Software Engineering Issues for Network Computing

Carlo Ghezzi and Giovanni Vigna

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32
20133 Milano, Italy
ghezzi|vigna@elet.polimi.it

Abstract. The Internet is becoming the infrastructure upon which an increasing number of new applications are being developed. These applications allow new services to be provided and even new business areas to be opened. The growth of Internet-based applications has been one of the most striking technological achievements of the past few years. Yet, there are some risks inherent in this growth. Rapid development and reduced time to market have probably been the highest priority concerns for application developers. Therefore, these developments proceed without following a disciplined approach. We argue that the resulting applications will become the legacy systems of the near future, when the quality of these systems will need improvement but, at the same time, modifications will be hard to make in an economical and reliable way. In this paper we discuss the need for a software engineering approach to the development of network applications. In particular, we discuss a possible research agenda for software engineering research by looking at two specific areas: the World Wide Web and applications based on mobile code.
Keywords and phrases: Internet, World Wide Web, mobile code, software engineering, software quality, software development process.

1 Introduction

Since the beginning of the 1990's, use of the Internet and the World Wide Web (WWW) has exploded. Today, there are more than 50 million users from about 200 countries; the yearly rate of growth has been more than 50%, and the number of forecasted users by the year 2000 is approximately 380 million [20].

In recent years, there has been a shift both in the way the Internet is used and in how its potential is perceived by technology developers, service providers, and users. The Internet is not merely seen as a communication infrastructure that allows people to communicate in a fast, cheap, and reliable way; it is increasingly seen as the infrastructure upon which new services, new applications, and even new and previously unforeseen types of social processes are becoming possible. For example, electronic commerce may revolutionize the way business is done.

As another example, interactive distance learning and tutoring may change the way knowledge is transferred and will support new kinds of learning processes.

A new field is therefore emerging: *network computing*. By this, we mean computing where the computational infrastructure is a large set of autonomous and geographically distributed computers, connected by the Internet. In this paper we discuss critically the current state of network computing, to understand the current risks that are inherent in its growth.

Rapid development and reduced time to market seem to be the major concerns that drive the developments of network computing applications. The implementation technologies used for new developments are often unstable; in some cases they are just partially developed prototypes. Applications are developed in an *ad hoc* manner, without following disciplined design approaches, and often with little concern for their qualities, such as reliability or modifiability. We argue that these systems are likely to become the legacy systems of the near future, when people will discover that these applications are difficult to maintain in an economical and reliable way. We see a similarity between the current situation and the one that existed in the sixties (see [9] and much of the work that was spurred on by that paper), when the risks due to the lack of appropriate mathematical foundations, methods, and tools were recognized, and a suitable research agenda was set for software engineering to tame those risks.

This paper is structured as follows. In Section 2, we introduce some some general concepts about network computing. In sections 3 and 4 we look into two specific important areas of the network computing domain: the World Wide Web and applications based on mobile code. For these two areas, we discuss what the main risks are and outline a possible research agenda. This is not meant to be an exhaustive account of what is needed, but rather reflects a subjective viewpoint that is based on our work and some initial results that have been achieved by our research group at Politecnico di Milano. Section 5 draws some conclusions and outlines future work.

2 Network Computing

The term network computing (or "Internet computing") is difficult to define precisely. The term is often used informally to denote the research efforts, technologies, products, and services that exploit the global network, known as the Internet (for example see [19]). One may object that network computing is just another name for the "traditional" distributed computing field. This is only partially true. Indeed, network computing is based on a distributed system, composed of a set of computational nodes connected by a network and therefore several of the methods and techniques developed so far by the distributed computing research community can be viewed as foundational background for network computing. However, there are some distinguishing characteristics of network computing that make it possible to identify a research field with new, unsolved problems. These characteristics are *large scale*, *autonomy*, *heterogene-*

ity, and *mobility*, which influences both the *communication layer* and the *computational layer* of the network.

In network computing, the communication layer is a global-scale internetwork composed of autonomous subnetworks. Each subnetwork is deployed, maintained, and evolved without a centralized control. In addition, the network is composed of heterogeneous technologies: from those used to connect computers in a LAN, to those used to interconnect LANs geographically, to those used to connect mobile computing devices. The different technologies used to provide connectivity, spanning from fiber optics to different kinds of wireless connections, provide different levels of quality of services, e.g., in terms of performance and reliability. Therefore, given two communication endpoints in the network, few assumptions can be made about the type of communication that can be established. In addition, wireless technology allows computational nodes to move while remaining still connected to the net. This supports mobile users, who may be using laptops or personal digital assistants (PDAs). Therefore, the protocols used to transfer data across the network must cope with topologies that may change dynamically.

The computational layer is the infrastructure that is responsible for supporting the execution of applications. This layer is composed of the network nodes and the operating system software associated with these nodes. Nodes are organized in clusters administered by different autonomous authorities, each with different objectives, access procedures, and use policies. In addition, the computational infrastructure includes heterogeneous platforms, ranging from PCs to workstation and mainframes. Therefore, a global-scale computational layer must be based on mechanisms that can be deployed on very diverse hardware and software platforms. The heterogeneous nature of the computational layer becomes more evident when support to mobile computations must be provided. In this case there is a need for mechanisms that allow execution of the same code on different platforms.

Network computing applications exploit the global-scale infrastructure provided by the communication and computational layer to provide services to distant users and to access resources that are dispersed across a large set of hosts. Based on a highly autonomous infrastructure, these applications tend to be autonomous themselves. This means that either a service is delivered to a user through the interaction of different autonomous components that are managed by different authorities (the World-Wide Web follows this approach), or that an application belonging to an administrative subdomain in the network can move to other domains to achieve its goals. These kinds of mobile applications are often called "mobile agents" or "software agents".

The potential of this pervasive and ubiquitous infrastructure is enormous, and it is quite difficult to anticipate the way it will evolve and how far it will go. New applications and new services are announced almost every day. Although in many cases they promise more than what they actually deliver, the speed and complexity of the evolution are such that they are difficult to dominate. It is therefore important to build a coherent framework of principles, abstractions,

methods, and tools that allow network computing to be understood and practiced in a systematic fashion [11]. We claim that these are the challenges that software engineering must face in this context.

At the foundational level, we need to identify the theories that are needed to describe, reason about, and analyze network computations, where the topology and structure of the computing layer changes dynamically, while users and computations can move. From a methodology viewpoint, we need to identify process models that are suitable for describing and managing application developments for the new computing infrastructure, where applications grow in a largely independent way, no precise pre-planning is possible, and evolution/reconfiguration are the norm in an inherently chaotic, self-regulating environment.

As far as technology is concerned, we need to define a common set of mechanisms and service infrastructures that enable exploitation of the potential of a world-wide computing system in a effective, secure way. In addition, we need to understand what are the new programming language abstractions and mechanisms that are suitable for the implementation of network computing applications.

This wide spectrum of problems provides a real challenge for software engineering research. A number of efforts are already in place, but much more focused work is needed. In the next section we focus on the problem of supporting the development of World Wide Web sites. In Section 4 we describe some work on supporting developers of mobile code applications. The efforts we describe in this paper are only a small sample of what could be done in this field.

3 Software Engineering for WWW Applications

From its introduction in 1990 [3], the World Wide Web (WWW) has been evolving at a fast pace. The number of WWW sites is increasing as Internet users realize the benefits that stem from a globally interconnected hypermedia system. Each site, in fact, provides structured information as an intertwined net of hypertext resources; links can point both to local resources and to non-local resources belonging to other Web sites, thus providing a way to navigate from local to geographically distributed information sources. Companies, organizations, and academic institutions exploit the WWW infrastructure to provide customers and users with information and services.

The expectations of both providers and consumers are driving R&D efforts aimed at improving the WWW technology. Examples are represented by the introduction of active contents in static hypertext pages by means of languages like Java [16] and JavaScript [10] and by the use of the Servlet technology [22] to customize the behavior of Web servers. This technological evolution has promoted a shift in the intended use of the WWW. The Web infrastructure is going beyond the mere distribution of information and services; it is becoming a platform for generic, distributed applications in a world-wide setting.

This promising scenario is endangered by the lack of quality of many existing WWW-based applications. Although there is no well-defined and widely

accepted notion of Web quality (and indeed, this would be a valuable research objective in its own), our claim is based on the following common observations that can be made as WWW users:

1. we know that a required piece of information is available in a certain WWW site, but we keep navigating through a number of pages without finding it;
2. we get lost in our navigation, i.e., we do not understand where we are in our search;
3. navigation style is not uniform (for example, the "next page in the list" link is in the bottom right corner for some pages, and in the top left corner for others);
4. we continuously encounter broken links;
5. the data we find are outdated (for example, we find the announcement of a "future" event that has already occurred);
6. duplicated information is inconsistent (for example, in a university Web site providing pages in two language versions, say English and Italian, the same instructor has different office hours).

This is only a sample list. Items 4 to 6 of the list can be defined as flaws; they affect "correctness" of the Web site. The others are related to style issues, and affect usability. Furthermore, even if we start from a Web site that does not exhibit these weaknesses, these are likely to occur as soon as the Web site undergoes modifications. Thus, maintenance of legacy Web sites becomes increasingly difficult, and Web site quality decreases. If we try to understand what the causes of these inconveniences are, we realize that they all stem from the lack of application of systematic design principles and the use of inadequate (low-level) tools during development.

Most current Web site designs are not guided by systematic design methodologies and do not follow well-defined development processes. Rather, they proceed in a unstructured, *ad hoc* manner. Developers focus too early, and predominantly, on low-level mechanisms that enable, for example, particular visual effects, without focusing on who the expected users are, what the conceptual contents of the information is, and how the information should be structured. In particular, they rarely focus on the underlying conceptual model of the information that will be made available through the Web. The lack of a conceptual model becomes evident to the users, who find it difficult to search the Web to retrieve the data they are interested in. In addition, even if a conceptual model of the information to be published has been developed, no design guidance nor adequate abstractions are available to help Web developers move down systematically towards an implementation, possibly being supported by suitable tools.

This situation reminds us of the childhood of software development when applications were developed without methodological support, without the right tools, simply based on good common sense and individual skills. WWW site development suffers from a similar problem. Most Web site developers delve directly into the implementation phase, paying little or no attention to such aspects as requirements acquisition, specification, and design. Too often, implementation is performed by using a low-level technology, such as the Hypertext

Markup Language (HTML) [24]. Using the analogy with conventional software development, this approach corresponds to implementing applications through direct mapping of very informal designs (if any) into an assembly-level language. Furthermore, the lack of suitable abstractions makes it difficult to reuse previously developed artifacts, or to develop frameworks that capture the common structure of classes of applications and allow for fast development by customization. Finally, the management of the resulting Web site is difficult and error prone, because change tracking and structural evolution must be performed directly at the implementation level. This problem is particularly critical since WWW sites, by their very nature, are subject to frequent updates and even redesigns.

Software engineering research has provided methods for requirements acquisition, languages and methods for specification, design paradigms, technologies (such as object-oriented programming languages), and tools (e.g., integrated development environments) that provide systematic support to the software development process. In principle, their availability should help software developers to deliver quality products in a timely and cost-effective manner. A similar approach has to be followed in order to bring WWW development out of its immaturity. The next two subsections discuss a possible solution to these problems by analyzing the characteristics of Web site development process and by introducing a tool that aims at providing systematic support for the development process.

3.1 A WWW Software Process

The benefits of a well-defined and supported software process are well known [14]. As for conventional software, the development of a Web site should be decomposed into a number of phases: requirements analysis and specification, design, implementation. After the site has been implemented and delivered, its structure and contents are maintained and evolved. By identifying these phases of the development process we do not imply any specific development process structure. Different process models (waterfall, spiral, prototype-based) can be accommodated in the framework. Actually, the continuous and rapid changes in business, which will be reflected in the evolution of the corresponding WWW sites, is likely to favor flexible process life cycles, based on rapid prototyping and continuous refinement. In the sequel, we briefly and informally outline the possible objectives of the different phases of WWW development, based on our own experience.

Requirements Analysis and Specification During requirements analysis, the developer collects the needs of the stakeholders, in terms of contents, structuring, access, and layout. Contents requirements define the domain-specific information that must be made available through the Web site. Structuring requirements specify how contents must be organized. This includes the definition of *relationships* and *views*. Relationships highlight semantic connections among

contents. For example, relationships could model generalization (*is-a*), composition (*is-composed-of*), or domain-dependent relationships. Views are perspectives on information structures that "customize" contents and relationships according to different use situations. Different views of the same contents could be provided to different classes of users (e.g., an abstract of a document can be made accessible to "external" users, while the complete document can be made accessible to "internal" users). Access requirements define the style of information access that must be provided by the Web site. This includes priorities on information presentation, indexing of contents, query facilities, and support for guided tours over sets of related information. Layout requirements define the general appearance properties of the Web site, such as emphasis on graphic effects vs. text-based layouts. Based on our experience, we argue that existing tools supporting requirements specification and traceability of requirements through all development artifacts can be used in this context too. Further research is needed to extend the above framework and to identify the additional specific features that a tool supporting requirements for Web based applications should exhibit.

Design Based on the requirements, the design phase defines the overall structure of a WWW site describing how information is organized and how users can navigate across it. A careful design activity should highlight the fundamental constituents of a site; it should abstract away from low-level implementation details, and should allow the designer to identify recurring structures and navigation patterns to be reused [13]. As such, a good design can survive frequent changes in the implementation, fostered by —say— the appearance of new technologies.

Being largely implementation-independent, the design activity can be carried out using notations and methodologies that are not primarily Web-oriented. Any design methodology for hypermedia applications could be used; e.g., HDM [12], RMDM [2], or OOHDM [25]. Our experience is based on the adoption of the HDM (Hypertext Design Model) notation [12].

In designing a hypermedia application, HDM distinguishes between the *hyperbase layer* and the *access layer*. The hyperbase layer is the backbone of the application and models the information structures that represent the domain, while the access layer provides entry points to access the hyperbase constituents. The hyperbase consists of *entities* connected by *application links*. Entities are structured pieces of information. They are used to represent conceptual or physical objects of the application domain. An example of an entity in a literature application is "Writer". Application links are used to describe domain-specific, non-structural relationships among different entities (e.g., an application link from a "writer" to the "novels" he or she wrote). Entities are structured into *components*, i.e., clusters of information that are perceived by the user as conceptual units (for example, a writer's "biography").

Complex components can be structured recursively in terms of other components. Information contained in components is modeled by means of *nodes*.

Usually, components contain just one node, but more than one node can be used to give different or alternative views (*perspectives*, in HDM) of the component information (e.g., to describe a book's review in different languages, or to present it in a "short" vs. an "extended" version). Navigation paths inside an entity are defined by means of *structural links*, which represent structural relationships among components. Structural links may, for example, define a tree structure that allows the user to move from a root component (for example, the data-sheet for a novel) to any other component of the same entity (e.g., credits, summary, reviews, etc.)

Once entities and components are specified, as well as their internal and external relationships, the access layer defines a set of *collections* that provide users with the structures to access the hyperbase. A collection groups a number of *members*, to make them accessible. Members can be either hyperbase elements or other collections (nested collections). Each collection owns a special component called *collection center* that represents the starting point of the collection. Examples of collections are *guided tours*, which support linear navigation across members (through next/previous, first/last links), or indexes, where the navigation pattern is from the center to the members and vice versa. For example, a guided tour can be defined to navigate across all horror novels; another one can represent a survey of 14^{th} century European writers.

Implementation The implementation phase creates an actual Web site from the site design. As a first step, the elements and relationships highlighted during design are mapped onto the constructs provided by the chosen implementation technology. As a second step, the site is populated. The actual information is inserted by instantiating the structures defined in the previous step and the cross-references representing structural and application links among the elements. Collections are then created to provide structured access to the hyperbase contents. The third step is delivery. The site implementation must be made accessible using standard WWW technologies, namely Web browsers like Netscape's Navigator or Microsoft's Internet Explorer that interact with Web servers using the Hypertext Transfer Protocol (HTTP). This can be achieved by publishing the site implementation into a set of files and directories that are served by a number of "standard" WWW servers (also called *http daemons* in the UNIX jargon).

The standard tools available today to implement Web sites are rather low-level and semantically poor. The basic abstractions available to Web developers are:

- HTML pages, i.e., text files formatted using a low-level markup language;
- directories, i.e., containers of pages; and
- references, i.e., strings of text embedded in HTML tags that denote a resource (e.g., an HTML page) using a common naming scheme.

There are neither systematic methods nor linguistic constructs to map the types that define the semantics of the application domain (entities) onto implementation-level types (pages). There are no constructs to define complex information structures, like sets of pages with particular navigational patterns, such as lists of

pages or indexes. These structured sets of information must be realized manually by composing the existing constructs and primitives. In addition, there is no way to create document templates and mechanisms to extend existing structures by customization. The development of a set of documents exhibiting the same structure is carried out in an ad hoc manner by customizing sample prototypes manually. There are no constructs or mechanisms to specify different views of the same information and to present these views depending on the access context. This hampers effective reuse of information. The only form of reuse is by copy. Some authoring tools like Microsoft's FrontPage [6] and NetObject's Fusion [18] try to overcome some of these limitations by providing a site-level view on the information hyperbase. Nonetheless, these tools are strictly based on the low-level concepts of HTML pages and directories. Therefore, the developer is faced with a gap between the high-level concepts defined during design and the low-level constructs available for implementation.

Maintenance The situation described above worsens in the maintenance phase. Web sites have an inherently dynamic nature. Contents and their corresponding structural organization may be changed continuously. Therefore, maintenance is a crucial phase, even more than in the case of conventional software applications. As in conventional software, we can classify Web site maintenance into three categories: *corrective*, *adaptive*, and *perfective* maintenance [14]. Corrective maintenance is the process of correcting errors that exist in the Web site implementation. Examples are represented by internal dangling references, errors in the indexing of resources, or access to outdated information (as in the case of published data with an expiration date). Adaptive maintenance involves adjusting the Web site to changes in the outside environment. A notable example is represented by verification of the references to documents and resources located at different sites. Outbound links become dangling as a consequence of events over which the site developer has no control. Thus, adaptive maintenance is a continuous process. Perfective maintenance involves changing the Web site in order to improve the way contents are structured or presented to the end user. Changes may be fostered by the introduction of new information or by the availability of new technologies. Perfective maintenance should reflect updates to the requirements and design documents. Maintenance in general, and perfective maintenance in particular, is by far the activity that takes most of the development effort.

Presently, Web site maintenance is carried out using tools like link verifiers or syntax checkers that operate directly on the low-level Web site implementation. This approach may be suitable for some cases of corrective and adaptive maintenance, but does not provide effective support for tasks that involve knowledge of the high-level structure of the Web site. For example, since reuse is achieved by copy, modifying a reused component, like a recurring introduction paragraph for a number of documents, involves the identification of every use of the component and its consistent update. In a similar way, modification of the structure or style of a set of similar documents requires updates in all

instances. For example, if we decide that the background color of the summary page of all "horror" novels must be changed to purple, this requires consistent change of all files representing these summaries. More generally, since perfective maintenance may require modification of the structure and organization of information, it should be supported by a structural view of the site and of the relationships between design elements and their implementation constructs. These relationships are of paramount importance because they allow the developer to reflect design changes onto the implementation and vice versa. Standard Web technologies do not provide the means to represent these relationships and the high-level organization of information. Another problem concerns maintenance of hypertext references. In the standard WWW technology, references are just strings embedded inside the HTML code of pages; they do not have the status of first-class objects. Therefore, their management and update is an error-prone activity.

3.2 The WOOM Approach

A number of research efforts are currently being developed to improve the methods and tools supporting WWW developments, trying to solve some of the critical issues discussed in the previous section. It will not be possible to provide a comprehensive view of such efforts here. Rather, we will bring to the readers' attention what we are currently doing in our group to support Web design, as an example of a research effort that tries to address some of the previously identified problems. In this project, we developed a WWW object-oriented modeling framework, called WOOM — *Web Object Oriented Model*. WOOM provides concepts, abstractions, and tools that help in the mapping from high-level design of a Web site (e.g., in HDM) into an implementation that uses "standard" WWW technology.

More precisely, WOOM offers three main modeling environments: a *design environment*, an *implementation environment*, and a *presentation environment*. In the design environment the developer designs the conceptual model of the information to be published through the Web site. WOOM provides a set of predefined classes that allow designs to be built following the HDM methodology. In the implementation environment the developer implements a Web site leveraging off of an object-oriented model. This model provides high-level constructs to implement the information architecture defined in the design environment. Relationships between design elements and implementation constructs are maintained explicitly to allow for change tracking and consistent updating. In the presentation environment the developer is provided with mechanisms to customize and put into context the user's view on the site contents. This is achieved by means of a dynamic publishing process.

In the following, we provide some details of the Web site model and the publishing process, which constitute the core elements of the implementation and presentation environment, respectively.

A Web Site Model According to WOOM, a Web site can be defined in terms of *components*, *links*, *sites* and *servers*.

Components are the fundamental entities of the model. They model the contents and structure of a collection of information. In WOOM there are many types of components that differ in granularity, in the type of information they represent, and in the role they play in the model. For instance, a component can be the whole contents of a site, a single hypertext page, an icon, or even a single word. In WOOM all the component types are organized by the inheritance relationship in a class hierarchy, whose root is the `Component` class. The hierarchy can be extended by developers to define new types of components. WOOM provides a predefined set of components types that can be distinguished into *resources*, *elements*, and *containers*.

Resources are the units of information. They are distinguished into *opaque resources* and *hyper pages*. Opaque resources are unstructured resources. Subclasses of this class are *images*, i.e., graphic objects, *applets*, i.e., programs that are activated on the client side, *scripts*, i.e., applications that are activated on the server side, and *external*. External resources are those types of information that are not directly supported by the current Web technology and are managed by means of external helper applications. These resources include audio and video information, PostScript files, binaries, and other similar entities. Hyper pages are hypertext pages, which may contain text, anchors, and references to pictures, sounds, and animations. HTML pages are a special kind of hyper pages.

The contents of a hyper page are modeled by a ordered list of *elements*. An element is an information fragment, like a text paragraph, an anchor, or a dotted list. Elements can be simple or complex. Simple elements are atomic data containers, while complex elements contain an ordered list of other elements. WOOM provides a predefined set of elements that model the elements of the HTML standard. For example, the image placeholder element (IMG) is a simple element, while the BODY element may be composed of some paragraphs, a table, etc.

Containers are collectors of resources. They are composed of an ordered list of components that can be resources or other containers. Containers are used by the site developer to define contexts or to organize other components in macro-objects that can be managed as a whole. WOOM provides a number of predefined container types: *lists*, *trees*, *indexes*, and *sets*. Lists organize the enclosed resources in a linear fashion. They are used to represent a sequential relationship inside a group of resources (e.g., the pages that compose a guided tour through the novels of a given writer). Trees impose a hierarchical structure to the enclosed resources. For example, the novels of a given writer can be classified into genres: horror, science fiction, etc.; science fiction novels, in turn, can be classified into, say genetics, astronomy, etc. Indexes organize the contained resources in two-level trees. For example, an author's novels can be grouped into "youth", "maturity", and "late" novels. Sets are simply used to group resources without any specific relationship among them, but characterized by some common vi-

sual or semantic property. Each container type exports an interface that allows other entities to access the enclosed resources without exposing the container's internal implementation details. Additional container types can be defined by the Web developer by extending the WOOM framework.

The containment relationship among containers, resources, and elements defines an ordered DAG in which compound components are nodes and atomic components are leaves. In a given DAG, there is a component that is not enclosed by others: it is called the DAG root. In general, a component can belong to more than one component, and there cannot be circular containment relationships. This is different from the existing Web technology in which files can be included in more than one directory only by using links in the file system. In WOOM even a single word can be shared by different hyper pages.

Every component object has an identifier. Components belonging to the same compound component cannot have the same identifier. Components are uniquely identified a pathname. The pathname for a component is the description of the path that must be followed to reach the component starting from the DAG root. The pathname is a string obtained by concatenating the identifiers of the components that constitute the path. This identification mechanism is similar to the well-known naming scheme based on pathnames adopted by file systems. A component can be identified by one or more pathnames. Each pathname is a different chain of enclosing components and identifies a different context for the component. As it will be explained later, contexts are an important concept in the WOOM model. When the information consumer requests the contents of a component, he/she specifies also one of the contexts for the component. The publishing process that produces the external representation of the component delivered to the user provides a different result on the basis of the chosen context. Thus, the same component information is "contextualized" in different ways.

Links model the navigational connections between components. Links are objects that associate a source component with a destination component within a particular context. Context information must be taken into account when creating links because a component in different contexts may assume different forms. By keeping track of the context of the destination component, WOOM links may lead to a particular version of the component. WOOM links are different from standard HTML links in two ways. First, they are first-class objects whereas in HTML links are just strings of text embedded in the HTML anchor element. Second, links can reference any component, e.g., a hyper page or a text paragraph inside a hyper page, whereas in standard HTML links may only refer to files in the file system.

Sites and servers model the mechanisms that allow the information consumer to access the components' data. A site is composed of a component DAG and one or more servers. The servers are used to define the network access points to the site contents. A server corresponds, at run-time, to an HTTP daemon process that replies to the end user's requests for components belonging to the site. Each server is characterized by a unique address and has an associated container and context that limit the scope of the components that are accessible through the

server. For instance, the information contained in a Web site could be made accessible by means of two servers that are associated with two containers such that one is not the ancestor of the other in the site's DAG. Therefore, they provide access to resources that are in different contexts and the information spaces served by the two servers are cleanly separated even if there may be some components that can be accessed by both servers.

Delivering Information to the User The end-user accesses the Web site contents by requesting a component from a server. The particular component is specified by means of a path in the Web site's DAG. The server replies providing an *external representation*, or *view*, of the component's contents. The external representation is the result of a recursive publishing process that propagates in a top-down fashion from the DAG's root to the component to be published, along the path specified in the request. Before detailing the process, two WOOM mechanisms must be introduced: *transformers* and *viewers*.

Transformers are objects with an associated state and a *transform* operation that takes as parameter a component object. The transform operation modifies the component passed as parameter and eventually returns it. Transformer objects are associated with component objects in the site's DAG. A transformer associated with object O influences the publishing process of O and of all its sub-components. WOOM provides a set of predefined transformers, e.g., transformers to publish only specific parts of a component, or to add some navigational garnishment to components that are part of containers such as lists or indexes. The set of available transformers can be extended by the Web site developer.

Viewers are responsible for building the external representation of components. When a component is passed to a viewer, the viewer uses the component's data (i.e., the values of its attributes) and the external representation of its sub-components to create the component view that will be delivered to the user. WOOM provides several generic viewers that are able to produce a simple external representation for WOOM's predefined components. The Web site developer must provide viewers for any new component types introduced in the component hierarchy.

Transformers and viewers are the key mechanisms in the two phases of the publishing process, namely the *transformation process* and the *translation process*.

The transformation process is responsible for modifying the information structure according to the user's access context. Let us consider a user request for a component, say C, identified by the path $A \mapsto B \mapsto C$ in the component DAG. For the sake of simplicity, we assume that the server that received the request is associated with the DAG root (A). The publishing process starts by applying the transformation process to A. A recursively invokes the process on B, and finally B invokes the process on C. Transformers are propagated as parameters in the chain of invocations. Suppose that a transformer t_A is associated with A, and transformers t_B^1 and t_B^2 are associated with B. Therefore, A invokes

the transformation process on B passing t_A as a parameter. Then, B invokes the transformation process on C passing as parameters t_A, t_B^1, and t_B^2.

The transformation process for a component follows a fixed schema. First, a shallow copy of the component is created. This copy is passed to the first transformer in the list received during the transformation invocation. The transformer modifies the copied object depending on its own state and the values of the object's attributes. The modified object is then passed to the transform method of the next transformer, until the last transformer has been applied. If the returned object is non-null then the transformation process is propagated to every contained component object. This way, the publishing process takes into account the context (i.e., the path) in which the object is accessed.

Once the transformers have modified the DAG[1] according to the context, a similar process is performed to translate the requested component object into an external representation. The translation process starts from the DAG root (A) and is recursively invoked until it reaches the requested component (C). The chain of invocations propagates the viewers from the root to the specified component. From that point on the process is invoked on any subcomponent to produce the corresponding external representation. These representations are then used to produce the external representation of the requested component. For example, suppose that C has two sub-components D and E. In addition, suppose that B is associated with viewer v_B and C is associated with viewer v_C. Then the translation process is the following: A invokes the translation on B. B invokes the translation on C passing as parameter its viewer, v_B. C invokes the translation on D and E passing as parameters v_B and v_C. Since D and E are leaves of the DAG, the recursion stops and one viewer is applied to obtain the external representation. The most specific viewer is chosen among those that have been propagated by the publishing process. If no viewers are present, the default one is used. Then, the external representations of D and E are returned. A viewer is then applied to C, passing as parameters the external representation of D and E. The result is the external representation of C. This result is passed back to the chain of recursive invocation without modifications, and it is eventually delivered to the user that requested the component.

This general publishing mechanism is used to contextualize and customize the presentation of information. For instance, consider a hyper page describing a novel (e.g., Primo Levi's *La Tregua*) that is placed in two containers. The first container is a list enclosing all the novels of a particular genre (e.g., "holocaust" novels). The second is a set collecting the novels of the same writer (e.g., Levi's books). In the first context the page must be modified to include information that highlight the relationship between the novel and the genre, and links to navigate inside the list. In the latter context, the resource must be modified to include references to the author's biography. This is achieved by associating two different transformers with the two containers. Depending on the access context (the publishing path) only one of the transformers will be applied to the hyper

[1] Note that since transformers are applied to copies of resources and elements, the original entities defined by the site developer are not modified.

page, resulting in the "customized" version. Since this done automatically by the publication tool, consistency is automatically preserved (only one instance of the writer's data is kept), and maintenance is greatly facilitated.

An important result of this approach is that it clearly separates the description of the data from the way the data are presented to the user. The same data can be presented differently in different contexts. This separation not only helps in designing the application, but also provides support to Web site evolution.

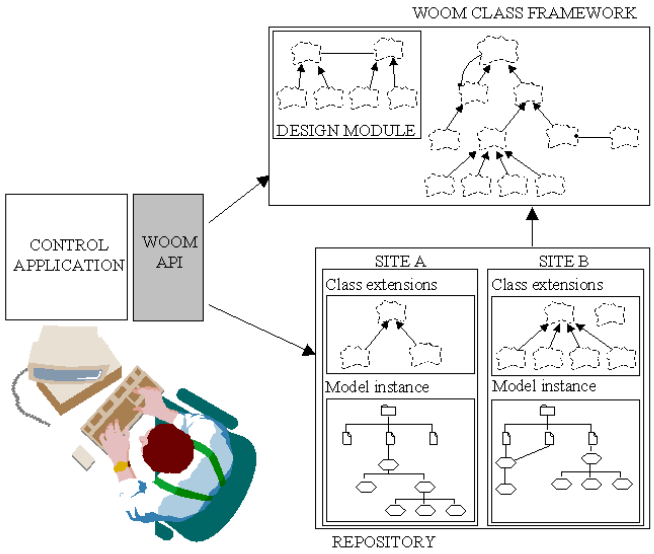

Fig. 1. The WOOM-based authoring tool.

A Tool for Web Development We developed a prototype authoring tool, written in Java, that implements the WOOM model. The tool allows the developer to use WOOM constructs to create the components in a Web site, to perform complex management operations, and to customize the publishing process that delivers a particular view of the site's information to the user. The main components of the tool are presented in Figure 1. A first component is the WOOM class framework. The framework provides the definitions of basic WOOM entities and provides some predefined constructs. The class framework provides support for representing Web site design elements into the model. This is achieved by means of an integrated, yet separate, design module. The design module is a plug-in component that provides support for a specific design notation. Currently, the HDM notation is supported. As a preliminary step in Web

site implementation, the WOOM class framework is imported into the development application. Then, the developer uses the instances of the classes provided by the design module to represent the entities defined during the design phase. Once the design elements have been represented, the developer chooses which type of component will be used to implement a particular design element. To do this, the developer may use the predefined constructs offered by the WOOM class framework or may create new application-specific constructs using inheritance and composition. After suitable constructs have been identified, relationships that associate a design element with the corresponding implementation construct are created. These relationships are used in tracking changes in the implementation to the site design and vice versa. The next step consists of populating the site, by instantiating component objects of the appropriate classes, and creating application links. Structural links are automatically managed by the semantics of structured objects that implements structured design elements. Once the site has been populated, the developer specifies how contents must be presented by defining transformers, viewers, and by configuring the servers that provide access to the site's information.

Web site maintenance and management operations are performed on the WOOM model instance. The WOOM framework provides support for a set of predefined tasks like syntax checking, link updating, resource restructuring, consistency checks, shared resource management, and design change management. Web site instances, composed of site-dependent schema extensions (classes and transformers) and component objects are persistently stored in a repository module. The control application accesses the WOOM schema and instances by means of the WOOM API. The control application is a Java application that uses the primitives and services offered by the API. We are currently working on a graphical interface that allows the developer to access WOOM services in an intuitive and user-friendly way.

4 Mobile Code Applications

The global computing infrastructure realized by the Internet is still in its infancy. Even though there exist efforts to realize mechanisms to distribute the computations on a large scale (for example, CORBA [23]) there is still the need for a general-purpose, flexible, programmable infrastructure that applications can exploit to access the computing power and the resources of the hosts connected to the network. One of the most promising approaches to providing this infrastructure is represented by *Mobile Code Systems* (MCSs). MCSs allow an application to relocate part or all of its code on remote hosts, possibly together with the corresponding execution state.

MCSs are based on a common conceptual framework. In this framework, the computational infrastructure is composed of a world-wide distributed environment with several autonomous *sites* that support the computations of *agents*, that are threads of execution. Agents may change their execution site and even

their code dynamically. Several MCSs, like Telescript [28], Agent Tcl [17], and Java Aglets [21] have been implemented.

The idea that software can migrate is not new. In particular, it has been exploited by several distributed systems to support load balancing. Mobile code, however, differs from distributed computing in many respects [11]. First, traditional distributed computing systems deal with a set of machines connected by a local area network, whereas in the mobile code framework mobility is exploited at a much larger scale (the Internet scale). Hosts are heterogeneous, they are managed by different authorities, and they are connected by heterogeneous links. Second, mobility is seldom supported in distributed systems. In the particular cases where it is supported, it is not provided as a feature given to the programmer to be exploited to achieve particular tasks. Rather, it is used to allow components to be automatically relocated by the system to achieve load balancing. Component relocation is not visible to the applications' programmer, since a software layer is provided on top of the network operating system to hide the concept of physical locality of software components. On the other hand, in the mobile code framework programming is location aware and mobility is under the programmer's control. Components can be moved to achieve specific goals, such as accessing resources located at remote sites.

This powerful concept originated a very interesting range of technical results recently. What is still lacking, however, is both a conceptual reference framework to describe and compare the various technical solutions and a methodological framework to support a sound development process of *Mobile Code Applications* (MCAs). In particular, we envision a process by which developers of MCAs are equipped with methods, notations, tools, techniques, and guidelines that support every phase of the development process. But much research is needed to reach this ideal stage from the current state of knowledge.

In the sequel, we illustrate some initial work done by our group in the areas of architectural design and implementation of MCAs. These phases are particularly critical because the distinction between design paradigms [5] and implementation technologies [8] is often blurred and not well understood. The goal here is to identify which are the concepts that are characteristic of architectural design, which are the issues that must be addressed during implementation, and what are the relationships between design choices and implementation choices. This way, it is possible to develop guidelines that allow the developers to select the most appropriate design paradigm for a specific application, and the most appropriate technology to implement the resulting software architecture.

4.1 Design

In the design phase the developer creates a software architecture for an application that must deliver a specified functionality. A software architecture is the decomposition of a software system in terms of software components and interactions among them [26]. Software architectures with similar patterns of interaction can be abstracted into *architectural styles* [1] or *design paradigms*,

which define architectural schemas that may be instantiated into actual software architectures.

The design of MCAs is a complex task. These applications are highly dynamic from the point of view of both code and location and therefore it is necessary to take into account these concepts at the design level. We identified three prototypical design paradigms that involve code mobility: *Remote Evaluation* (REV), *Code on Demand* (COD), and *Mobile Agent* (MA). Although we cannot claim that these paradigms cover all possible design structuring styles for network-centric applications, they can be viewed as the most typical representatives.

Given two interacting components A and B of a distributed architecture, these paradigms differ in how the *know-how* of the application, i.e., the code that is necessary to accomplish a computation, the *resources*, i.e., the inputs of the computation, and the *processor*, i.e., the component responsible for the execution of the code, are distributed between the involved sites. In order to let this computation to take place, the know-how, the resources, and the component that will process the resources using the know-how have to be present at the same site. Let us assume that component A is located at site S_A and component B is located on site S_B. In addition, let A be the entity that causes the interaction and the one that is interested in its effects. Table 1 shows the distribution of the different elements before and after the interaction. The table also lists the *Client-Server* (CS) paradigm. Although CS is not a paradigm for mobile computations (no mobility of code takes place), it has been included because it is often used in designing network computing applications.

Paradigm	**Before**		**After**	
	S_A	S_B	S_A	S_B
Client-Server	A	know-how resources B	A	know-how resources **B**
Remote Evaluation	know-how A	resources B	A	*know-how* resources **B**
Code on Demand	resources A	know-how B	resources *know-how* **A**	B
Mobile Agent	know-how A	resources	—	*know-how* resources ***A***

Table 1. Mobile code paradigms. This table shows the location of the components before and after the service execution. For each paradigm, the component that is the processor is in bold face. Components in italics are those that have been moved.

In the CS paradigm, a server component (B in Table 1) exports a set of services. The code that implements such services is owned by the server component; thus, we say that the server holds the *know-how*. It is the server itself that executes the service; thus it is the *processor*. The client (A in Table 1) is interested in accessing some entity managed by the server, and therefore it is the server that has the *resources*.

In the REV paradigm, the executor component (B) offers its computational power (i.e., it is the *processor*) and its *resources*, but does not provide any "specific" service. It is A that sends the service code (the *know-how*) that will be executed by B in its location.

In the COD paradigm, component A initially is unable to execute its task. It is B that provides the code (i.e., the *know-how*). Once the code is received by A, the computation is carried out on A's location, thus, A is the *processor*. The computation involves only local files and local devices; thus, A holds the *resources*.

In the MA paradigm, A has the *know-how* and it is the component responsible for the execution of the task. The computation must access the *resources* that are located at B's site. Therefore, A migrates to S_B and performs the computation there.

Usually, an application (or parts thereof) may be designed following different paradigms. In principle, it would be helpful to be able to analyze the tradeoffs among the different solutions based on different paradigms at the design level, before proceeding to an implementation. For example [5] discusses a set of possible designs and evaluates their tradeoffs in the case of a distributed information retrieval application. The tradeoffs are evaluated in terms of a simple quantitative measure: network traffic. The case study shows that, in general, there is no definite winner among the different paradigms, but rather the choice depends on a number of parameters that characterize the specific problem instance.

Therefore, the developer that approaches the design of an MCA should select some parameters that describe the application behavior, together with some criteria to evaluate the parameters values. For example, one may want to minimize the number of interactions, the CPU costs, or the generated network traffic. Given different possible architectural designs, the developer should analyze the selected parameters looking for the design that optimizes the chosen criteria. This way it is possible to determine which is the most reasonable design.

4.2 Implementation

Having designed an MCA according to some paradigm, one has to choose a technology to implement it. Given a particular paradigm, which technology should be used?

We identify three classes of technologies [7]:

Message-based These technologies enable the communication between remote processes in the form of message exchange. They do not provide any native mechanism for the migration of code. A typical example is RPC [4].

Weakly mobile These technologies provide mechanisms that enable an agent to send code to be executed in a remote site together with some initialization data or to bind dynamically code downloaded from a remote site. Examples of such technologies are the *rsh* facility in UNIX, and languages like M0 [27] or Java [16].

Strongly mobile These technologies enable agents to move with their code and execution state to a different site. An example is represented by the Telescript technology.

In principle, it is possible to implement applications developed with any paradigm by using any kind of technology, given that such technologies allow for the communication between agents. However, we have found that some technologies are more suitable to implement applications designed using particular paradigms [15]. Unsuitable technologies force the developer to program, at the application level, some mobility mechanisms or force an inefficient, counter-intuitive use of the existing ones.

Technologies	Paradigms		
	CS	COD/REV	MA
Messsage-based	Well suited	Code as data Program interpretation	Code and state as data Program state restoring Program interpretation
Weakly mobile	Code is a single instruction Creates unnecessary execution threads	Well suited	State as data Program state restoring
Strongly mobile	Code is a single instruction Creates unnecessary execution units Move state back and forth	Manage migration Move state back and forth	Well suited

Table 2. Relationships among paradigms and technologies.

As shown in Table 2, message-based technologies are well suited for implementing architectures based on the CS paradigm. If they are used to implement COD-based or REV-based architectures, they force the implementation to use the basic message exchange mechanism to transfer code (viewed as data) and to program the evaluation of such code explicitly. Even worse, if message-based technologies are used to implement MA-based architectures, the programmer also has to explicitly manage state marshalling, transfer, and unmarshalling; i.e., auxiliary variables must be used to keep the state of the computation and unnatural code structures must be used to restore the state of a component after migration to a different site.

Weakly mobile technologies that allow segments of code to be executed remotely are naturally oriented towards the implementation of applications designed according to the REV and COD paradigms. These technologies provide inefficient implementations of CS architectures since they force the remote execution of segments of code composed of a single instruction. Therefore, a new thread of execution is created in order to execute this "degenerate" code. On the contrary, in order to implement applications based on the MA paradigm, the programmer has to manage, at the program level, the packing/unpacking of the variables representing the state and the restoring of the agent execution flow[2].

Strongly mobile technologies are oriented towards MA-based applications while they are not suited for implementing applications based on the CS and REV/COD paradigms. In the former case, the programmer has to "overcode" an agent in order to have it moved to the server site, execute a single operation and jump back with the results. Such implementations could be rather inefficient since the whole thread state is transmitted back and forth across the network. In the latter case, in addition to the code to be executed remotely, the implementor has to add the migration procedures. Furthermore, the state of the execution thread is to be transmitted over the network.

Summing up, technologies may reveal to be too powerful or too limited to implement a particular architecture. In the first case resources are wasted, resulting in inefficiency. In the second case, the programmer has to code all mechanisms and policies that the technology does not provide. It is therefore important to select an appropriate technology that can implement an architectural design in a natural way.

5 Conclusions

Network computing is a rapidly evolving field, which is raising much interests, both in industry and in research institutions. Indeed, it is a very promising field. Unfortunately, however, at its current stage of maturity, it is perhaps raising too many unjustified expectations. We see many interesting things being done in practice which are not backed up by adequate methods and tools. A challenge exists for software engineering research to evaluate critically how things are done today in order to identify systematic approaches to the development of network computing applications. The "just do it" approach that seems to be behind the current efforts is simply inadequate to reach the desired levels of quality standards of network applications, for example in terms of reliability and ease of change. We must, of course, keep into account what makes network applications different from most traditional applications. In particular, their intrinsic levels of flexibility, autonomy, decentralization, and continuous change that cannot be

[2] This is a very common situation. In fact, most existing mobile code technologies are weakly mobile, because weak mobility is easier to implement. Nonetheless, practitioners tend to think in terms of the MA paradigm. Therefore, there are many examples of mobile code applications that use awkward conditional structures to restore, at the logical level, the computational flow after migration.

pre-planned centrally. These properties must eventually be combined with the necessary discipline that allows the desired level of reliability to be reached in a measurable and economical way.

In this paper we tried to identify a possible research agenda for software engineering research in the area of network computing. We also summarized some initial work that has been done in the past few years by the Software Engineering Group at Politecnico di Milano. The work described here is the result of the joint work of several people: Antonio Carzaniga, Francesco Coda, Gianpaolo Cugola, Alfonso Fuggetta, Franca Garzotto, Gian Pietro Picco, and the authors of this paper. The results described here are only representative of some initial steps in the direction of providing systematic support for the development of network computing applications. More research is needed before we can identify a comprehensive, integrated set of useful methods and techniques, and provide tools to support them.

The authors wish to thank the participants at RTSE'97 in Bernried for very stimulating discussions and comments on the work reported here. The reviewers of the initial version of this paper provided further insightful suggestions.

References

1. G. Abowd, R. Allen, and D. Garlan. Using Style to Understand Descriptions of Software Architecture. In *Proc. of SIGSOFT'93: Foundations of Software Engineering*, December 1993.
2. V. Balasubramanian, T. Isakowitz, and E.A. Stohr. RMM: A Methodology for Structured Hypermedia Design. *Communications of the ACM*, 38(8), August 1995.
3. T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen, and A. Secret. The World Wide Web. *Communications of the ACM*, 37(8), August 1994.
4. A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, 2(1):29–59, February 1984.
5. A. Carzaniga, G.P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In R. Taylor, editor, *Proceedings of the 19^{th} International Conference on Software Engineering (ICSE'97)*, pages 22–32. ACM Press, 1997.
6. Microsoft Corp. FrontPage Home Page. *http://www.microsoft.com/FrontPage/*.
7. G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. A Characterization of Mobility and State Distribution in Mobile Code Languages. In M. Mühlaüser, editor, *Special Issues in Object-Oriented Programming: Workshop Reader of the 10^{th} European Conf. on Object-Oriented Programming ECOOP'96*. dpunkt, July 1996.
8. G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes on Computer Science*. Springer, April 1997.
9. E.W. Dijkstra. GOTO Statement Considered Harmful.
10. D. Flanagan. *JavaScript — The Definitive Guide*. O'Reilly & Ass., 2^{nd} edition edition, January 1997.
11. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5), May 1998.
12. F. Garzotto, L. Mainetti, and P. Paolini. Hypermedia Design, Analysis, and Evaluation Issues. *Communications of the ACM*, 38(8), August 1995.

13. F. Garzotto, L. Mainetti, and P. Paolini. Information Reuse in Hypermedia Applications. In *Proceedings of ACM Hypertext '96*, Washington DC, March 1996. ACM Press.
14. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.
15. C. Ghezzi and G. Vigna. Mobile Code Paradigms and Technologies: A Case Study. In K. Rothermel and R. Popescu-Zeletin, editors, *Proceedings of the 1st International Workshop on Mobile Agents (MA '97)*, volume 1219 of *Lecture Notes on Computer Science*. Springer, April 1997.
16. J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, October 1995.
17. R.S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents*, Baltimore, Md., December 1995.
18. NetObjects Inc. Fusion Home Page. *http://www.netobjects.com/*.
19. IEEE Internet Computing Magazine. IEEE Computer Society, 1997.
20. A. Kambil. Doing Business in the Wired World. *IEEE Computer*, 30(5):56–61, May 1997.
21. D.B. Lange and D.T. Chang. IBM Aglets Workbench—Programming Mobile Agents in Java. IBM Corp. White Paper, September 1996.
22. Sun Microsystems. The Java Servlet API . White Paper, 1997.
23. Object Management Group. *CORBA: Architecture and Specification*, August 1995.
24. D. Ragget, A. Le Hors, and I. Jacobs. Hypertext Markup Language 4.0 Specification. W3C Recommendation, April 1998.
25. D. Schwabe and G. Rossi. From Domain Models to Hypermedia Applications: An Object-Oriented Approach. In *Proceedings of the International Workshop on Methodologies for Designing and Developing Hypermedia Applications*, Edimburgh, September 1994.
26. M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice Hall, 1996.
27. C. Tschudin. *An Introduction to the* `MO` *Messenger Language*. Univ. of Geneva, Switzerland, 1994.
28. J.E. White. Telescript Technology: Mobile Agents. In Jeffrey Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.