Journal of Digital Libraries Manuscript Nr.
(will be inserted by hand later)

Linguistic and Architectural Requirements for Personalized Digital Libraries

Joachim W. Schmidt, Gerald Schröder, Claudia Niederée, and Florian Matthes

Technische Universität Hamburg-Harburg, Arbeitsbereich Softwaresysteme, Harburger Schloßstraße 20, D-21079 Hamburg, Germany

Received December 10, 1996; accepted December 10, 1996

Abstract. Our vision of digital libraries is influenced by our experience with systems for persistent and networked object management and with polymorphic programming languages for their implementation. When viewed from this perspective, the essence of digital libraries can be captured by the following three essentials:

- the content of a digital library is represented by two kinds of information entities: on the basic level there are *information tokens* as supplied by information providers on the net; value is added to such tokens by individually constructing *information artifacts* over them with the goal of information consumer satisfaction;
- the services required for artifact construction and use – on the information level as well as on the level of the software artifacts required for these processes – rely heavily on powerful *binding environments* for multi-medial, persistent and networked information;
- the processes of artifact construction and use are in themselves valuable sources of information about artifacts; for the exploitation of such process information, digital libraries employ advanced *tracing environments*.

We derive linguistic and architectural requirements for digital libraries from these above essentials. On the language level we concentrate on generalized requirements for the *typing*, *binding* and *scoping* of library entities and services. On the system level we discuss architectural requirements in terms of *orthogonal persistence*, *open extensibility*, *platform independence*, *mobility* and *reflection*.

We present Tycoon [Matthes and Schmidt 1992; Matthes *et al.* 1995], a polymorphic, higher-order language and its system, and demonstrate its potential for digital libraries. We evaluate Tycoon's rich conceptual basis (*data*, *functions* and *threads*), library-based extensibility, powerful binding mechanisms, its orthogonal persistence and its capability of network-wide data, code and thread migration.

We conclude by referring to an interdisciplinary digital library project in Art History Research based on icons, texts and data. Here, Tycoon effectively supports the process of individually customizing and scaling library services thus generalizing the notion of a query language into that of a persistent personal reference library.

1 Introduction

Digital libraries are coming into their own at a time when computer and network technology is improving rapidly resulting in increased global acceptance of digital information and communication services.

As a first consequence, an enormous quantity of information on virtually any medium is now being exchanged between a world-wide community of information providers and consumers. This potential will virtually revolutionalize all information processing activities. Digital libraries are intended to provide the software which helps to structure this global information space and to improve its use.

World-wide dissemination of digital contents is, however, not the only achievement of the rapidly improving information and communication technology. Liberating computer scientists from too narrow technological restrictions also gives rise to the development of highly advanced models for computation and communication which aim at pushing computers towards the perfect information handling devices – and rapid absorption of the new enabling technologies.

Our vision of digital libraries is founded on both of the above achievements. We firmly believe that the great potential of the global information space can be exploited only if digital libraries make best use of highly advanced models, languages and systems. If not, we are filling new wine into old barrels thereby creating another Pandora's box of legacy problems for the upcoming *information age*.

Based on our vision of digital libraries outlined in Section 2 we will present in Section 3 the essential linguistic and architectural requirements which we consider necessary for the construction and use of such systems. In Section 4 we will evaluate Tycoon [Matthes and Schmidt 1992; Matthes *et al.* 1995], the language and its system platform, and demonstrate its potential for digital libraries. Section 5 presents our initial experience with the *Warburg Electronic Library*, a cooperation project with the Art History community which involves data, texts and large amounts of iconic images.

Correspondence to: j.w.schmidt@tu-harburg.d400.de

2 Vision of Digital Libraries

In our view, the final purpose of a digital library is to improve the exploitation of the globally networked information universe with a clear focus on the particular information needs of an individual user and his task at hand [Van House 1995]. In other words, a digital library is the software needed to bridge the gap between the virtually infinite amount of information out there on the net and the very specific information needs of a particular person and task.

With this goal in mind, we distinguish on the level of *library content* two kinds of *library entities*:

Information tokens represent the information content which is collectively provided by the net; information tokens are multi-medial and may represent texts, data, images, sound, videos etc. [McNab *et al.* 1996; Li *et al.* 1996]. In addition to information tokens digital libraries also contain software tokens, like applets or generic functions (such as filters, constructors, evaluators, presentors, binders or wrappers).

Information artifacts are (recursively) created views over information tokens. Selecting, structuring, combining, and annotating are typical steps in this process. Artifact construction aims at adding value to information and thereby increasing the satisfaction of information consumers. Information artifacts may be image tokens associated with texts on their content as well as entire multi-media libraries including sound and videos.

Two main library processes can be identified: value-adding artifact construction and personalized exploitation. Although this paper concentrates on information artifacts it should be emphasized that the software required for personalized library processes puts heavy demands on software customization. Indeed, there is a striking similarity between processes which personalize the library's information content and those which customize its software required for doing so. Such software is also represented by software tokens out of which customized software artifacts can be constructed which are finally bound and applied to information artifacts. In our closing remarks we will come back briefly to this similarity between information and software libraries.

These *library processes* are supported – on the information as well as on the software level – by two kinds of library environments:

Binding environments enable artifact construction and exploitation on the two intertwined levels of information artifacts and information handling software artifacts. Binding environments provide a wide range of binding capabilities and support their safe and disciplined use in heterogeneous, open, networked and multi-medial settings.

Tracing environments are highly desirable because library processes themselves are sources of valuable information on artifacts. Tracing such processes and extracting context information (on the who and when, the what and where, the why?, but also on software versions, hardware platforms etc.) may substantially augment the artifact's value.

2.1 Information Tokens

Information tokens represent the basic multi-media content of digital libraries and are provided by a digital and networked information universe. Their main purpose is to conceptualize information on an elementary level completely independent of any specific contexts in which information is referenced, aggregated and consumed.

Examples of information tokens are

- pictures in digital formats such as JPEG or GIF;
- texts in formats such as ASCII, Postscript or Word;
- time dependent tokens such as audio and video sequences in QuickTime or MPEG formats;
- software tokens such as JAVA applets or generic sorting functions in source or compiled bytecode or even native (machine-dependent) code.

To best serve their role as basic building blocks in value-adding digital libraries, tokens should be unbiased in their representation. Furthermore they should be self-describing, equipped at least with basic methods for presentation, copying, migration, storage etc. Note that the information tokens may inherently provide further more specific methods. A Word document, for example, may come with methods for reformatting or generating a table of content. For software tokens the basic methods may be compilation, execution or interpretation.

Independent of their content and the platform on which they reside, tokens have to share the following basic properties:

- digital representation
- atomic view, i.e., context-independent interpretation
- unconstrained content described by automorphic typing.

Automorphic typing allows the basic classification of information tokens, stresses their self-contained character of the information tokens and avoids restriction to predefined formats.

Information tokens have to be handled uniformly, independent of their content, lifetime or location in the network space. Therefore, the following additional properties are requested uniformly over all kinds of information tokens:

- network-wide, stable identity
- universal referentiability via the net.

Any specific demands on information which are derived from the needs of individual users or user groups are captured by the concept of information artifacts.

2.2 Information Artifacts

Information artifacts are constructed (recursively) from tokens with the intent to add, step by step, value to the information on its way from the network of information providers to the individual information consumer.

The structure of artifact constructing processes is shown in Figure 1. Information tokens provided by a networked information universe are composed into information artifacts, a process guided by the two related principles that value be added by information composition, but also by personalization [Röscheisen *et al.* 1995; Röscheisen *et al.* 1994]. Conceptually, a digital library has manifold artifacts visible to information consumers with high emphasis on personalized artifacts customized for the consumers preferences and tasks.

Typical examples of personalized information artifacts are individual collections of text and image tokens (references or copies) augmented by hyper-text artifacts and personal annotations. Artifacts may be attached to agents which search autonomously for further material on the net by employing text retrieval systems or SQL engines. Other examples are video sequences augmented by data on title, producer, costs, list of actors, comments and ratings etc.

Examples of software artifacts are application packages such as Oracle's graphical query tools with GUI and SQL components or Microsoft's application package *Office* composed of basic modules such as Word, Excel, and Powerpoint.

After all, effective exploitation of information artifacts relies on additional and more traditional library services [Graham 1995; Cousins *et al.* 1995; Paepcke 1996] such as

- ▷ information filtering capabilities (static and dynamic views)
- ▷ mobility in heterogeneous environments (platform independence)
- ▷ exchange between libraries (external gateways)
- ▷ persistent storability, multiple user support and recovery (database system support)
- ▷ security and accounting services (authorization and authentication).

However, it is the *binding technology* which, in our view, provides the core capabilities for artifact construction and use – on the information as well as on the software level. Therefore, binding requires particularly strong support in any modern digital library system.

2.3 Binding Environments

On the technical level, extended *binding capabilities* are most crucial for our approach to digital libraries. It is through binding that selected information entities can be organized into artifacts and that information consumers can access the artifacts of their choice. It is also bindings which connect information artifacts to software artifacts when value-adding software services are requested for artifact exploitation.

As outlined above, library entities may be found on any medium, anywhere in the net, privately owned or shared, short-lived or persistent, mutable or not, etc. Consequently, our demands on binding capabilities are multiple [Morrison *et al.* 1990]: Bindings

- ▷ have to be possible between all kinds of entities on an *open set of media*;
- ▷ must also be complete in the sense that all computational entities, *data*, *code* and *threads* can be freely combined by bindings;
- ▷ may be *fixed* over time or *mutable* on demand;
- ▷ may refer to *transient* or *persistent* entities;
- ▷ may involve *local* entities or reach out to *remote* sites of the net.

Bindings to entities *internal* to a system have to be defined and managed as well as *external* bindings to entities spread over heterogeneous systems. Artifact construction may require both kinds of semantics – *copy semantics* and *reference semantics* – via bindings; the transition between both has to be supported.

2.4 Tracing Environments

People, when working with libraries, usually follow certain patterns of use. Frequently they work top down, from vague ideas via keywords and catalogues to volumes and individual texts. Library users work continuously in the sense that they often resume sessions at the point of interrupt some time earlier. There also exist distinguished modes of operation which people apply when using a library:

- ▷ quick look-up of references to dictionaries or texts, or
- ▷ evaluation and memorizing of a concrete text performed in a series of sessions, or
- ▷ construction, maintenance and use of topic-oriented personal reference libraries over years.

The individual context in which such patterns are applied varies, of course, with the user and his task (the who and when, the what and where etc.). With digital libraries we are in a position to trace library processes on an individual level. We can discover individual patterns behind processes and provide environments to support them. Individual process execution contexts can be recorded and the information gained can be used for improving the library content as well as its usability [Böhm and Rakow 1994; Kashyap *et al.* 1996].

Typical information gained from process traces answers questions such as

- ▷ who is involved in artifact construction? when is it constructed?
- ▷ which digital libraries are consulted to find information? where do they reside?
- ▷ which queries are posted and what are the results of these queries?
- ▷ what alternatives or versions exist for artifact construction?

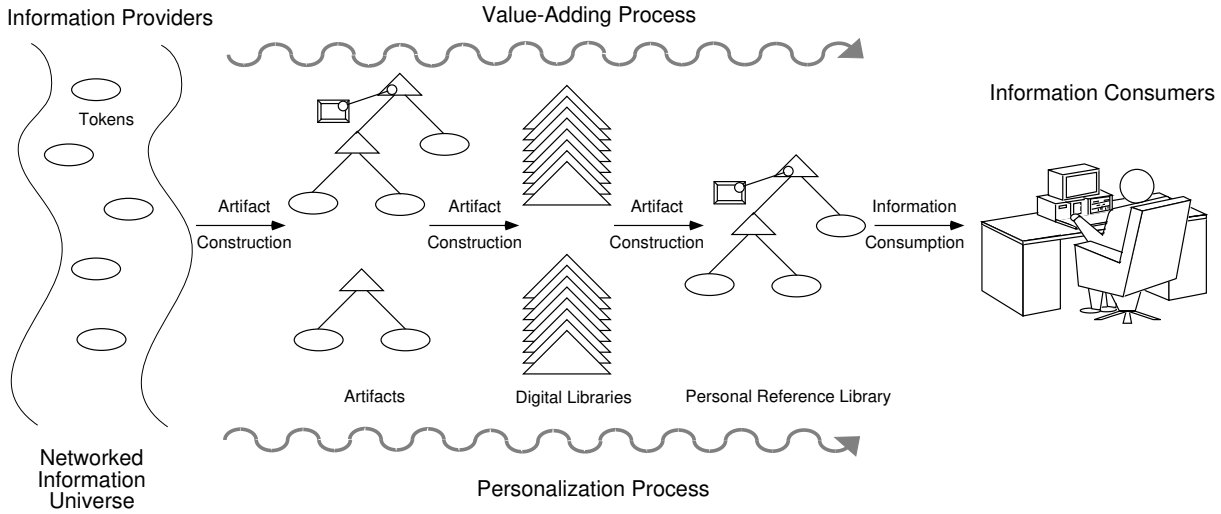

Fig. 1. Constructing Personalized, Value-Added Information Artifacts

▷ which design decisions lead to the artifact?

Some of this information (see Figure 2) can only be given explicitly by the person who constructs the artifact (e.g. information on the why?). Other information can be deduced automatically by the tracing process attributes (e.g. owner, time), statement sequences (e.g. retrieval operations) or process bindings (retrieval results).

We conclude our vision of digital libraries by contrasting them with traditional database systems: databases contain homogeneous data (e.g. relations) and database architectures provide a centralized view on data and software (global schema; distribution transparency). Furthermore, databases are accessed by isolated transactions (ACID principle). Digital libraries differ substantially from databases in all three dimensions just mentioned: they access heterogeneous information represented by multiple models on multiple media; their architecture is a dynamic and highly personalized construction of information and services, and their users work with contexted processes which are traced and evaluated by the library environment. Some of these extended requirements of digital libraries are addressed in the research area of information integration and heterogeneous databases (see e.g. [Garcia-Molina *et al.* 1995; Hammer and McLeod 1993]).

In the following section we discuss some of the linguistic and architectural demands on software platforms on which digital libraries are best developed, maintained and used.

3 Software Requirements for Digital Libraries

Our view of digital libraries imposes several requirements on software platforms on which such libraries are developed, used and maintained. Major demands result from the following properties in digital library systems:

▷ safe extensibility to an open set of new media types, novel library services, extended user requirements etc. [Nürnberg *et al.* 1995]
▷ scalability from private stand-alone libraries to shared group libraries or globally networked libraries
▷ global connectivity with free data exchange, open systems communication, unrestricted software migration etc.
▷ customizability for personalized, value-added information and software artifacts.

Some of these demands involve language requirements, others imply architectural requirements for the digital library system itself. Both sets of requirements are clearly not completely independent of each other as we will see with requirements such as binding or reflection.

3.1 Linguistic Requirements

Our vision of digital libraries raises specific demands on the naming, typing, binding and scoping capabilities of the language of our choice. We argue here for some fairly novel language concepts by which data, code and threads become first-class citizens, i.e., players with equal linguistic rights.

3.1.1 Typing

Digital libraries essentially impose three important demands on the underlying typing system:

▷ type expressiveness for extensive model definition,
▷ type control for safe model execution, and
▷ type inspection for generic programming.

Types for modeling: The information modeling needs of a digital library are abundant and may vary over its

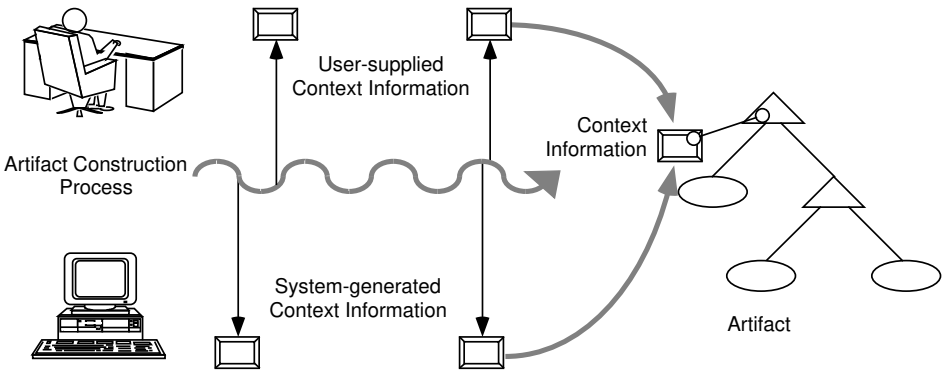

Fig. 2. Context Information for Information Artifacts

lifetime. Consequently, languages with a fixed, built-in set of models do not suffice. Instead, an add-on approach to model definition is requested, a demand which can best be met if such models can be expressed by the language itself. Such a demand leads directly to languages with rich polymorphic type systems, (recursive, parametric) type constructors, subtyping and higher-order functions.

Tasks to be addressed by model extensions are bulk data management [Schmidt 1977; Ackerman and Fielding 1995], multi-media modeling, safe service integration, etc. There is a particular need for a wide range of bulk or collection types such as lists, sets, keyed sets and bags. The framework should provide an add-on bulk type library which satisfies different requirements for search, insert, order or store. Bulk types dynamically added on should, at the same time, be safe, disallowing, for example, the insertion of image tokens into book collections.

In our approach expressive typing schemes come into effect on the artifact level; tokens are automorphically typed (i.e. self describing) bit vectors, a representation which simplifies generic low-level operations such as copying, storage, migration or garbage collection. Furthermore, the automorphic typing enables more specific methods, e.g., for token presentation and basic manipulations.

Types for control: Types capture constraints on computations. Good type systems allow a rich set of constraints to be expressed and to be checked as early as possible. Type checking is usually done statically, i.e., at compile-time: at that time the price for type checking is cheapest and type violations can most easily be fixed. However, long-lived and distributed systems, such as digital libraries, are frequently faced with constraints which arise at different sites or change during their lifetime. This requires more dynamic approaches to typing. If, for example, values are transmitted (via files or communication channels) between independently developed digital libraries, there is no common scope in which a static type check could be performed to guarantee compatibility between data and programs. For tasks like these, (value, type)-pairs have to be available at run-time, and a rich functionality is required for run-time evaluation and exception handling.

Types for genericity: Another need for dynamic types results from the desire to implement generic functions with type-dependent behavior. Such functions take a type representation, usually along with a value of this type, inspect the type and exhibit different behavior depending on the type. Type inspection allows, for example, iteration over the attribute types and attribute values of an aggregate or construction of an aggregate from a list of typed bindings. Such possibilities are required, for example, for generic library browsers capable of displaying and manipulating artifacts of any type.

3.1.2 Binding capabilities

Different kinds of bindings between information artifacts and tokens as well as between information and software artifacts are required. It should be possible to combine them orthogonally:

static and dynamic: In traditional applications bindings to entities are stable over time. For digital libraries such static bindings are, however, not sufficient since artifacts are created dynamically and may be updated. Dynamic bindings are functions (queries or other computations) that are dynamically evaluated and establish bindings on demand.

internal and external: Internal bindings refer to entities which are under the control of the active system while external bindings refer to entities outside. External bindings frequently require special treatment because essential properties of the active system may not apply to external entities. Typical examples of such properties are orthogonal persistence or transactional access.

local and remote: Local bindings refer to entities on the site where the process is currently executing. Digital libraries also require remote bindings which can refer to entities that reside on a different site in the network. Remote bindings imply automatic transfer to the local site if the bound entity is referenced, e.g. for display.

copy and reference: A reference to an entity may be replaced by copying the entity into the local context, i.e., the context of the artifact that contains the reference. This facilitates the access to the referenced entity and increases the autonomy of artifacts. Increased autonomy is required for the construction of a stand-alone artifact collection that may be used offline, e.g. on a CD-ROM. Further aspects of copying in digital libraries are discussed in [Cameron 1994; Shivakumar and Garcia-Molina 1996].

3.1.3 Scoping

The space of library entities reachable from a given information artifact has to be individually structured and organized. Therefore, in addition to static binding environments (scopes) there is a need for dynamic environments [Dearle 1989] in the sense that new bindings may be added or existing bindings may be hidden or removed. Scopes may be public or protected for private use. Libraries may require that scopes be named or even possess first-class status.

3.1.4 First-class status for data, code, and threads

Besides passive information (data), artifacts may also include entities which can be activated (code) or even activities themselves (threads). Instead of constructing artifacts 'in advance', code entities are frequently used to dynamically construct artifacts 'just in time'. Thread entities are required to realize artifacts with autonomous search agents attached to them. Threads may be in the state of running, frozen, waiting for some event, etc. Requesting code having first-class status implies that our language of choice be algorithmically complete.

3.1.5 Exception handling

In dynamically changing and heterogeneous environments not all services are available always and everywhere. This results in exceptional behavior that has to to be handled flexibly and safely.

3.2 Architectural Requirements

A great number of software services have to cooperate smoothly and steadily to collectively provide the services to be expected from an advanced digital library. To do so, software has to show a certain degree of 'social behavior' which cannot yet be taken for granted on current platforms. In addition, functionality which is crucial for libraries, such as persistence and mobility, has to be substantially supported for all kinds of library entities.

3.2.1 Orthogonal Persistence

Generally speaking there is no argument supporting why the lifetime of any digital token or artifact should depend on its type or on the fact as to whether it is composed of data, code or threads. Only if persistence is provided orthogonally to such dimensions, general information (data), the knowledge of how to use it (code), and the actual use made of it (threads) can be preserved and made available any time.

Having a system with orthogonal persistence means, in technical terms, that arbitrary bindings (i.e., arbitrary data structures, function closures, activation environments) can be converted between execution environments and stable storage environments. For elaborate typing and binding models as discussed in the previous subsection this is obviously not a trivial task. However, once this problem is solved other requirements such as free data, code and thread migration or powerful tracing environments are almost free for the asking.

Since persistent storage systems have to deal with all kinds of information tokens and artifacts they do so according to the lowest common denominator as given by bit vectors and references. Further requirements for persistent stores are

- scalability to different implementations
- transactional semantics
- standardized import and export formats
- garbage collection based on reachability.

3.2.2 Open Service Integration

Digital libraries have to be prepared to accept services from 'alien' systems. The diversity of services required by a digital library could never be provided without external support. The demand for novel services is driven by both sides, providers and consumers, and is expected to grow rapidly with the number of libraries on the net. External services may be required, for example, in the following situations:

- external tokens – images, sounds, but also native code – can frequently only be interpreted by software which is only available on the site on which the token resides;
- some sites specialize in services not available at the home site; examples are expensive printing or scanning facilities;
- many standard services, e.g., for presentation, compression, encryption, communication, etc., are factored out to generic subsystems; it is often the case that the complexity of a service demands a specialized service provider; examples are information retrieval and SQL engines;
- finally, legacy awareness requires that external service integration is approached on a general level.

On the modeling level safe service integration can be achieved by expressive typing (see previous subsection); on the system level it requires gateways and call/callback

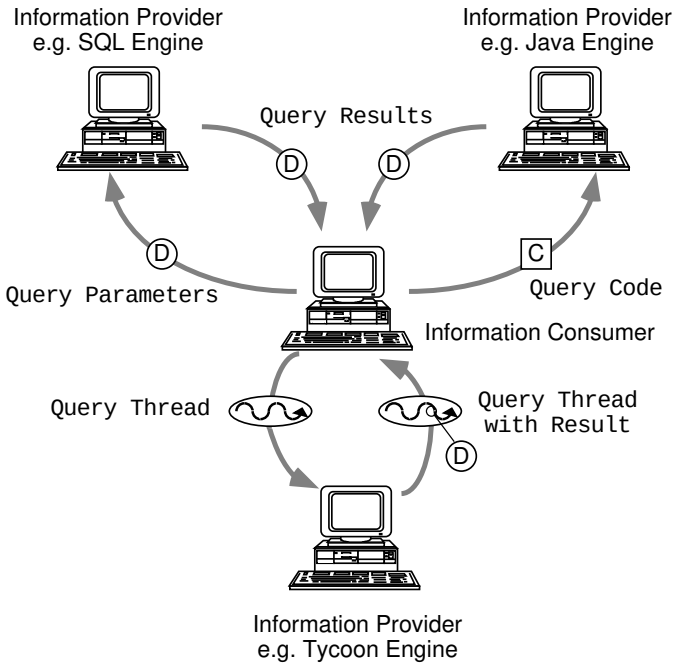

Fig. 3. Data, Code and Thread Mobility

interfaces to the external service's languages (C, C++, ...). In combination, both requirements support the definition and implementation of new software libraries which integrate external services safely into our language of choice. In addition to what is required for service integration in general, standards for export interfaces and for exported references to library entities are necessary. Finally, each digital library also should be prepared to export its services to others and to act as information provider in the net.

3.2.3 Platform Independence

Digital libraries require that information and services must be made available network-wide on heterogeneous platforms (see Figure 3). Library exploitation can be improved considerably if a library's core functionality for searching, querying and storage is provided platform-independently. This means that data, code and thread representations abstract from all platform-dependent details. Technically speaking, this leads to simplified abstract machine architectures based on bytecode representation and interpretation as well as to run-time systems with improved portability, minimal in size and composed of standardized components.

3.2.4 Migration Technology

The usability and autonomy of digital libraries is substantially improved by agents [Birmingham 1995] capable of migrating freely in open networks to visit data sets and collect information [Balabanovic and Shoham 1995]. Mobility, in particular that of threads, means portability at run-time. This requires an extremely flexible binding technology which can dynamically dissolve and establish bindings, replace local bindings by remote ones and implement the notion of ubiquitous resources to which bindings can be established at any site.

The flexibility gained by migration technology should be contrasted with the traditional client-server paradigm which heavily depends on assumptions on the client's home site and on the servers available in the net.

3.2.5 Reflection

There are numerous reasons why digital artifacts may have to be occasionally rearranged when used in networked and long-lived environments. Software migration, integration, (re-)binding, tracing are some such reasons, but also wrapping, persistent storage or optimization. Frequently, such tasks are highly regular and can be systematically performed by algorithms.

Reflection is a powerful yet (relatively) safe technology for the algorithmic inspection and readjustment of digital representation. Reflective program analysis and transformation has to rely on detailed knowledge about the programming language model and its implementation. Reflection may not only be used for generating or altering data or code (at compile-time or run-time) but also for transferring compile-time information (e.g. on types) to run-time. Tasks for which reflection is required in our setting are

- binding environments: reflective manipulation of bindings may, for example, transform local references to remote ones or replace remote references with copies;
- tracing environments: reflection can exploit information in the environment in which threads are executed (e.g. date, user-id, software version numbers, etc.) and attach such meta-information to the executing thread. Such information is collected automatically as a side-effect of user actions. This information collection process can be parameterized by user-defined filters according to user preferences.

4 Tycoon as a Framework for Digital Libraries

The goal of the Tycoon project[1] [Tycoon 1992] is to provide modeling flexibility as well as system stability for multi-functional, long-lived application systems operating in heterogeneous, open and networked environments.

Tycoon contributes to this goal on two levels [Matthes *et al.* 1995].

- The Tycoon language is a persistent polymorphic programming language with an elaborate higher-order type system [Matthes and Schmidt 1991; Schmidt *et al.* 1993; Schmidt and Matthes 1993; Matthes and Schmidt 1993].

[1] Tycoon: Typed communicating objects in open environments.

▷ The Tycoon system is architectured to substantially improve portability, scalability and interoperability as required by applications in open heterogeneous networks [Matthes *et al.* 1996; Mathiske *et al.* 1996; Mathiske *et al.* 1995a; Mathiske *et al.* 1995b; Matthes and Schmidt 1994].

In comparison, standards activities such as CORBA, DCOM, OLE, etc. focus on provider-independent and distribution-transparent service interfaces. In the context of digital libraries, these interfaces encapsulate tokens describing the methods applicable to the tokens. Tycoon supplements these basic token services by supporting the process of artifact construction and the tracing of this process. Therefore, we concentrate on algorithmic aspects, flexible binding capabilities and rich-typed modeling of digital libraries. In this area Tycoon is comparable to portable high-level languages such as Java which, however, is lacking some properties essential to digital libraries, e.g. orthogonal persistence or thread mobility. In the long run we expect a separation of concerns: approaches such as CORBA will concentrate on industrial-strength basic object libraries on the token level; languages like Tycoon, Java and its future versions will provide service integration and are intended to support value-adding processes such as artifact construction.

4.1 Achievements of the Tycoon Language

Tycoon typifies one of those recent language developments [Gosling and McGilton 1995; Morrison *et al.* 1994] which fully exploit the increased capacity of modern computing and storage facilities. Although, according to conventional standards, Tycoon's computational model may be considered resource consuming, it is our firm belief that such an investment is well justified by the substantially improved quality of systems developed in languages such as Tycoon.

4.1.1 Typing

The Tycoon Language (TL, [Matthes and Schmidt 1992; Matthes 1993]) excels in its expressive type system based on existential and universal type quantification, recursive types and structural subtyping. With respect to typing, TL follows the tradition of the experimental polymorphic language Quest [Cardelli 1989].

TL is based on very few built-in types and a small set of type constructors. Orthogonal combination along with recursive type definition and (recursive) type operators provide virtually all data structures of interest.

For a simple digital library, books composed of texts and images may be modelled by:

```
Let BookComponent = Tuple
  case text with content :String
  case image with content :jpeg.T end
Let Book = Tuple
  authors :set.T(Author)  title :String
  contents :list.T(BookComponent) end
Let Library = set.T(Book)
```

The **Let** construct introduces type bindings. In the example above, the two variants (*text* and *image*) of the tuple type *Book* contain a field labeled *content* which is of type *String* in the case of text and of type *jpeg.T* in the case of image.

The second type defines a tuple with three fields (*authors*, *title*, *contents*). The field *authors* is defined as a set of authors, where *set.T* is a polymorphic type operator that takes the element type as actual parameter.

By structural subtyping a (recursive) type *Publication* becomes a subtype of *Book* so that all operations on books, e.g. the displaying of the contents, are also applicable to publications:

```
Let Rec Publication <:Ok = Tuple
  authors :set.T(String)  title :String
  contents :list.T(BookComponent)
  references :list.T(Publication) end
let displayBook(book :Book) :Ok = ...
let napoleonicWars :Publication = ...
displayBook(napoleonicWars)
```

The notation <:**Ok** in the type definition defines a super type needed for type checking of recursive types, in this case the super type of all unparametrized types, **Ok**. The **let** construct introduces value bindings. *displayBook* is a function taking one parameter of type *Book* and returning nothing (represented by the type **Ok**). *napoleonicWars* is a value of type *Publication*.

Generic structures such as sets or lists can be defined within TL through its higher-order polymorphic type system [Schmidt and Matthes 1994; Matthes and Schmidt 1991]. In the following example the recursive type operator for lists depends on an element type E so that lists of type $T(E)$ accept only elements of type E:

```
Let Rec T(E <:Ok) <:Ok =
  Tuple case nil case cons with head :E  tail :T(E) end
let new(E <:Ok) :T(E) = ...
let cons(E <:Ok  element :E  list :T(E)) :T(E) = ...
```

The functions *new* and *cons* take a type parameter E constraining the types of the actual values passed to the functions resp. that describes the result value returned by the functions. This enables static type checking of generic operators.

TL supports dynamic types and dynamically-typed values. A dynamic value is a pair of a value v and a run-time representation t of its type. If a dynamic value component v is extracted, its associated type representation t can be inspected. This enables a boolean subtype test whether t is a subtype of a given supertype T. Such functionality is required to assure, for example, that an artifact transferred from an external site is a book:

```
(* -- transfer a dynamic value: *)
let dynamicValue :dynamic.T = receive(channel)
(* -- test if this value is a book: *)
let book :Book = dynamic.be(:Book dynamicValue)
(* -- book is now statically typed *)
displayBook(book)
```

The actual type parameter *:Book* is passed to the function *dynamic.be*. It instantiates a formal type parameter like *T* in the following example.

Another application of dynamic types is, for example, a generic artifact browser that is capable of working on arbitrarily structured artifacts:

```
let artifactBrowse(Dyn T <:Ok  artifact :T) :Ok = ...
artifactBrowse(:Library historyLibrary)
artifactBrowse(:Book napoleonicWars)
artifactBrowse(:BookComponent napoleonPicture)
```

4.1.2 Binding Capabilities

Tycoon supports a wide variety of binding concepts covering a substantial range of what is requested in Section 3.1. The more traditional binding alternatives include immutable, mutable, static and dynamic bindings to types and values:

```
(* -- type binding: *)
Let Book = Tuple ... end
(* -- value binding: *)
let napoleonicWars = tuple ... end
(* -- mutable binding: *)
let var actualBook = tuple ... end
(* -- static binding: *)
let bookMark = actualBook
(* -- assignment: *)
actualBook := tuple ... end
(* -- dynamic binding: *)
let displayBook(book :Book) :Ok = ...
displayBook(napoleonicWars)
```

In Tycoon it is completely transparent whether a binding refers to a local or a remote entity [Mathiske *et al.* 1995a; Mathiske *et al.* 1995b]. If an entity leaves the local scope in which it is bound, two alternatives exist: Either the entity (and its closure) is copied or the binding is replaced by a remote reference. On request remote references may be eliminated by copying.

Bindings to external functions are established by the predefined *bind* function. Calls of external functions are indistinguishable from calls of internal functions. Tycoon takes care of the machine-dependent part of the parameter conversion and call mechanisms. The following example shows the binding to a machine-dependent function which displays JPEG images:

```
let displayJPEG = bind(:Fun(:JPEG) :Ok
   "imageLib" "display_jpeg_image")
displayJPEG(image)
```

The type *:**Fun**(:JPEG) :**Ok*** describes the signature of the external function. This is, again, a type parameter that enables static type checking.

It should be noted that call and binding transparency is a language requirement with heavy demands on the architecture of the system.

4.1.3 First-Class Data, Code and Thread Entities

Artifacts contain data but may also require code and thread components. Code entities are used to construct artifacts dynamically 'just in time'. Each time the subsequent function *newsOnElection* is evaluated the Reuters news service is asked for the latest news on the election:

```
let newsOnElection() :set.T(News) = begin
 let reuters = openConnectionTo("Reuters News Service")
  searchFor(reuters "Election")
end
let actualNews = newsOnElection()
```

Even more value may be added to artifacts by higher-order functions expressing specific personal preferences, such as an interest in news which mention both candidates:

```
let newsOnElection(pref(:News) :Bool) :set.T(News) = ...
let bothCandidates(news :News) :Bool =
  news.includes("Clinton") andif news.includes("Dole")
let actualNews = newsOnElection(bothCandidates)
```

With threads components artifacts can work off-line based on local state information. To collect only the latest news from a news service a thread may be employed which remembers the time it last collected the news:

```
let collectLatestNews() :Ok = begin
 let var lastCollectionTime = never
 let isActualNews(news :News) :Bool
   news.time > lastCollectionTime
 loop
   let latestNews = newsOnElection(isActualNews)
   lastCollectionTime:= now()
   deliverNews(news)
   thread.sleep()
 end
end
let newsThread = thread.start(collectLatestNews)
```

Our *newsThread* collects (in parallel to other activities) all news available since its last wakeup; then it goes to sleep until reactivated by *thread.wakeup(newsThread)*.

Threads are first-class entities in Tycoon and can be observed (traced) reflectively. In this way context data of artifact construction processes can be created and stored persistently [Matthes and Schmidt 1994]. With Tycoon's reflective capabilities such processes can be parameterized by user-preferred filters for the information to be attached to the artifact. This may not only add value to the constructed artifact but also help improving the artifact construction process.

4.1.4 Exception Handling

Since unexpected events are frequent in open systems, Tycoon provides strong support for the handling of exceptional situations. If an exception is raised, execution stops and continues where the last exception handler was

installed, possibly crossing function or module boundaries.

The exception handling routine decides which action is sufficient to handle the exceptional situation. For example, when the display routine for a JPEG image is not available on a particular workstation, a conversion from JPEG to GIF may help:

```
try displayJPEG(jpegPicture) when notAvailable then
  try displayGIF(convertJPEGtoGIF(jpegPicture))
  when notAvailable with name :String then
    message("Cannot display picture " <> name)
  end
end
```

4.1.5 Scopes

Tycoon applications, frequently, are very large. As a result, they can be structured into interfaces, modules and libraries which restrict the scope of bindings [Schmidt *et al.* 1993]. Tycoon libraries are nested high-level entities that declare a definition order for interfaces and modules so that they can only use identifiers which were defined previously, preventing cyclic dependencies. Interfaces export public identifiers while modules define the bindings for these identifiers using also private bindings. Through nesting and hiding mechanisms the scope of identifiers can be controlled.

Currently, an orthogonal suite of scoping primitives is under investigation which specializes in the requirements of personalized scopes in open networked environments. It is provided by a generic Tycoon module that may be combined with modules for distribution, versioning, access control [Rudloff *et al.* 1995], etc. thus enabling the flexibility required by personalized digital libraries.

4.2 The Tycoon System Architecture

The architectural requirements for digital libraries can be seen from two perspectives.

- ▷ Since digital libraries hold a vast variety of information content and provide extended information handling services there is an intrinsic complexity involved in libraries seen as a performing software system.
- ▷ Complexity also results from the fact that library systems will never stay stable; instead it has to be anticipated that libraries will develop during their virtually infinite lifetime, even more so when connected to a virtually unlimited network space.

Tycoon reduces the complexity of both of the above tasks, first by enabling the construction of systems with a higher degree of regularity (through orthogonal persistence, platform independence, migration technology etc.), second by supporting system evolution and change management (open service integration, scalability, reflection). The layered architecture of the Tycoon System (see Figure 4) scales from single-user PC-based applications to distributed applications in open heterogeneous networks.

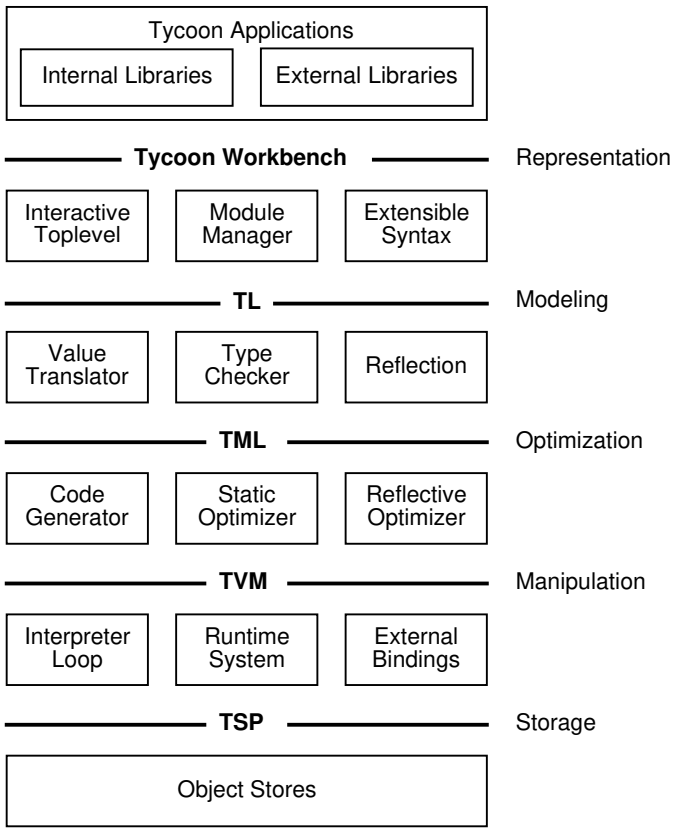

Fig. 4. Layered Architecture of the Tycoon System

4.2.1 Persistence

Tycoon provides orthogonal persistence for data, code and threads. The Tycoon Store Protocol, TSP [Matthes *et al.* 1996], defines a uniform, data model-independent call-level interface to multiple (commercially distributed) persistent object stores abstracting from implementation details of the underlying persistent stores. A typical TSP server is implemented by a store adaptor which maps TSP data structures and functions to data structures and operations of an existing object store.

TSP contributes significantly to Tycoon's system scalability since TSP clients can decide between different

- ▷ garbage collection strategies
- ▷ object faulting mechanisms
- ▷ error recovery, logging and persistent savepoint mechanisms
- ▷ commit protocols (for distributed systems)
- ▷ security and authentication support

or choose between a single- or multi-user environment.

A primary design goal of TSP is to provide efficient data storage independent of the data and language model used by TSP store clients. TSP supports polymorphically typed models where store objects may contain values of different types. TSP therefore uses an untyped

low-level store model which distinguishes only (uninterpreted) bit vectors, used for information tokens, and arrays of references used for artifacts. The lifetime of TSP store objects is defined by reachability and storage is reclaimed by garbage collection.

4.2.2 Open Service Integration

External services, e.g. visualization systems or database systems, can be integrated into the Tycoon system through bindings to external functions [Schmidt and Matthes 1993]. The binding mechanism allows the Tycoon system to call external functions as well as it allows the external system to call back the Tycoon system via higher-order functions. Examples are event-controlled windowing systems that call application-defined functions when a button is pressed or a window closed etc.

Such services are wrapped by Tycoon in portable and type-safe functions exploiting the full polymorphic power of the Tycoon type system even in cases in which external functions are completely untyped. When calling external functions from Tycoon or, in reverse, calling (back) Tycoon functions from external systems, parameter conversion is handled by the Tycoon run-time system.

The following example shows an interface *SQL* that exports polymorphic functions to access different SQL database systems:

```
interface SQL import ... export
  error :Exception with sqlError:String end
  Table(E, K <:Tuple end)  <:Ok
  ...
  openTable(Dyn E, K <:Ok
    tableName :String) :Table(E K)
  ...
  lookup(E, K <:Ok  from :Table(E K)  key :K) :E
  ...
  selectFromWhere(E, K, R <:Ok  project(:E) :R
    from :Table(E K)  where(:E) :Bool) :Iter.T(R)
  ...
end
```

Tycoon libraries hold, for example, two different modules *ingresSQL* and *oracleSQL* which implement the above interface by bindings to the dynamic SQL call interfaces of the Ingres and Oracle relational database management systems. The type operator *Table(E K)* exported from the interface describes the type of SQL tables with element type E and key K. For example, a value of type *oracleSQL.Table(News Int)* is an Oracle table with rows that have attributes as defined by the Tycoon tuple type *News* where a running number identifies a row.

The polymorphic function *openTable* opens a named table for further processing and takes dynamic type variables E and K as its first arguments to ensure that the database table structure matches the Tycoon type information. If a schema mismatch is detected, the Tycoon exception *error* is raised at run-time. All other operations of the SQL interface (queries, table updates) can be checked statically by the Tycoon compiler based on the polymorphic signatures assigned to the SQL functions. For example, the signature of the function *lookup* expresses the type constraint that from a table of type *Table(E K)* only tuples of a matching type E can be retrieved.

The function *select* shows how polymorphic typing can be used to statically describe a (unary) select statement. The selection on relation *from* is controlled by the predicate *where* that works on relation elements. *project* builds the resulting tuples. Note that this works only if the external database system is capable of including arbitrary function calls dynamically in queries.

The Tycoon service integration technology sketched above is open to arbitrary services of interest. Since, besides formatted data and SQL, texts are essential for digital libraries, information retrieval technology has to be available based on fuzzy queries and ranking lists . Therefore, Tycoon also maintains a text retrieval library which integrates the services provided by the information retrieval system Inquery [Callan *et al.* 1991; Broglio *et al.* 1994].

Besides the integration of external services, Tycoon is also open for being integrated as a server into external systems. For this purpose, Tycoon provides a callback mechanism which allows external systems to invoke Tycoon functions.

In the following example, Tycoon is set up as a server, e.g. as WWW server, that accepts external communication requests. In our example, remote procedure calls are used for communication and a RPC server is installed and started which stores and retrieves JPEG images:

```
let jpegDB = tuple
  let store(image :jpeg.T  name :String) :Ok = ...
  let retrieve(name :String) :jpeg.T = ...
end
let server = rpcServer.new()
rpcServer.register(server  "JPEG DB"  jpegDB)
rpcServer.dispatch(server)
```

4.2.3 Platform Independence

The Tycoon Virtual Machine (TVM) is an abstract call interface above the TSP layer that defines a bytecoded instruction set based on a higher-order, functional execution model. TVM bytecode is either interpreted by a virtual machine or is compiled on the fly into target machine code. The TVM interpreter and its associated run-time system are written in ANSI-C.

The platform-independence of the TVM model makes it possible to dynamically transfer portable bytecode between heterogeneous nodes in distributed digital libraries without recompilation. Utilizing TSP's linear external data representation (TXR), it is also possible to migrate a thread across system boundaries [Mathiske *et al.* 1995a; Mathiske *et al.* 1995b; Matthes and Schmidt 1994].

For example, the higher-order query function *newsOnElection* can be rewritten in the following way for

a client-server approach where the function *bothCandidates* is shipped to a server via RPC and evaluated at the server site:

```
let newsOnElection =
  rpc.bindTo(:Fun(pref(:News) :Bool) :set.T(News)
    serverSite "newsOnElection")
let bothCandidates(news :News) :Bool =
  news.includes("Clinton") andif news.includes("Dole")
let actualNews = newsOnElection(bothCandidates)
```

4.2.4 Migration Technology

Even more flexibility is achieved in Tycoon because threads can migrate between the nodes in a network [Mathiske *et al.* 1995a; Mathiske *et al.* 1996]. This enables, among others, the implementation of agents. An agent may include the trace of the search in its search process, e.g. the items it has already found, the number of nodes visited or the nodes it has been recommended to visit:

```
Let SiteData = Tuple news :set.T(News)
  sitesRecommended :set.T(Site) end
let homeSite = agent.thisSite()
let sitesVisited = set.new(:Site)
let sitesToVisit = set.create(newYork washington)
let newsFound = set.new(:News)
while not(set.empty(sitesToVisit)) do
  let nextSite = set.getAny(sitesToVisit)
  let local :SiteData = agent.migrateTo(nextSite)
  set.insert(sitesVisited nextSite)
  set.include(newsFound local.news.search("Election"))
  let newSites =
    set.exclude(local.sitesRecommended sitesVisited)
  set.include(sitesToVisit newSites)
end
agent.migrate(homeSite)
show(newsFound)
```

4.2.5 Reflection

The Tycoon system supports different manners of reflection. By *linguistic reflection* [Stemple *et al.* 1991; Stemple *et al.* 1992] the execution of the compiler (compile-time reflection) or the application (run-time reflection) can be influenced.

Compile-time reflection is achieved by executing user-defined code during compilation. Dynamic types, for example, are implemented by compile-time reflection. The compiler collects type information and stores it persistently for run-time use. Similarly, operations on compile-time information are performed as, for example, by the **Repeat** construct which repeats a list of Tycoon signatures (for the definition of *Book*, see above):

```
let createBook(Repeat Book) :Book = ...
```

Here, **Repeat** *Book* is reflectively replaced by

```
authors :set.T(Author)  title :String ...
```

Run-time reflection is achieved by typed bindings to compiler subcomponents (parser, type checker, code generator, evaluator, module manager, ...), thus made available to applications at run-time. In this way code may be evaluated or even generated depending on run-time (computed) information. In Tycoon full type safety is guaranteed, even in the presence of run-time and compile-time reflection, by the consistent use of dynamic types in the Tycoon compile-time and run-time environments. The following example shows the binding of the subtype checking function of the compiler and its use for dynamic values at run-time:

```
let isSubType = reflect(:Fun(t1, t2 :typeRep_T) :Bool
  "isSubType")
(* -- transmit dynamic value, e.g. via network: *)
let dynamicValue = receive(...)
if isSubType(typeOf(dynamicValue) :Book) then ... end
```

Behavioral reflection [Kirby *et al.* 1996] is a second kind of reflection supported by Tycoon. It is achieved by influencing Tycoon's interpreter loop and the run-time system of the Tycoon Machine. For example, the following code marks the image *napoleon* as immobile. Consequently, if the book *napoleonicWars* which references the image *napoleon* is transferred, the image is not copied; instead a remote reference to the image is introduced:

```
markAsImmobile(napoleon)
sendTo(newYork napoleonicWars)
```

In summary, Tycoon is a powerful framework which provides much of the functionality required to make multi-functional and multi-media application systems persist in turbulent environments such as open, heterogeneous and networked information infrastructures.

5 The Warburg Electronic Library Project

The Warburg Electronic Library Project (WEL) began within the framework of an interdisciplinary cooperation [Niederée *et al.* 1996] between our group and the Art History department at the University of Hamburg. The goal of this cooperation is the examination, development and application of digital libraries for art history research purposes.

The development of this particular digital library focuses on two topics:

- ▷ understanding the special requirements of the application domain and their adequate realization;
- ▷ personalization of a working environment for art history research by tailoring personal digital reference libraries to the needs of individual information consumers and specific tasks.

5.1 The Application Domain

The application domain of art history is dominated by image material augmented by texts and multi-media artifacts such as films and audio documents. Art history

research stresses the examination of themes, the identification of represented icons (symbols), events and persons and their classification according to these issues. In the handling of image material automatic image processing plays only a subordinate role. Most of the information has to be added by people in a value adding process.

This style of image handling is not restricted to art history. Other areas such as marketing and press archives are also primarily interested in themes and messages transported by their artifacts as well as the represented objects and persons.

The PI-Index

Political Iconography (PI) is the area of art history which examines political messages conveyed in images showing regents, politicians, ceremonies, political acts, etc. The underlying assumption of the PI is that the effects of political actions are not restricted to contracts and political documents but are also depicted in paintings, monuments and buildings.

The art history department has developed an elaborated ontology for the classification of images according to their political messages called the *PI-Index*. This ontology consists of a hierarchy of terms referring to politics, political acts, and social phenomena. It includes terms as varying as science, marriage, democracy, shepherd, and revolution.

The classification of image material according to this ontology cannot be done automatically. It is itself a result of scientific work in the area of art history. About 250,000 cards with photographs of paintings, etc. showing politicians, political acts and ceremonies, battles and social events from all epochs and countries are already classified according to this scheme. The classification is not disjoint, many cards are assigned to several terms of the index.

5.2 Tokens and Artifacts in the Warburg Electronic Library

Photographs or prints of paintings showing regents or related matters, associated texts, speeches of politicians, and documentary films about regents like Napoleon, form the relevant information tokens in this digital library. The tokens are created and included into the information universe through scanning or an equivalent digitalization process. A further source of tokens is text editing. Scanned image tokens are stored as bit vectors in formats like JPEG or GIF. In Tycoon types are associated to these bit vectors and routines, preventing the erroneous use of bit vectors and routines. The following example shows the association of a type T and the routines *scan* and *display* for JPEG images:

```
let jpeg = tuple
  Let T = ...
  let scan = bind(:Fun() :T  "imageLib"
    "scan_jpeg_image")
  let display = bind(:Fun(:T) :Ok  "imageLib"
    "display_jpeg_image")
end
let jpegImage = jpeg.scan()
jpeg.display(jpegImage)
```

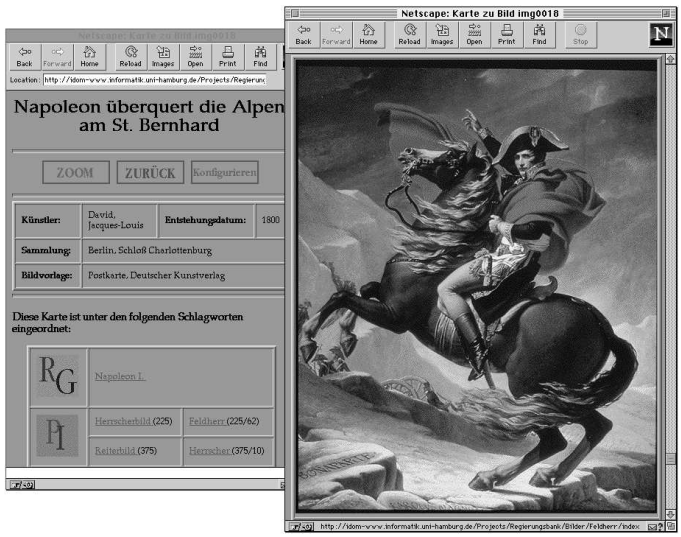

Fig. 5. Napoleon crossing the Alpes at St. Bernhard

As mentioned in Section 4 the *bind* function in Tycoon is used to include external, machine-dependent routines *scan_jpeg_image* and *display_jpeg_image* that work on JPEG files.

On this level the tracing environment may provide information about the date of digitalization and the identification of the person creating this token.

The tokens are taken as starting points for the artifact construction. The image token showing Napoleon crossing the Alps, for example, is taken as a basis for an artifact as illustrated in Figure 5. The artifact contains a reference to the scanned painting (image token) and a thumbnail copy of it, as well as information about the artist, the title, and the date of the the painting as well as the location where the original painting can be found and the source of the scanned print of the painting.

In addition, the artifact is accompanied by content-descriptive metadata: It is classified according to the ontology of the *PI-Index* and the regent pictured by the token, here Napoleon, is specified. This additional information can be considered as information tokens from a private information source.

The type representation of this artifact in Tycoon looks as follows:

```
Let PIArtifact = Tuple
  originalImage :Reference(jpeg.T)
  thumbnailImage :jpeg.T
  title :String
  date :Date
  ...
end
```

Reference is a data type which abstracts from the distinct kinds of possible bindings like external/internal, local/remote etc.

The Warburg Electronic Library maintains a large collection of these artifacts together with services for their persistent storage, transmission, retrieval, and presentation. On the language level retrieved artifacts may be included into a list of type *list.T(PIArtifact)* and the associated thumbnail images may be displayed by using iteration abstraction provided as generic services by a Tycoon library.

```
list.forEach(artifactSelection fun(a :PIArtifact)
  jpeg.display(a.thumbnailImage))
```

The library is augmented by an Oracle database containing historical data on regents of all centuries. The Tycoon SQL-gateway to Oracle databases is described in Section 4. Looking up historical data about Napoleon is accomplished by the following function of the SQL interface.

```
oracleSQL.lookup(regents "Napoleon")
```

In addition to image artifacts and structured data the Warburg Electronic Library also contains text collections, e.g. about political iconography and political events. These collections can be searched by an Inquery information retrieval engine. Employing the Tycoon Inquery gateway the collection can be searched in the same language framework as the Oracle database.

```
inquery.eval(politicalEvents
  "Anything about Napoleon and Waterloo")
```

The query results thus can be easily combined, e.g. in order to prepare a presentation about Napoleon at Waterloo with historical dates from the database *regents* and text information from the text collection *politicalEvents*.

5.3 Personal Reference Libraries

Digital libraries provide multi-media material for groups of information consumers. To secure the success of a complex task a working environment tailored to the individual requirements of the consumer and to the specific task is necessary. The development of such personalized, cooperative working environments called *personal reference libraries* is currently being examined in our group. Personal digital reference libraries can be stored persistently and exchanged with other persons who work on the same or a similar topic.

An example of the construction of a personal reference library is the preparation of a publication and/or demonstration on equestrian portraits of regents throughout the centuries.

The ontology PI-Index includes the terms *Reiterbild*, i.e. equestrian portrait, and *Herrscher*, i.e. regent. The set of pictures and other material classified into both categories can thus be easily accessed using the services of the digital library.

In the next step a personal reference library can be created for the special task by selecting representative pictures and related text material from the different centuries. The selected pictures can be annotated with comments giving, for example, the reason for the choice, or ideas for the publication. Further information such as the date of inclusion, the creator of the artifact, and the source of the information can be added automatically from the tracing environment.

Since personal reference libraries are also artifacts (compare Figure 1) the different supported binding mechanisms can be exploited to realize the separation between the public scope of the digital library and the private tailored scope of the personal reference library.

- Remote bindings to relevant paintings in other art collections and local bindings to paintings organized in the *PI-Index* can be established and stored in local object stores.
- Information on artists, available on CD-ROM, may be included in the personal library by external references in order to gain a better understanding of individual paintings.
- Stored parameterized queries can be used to establish dynamic bindings to contemporary paintings of horses and other animals, evaluating the query on demand for different parameters specifying the time frame of interest.
- Text information about the illustrated event, the regent or equestrian portraits in general can be included by remote references to entire publications complemented by copies of text passages considered important for the publication in work.
- Autonomous bindings (agent technology) can be exploited to search further art collections for representative equestrian portraits of regents. The respective agent may be resent periodically in order to retrieve the most actual publications and newly created tokens and artifacts.

Personal reference libraries are created stepwise, scanning the available material in several sessions, including new more adequate tokens and artifacts, possibly rejecting previous choices.

This requires a binding environment that can be updated dynamically (inserting and deleting bindings). Dynamic environments as proposed in [Dearle 1989] are considered a good starting point for the realization of these dynamic artifacts. The combination of the various binding mechanisms with the concept of a dynamic environment is a topic of current research.

6 Conclusions and Future Research

Central to our view of digital libraries over networked information tokens is the value-added construction of information artifacts and their use in personalized information environments. The library processes of artifact construction and use are enabled and supported by advanced binding and tracing environments which place substantial demands on the linguistic and architectural

requirements for digital libraries. Although languages and systems such as Tycoon meet many of the requirements requested, there remains the practical problem of how to best organize and customize the concrete software which implements such advanced environments for library process support.

On various occasions in this paper we mentioned already the similarity between digital libraries with information artifacts and software libraries (e.g. [Meyer 1990; Meyer 1994]) with generic software tokens and customized software artifacts constructed for specific use. Based on our initial experience we expect that much of the functionality of the binding and tracing environments for information artifacts can also be used for the construction and use of the software artifacts required by digital libraries. The first-class status of data, code and threads in Tycoon provides a promising framework for such an approach.

On the technology level, we have a vested interest in the extension of our binding technology, e.g., by the concept of versioned bindings and by extended operations on structured collections of bindings.

The most relevant input, however, for the future development of digital libraries we expect from our cooperation with potential user communities which have both the relevant library contents as well as the pragmatics of how to use them. From their feedback we expect to gain the domain knowledge which is required to design and realize the adequate abstractions for future digital libraries.

Acknowledgement. This research was supported by Esprit Projects ISC-CAN-080 and 22552-PASTEL.

Ackerman and Fielding 1995: Ackerman, M.S. and Fielding, R.T. Collection maintenance in the digital library. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, June 1995.

Balabanovic and Shoham 1995: Balabanovic, M. and Shoham, Y. Learning inforamtion retrieval agents: Experiments with automated web browsing. In *Proceedings of the AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Resources*, March 1995.

Birmingham 1995: Birmingham, W.P. An agent-based architecture for digital libraries. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, June 1995.

Böhm and Rakow 1994: Böhm, K. and Rakow, T.C. Metadata for multimedia documents. *Sigmod Record*, 23(4):21–26, December 1994.

Broglio et al. *1994:* Broglio, J., Callan, J.P., and Croft, W.B. Inquery system overview. In *Proceedings of the TIPSTER Text Programm (Phase 1)*, pages 47–67. Morgan Kaufmann Publishers, 1994.

Callan et al. *1991:* Callan, J.P., Croft, W.B., and Harding, S.M. The Inquery retrieval system. In *Proceedings of the Third International Conference on Database and Expert Systems*, 1991.

Cameron 1994: Cameron, R.D. To link or to copy? Four principles for materials acquisition in internet electronic libraries. Technical Report CMPT TR 94-08, School of Computing Science, Simon Fraser University, December 1994.

Cardelli 1989: Cardelli, L. Typeful programming. Technical Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, May 1989.

Cousins et al. *1995:* Cousins, S., Ketchpel, S., Paepcke, A., Garcia-Molina, H., Hassan, S., and Röscheisen, M. Interpay: Managing multiple payment mechanisms in digital libraries. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, 1995.

Dearle 1989: Dearle, A. Environments: a flexible binding mechanism to support system evolution. In *Proc. HICSS-22, Hawaii*, volume II, pages 46–55, January 1989.

Garcia-Molina et al. *1995:* Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J. Integrating and accessing heterogeneous information sources in tsimmis. In *Proceedings of the AAAI Symposium on Information Gathering*, pages 61 –64, Stanford, California, March 1995.

Gosling and McGilton 1995: Gosling, J. and McGilton, H. The Java language environment — A whitepaper. Technical report, Sun Microsystems, October 1995.

Graham 1995: Graham, P.S. The digital research library: Tasks and commitments. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, June 1995.

Hammer and McLeod 1993: Hammer, J. and McLeod, D. An approach to resolving semantic heterogeneity in a federation of autonomous heterogeneous database systems. *International Journal of Intelligent and Cooperative Information Systems*, 2(1):51 – 83, March 1993. 5.

Kashyap et al. *1996:* Kashyap, V., Shah, K., and Sheth, A. Metadata for building the mutimedia patchquilt. In Subrahmenian, V.S. and Jajodia, S., editors, *Multimedia Database Systems*, pages 297–319. Springer, 1996.

Kirby et al. *1996:* Kirby, G.N.C., Connor, R.C.H., Morrison, R., and Stemple, D. Using reflection to support type-safe evolution in persistent systems. Technical Report CS/96/10, University of St. Andrews, 1996.

Li et al. *1996:* Li, W., Gauch, S., Gauch, J., and Pua, K.M. Vision: A digital video library. In *Digital Libraries '96, 1st ACM International Conference on Digital Libraries*, 1996.

Mathiske et al. *1995a:* Mathiske, B., Matthes, F., and Schmidt, J.W. On migrating threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, June 1995. (Also appeared as TR FIDE/95/136).

Mathiske et al. *1995b:* Mathiske, B., Matthes, F., and Schmidt, J.W. Scaling database languages to higher-order distributed programming. In *Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy*. Springer-Verlag, September 1995. (Also appeared as TR FIDE/95/137).

Mathiske et al. *1996:* Mathiske, B., Matthes, F., and Schmidt, J.W. On migrating threads. To appear in the Journal of Intelligent Information Systems, 1996.

Matthes and Schmidt 1991: Matthes, F. and Schmidt, J.W. Bulk types: Built-in or add-on? In *Database Programming Languages: Bulk Types and Persistent Data*. Morgan Kaufmann Publishers, September 1991.

Matthes and Schmidt 1992: Matthes, F. and Schmidt, J.W. Definition of the Tycoon language TL – a preliminary report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

Matthes and Schmidt 1993: Matthes, F. and Schmidt, J.W. System construction in the Tycoon environment: Architectures, interfaces and gateways. In Spies, P.P., editor, *Proceedings of Euro-Arch'93 Congress*, pages 301–317. Springer-Verlag, October 1993.

Matthes and Schmidt 1994: Matthes, F. and Schmidt, J.W. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pages 403–414, Santiago, Chile, September 1994.

Matthes et al. *1995:* Matthes, F., Schröder, G., and Schmidt, J.W. Tycoon: A scalable and interoperable persistent system environment. In Atkinson, M.P., editor, *Fully Integrated Data Environments*. Springer-Verlag (to appear), 1995.

Matthes et al. *1996:* Matthes, F., Müller, R., and Schmidt, J.W. Towards a unified model of untyped object stores: Experience with the Tycoon store protocol. In *Advances in Databases and Information Systems (ADBIS'96), Proceedings of the Third International Workshop of the Moscow ACM SIGMOD Chapter*, 1996.

Matthes 1993: Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993. (In German).

McNab et al. *1996:* McNab, R.J., Smith, L.A., Witten, I.H., and Henderson, C.L. Towards the music library: Tune retrieval from acoustic input. In *Digital Libraries '96, 1st ACM International Conference on Digital Libraries*, 1996.

Meyer 1990: Meyer, B. Lessons from the design of the Eiffel libraries. *Communications of the ACM*, 33(9):69–88, September 1990.

Meyer 1994: Meyer, B. *The Base Object-oriented Component Libraries.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

Morrison et al. *1990:* Morrison, R., Atkinson, M.P., Brown, A.L., and Dearle, A. On the classification of binding mechanisms. *Infomation Processing Letters*, 34(2):51–55, 1990.

Morrison et al. *1994:* Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.J., Dearle, A., Kirby, G.N.C., and Munro, D.S. The Napier88 reference manual (release 2.0). FIDE Technical Report Series FIDE/94/104, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1994.

Niederée et al. *1996:* Niederée, C., Hattendorf, C., Müßig, S., Warnke, M., and Schmidt, J.W. Warburg electronic library: Eine digitale Bibliothek für die Politische Ikonographie. *unihh forschung*, XXXI, 1996.

Nürnberg et al. *1995:* Nürnberg, P.J., Furuta, R., Leggett, J.L., Marshall, C.C., and Shipman III, F.M. Digital libraries: Issues and architectures. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, June 1995.

Paepcke 1996: Paepcke, A. Digital libraries: Searching is not enough. *D-Lib Magazine*, May 1996. http://ukoln.bath.ac.uk/dlib/dlib/may96.

Röscheisen et al. *1994:* Röscheisen, M., Mogensen, C., and Winograd, T. Shared web annotations as a platform for third-party value-added information providers: Architecture, protocols, and usage examples. Technical Report CSDTR/DLTR, Computer Science Department, Stanford University, Stanford, CA 94305, U.S.A., November 1994. http://www-pcd.stanford.edu/COMMENTOR.

Röscheisen et al. *1995:* Röscheisen, M., Winograd, T., and Paepcke, A. Content ratings and other third-party value-added information - defining an enabling platform. *D-Lib Magazine*, August 1995. http://ukoln.bath.ac.uk/dlib/dlib/august95.

Rudloff et al. *1995:* Rudloff, A., Matthes, F., and Schmidt, J.W. Security as an add-on quality in persistent object systems. In *Second International East/West Database Workshop, Klagenfurt, Austria*, Workshops in Computing, pages 90–108. Springer-Verlag, 1995. (Also appeared as TR FIDE/95/138).

Schmidt and Matthes 1993: Schmidt, J.W. and Matthes, F. Lean languages and models: Towards an interoperable kernel for persistent object systems. In *Proceedings of the IEEE International Workshop on Research Issues in Data Engineering*, pages 2–16, April 1993.

Schmidt and Matthes 1994: Schmidt, J.W. and Matthes, F. The DBPL project: Advances in modular database programming. *Information Systems*, 19(2):121–140, 1994.

Schmidt et al. *1993:* Schmidt, J.W., Matthes, F., and Valduriez, P. Building persistent application systems in fully integrated data environments: Modularization, abstraction and interoperability. In *Proceedings of Euro-Arch'93 Congress*, pages 270–287. Springer-Verlag, October 1993.

Schmidt 1977: Schmidt, J.W. Some high level language constructs for data of type relation. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Toronto, Canada*, August 1977.

Shivakumar and Garcia-Molina 1996: Shivakumar, N. and Garcia-Molina, H. Building a scalable and accurate copy detection mechanism. In *Digital Libraries '96, 1st ACM International Conference on Digital Libraries*, 1996.

Stemple et al. *1991:* Stemple, D., Morrison, R., and M., Atkinson. Type-safe linguistic reflection. In *Database Programming Languages: Bulk Types and Persistent Data*, pages 357–362. Morgan Kaufmann Publishers, 1991.

Stemple et al. *1992:* Stemple, D., Sheard, T., and Fegaras, L. Linguistic reflection: A bridge from programming to database languages. In *Proceedings 25th Annual Hawaii International Conference on System Sciences*, pages 46–55, 1992.

Tycoon 1992: WWW home page for the Tycoon project. http://idom-www.informatik.uni-hamburg.de/Projects/-Tycoon/entry.html, 1992.

Van House 1995: Van House, N.A. User needs assessment and evaluation for the uc berkeley electronic environmental library project: a preliminary report. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, School of Library and Information Studies University of California Berkeley, CA 94720-4600, USA, June 1995. Austin, Texas, USA.

This article was processed by the author using the LaTeX style file *cljour2* from Springer-Verlag.