Meta-Design Principles for Literate Workflows in Scientific Computing

Matthew Dinmore

Department of Information Systems, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250. E-mail: mdinmo1@umbc.edu

Abstract

Scientists and other domain experts often must construct elaborate programs to carry out their analyses, even though they are not professional programmers. Moreover, beyond simply performing computations, that have the ultimate objective of creating and sharing knowledge, of which these workflows are an intricate part. We introduce the concept of the literate workflow, which combines end-user programmability and knowledge capture to produce unified, reusable software/knowledge artifacts. In support of this paradigm, we enumerate five meta-design principles around which systems supporting literate workflows can be developed. A prototype incorporating these principles is presented.

1. Introduction

Scientists and others working in computationally-supported domains such as biology, physics and geospatial analysis often develop software as a component of their problem-solving process. These solutions, typically referred to as scientific workflows[1], manipulate data to produce an analytical result. However, though this process results in the creation of software programs, they are neither the principle focus of the work, nor are those creating them generally programmers; the real goal of the work is knowledge creation, and the domain experts performing this work can be classified as end-user developers. With this in mind, we argue that many of the findings from research in end-user programming can inform the development of domain-oriented environments in which these practitioners can more effectively work.

This discussion is not to suggest that concepts from end-user programming have not been previously applied to scientific workflows; on the contrary, the scientific and allied problem-solving communities have often been the subject of end-user programming research. Letondal and Mackay (2004) have studied bioinformaticists, the tensions between scientific work and the need to develop software in support of it. Whitley and Blackwell (2001) examined the efficacy of visual environments such as Labview in the sciences. Traynor and Williams (1997) studied end-user programming in the closely-related field of geospatial analysis, and Gantt and Nardi (1992) in their well-known paper illuminating various roles in end-user development looked at engineers using computer-

1. While we use the terms scientific computing and scientific workflow because of their recognition in the field, we intend this in the broadest sense, that of persons performing computationally-assisted problem solving in data intensive applications.

aided design systems. Despite this history, these research communities still have limited cross-domain recognition. For example, in the end-user development community's Research Agenda & Roadmap document (Paternò, Klann, & Wulf, 2003), the only mention of workflow is centered around the business environment, though past work and the need for future collaboration with the scientists as a class of end users is cited. Similarly, outlines for future work from the scientific workflow community (Gil et al., 2006; Deelman, 2007) make no mention of the potential contributions of end-user programming research to the development of workflows.

This paper is organized as follows. First, we examine the user experience requirements as elaborated upon in the workflow literature. We then look at the corresponding research results and direction from end-user development research. We introduce the concept of the literate workflow, which advocates literate programming as a paradigm to respond to a stated needs in scientific workflow for which there is not an adequate approach in the existing end-user programming body of work. Drawing on empirical findings from literate programming, end-user programming, cognitive psychology, knowledge management and documentation research, we present five principles for the design of literate workflow environments. Finally, we present the results of an exploratory case study into the efficacy literate workflows in satisfying their primary goal of knowledge representation and transfer through a unified software/knowledge artifact.

1.1 User-Centric Requirements in Scientific Workflow

We focus here on scientific workflows in order to differentiate them from business workflows (Ludäscher & Goble, 2005). In particular, a scientific workflow is data-centric; data flows though the workflow undergoing various transformations and processing that is generally computationally intensive. Business workflows, in contrast, are control-oriented, requiring authorized users to manipulate, review and approve documents passing through the workflow at each step. Control of business workflows is often exercised through centralized development and administration, while scientific workflows tend to be created by individuals or small, decentralized teams of scientists.

Another attribute of the scientific workflow is its role in domain and procedural knowledge creation and sharing leading to a need for annotation of the workflow (Chin, Ruby Leung, Schuchardt, & Gracio, 2002; Ludäscher & Goble, 2005). In fact, we should recognize this as the central goal of the work being performed rather than the creation of a piece of computer software; this perspective should drive our design model for systems supporting scientific computing.

The scientific workflow community has put forth a collection of research challenges (Gil et al., 2006). Among these are the capture and sharing of workflow descriptions and data provenance; reuse and repeatability; dynamicity, user interaction in the workflow and tailorability; intelligent assistance in creating and optimizing workflows; and collaborative development throughout an evolutionary life cycle for workflows. These requirements reiterate the primacy of knowledge capture and dissemination to scientific workflows, as well as being suggestive of a elements of the development paradigm in which they are created and evolved.

1.2 Survey of Applicable End-User Programming Research

If we consider the development of scientific workflows in the context of end-user programming (Sutcliffe & Mehandjiev, 2004), we can begin to apply the research results from that field to the creation of scientific workflow development environments. Many of the challenges and of objectives of the two research communities are similar. In particular, end-user programming offers guidance in three compelling areas. First, there has been substantial work in the development of tools, often visually-oriented, for non-programmers (Burnett, 1999). Most scientific workflow environments are visual programming systems and can benefit from human-computer interaction research findings. Additionally, other metaphors, including programming by demonstration (Cypher, 1993) and natural programming (Myers, Pane, & Ko, 2004) are applicable. Recent work offers support in the area of intelligent assistance by researching methods for converting narrative descriptions into programmatic structures (Liu & Lieberman, 2005).

Second, there is ongoing work in bringing software engineering principles to end users, especially in the area of debugging support (Burnett, Cook, & Rothermel, 2004). Using intelligent assistance, methods for incremental and interactive testing are offered to the user. Much of this work centers on the prevention of errors in spreadsheets, the most common end-user development tool, but also prone to costly errors (Panko, 1998). In scientific workflow applications, this is often addressed through efforts to document data provenance (Simmhan, Plale, & Gannon, 2005), though an expanded definition that includes elements of the workflow design itself is given by Greenwood, et. al. (2003). However, there appear to be no empirical studies of the frequency or impact of errors in scientific workflow.

Finally, we must recognize that end-user programming as a process is in many ways different from professional programming; as previously noted, the production of software is not the ultimate goal, but rather a means to an end. The concept of meta-design has emerged in the end-user development research community in response to this (Fischer, Giaccardi, Ye, Sutcliffe, & Mehandjiev, 2004). Meta-design refers broadly to the design of environments in which the users become owners of their own problems and designers of their own solutions. This notion extends beyond the software to the socio-technical setting in which this occurs, shaping the relationship among users and between professional developers and the user community, as described by the seeding, evolution-ary growth and reseeding process (Fischer, McCall, Ostwald, Reeves, & Shipman, 1994). With respect to workflows, the initial, scientifically-informed introduction of the compo-nent tools and services is an example of seeding, and the development of the workflows themselves represent evolutionary growth. Reseeding occurs when workflows are re-eng-ineered into new components and re-introduced into the users' toolbox.

A principle mechanism for achieving the goals of meta-design is underdesign. In underdesign, developers of the software environment intentionally design with the pur-pose of allowing for flexibility to address unanticipated requirements when the system is in use, thereby enabling the users to adapt the system to their evolving requirements. This is a significant change to the traditional waterfall, and even iterative spiral models of development, and is in many ways more of design challenge for professional software engineers. The developers of workflow environments are faced with this necessity, as well, and use-time adaptability (often referred to as support for dynamicity and experimentation) is a primary requirement in scientific workflow (Chin, Ruby Leung, Schuchardt, & Gracio, 2002; Gil et al., 2006).

Software is not the only artifact that falls within the SER life cycle; knowledge also evolves throughout this process, and the environment must support and encourage the creativity inherent in this kind of work (Giaccardi, & Fischer, 2005). We argue that the explicit representation of knowledge should be a first-class object. From the users' perspective, knowledge is the primary product, and this is well-understood in the scientific workflow community; Ludäscher and Goble remark that a workflow represents "...know-how about a scientific method that can be shared and reused" (Ludäscher & Goble, 2005p. 4). Deelman (2007) lists expert knowledge capture, as represented in workflows, as the number one future research requirement. Support for creativity has also been a research objective as it relates to workflow, and Faroq, Carroll and Ganoe (2005) enumerate requirements for doing this effectively.

Conversely, knowledge capture is an area notably absent from the end-user development literature is research into mechanisms for domain knowledge capture in conjunction with programming activities. As we have seen, this is the leading requirement in the scientific community, and is likely a requirement in other areas of knowledge work. We believe that a user-centric view of this challenge leads to re-centering the focus of end-user development environments for this class of users to first support explicit knowledge capture and secondarily (though not less effectively) support programming better aligns the resulting platform with the needs of the user. This work proposes a paradigm for an environment supporting end-user developed scientific workflows, and offers several meta-design principles for instantiating such an environment. We base this approach on the concept of literate programming.

1.3 Literate Programming

Literate programming (Knuth, 1992) represents a different way to think about programming, changing the focus of a program from something that tells a computer what to do to a means for humans to communicate with one another what the program does. Knuth's goal was to transform programs into literary works, structured to be easily read and shared among computer scientists. This was accomplished by organizing the program into chunks of marked-up narrative associated with the actual implementing the program; the chunks are ordered in way natural for explaining and understanding the program, not necessarily the compilation or execution order.

Literate programming never gained a significant foothold among computer scientists, and since it was generally held to be only for the explanation of algorithms by them rather than a practical tool for day-to-day programmers, there was also never broad adoption among programmers. Further, limited tool support required the programmer to effectively use two languages – one for the code and one for typesetting – and additional markup to glue all of the parts together.

There was limited empirical research into the efficacy of the approach over traditional programming though, for example, Bertholf and Scholtz (1993) demonstrated that literate programs are more comprehensible. Oman and Cook (1990) showed increased performance in maintenance tasks for code presented in a book-like style, and Shum and Cook (1994) compared comments in structured and literate programs, finding more descriptive narrative in the latter, though not necessarily more text, suggesting greater information content.

In recent years, there has been a resurgence in interest in literate programming (Pieterse, Kourie, & Boake, 2004), partially fueled by the availability of new technologies such as integrated development environments which can naturally capture and present a literate program without the need for additional markup. Technologies such as extensible markup language (XML) offer a standard persistence format, and concepts such as theme-based literate programming (Kacofegitis & Churcher, 2002) present a mechanism for embedding multiple representations targeted to different audiences within a single program.

We are offered a step in the direction toward the end user for literate programming in an argument made by Holmes (2003) on the need for a perspicuous approach to programming. In this paradigm, the programmer incorporates documentation into the program, as is done in literate programming, but rather than simply documenting the algorithms, the narrative becomes a sort of user manual, explaining how to use the program. We can easily envision a themed view in which both algorithmics and user guidance are contained within a single object, thereby capturing the design and application knowledge in an executable, reusable artifact. If we now place the end user in the role of author and assembler of all of these elements, we arrive at a model for end-user development in which both the knowledge capture and programming objectives are achieved, a concept we have previously presented (Dinmore, & Norcio, 2007) and now expand upon with respect to the scientific workflow domain.

2. Literate Workflows

A literate workflow is a user-assembled program instantiating a procedure and organized around a structured documentary form. This places the emphasis on the explicit representation of knowledge to support dissemination and reusability, while simultaneously incorporating the software components (or representative proxies of them) in a unified artifact.

A literate workflow is composed of chunks, each of which has one or more elements. Elements are the basic building blocks assembled by the user, and chunks offer a mechanism for grouping related elements in a meaningful way in the user domain For example, a chunk might consist of a narrative, a software component that implements an action, and an image showing sample results. Elements within a chunk can be assigned to one or more themes, and would only be displayed when that theme is active.

As in literate programming, the presentation order of a literate workflow does not have to match the execution order, however we expect that it often will because of the intuitive relationship between the two. When used in combination with theme assignments to chunks and elements, different navigation paths through the program may be presented (Kacofegitis & Churcher, 2002). Consider a purely narrative view that is not related to the execution order, or a tutorial view for novice users. This is enabled through another construct from literate programming, the use of contents, an index and cross reference system; each theme would have its own ordered subset of the master contents.

Literate workflows achieve versimilitude, or the inclusion of software and documentation in a single artifact, by combining persistent representations of each element in a standard, serializable formate. The most obvious choice for this is XML, and several candidate schemas already exist, for example the Business Process Execution Language (BPEL) that has been previously examined for its ability to represent scientific workflows (Slominski, 2007; Akram, Meredith, & Allan, 2006); it satisfies most, but not all requirements, though some users have implemented extensions to remedy this (Wasserman et al., 2007). XML's flexibility in allowing embedding of multiple schemas enables us to create a schema for those additional elements we require to support literate workflows and combine them with a suitable extant schema; a number of recent literate programming projects have used XML, suggesting a starting point for a workflow-centric schema (Pieterse, Kourie, & Boake, 2004; Aguiar, & David, 2005; Kacofegitis & Churcher, 2002). Additionally, we can use documentation-centric XML schemas, for example the Darwin Information Typing Architecture (DITA) (Priestley, 2001), to add documentary structure to the artifact.

2.1 Design Principles for Literate Workflows

With this as a foundation, we now present and discuss five meta-design principles for literate workflow environments; these are summarized in table 1. Throughout, we will use the Cognitive Dimensions of Notations (CDN) (Green, 1989) as a framework for discussing our design decisions. This has been used in end-user development research previously (Blackwell, Burnett, & Peyton-Jones, 2004) because of its focus on the user interaction aspects of notations as we would find in computer programs; it has been specifically applied to visual programming environments (Green & Petre, 1996), research that informs several of our principles.

Principle	**Key Points**
Documentation-centric interaction	• Linear, notebook presentation • Natural interface
Chunk everything	• Discrete, interwoven knowledge • Component-based architecture
Provide multiple views	• Address different audiences • Provide for uses beyond workflow execution
Support reuse processes throughout	• Locating, comprehending, modifying and sharing • Knowledge and software reuse processes • Component-level reuse
Offer intelligent assistance	• Component selection • Workflow composition • Debugging

Table 1. Literate workflow meta-design principles.

2.1.1 Provide a Documentation-centric Interaction Paradigm

As a general model of interaction, we propose a visual, integrated environment within which users create and edit narrative and embed presentation interfaces to local and remote components that perform that workflow; this is a conceptual implementation of the "scientist's notebook" (Chin, Ruby Leung, Schuchardt, & Gracio, 2002), and offers a natural (Myers, Pane, & Ko, 2004) and domain-oriented (Fischer, 1994) end-user development environment. This model is presented in figure 1. We should be clear here in noting that a notebook-style view for implementing literate programming principles is not novel and can be found in commercial products such as Mathematica (Wolfram Research, 2007; Hu, 2004) and research tools such as MathModelica (MathCore, 2006; Fritzson, Gunnarsson, & Jirstrand, 2002).

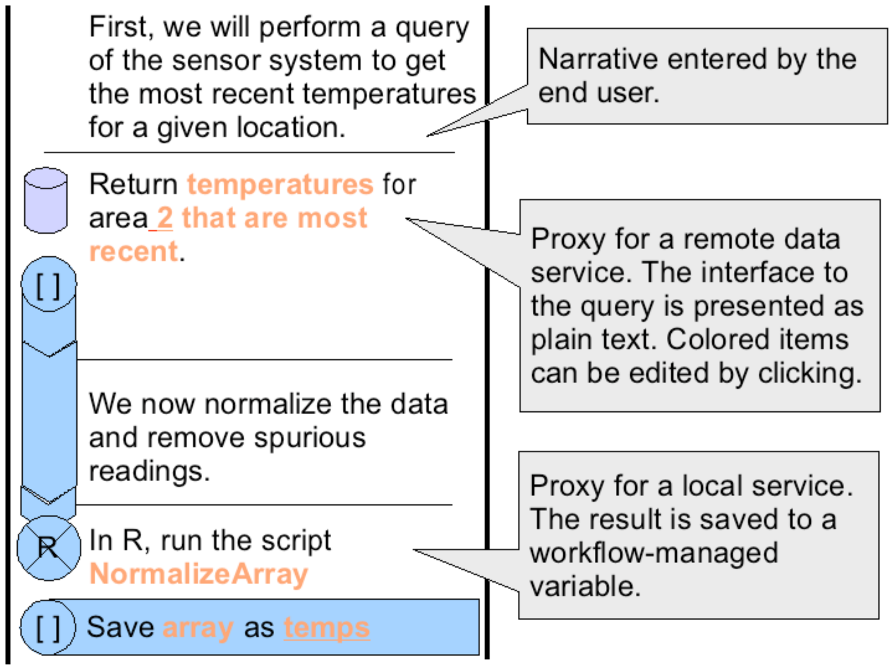
First, we will perform a query of the sensor system to get the most recent temperatures for a given location.
Narrative entered by the end user.
Return temperatures for area 2 that are most recent.
[ ]
Proxy for a remote data service. The interface to the query is presented as plain text. Colored items can be edited by clicking.
We now normalize the data and remove spurious readings.
Proxy for a local service. The result is saved to a workflow-managed variable.
R
In R, run the script NormalizeArray
[ ]
Save array as temps

Figure 1. Notional display for a portion of a literate workflow showing narrative interleaved with interface proxies for component services.

From a CDN perspective, this design provides a closeness of mapping between the problem and the instantiation of the solution. Dataflow is a natural way to think about scientific computational problems, as is a top-down, book-style approach to documentation for scientists. The representation also lends itself to simplicity and consistency, though consistency can only be completely judged against an actual implementation. The interleaving of narrative and active components allows for interactive workflows, in which the user performs decision-making based previous results; this is a stated requirement in scientific workflows (Gil et al., 2006). This interleaved model has also been found to be a more effective presentation for understanding and performing a procedure (Duggan & Payne, 2001).

In keeping with meta-design concepts, we combine the design and use-time environment, though themes enable us to target use-time presentations. We adopt a spreadsheet-like execution metaphor, as has been done in other scientific workflow tools (Rygg, Roe, & Wong, 2006). For simple, linear workflows, which seem to constitute the majority of the requirements (Rygg, Roe, & Wong, 2006), this model enables a straightforward data flow-oriented construction of a workflow. For more complex designs, the use of chunking from literate programming and variables to maintain and transfer state and data among chunks, provides for a wide range of solutions. These mechanisms also support the visibility and juxtaposability in the CDN framework. Finally, we note here and will expand upon below that the use of themes allows us to provide graduated steps in the presentation between the design and use-time views to accommodate a range os user needs.

Though we do not wish to overly constrain the implementation of literate workflow environments, there is one aspect of the design that differentiates the literate workflow from many other workflow and visual programming tools, and that is the linear, or 1-dimensional presentation of the program. Moat visual development tools provide a 2-dimensional “canvas” upon which components can be arranged and interconnected. With our focus on a narrative, book-like flow, a linear presentation is more appropriate; this style has also been adopted by other tools recently, including Automator (Apple, 2005), part of Apple Inc.’s Mac OS X, and the Automator-inspired Marmite web programming research project (Wong, & Hong, 2006).

A linear presentation also addresses a potential conceptual problem found in 2-dimensional presentations, namely that of secondary notations. Petre (1995) describes how experts in a domain may make use of the physical arrangement, color and other display attributes of objects on the canvas to convey knowledge, and that novices have difficulty in both producing and interpreting this kind of secondary notation. Green and Petre (1992), in examining visual programming environments, find that secondary notation and “viscosity” are two considerations designers of visual environments must understand. Viscosity, which is resistance to making changes in a program, is increased in a visual environment unless, as they note, “a good diagram editor is used” (Green & Petre, 1992, p. 157). This is because the myriad connections to other components must be re-established if an insertion is made, and this may also require physical re-arrangement of components on the canvas. In a 1-dimensional presentation, insertion, deletion and movement (of individual or groups of items) are the only allowed editing options, eliminating the need for both re-linking and re-arranging.

A 1-dimensional view offers its own challenges, for example how to visualize parallel constructs, which are often found in scientific computing. Workflows may also tend to become long, requiring the user to scroll about more frequently. Without being prescriptive, potential solutions to both issues include split-views, as commonly found in text editors, selective code or narrative elision (Cockburn & Smith, 2003) to hide segments that are not of immediate interest, and overviews and fisheye views (Cockburn, 2001).

2.1.2 Chunk Everything

Chunking (Miller, 1956) is the cognitive process of organizing information into recognizable groupings and is fundamental to how humans process information about and understand the world. Green (1989) discusses chunks extensively in the exposition of the CDNs, particularly around the dimensions of role-expressiveness, in which chunks visually indicate the part they play in the plan of the program, and in reducing the complexity of hard mental operations through decomposition. Chunking has been a part of the literate programming paradigm from the beginning. Each logical piece of a literate program is a chunk and contains information such as code, variable definitions or narrative descriptions. Chunking like items improves recall, and recall – or the ability to locate and comprehend an object – is central to reusability (Fischer, 2001). This concept has been shown to apply to the organization of computer programs (Norcio & Kerst, 1983), and we argue here that literate workflows, as computer programs, will benefit from chunking both in their software and knowledge aspects.

Chunking of the elements of a literate workflow should therefore increase their reusability within the user domain, but we can achieve additional reuse benefit by enabling the incorporation of well-defined, professionally-developed software artifacts; this idea is the basis for component-based software engineering.

Component-based software engineering (CBSE) is an approach in which functionality in a software system is encapsulated in relatively independent executable objects with well-defined interfaces to enable them to be integrated (Ning, 1997). This allows traditional developers to plug-and-play with reusable pieces of code, but it has also been recognized as an appropriate approach to abstracting the details of professionally-developed code for end-user developers (Mørch et al., 2004).

Further, they enable the kind of interaction envisioned in the meta-design paradigm by enabling professional developers to introduce componentized seeds to the shared design environment and for end-users to modify these, resulting in tailored and assembled components that can themselves be presented as reusable components. This also facilitates re-engineering of user-designed solutions and their reseeding to the design environment.

Componentization is also a recommended design practice for successful reuse in knowledge management systems, particularly those supporting emergent knowledge processes (EKP), such as scientific problem solving (Markus, Majchrzak, & Gasser, 2002). This is necessary because of the rapidly evolving and unpredictable nature of of this class of problems, and fits well with the SER model.

Component-based approaches are common in scientific workflow (Gannon, 2007), but there are some important differences from typical CBSE. In workflow applications, components are arranged in time for human-scale applications, rather than assembled to make a single executable. Also, components are usually distributed or are, in fact, proxies for remote services. This means that information about state is not persisted within the component itself, but in some other location; the workflow engine must maintain the references necessary to access this state information at runtime.

2.1.3 Provide Multiple Views

As we saw in our review of the literate programming literature, themes provide a powerful means to providing multiple views tailored to particular audiences within the same artifact; with chunking, they are relatively easy to implement and make use of through the assignment of tags and, if desired, alternate presentation orders.

The user-selected level of presentation offered by themes supports the multi-level model of tailoring (Kyng, & Mathiassen, 1997). In this model, users have access to increasing levels of tailorability at which they can make changes to the workflow, from customization in which the user modifies parameters, to integration, in which they add new components to the workflow, to extension, in which they are able to wrap external entities to provide a workflow-compatible presentation interface. In that users learn by studying and modifying programs written by others (Mackay, 1990), these successive levels of interaction – selectable by the user – allow gradually increased participation in the SER process.

A literate workflow is largely intended to be a documentary form, and themes enable us to offer a variety of types of users what they need in their documentation. Novick and Ward (2006), in a study of what users want in documentation, identify five key areas: navigation, appropriateness of explanations, problem-oriented organization, presentation, and completeness and correctness. To highlight a few ways themed literate workflows can satisfy these needs, consider the following. Navigation is supported by the index and table of contents structures from literate programming; each theme may offer a different view of these. A search capability, as we would expect to find in a word processor, also supports this. Themes, with selective elision, enable a user to set the level of explanation to their needs, and the use of narrative text offers a lowest-common denominator form of symbolic presentation. Themes, with alternate traversal paths, provide problem-oriented organization, and the interleaving of components, text and other media delivers a rich presentation. Finally, the narrative's almost journal paper-like form encourages completeness.

Providing multiple views addresses the CDN concepts of visibility, premature commitment, role expressiveness and abstraction. Views act as a filter, enabling users to see only what they need to for their application and level of knowledge. Because the workflow is the interface, it is possible to see what subsequent steps will be, thereby avoiding premature commitment. As noted previously, the visibility of elements of the workflow – enhanced by the use of multiple views – helps make clear the role of each object. Finally, the workflow is a natural abstraction of the user's work, and the embedded components are themselves abstractions of computational processes.

2.1.4 Support Knowledge and Software Reuse Cycles Throughout

In addition to knowledge created by the experiment, the problem solving, design and provenance information that is part of the workflow are valuable, reusable knowledge, and the system should actively encourage their capture (Ludäscher & Goble, 2005). Associating this knowledge with the software in a single artifact supports reusability. The system itself must encourage reuse throughout the process by making reusable objects easy to create, find and apply.

A literate workflow system must support both software and knowledge reuse, and do so through unified knowledge/software artifacts. There are a number of models of reuse from which we can draw insights about both processes. Basili and Rombach (1991) summarize the reuse literature and propose a comprehensive model of reuse, which takes an object and activity-based view: reuse candidate objects are selected via a reuse process to satisfy needs in the development context. Fischer, Henninger and Redmiles (1991) presents a conceptual model for software reuse in which there are three defined phases: creation, location and comprehension; in a cyclic environment, such as the SER model suggests, creation can alternately be conceptualized as modification (Fischer, 2001). We can further extend this model with separate recognition for a sharing phase (Sumner, & Dawe, 2001). Markus (2001) identifies four stages in knowledge reuse: capturing or documenting, packaging, distributing or disseminating and reusing, which consists of recall and recognition. A literate workflow system must encompass all of these aspects of reuse in intrinsically.

Krueger (1992), in his survey of software reuse, notes four truisms: (1) reuse techniques must reduce the cognitive distance between the conceptual solution and its implementation; (2) a reuse technique is only effective if it is easier to reuse an object than develop it in the first place; (3) successful selection of a reusable object requires understanding what that object does; and (4) reuse is ineffective is a developer can build an object faster than finding it.

Markus (2001) presents a typology of knowledge reuse situations, of which there are four: (1) reuse by shared knowledge producers; (2) reuse by shared work practitioners; (3) reuse by expertise-seeking novices; and (4) reuse by secondary knowledge miners. Literate workflows attempt to satisfy all four situations, though how a producer of a workflow shapes its knowledge content will largely determine its effectiveness in each situation, and we can make the generalization that each of these situations in order is better served by more explicit and complete representations, as each represents a greater distance from the producer. However, the cost of capturing this knowledge also increases, as does the challenge. Markus notes the increasing difficulty as we move from documenting for ourselves, to documenting for similar others, to finally attempting to document for dissimilar others.

A recurring theme in the reuse literature is the need for some documentation beyond the software to enable it to be found in context, understood and applied. This may take the form of metadata attributes assigned to reusable objects; Basili and Rombach (1991) enumerate a number of attributes which can characterize a reuse candidate. Markus (2001) also recognizes the need for "repository records" in knowledge reuse, that enable others to find stored knowledge, but notes the inherent difficulties in maintaining these records, largely due to the cost in time and resources. Incentivizing workers may be effective, depending on the situation and whether the benefit (even with incentives) outweighs the perceived cost; Grudin (1994) observed this as a key issue in the acceptance of computer-supported cooperative work systems, of which workflow is a clear example, though it is certainly more broadly applicable.

We hope to lower the barrier to preparing literate workflows for reuse by gathering much, if not all of the reuse metadata necessary to share the workflow in a repository from within the narrative of the workflow itself. In a well-documented workflow, the combination of explicit text provided by the author, as well as inferences that can be made automatically from the components used, data sources selected, and outputs presented, offer a rich context for describing the workflow. Medeiros, et. al. (2005) describe their work on a workflow-based geospatial analysis system that provides a common component structure for encapsulating any kind of digital content as part of a workflow, and leveraging Semantic Web technologies to enable reuse. In particular, richly-described components support both component selection and workflow composition, which, as we will see later, enables intelligent assistance in these tasks.

It is therefore clear that knowledge capture in association with the software artifact is critical to successful reuse. Annotation of components and workflows enables location and comprehension, the latter which is consistent with the CDN dimension of role-expressiveness. We argue here that, in an end-user programming situation, the knowledge about the software artifact and the task it represents are largely the same, and that encouraging problem-solving knowledge reuse through appropriate mechanisms within the literate workflow environment drives software reuse.

2.1.5 Offer Intelligent Assistance

A final design principle is to provide intelligent assistance. This capability offers the user appropriately timed help to complete a task, while not interfering with the creative process. This can be implemented through a number of technologies and to a range of degrees, from simple, embedded contextual help (Ellison, 2007) to a mixed-initiative system (Hearst, Allen, Guinn, & Horvtz, 1999). There are three key stages in the development process where this can be effective: selecting components and task automation, solution reuse, and debugging a workflow.

Component selection is one example in our environment of the location phase of the software reuse process model (Fischer, Henninger, & Redmiles, 1991), though as we previously discussed, we are not limited to only reusing components. Intelligent assistance can help in locating a needed component from a library by using cues provided by the user, including the desired inputs and outputs and descriptions of what the user wishes to accomplish. Henninger (1991) discusses CodeFinder, a system that makes recommendations about reusable code selection based on examples. Similar work has been done in scientific workflow, where semantic analysis and templates to perform assisted composition (Gil, 2007).

A step beyond this is task automation. Here, the intelligent assistant recognizes what intermediate steps are required to complete a task and offers them as a recommendation to the user. While this may involve component selection, as above, this is more sophisticated because multiple steps are involved and some understanding of the desired outcome is necessary. Solutions may be provided through template matching or case-based reasoning (Leake, 1996).

Generalizing further, we can offer complete, reusable workflows to users based on their needs. Gong, Nakamura, Yura and Go (2006) discuss a recommendation system (BAAQ) that allows a user to ask a question, and based on indexed annotations and feedback from previous users, creates a ranked list of candidate solutions. This is in many ways similar to analogical matching found in the reuse literature, e.g. Maiden & Sutcliffe, 1992; Bhansali, 1995.

The last area in which intelligent assistance can be effective is debugging support. In end-user programming research, there has been significant work toward this objective. Burnett, Rothermel and Cook (2004) discuss methods such as “What You See Is What

You Test" (WYSIWYT) in which untested portions of a program under development are visually highlighted, "Help Me Test" that intelligently generate test vectors for the user upon their request (perhaps coupled with WYSIWYT), the use of assertions and a surprise-explain-reward strategy for encouraging testing. While designed around a spreadsheet environment, these devices are equally applicable in a literate workflow paradigm. Active assistance in composing a workflow, as well as assistance in the debugging process should reduce errors, and thereby decrease the propensity towards the CDN dimension of error-proneness.

3. "Blackbox" Empirical Study

We performed in initial empirical study to validate the central claim of literate workflows, that they improve reuse and domain knowledge transfer. This study does not examine the individual design principles as factors, but rather treats them holistically; clearly, future studies will have to consider the design ranges and impacts of each in turn.

For this study, we draw on Fischer's Location-Comprehension-Modification Cycle for reuse (Fischer, 2001), noting that literate workflows should be both easier to locate and easier to comprehend than their non-literate counterparts. Comprehension is not only critical to reuse, but fundamental to knowledge transfer; measuring the relative improvement in ability to locate and comprehend therefore demonstrates the efficacy of the literate workflow concept.

3.1 Methodology

We employed a between-subjects design in which each subject was assigned to a group using either traditional workflows or a literate workflow. Each subject performed two tasks: first, to compare the ability to locate an object, the subject was given a written problem statement and asked to retrieve by keyword search a workflow which would best solve the problem. Workflows were created in a traditional workflow tool (KEPLER?) and in a literate workflow style, then indexed and made accessible through a common search engine (LUCENE OR GOOGLE DESKTOP?).

In the second task, studying comprehension, subjects were provided with a work-flow in either traditional (KEPLER) or literate format and asked to answer several ques-tions about what it does.

3.2 Results

4. Discussion

5. Future Research

The study presented here is preliminary. As noted previously, each principle must be considered, and different implementation options for each weighed. It is also unclear whether the principles defined here are exhaustive. Finally, this study did not examine the Modification/Creation aspect of the Location-Comprehension-Modification Cycle, or, in the extended model, the sharing aspect. Modification can be studied in a straightforward manner; studies in both literate programming (CITE CODE MOD IN LP) and end-user programming (SOMETHING ON TAILORING FROM EUD) demonstrate potential approaches to this. Creation is more challenging to study empirically because, in a rich environment, the possible solution space for a problem is large, leading to potentially subjective considerations of "correctness," especially in judging the literate aspect of the composition. Sharing is also difficult to study in the laboratory because of the significant impact the socio-technical environment on it, including issues of incentivization, as we have discussed.

6. Conclusions

7. References

Aguiar, A. & David, G. (2005). *WikiWiki weaving heterogeneous software artifacts*. San Diego, California.

Akram, A., Meredith, D., & Allan, R. (2006). *Evaluation of BPEL to Scientific Workflows*. Paper presented at the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06), Los Alamitos, CA.

Apple. (2005). Automator: Your personal automation assistant. Retrieved June 28, 2007, from http://www.apple.com/macosx/features/automator/

Basili, V. R. & Rombach, H. D. (1991). Support for comprehensive reuse. *Softw. Eng. J*, *6*(5), 303-316.

Bertholf, C. F. & Scholtz, J. (1993). *Program Comprehension of Literate Programs by Novice Programmers*. Paper presented at the Empirical Studies of Programmers - Fifth Workshop, Pal Alto, California.

Bhansali, S. (1995). A hybrid approach to software reuse. *Proceedings of the 1995 Symposium on Software reusability*, 215-218.

Blackwell, A. F., Burnett, M., & Peyton-Jones, S. (2004). *Champagne Prototyping: A research technique for early evaluation of complex end-user programming systems*. Paper presented at the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC04).

Burnett, M. (1999). *Visual Programming*. New York.

Burnett, M., Cook, C., & Rothermel, G. (2004). End-user software engineering. *Commun. ACM*, *47*(9), 53-58.

Chin, G. J., Ruby Leung, L., Schuchardt, K., & Gracio, D. (2002). *New paradigms in problem solving environments for scientific computing*. San Francisco, California, USA.

Cockburn, A. & Smith, M. (2003). Hidden messages: evaluating the efficiency of code elision in program navigation. *Interacting with Computers*, *15*, 387-407(21).

Cockburn, A. (2001). Supporting Tailorable Program Visualization Through Literate Programming and Fisheye Views. *Information and Software Technology*, *43*(13), 745-758.

Cypher, A. (Ed.). (1993). *Watch What I Do: Programming by Demonstration*. Cambridge: MIT Press.

Deelman, E. (2007). Looking into the Future of Workflows: The Challenges Ahead. In I. Taylor, e. Deelman, D. Gannon, & M. Shields (Eds.), *Workflows for e-Science*. (pp. 475-481). London: Springer-Verlag.

Dinmore, M. & Norcio, A. F. (2007). *Literacy for the Masses: Integrating Software and Knowledge Reuse for End-User Developers Through Literate Programming*. Paper presented at the IEEE Information Reuse and Integration '07, Las Vegas, NV.

Duggan, G. B. & Payne, S. J. (2001). Interleaving Reading and Acting While Following Procedural Instructions. *Journal of Experimental Psychology*, *7*(4), 297-307.

Ellison, M. (2007). Embedded user assistance: the future for software help? *interactions*, *14*(1), 30-31.

Farooq, U., Carroll, J. M., & Ganoe, C. H. (2005). *Supporting creativity in distributed scientific communities*. Sanibel Island, Florida, USA.

Fischer, G. (1994). Domain-oriented design environments. *Automated Software Engineering*, *1*(2), 177-203.

Fischer, G. (2001). *The Software Technology of the 21st Century: From Software Reuse to Collaborative Software Design*. Paper presented at the International Symposium on Future Software Design, ZhengZhou, China.

Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A. G., & Mehandjiev, N. (2004). Meta-design: a manifesto for end-user development. *Commun. ACM*, *47*(9), 33-37.

Fischer, G., Henninger, S., & Redmiles, D. (1991). Cognitive tools for locating and comprehending software objects for reuse. *Software Engineering, 1991. Proceedings., 13th International Conference on,* 318-328.

Fischer, G., McCall, R., Ostwald, J., Reeves, B., & Shipman, F. (1994). Seeding, evolutionary growth and reseeding: supporting the incremental development of design environments. *SIGCHI conference on Human factors in computing systems: celebrating interdependence*, *Proceedings of the SIGCHI conference on Human factors in computing systems: celebrating interdependence*, 292-298.

Fritzson, P., Gunnarsson, J., & Jirstrand, M. (2002). *MathModelica - An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming*. Paper presented at the 2nd International Modelica Conference, Oberpfaffenhofen, Germany.

Gannon, D. (2007). Component Architectures and Services: From Application Construction to Scientific Workflows. In I. Taylor, E. Deelman, D. Gannon, & M. Shields (Eds.), *Workflows for e-Science*. (pp. 174-189). London: Springer-Verlag.

Gantt, M. & Nardi, B. A. (1992). Gardeners and gurus: patterns of cooperation among CAD users. *SIGCHI conference on Human factors in computing systems*, *Proceedings of the SIGCHI conference on Human factors in computing systems*, 107-117.

Giaccardi, E. & Fischer, G. (2005). *Creativity and Evolution: A Metadesign Perspective*. Paper presented at the Design System Evolution: Sixth International Conference of the European Academy of Design (EAD06), Bremen, Germany.

Gil, Y. (2007). Workflow Composition: Semantic Representations for Flexible Automation. In I. Taylor, E. Deelman, D. Gannon, & M. Shields (Eds.), *Workflows for e-Science*. (pp. 244-257). London: Springer-Verlag.

Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., et al. (2006). Report on the 2006 NSF Workshop in Challenges of Scientific Workflows. Retrieved June 25, 2007, from http://vtcpc.isi.edu/wiki/images/b/bf/NSFWorkflow-Final.pdf

Gong, X., Nakamura, K., Yura, K., & Go, N. (2006). *Toward building grid applications in bioinformatics*. Hobart, Tasmania, Australia.

Green, T. R. G. & Petre, M. (1992). When Visual Programs are Harder to Read than Textual Programs. *ECCE-6 (6th European Conference on Cognitive Ergonomics)*, *ECCE-6 (6th European Conference on Cognitive Ergonomics)*(6th European Conference on Cognitive Ergonomics).

Green, T. R. G. (1989). Cognitive Dimensions of Notations. In A. Sutcliffe & L. Macaulay (Eds.), *People and Computers V.* (pp. 443-460). Cambridge, UK: Cambridge University Press.

Green, T. R. G. & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, *7*, 131-174.

Greenwood, M., Goble, C., Stevens, R., Zhao, J., Addis, M., Marvin, D., et al. (2003). *Provenance of e-Science Experiments - experience from Bioinformatics*. Paper presented at the The UK OST e-Science second All Hands Meeting (AHM'03), Nottingham, UK.

Grudin, J. (1994). Groupware and social dynamics: eight challenges for developers. *Commun. ACM*, *37*(1), 92-105.

Hearst, M., Allen, J. E., Guinn, C. I., & Horvtz, E. (1999). Mixed-initiative interaction. *Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems]*, *14*, 14-23.

Henninger, S. (1991). Retrieving software objects in an example-based programming environment. *Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*, 251-260.

Holmes, N. (2003). The case for perspicuous programming. *Computer*, *36*, 102-104.

Hu, Z. (2004). A Simple Mesh Generator in Mathematica. Retrieved June 28, 2007, from http://library.wolfram.com/infocenter/MathSource/5475/

Kacofegitis, A. & Churcher, N. (2002). Theme-based literate programming. *Software Engineering Conference, 2002. Ninth Asia-Pacific,* 549-557.

Knuth, D. (1992). *Literate Programming*. Stanford: Center for the Study of Language and Information.

Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv*, *24*(2), 131-183.

Leake, D. (1996). CBR in Context: The Present and Future. In D. Leake (Ed.), *Case-Based Reasoning: Experiences, Lessons & Future Directions*. (pp. 1-35). Menlo Park: AAAI Press/MIT Press.

Letondal, C. & Mackay, W. E. (2004). *Participatory programming and the scope of mutual responsibility: balancing scientific, design and software commitment*. Toronto, Ontario, Canada.

Liu, H. & Lieberman, H. (2005). Metafor: visualizing stories as code. *10th international conference on Intelligent user interfaces*, *Proceedings of the 10th international conference on Intelligent user interfaces*, 305-307.

Ludäscher, B. & Goble, C. (2005). Guest Editors' Introduction to the Special Section on Scientific Workflows. *SIGMOD Record*, *34*(3), 3-4.

Mackay, W. E. (1990). *Patterns of sharing customizable software*. Los Angeles, California, United States.

Maiden, N. & Sutcliffe, A. (1992). Exploiting reusable specifications through analogy. *Commun. ACM*, *35*(4), 55-64.

Markus, L., Majchrzak, A., & Gasser, L. (2002). A Design Theory for Systems that Support Emergent Knowledge Processes. *MIS Quarterly*, *26*(3), 179-212.

Markus, M. L. (2001). Toward a Theory of Knowledge Reuse: Types of Knowledge Reuse Situations and Factors in Reuse. *Journal of Management Information Systems*, *18*(1), 57-94.

MathCore. (2006). About MathModelica System Designer. Retrieved June 28, 2007, from http://www.mathcore.com/products/mathmodelica/

Medeiros, C. B., Perez-Alcazar, J., Digiampietri, L., Pastorello, G. J., Santanche, A., Torres, R., et al. (2005). Woodss and the Web: annotating and reusing scientific workflows. *SIGMOD Review*, *34*(3), 18-23.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, *63*(2), 81-96.

Kyng, M. & Mathiassen, L. (Eds.). (1997). *Three Levels of End-User Tailoring: Customization, Integration, and Extension*. Cambridge: The MIT Press.

Mørch, A., Stevens, G., Won, M., Klann, M., Dittrich, Y., & Wulf, V. (2004). Component-based technologies for end-user development. *Commun. ACM*, *47*(9), 59-62.

Myers, B. A., Pane, J. F., & Ko, A. (2004). Natural programming languages and environments. *Commun. ACM*, *47*(9), 47-52.

Ning, J. Q. (1997). *Component-Based Software Engineering (CBSE).*

Norcio, A. F. & Kerst, S. (1983). Human Memory Organization for Computer Programs. *Journal of the American Society for Information Science*, *34*(2), 109-114.

Novick, D. G. & Ward, K. (2006). *What users say they want in documentation*. Myrtle Beach, SC, USA.

Oman, P. W. & Cook, C. R. (1990). Typographic style is more than cosmetic. *Commun. ACM*, *33*(5), 506-520.

Panko, R. (1998). What We Know About Spreadsheet Errors. *Journal of End User Computing*, *10*(2), 15-21.

Paternò, F., Klann, M., & Wulf, V. (2003). Research Agenda and Roadmap for EUD. *Deliverables of the Network of Excellence on End-User Development.*

Petre, M. (1995). Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, *38*(6), 33-44.

Pieterse, V., Kourie, D. G., & Boake, A. (2004). *A case for contemporary literate programming*. Stellenbosch, Western Cape, South Africa.

Priestley, M. (2001). *DITA XML: a reuse by reference architecture for technical documentation*. Sante Fe, New Mexico, USA.

Rygg, A., Roe, P., & Wong, O. (2006). *GPFlow: An Intuitive Environment for Web Based Scientific Workflow*.

Shum, S. & Cook, C. (1994). *Using literate programming to teach good programming practices*. Phoenix, Arizona, United States.

Simmhan, Y. L., Plale, B., & Gannon, D. (2005). A survey of data provenance in e-science. *SIGMOD Rec*, *34*(3), 31-36.

Slominski, A. (2007). Adapting BPEL to Scientific Workflows. In I. Taylor, E. Deelman, D. Gannon, & M. Shields (Eds.), *Workflows for e-Science*. (pp. 208-226). London: Springer-Verlag.

Sumner, T. & Dawe, M. (2001). *Looking at digital library usability from a reuse perspective*. Roanoke, Virginia, United States.

Sutcliffe, A. & Mehandjiev, N. (2004). Introduction to Special Edition on End-User Development. *Commun. ACM*, *47*(9), 31-32.

Traynor, C. & Williams, M. G. (1997). *A study of end-user programming for geographic information systems*. Alexandria, Virginia, United States.

Wasserman, B., Emmerich, W., Butchart, B., Cameron, N., Chen, L., & Patel, J. (2007). Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling. In I. Taylor, E. Deelman, D. Gannon, & M. Shields (Eds.), *Workflows for e-Science*. (pp. 428-449). London: Springer-Verlag.

Whitley, K. N. & Blackwell, A. F. (2001). Visual Programming in the Wild: A Survey of LabVIEW Programmers. *J. Vis. Lang. Comput*, *12*(4), 435-472}, bibsource = {DBLP, http://dblp.uni-trier.de.

Wolfram Research. (2007). Wolfram Mathematica: Features. Retrieved June 28, 2007, from http://www.wolfram.com/products/mathematica/features/

Wong, J. & Hong, J. (2006). *Marmite: end-user programming for the web*. Montr\&\#233;al, Qu\&\#233;bec, Canada.