Published in Lecture Notes in Computer Science LNCS 3263, Springer Verlag
Proceedings of Net.ObjectDays 2004
Volume "Object-Oriented and Internet-Based Technologies"

White Paper
Ercatons: Thing-oriented Programming

Oliver Imbusch, Dr. Falk Langhammer and Guido von Walter

June 25, 2004

Living Pages Research GmbH
Kolosseumstrasse 1a, 80469 Munich, Germany
{flabes|falk|guido}@livis.com

Abstract. Thing-oriented programming (TP) is an emerging programming model which overcomes some of the limitations of current practice in software development in general and of object-oriented programming (OOP) in particular. Ercatons provide an implementation of the model. TP does not describe a new programming language. The so-called "ercato virtual machine" or "ercato engine" implements the ideas discussed and has been used to validate the concepts described here.

Thing-oriented programming is centered around the concept of a "Thing". A thing in an executing software system is bound to behave like an object does in our real world. Software objects do not. Aspect-oriented programming (AOP) or extreme programming methods (XP) provide no exception. This fact is traced back to be the root why current software development often fails to meet our expectations. TP should then be able to provide the means to make software development achieve what other engineering disciplines have achieved a long time ago: that projects are deterministic (completed in time and effort scales sub-linearly with size).

This paper is a revised version of the original paper published in the conference proceedings of the Net.ObjectDays 2004 conference, by Springer Verlag LNCS 3263.

1 Motivation

The development of ercatons was inspired by the way we all deal with real-world objects and how recent object-oriented techniques have deviated from this ideal. An anecdote may illustrate this best.

"One morning, we noticed some workers tile our office's backyard. The day before, piles of square tiles had been delivered and the workers now seemed to make good progress covering the backyard's middle section. All of a sudden, loud noise stopped us working. What had happened? The workers had finished to cover about 90% of the surface with square tiles and had started to *cut* tiles using a stone saw. Due to corners and the irregular shape of the backyard, the produced tiles had all shapes one could possibly think of. The end of the day, the entire backyard was nicely tiled."

What if the tiles were software objects? We would have two options: First, the textbook option with *tile* as a base class, *square*, *rectangular* and *polygon-bounded tile* as its derived classes (where we already encounter and avoid the meaningless discussion if *rectangle* should be inherited from *square*...). Or second, the pragmatic option of a *generic tile* with appropriate constructors. And we would have to create objects on demand while tiling or we would have to algorithmically solve the tiling problem in order to create all required classes upfront. Still, we probably would miss cases where we need a hole in the tile etc. And still, our software objects do not model tiles – the defined classes model a *factory* which produces them.

In the real world, we use square tiles and cut to fit. In the software world, we use tile factories which must be able to produce tiles of all shapes we analyzed beforehand to be required.

This example, as trivial as it appears to be, does matter. Initially, the problem seems to go away if we look how software is really being built: We may always modify a generic tile class and recompile and test until the software is finished. Source code modification then is what corresponds to "cut-to-fit" in the real world. At a second analysis however, this is not true anymore for large enough systems. First of all, complex systems are never finished. But more fundamentally, complex systems must be layered and modular with lower level modules considered stable at some point. And now the problem of a lack of reusability of objects emerges even *within* a single system, if only complex enough.

The best technologies to cope with this situation are a combination of waterfall methods (specify as much as possible as early as possible), model-driven architectures (same reasoning), and powerful modules with well-defined contracts (increase chances of reusability as much as possible). However, projects must, by definition fail to scale linearly with size this way. Interestingly, real-world engineering projects do not.

Thing-oriented programming (TP) is the attempt to overcome the limitations of OOP. The philosophy behind it may be summarized by the following "manifesto".

The Ercato Manifesto of Thing-Oriented Programming:

§0 **T***he exception is the rule.*

§1 **O***ur world is* ***rich*** *and complex*
rather than well-structured and simple.

§2 **S***oftware must cover* ***irregular****, changing patterns*
rather than regular patterns.

§3 **A** *software system is an* ***organic*** *being*
rather than a set of mathematical algorithms.

§4 **S***oftware components are an* ***integral part*** *of our rich world*
rather than entities at some meta level.

§5 **S***oftware engineering evolves from small to* ***large***
rather than from concrete to abstract.

We will not go into too much detail regarding the manifesto here. While the first three points are widely accepted within the OO community, the latter three points may be what constitutes the essence of the emerging TP model.

Ercatons are a development aimed at addressing *all* of the points of the Ercato manifesto. Ercatons have been developed over the past four years and this paper aims at presenting its core ideas and to set it into a larger context, namely that of Thing-oriented programming.

2 The emerging Thing-oriented programming paradigm

We distinguish between a mainstream trend within OOP and an emerging Thing-oriented trend.

Mainstream. The mainstream trend is dominated by generative methods such as model-driven architecture (MDA), aspect-oriented programming (AOP), various wizards in IDEs and an increasing level of abstraction. It is also characterized by pattern frameworks (such as Struts for the MVC paradigm for the web or J2EE for business logic) and a strong emphasis on an architecture which is as complete as possible upfront.

The evolution of programming languages has long be characterized by an attempt to conceptualize real-world entities into software entities – object-oriented languages being the current end point. Modeling languages such as UML2 are no exception as they emphasize visualization without escaping the limitations of an object-oriented language. Component-based software is a facet. Aspect-oriented programming deviates a bit as it depends on formal aspects such as grammar. Both are pushing those limitations a bit. Altogether, we found it disappointing how different software objects still are from real-world entities after a quarter century of research. They are poor when it comes to representing them while *using* or *growing* a system, as opposed to representing them while modeling.

The mainstream seems to disagree, given the movement for model-driven architectures (MDA) or executable models; or the generation of executable systems from models. The promise is: Once you have the model, you are done. And if objects are fine to model a system, where is the problem? The problem simply is that it may be impossible or too hard to ever create this model!

We believe that it may be possible and worthwhile to create a model for a simpler, special case such as a given algorithm, or some important processes, or a less accurate one for better overview; but not in general for an entire problem, not within time and budget, not without mistakes. We are convinced that being forced to model every detail of a system is against the general engineering principle of keeping things as simple as could possibly work.

Thing-orientation. The trend towards Thing-oriented programming may first have emerged in 1979 with "ThingLab" [1]. The publication of the programming language "Self" [2] in 1987 and of prototype-based languages in general were important milestones. The language "NewtonScript" [3] was inspired by Self and led itself to "JavaScript" [4]. Both are prototype-based. More recently, XML-based language "Water" [5] and Java-based system "NakedObjects" [6] emerged. Water is another prototype-based language linking every XML-tag with an object and calls its top-level ancestor "Thing" rather than "Object". NakedObjects is not a language (it uses Java) but drops the MVC pattern in favor of the idea that every object should expose an intrinsic user interface (i.e., that every object should be usable by itself). We notice growing interest in plugin architectures as well, as recently demonstrated by the popularity of Eclipse [7]. A plugin has some characteristics of a Thing which an object does not.

Orthogonally, new engineering methods emerged, with Extreme Programming [8] and the Agile Manifesto [9] being examples. Those methods contradict the mainstream trend towards even more abstract and complex models or architectures as well as the waterfall method. All of these developments address some points of the Ercato manifesto. Ercatons are meant to address *all* aspects of the Ercato manifesto and to be fully Thing-oriented. First information about it was released in 2003 [10].

As we are not aware of a publication bringing all of the above developments into the single context of a trend towards Thing-orientation (as opposed to model-driven), we would like to start with our definition of the notion of a Thing.

3 Definition of Thing-oriented Programming

The basic idea behind a "Thing" is to be able to represent a real-world entity without the absolute requirement to model it. If we do not model, we still describe, visualize, compare etc. Document-like properties of a Thing acknowledge this. During usage of a system, while we learn more about various aspects, Things are designated to morph and formalize aspects to reflect the increase in knowledge. Also, models can still be expressed by projection of knowledge "to the essentials" where what is essential and what is not may be a function of time.

Definition: A "Thing" is a software entity within a production software system with the following properties:

1. **Uniqueness** – It is unique and no two Things can be *exactly* equal:
   It has exactly one unique "name" and two Things are always distinguishable.
2. **Structure** – It has *inner structure* which is both state *and* behavior:
   Its state is a data tree or structure of equivalent complexity, not restricted to slots.
   Its behavior is defined by reaction to received messages, or events such as elapse of time.
3. **Object** – Its inner structure is inherited, polymorphic, delegated and encapsulated:
   Inheritance: a Thing may inherit (or clone) inner structure from another Thing.
   Polymorphism: two Things may behave qualitatively different for the same message.
   Delegation: a Thing's inner structure may refer to other Things.
   Encapsulation: a Thing defines how much of its inner structure is exposed to another Thing.
4. **Document** – It is human-readable:
   There is an equivalent externalizable and human-readable form accessible by name.
   It has one owner controlling its visibility and encapsulation.
   It determines its life-cycle, incl. infinite life (persistence).
   It has a (non-perfect) memory of previous inner structures it had over time (versioning).
5. **Morph** – Its behavior, state, owner, inheritance relation etc. may change during lifetime.
6. **Projection** – It defines its interactions with components of a production software system:
   It has one or more representations when manipulated by algorithms (programming interface).
   It has one or more representations when manipulated by humans (user interface).
   It determines its formal properties such as search index entries it may contain.
7. **Deterministic** – It will only temporarily maintain inconsistent inner structure (transaction safety).

Definition: A "production software system" is a real-world system comprised of hardware, software and humans currently solving or able to solve the problem the software is being crafted for. Such systems may be distributed or not. A program source editor or IDE is no such system.

To support Things, such systems will most likely be composed of a grid of virtual machines executing one or several flavors of a Thing execution environment.

The semantics of a Thing may and should be independent of a production software system.

Definition: "Thing-oriented programming" is the art of creating software composed of Things.

Because this activity resembles growing and building rather than modeling and programming, we sometimes call Thing-oriented programming "to *build* rather than *model*".

Note that some of the properties of a Thing are common properties for object instances, other for documents. The following simplifying statement shall summarize the above points for the rest of this paper:

"A Thing is unification and super-set of an object instance and a document."

A remark is in order: It is of course true that a document could be created or an object could be instantiated that has all claimed properties, e.g., a generic object instantiated from a Java class. However, a second such instance inheriting from the first would not be inherited in the Java sense. This means that Java as such lacks some of the required properties and could be used to *implement* an environment for Things only. To make this clear:

"Objects are not Things. Documents are not Things."

Untyped object-oriented languages such as Smalltalk are closer to Things than strongly typed languages such as Eiffel or Java. Prototype-based languages such as Self are even closer because they do not depend on classes and are able to change their behavior during lifetime. Many properties we deem important for Thing-oriented programming are missing from prototype-based languages and therefore, we do not classify TP as prototype-based.

The discussion so far left open the question why we call the approach "Thing-oriented". Both, objects and documents are normally combined to represent real-world entities. But there has been no unifying concept. Therefore, the fact that Things must be a superset of objects and documents immediately follows from the initial question:

"What is the software entity which most closely represents a given real-world entity?"

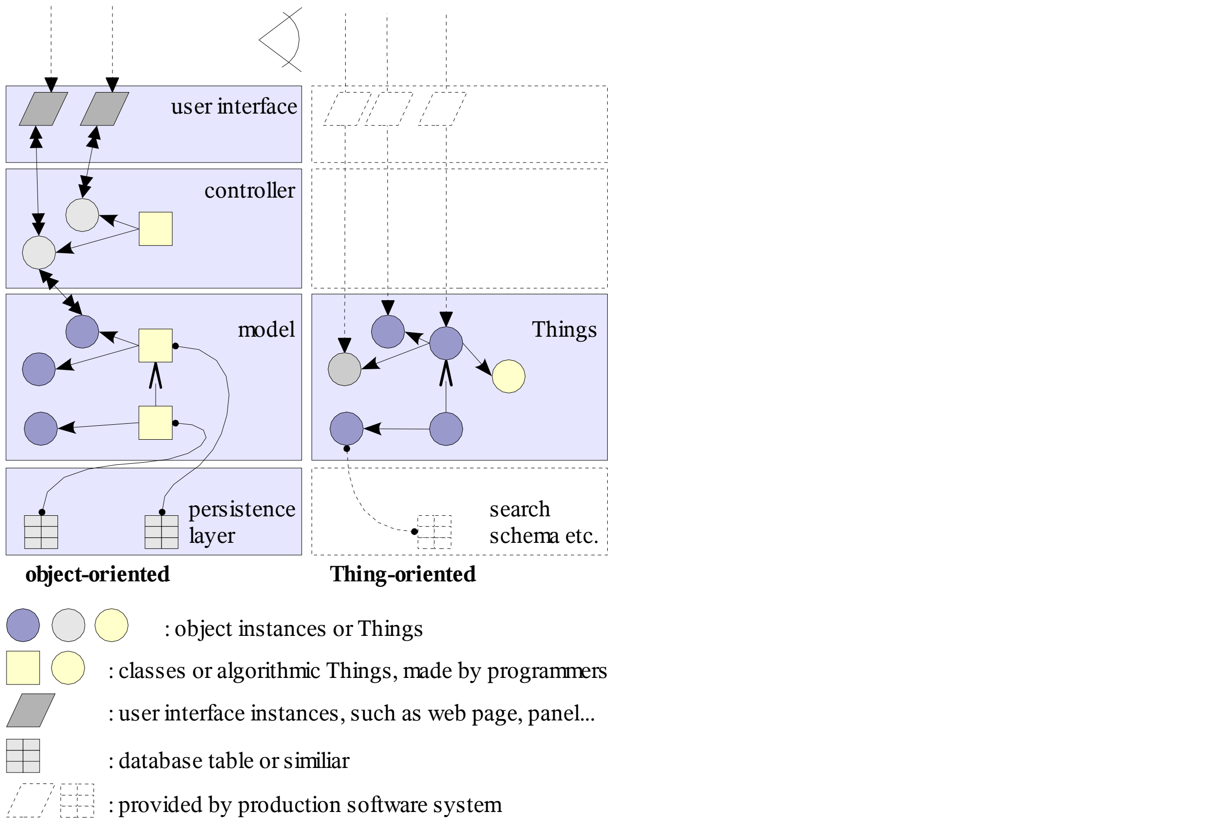

user interface
controller
model
Things
persistence layer
search schema etc.
object-oriented
Thing-oriented
: object instances or Things
: classes or algorithmic Things, made by programmers
: user interface instances, such as web page, panel...
: database table or similiar
: provided by production software system

Fig. 1 A Thing defines all relevant aspects within a software system, i.e., every single instance does. There is no intrinsic difference between a user's and a programmer's view.

In Fig. 1 we conclude this section with an overview about how a Thing-oriented software system differs from a traditional object-oriented one.

4 Definition of Ercatons

Ercatons are specified to be Things and to address all of the points of the Ercato manifesto, hence the name.

Ercatons separate inner structure from algorithm. Inner structure is expressed in one language (such as XML) with algorithms being expressed in another (such as Java or XSLT). This separation could easily be removed by creation of yet another language. The most elegant way to do so would be by treating code-closures [11] as ercatons. However, this would not be in line with both clauses of §3 of the Ercato Manifesto. All typical OOP features (such as polymorphism, inheritance, encapsulation, method signatures etc.) are features of an ercaton independent of algorithms. Actually, any piece of code written in a language which has XML syntax is an ercaton and a closure and removes the separation. It is just that currently, we do not recommend this style.

The specification of an ercaton is such that the difference between itself and the real-world entity described by it are reduced to an absolute minimum. As a consequence, any ercaton describing a *virtual* real-world entity (such as a bank account) ***is*** the entity. Algorithms are ercatons in turn, because code implementing an algorithm ***is*** a virtual real-world entity.

Definition: An "ercaton" is a Thing (as previously defined) with following additional properties and specifications:

1. **Name** – an ercaton has a structured local name:

   The local name is a string of the form "~owner/path[,version]" and follows Unix shell conventions.

2. **Syntax** – its equivalent externalizable and human-readable form is XML:

   Every XML document (if it includes a valid name) is an ercaton.

   It uses ercato markup (XML elements and attributes in given separate namespaces) to alter its semantics. This markup is not bound to an XML schema and will coexist with most XML applications. The ercato markup may be considered the syntax of an ercaton.

   Syntax exists to express all of the semantic properties of a Thing.

3. **Flavors** – ercatons come in six flavors or more:

plain	an ordinary ercaton.
role	a user, owner, or rôle; **user** is a sub-flavor. Users must be authenticated.
prototype	has reduced semantic power, e.g., serving as a template to be cloned.
resource	contains binary data such as a movie or a code archive (the XML form of a resource ercaton is, as an exception not equivalent to it).
index	a formal data schema or other formal information; ercatons are able to index part of their state in relational databases; persistence of ercatons must not depend on formal information.
version	a history of versions of an ercaton.

   Ercatons may change flavor (morph), but cannot be of more than one flavor at a time.

4. **Commutativity** – federations of ercatons form unordered sets:

   The state of an *ercato engine* is fully defined by all ercatons, independent of the order they have been manipulated in.

   This implies that an ercaton action cannot store data outside of ercatons, i.e., in a database.

5. **Algebra** – ercatons may be added (+) and subtracted (-):

Inheritance is supported where syntax is as general as XML (the infoset tree). The algebra is defined such that inheritance corresponds to addition. Let *a* and *b* be ercatons, then it holds true:

$$a = a + b \quad \Leftrightarrow \quad a \text{ inherits from } b$$

Subtraction is defined to be the inverse operation, $(a - b) + b = a$, and $a - a = 0$.

6. **Behavior** – ercatons contain actions, triggers, targets and objects:

action	specifies behavior upon receipt of a message; consumes and produces ercatons; actions are protected by sets of rôle-based permissions; an ercato engine must be able to discover and export actions as WebServices.
trigger	specifies behavior upon an event; events include elapse of time, change of state, asynchronous events such as receipt of email.
target	pipeline specifying projection onto a named user interface; an ercato engine must support at least a web-based interface.
object	specifies projection onto named API for an algorithm; an ercato engine must support at least a Java-based object model.

Behavior may be specified by a closure (in some XML language such as XSLT), delegated to an action of another ercaton, or implemented by an algorithm. An algorithm may be a resource ercaton containing a Java jar-file and is then identified as ercaton/class/method().

7. **Permission** – ercatons encapsulate their inner structure:

State and actions are guarded by the following rôle-based permissions

r	**r**eadable state.
w	**w**ritable – ercaton can be morphed or deleted as well.
b	**b**rowsable; state which isn't browsable for a rôle cannot be retrieved indirectly, i.e., by a database query performed by that rôle.
x	action is e**x**ecutable.
s	**s**ubstitute rôle by owner of action before executing it (aka s-bit); permissions are carried along action delegations forming a capability chain.
t	action is executable and may modify **t**his ercaton (secondary s-bit).

Permissions provide both the rôle-based business logic and the package/module-based OO encapsulation (private, package-private, public, friend, etc.) – using package names as rôles.

8. **Distribution** – ercatons may be distributed:

Access or invocation of actions is not bound to one address space and ercatons may freely migrate (using a global name). The transmission protocol is HTTP or SOAP [12] using the externalized form. The transmission of authentication depends on the trust between two ercato engines.

The ercato engine employs an optimistic locking strategy for concurrent operations. It may optionally use the XOperator (c.f. below) to avoid detection of false collisions.

Definition: An "ercato engine" is a production software system supporting ercatons as Things.

Definition: The "ercatoJ engine" is the ercato engine we have implemented on top of J2EE [13]. Note however, that ercatons are language-independent. The ercatoJ engine exists and is in mission-critical production use, e.g., at Henkel KGaA [14].

All of the above 8 points are to be discussed in more detail elsewhere. This paper, however, aims at providing an overview and introducing into the general idea. We will therefore now highlight and give examples for the aspects we deem to be particularly interesting.

5 Examples

Some examples may clarify things and introduce into the syntax, the ercato markup.

Example 1. A hello ercaton in its XML form (in following examples, we will leave out the <?xml> processing instruction and the erc: namespace declaration).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<hello xmlns:erc="http://ercato.com/xmlns/ErcatoCore">
  <erc:id>~sample/hello</erc:id>       <!-- the name or ercato-id    -->
  <p>Hello, world!</p>                 <!-- the ercaton's only state -->
</hello>
```

This example is a valid hello-world example because it will produce the right string in the executing system. To see it, the expression `~sample/hello` must be viewed.

Example 2. An improved hello ercaton. It returns a variable string.

```
<hello>
  <erc:id>~sample/hello2</erc:id>
  <erc:action> /bin/echo (text=xp{concat('Hello, ',$name,'!')}) </erc:action>
</hello>
```

This example uses another ercaton, `/bin/echo`, to delegate the implementation. Beforehand, the argument within the `xp{...}` clause is evaluated as XPath [15] expression. To see the result, the expression `~sample/hello2(name="world")` must be viewed.

Example 3. A counter ercaton which updates its state.

```
<counter xmlns:erc="..." xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <erc:id>~sample/count</erc:id>
  <count>0</count>                          <!-- state -->
  <erc:action name="main">
    <erc:arg name="amount">1</erc:arg>        <!-- default value -->
    <erc:native lang="Xslt">                <!-- a closure     -->
      <xsl:template match="count">
        <count><xsl:value-of select=". + $amount"/></count>
      </xsl:template>
    </erc:native >
  </erc:action>
</counter>
```

This simple ercaton contains the entire logic required to maintain a counter to be incremented, stored, viewed and used (in a transaction-safe way). This unique counter is identified as `~sample/count`. This example uses a closure in order to be self-contained (not referring to any other ercaton). However, real usage would more be like

Example 4. Counter ercaton rewritten (after incrementation is refactored).

```
<counter>
  <erc:id>~sample/count</erc:id>
  <count>0</count>
  <erc:action name="main"> /bin/increment (xpath="//count") </erc:action>
</counter>
```

Its owner (or another ercaton) may do the rewriting during production. There is no big difference between the rewrite and an invocation of an action like `main`. The `amount` argument is implicitely passed down the delegation chain.

Due to the properties of an ercato engine, ercatons may be displayed in a browser window with one URL per ercaton-id. In the ercatoJ engine, the counter `~sample/count` looks as follows:

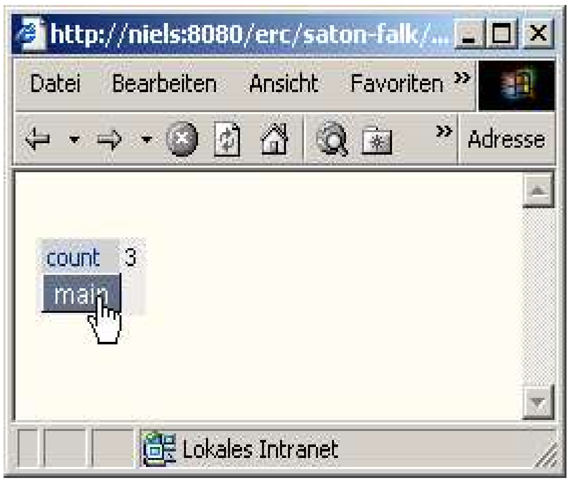

http://niels:8080/erc/saton-falk/...
Datei Bearbeiten Ansicht Favoriten »
Adresse
count 3
main
Lokales Intranet

Fig. 1. The count ercaton of Example 3/4 in its default web browser look&feel (after three clicks onto 'main').

A click onto the "main"-button would increment the count to 4, equivalent to expressions `~sample/count!main (amount=1)` or just `~sample/count()`. *Every* ercaton has, by definition, a "look&feel" as is seen here. In a different context, an ercaton may be displayed by a panel window within a GUI, or in a console window.

We will now discuss some selected features in more detail.

6 The programming model

Rather than featuring a programming language, ercatons are programming language-independent. It is therefore necessary to show how ercato programs are written. It is obvious that the ercato programming model is centered around the idea of creation, modification and use of ercatons. An ercaton may be used by inspecting its state, executing its actions, sending it to the user as part of the user interface, or sending it to another system.

Because ercatons may contain triggers which fire upon elapse of time, ercatons may also show autonomous behavior (or life). This is useful to implement agent-based patterns.

Ercatons are meant to directly implement entities of the real world also known as the problem domain or business logic. Most ercatons do. Their names are like `~flight/booking/lh6361/ma34`.

Other ercatons provide utility services. Their names are like `/bin/cp`. We call the former *business ercatons* and the latter *service ercatons*.

When it comes to implementation of actions, we distinguish between three typical cases:

- Structural, administrative tasks such as editing state etc.: Delegated to service ercatons.

  Note that delegation to an ercaton such as `/bin/increment` does only look like a shell script call. Upon invocation, the delegation chain will actually be followed and its end will be invoked. Final invocation is also cheap as it normally takes place within the local address space.

  Typical programming language primitives (instance creation, copy, a full edit cycle, print, change of part of its state, queries, etc.) are all provided by service ercatons.

- Preparation of state for a user interface or exchange: Implemented by service ercatons which are XSLT stylesheets.
- Algorithmically non-trivial tasks: Implemented in an OO language (Java) after projecting the state of involved ercatons onto appropriate object instances.

  There is a language binding which maps ercatons to Java objects (incl. their inheritance relation) together with an API to expose more features. Using this language binding, any framework to bind Java objects to XML and back may be used. The Ercato API for Java includes a lightweight framework for this task.

  Within the ercatoJ engine, invocation of an action implemented in Java incurs minimal overhead only.

Example 5. The counter ercaton is again rewritten, now with a Java implementation.

```
<counter>
  <erc:id>~sample/count</erc:id>
  <count>0</count>
  <erc:object lang="Java">
    <erc:archive> ~sample/lib.jar </erc:archive>
    <erc:class> sample.Counter </erc:class>
  </erc:object>
  <erc:action name="main">
    <erc:arg name="amount">1</erc:arg>
    <erc:native lang="Java">
      <erc:method> increment </erc:method>
      <erc:parameter name="amount" type="int"/>
    </erc:native>
  </erc:action>
</counter>
```

with (complete and working) Java code as follows:

```
package sample;
import  com.ercato.core.*;
import  org.w3c.dom.Text;

public class Counter extends ErcatonObject implements Action {
   public void increment (int amount) {
      count += amount;
      if (amount != 0) touch ();
   }
   protected void evaluateElement (EvaluationContext ec, String tag, String ns){
      if (!"count".equals (tag)) return;
      counter = ec.getTextNode (false);
      count   = Integer.parseInt (counter.getData ());
   }
   protected void approve () {
      counter.setData (String.valueOf (count));
   }
   private Text counter;
   private int  count;
}
```

Support for other languages may be added later provided the language supports a sandbox security model. The permission model is such that a user must not gain additional privileges by creating an ercaton with a malicious action written in any language. The implementation language of actions is hidden and actions may invoke each other even when implemented in different languages.

It is important to observe that the `Counter` Java-class is reusable within *any* ercaton which contains a count-tag with numeric content. This is a general observation and the following relationship is deduced:

Construct:	Thing-oriented language:	Object-oriented language:
Signature for algorithms	**OO-class**	OO-interface
Signature for state	Missing or XML-schema	**OO-class**

Ironically, this shift of usage of OO-classes in a Thing-oriented system makes them small and reusable.

7 Inheritance

Inheritance may be the most exciting single aspect of ercatons. It is supported where syntax is specified to be as general as XML (the infoset tree). The general idea is to provide an algebra for infoset trees defined such that inheritance corresponds to addition. Let a and b be infoset trees (ercatons except resource ercatons), then it holds true:

$$a \equiv a + b \quad \Leftrightarrow \quad a \text{ inherits from } b \qquad \textbf{(Eq. 1)}$$

Subtraction is defined to be the inverse operation, and the following equations all hold true:

$$a \equiv a + a \qquad \textbf{(Eq. 2)}$$

$$a - a \equiv 0 \qquad \textbf{(Eq. 3)}$$

$$(a - b) + b \equiv a \qquad \textbf{(Eq. 4)}$$

$$\exists\ a,b\text{: } a + b \neq b + a \qquad \textbf{(Eq. 5)}$$

This way, object-oriented inheritance turns out to be just a special case of a more powerful mathematical operator which we call the "XOperator" [16]. Morever, ercatons can continue to be standalone entities unrelated to each other and be unrestricted by a type system or type-safety.

An ercato engine must not make a difference between two ercatons *a* and *a'* where the left clause of Eq. 1 holds true for *a'* and *b*, and where *a* declares *"to be clone of" b*. In the latter case, *b* is called *"clonebase"* of *a*. An ercato engine is required to maintain Eq. 1 to hold true over changes of a clonebase, i.e., clone and clonebase must be kept synchronized.

This model provides full support for situations where no instance is like any other, but does support several classical OO techniques for more regular situations too. These techniques include:

- An object inheriting from an ancestor.
- An object instantiated from a class.
- Two objects sharing a "common part".
- Instance data filled into a template, or overriding default data.

To illustrate the above, we will evolve the counter ercaton example.

Example 6. An ercaton inheriting from another. The `~sample/count` ercaton was defined in Example 3.

```
<counter>
  <erc:clone>~sample/count</erc:clone>
  <erc:id>~sample/count2</erc:id>
</counter>
```

This example defines an ercaton `~sample/count2` which is kept *equal* to `~sample/count` except, of course for its ercato-id which must be unique. Ercaton `~sample/count` is now clonebase of `~sample/count2` and every change to the clonebase is a change to `~sample/count2` as well. Both ercatons being equal means that accessing their XML form yields equal results. This implies that the XML form of `~sample/count2` must differ from the listing in Example 6 after creation when inspected. At most one clonebase is allowed. However, an arbitrary number of "bases" is allowed. A base is like a clonebase except that changes after creation are not tracked.

Example 6 lists a clone which is almost empty. In the general case however, an ercaton will have rich inner structure and may be declared clone of another ercaton which has rich inner structure too. In this case, the XOperator-addition of infosets of both ercatons results in the following rules:

1. Clone ***a*** and clonebase ***b*** are compared.
2. What in ***a*** is missing is inherited from ***b***.
3. What in ***a*** is an extension of inner structure in ***b*** extends that inner structure.
4. What in ***a*** is additional is an extension of ***b*** altogether.

Annotations and additional rules refine the rules. We found that the XOperator is roughly as powerful as XSLT transformations. Lets present a simple example of non-empty inheritance:

Example 7. A non-empty clone. The `~sample/count2` ercaton was defined in Example 6.

```
<counter>
  <erc:clone>~sample/count2</erc:clone>
  <erc:id>~sample/count3</erc:id>
  <erc:action name="main">
    <erc:arg name="amount">2</erc:arg>
  </erc:action>
</counter>
```

Because `~sample/count2` is a clone of `~sample/count`, this definition is equal to:

```
<counter xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <count>0</count>
  <erc:id>~sample/count3</erc:id>
  <erc:action name="main">
    <erc:arg name="amount">2</erc:arg>
    <erc:native lang="Xslt">
      ... detail from Example 3 ommitted here ...
    </erc:native>
  </erc:action>
</counter>
```

Note that the default value for the action argument is overridden only. It would have been equivalent to first create the clone from the empty declaration as in Example 6 and then to change the default value in the resulting ercaton; or to first copy and change, then to declare it clone of another ercaton. It is therefore not necessary to model the inheritance relationship as it is discovered automatically. In doing so, the XOperator-subtraction is used to invert inheritance. An ercaton of prototype flavor may be used if its sole purpose is to act as a clonebase. This corresponds to an abstract class in a class-based OO system.

The above example of inheritance has been very object-oriented, just overriding (part of) an action. A very Thing-oriented example is one where two ercatons containing state expressed as scalable vector graphics (SVG) do inherit from each other.

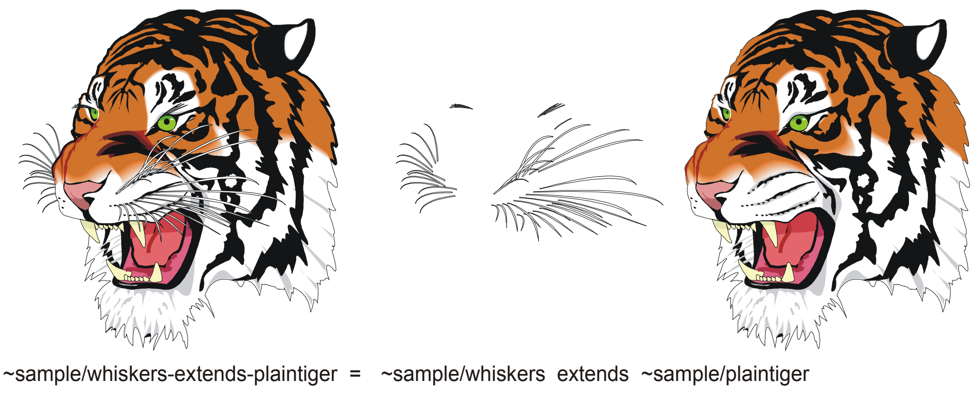

~sample/whiskers-extends-plaintiger = ~sample/whiskers extends ~sample/plaintiger

Fig. 2. A “whiskers” ercaton inheriting from a “shaven tiger”.

Ercaton `~sample/whiskers-extends-plaintiger` differs from ercaton `~sample/whiskers` by a single `<erc:clone>`-tag only.

8 Indexing

A long-lasting problem of object-oriented programming has been the tedious mapping of objects to relational databases and the insuffience of object databases. Thing-oriented programming should have the same problem. However, Thing-oriented programming allows *every* Thing to be persistent. Therefore, the problem is reduced to the use of databases for structured queries across relations.

"Persistence and Structured Queries are orthogonal and shall be implemented independently."

The internet is a good example that virtual real-world entities already obey this rule (a network of websites for persistence, "Google" for queries). However, we are not aware of a publication about this as a deeper insight. In order to serve both ends, the ercato programming model includes a powerful mechanism to fully get rid of the "persistence problem". It is composed of three cornerstones:

1. Index ercatons.
2. Index attributes in arbitrary ercatons.
3. Query ercatons for structured queries across relations and API for supported OO-languages.

A small example illustrates this:

Example 8. A index ercaton defining two indices where one points to ercatons.

```
<goods>
  <erc:type>index</erc:type>
  <erc:id>~sample/goods</erc:id>
  <erc:index>
    <erc:name>name</erc:name>
  </erc:index>
  <erc:index>
    <erc:name>price</erc:name>
    <erc:index-type>idref</erc:index-type>
  </erc:index>
</goods>
```

The first index is for a name, the second for a reference to an ercaton holding a price. Index ercatons may also contain references to other index ercatons themselves, from one index to another. This is the equivalent of SQL join statements and inner as well as outer joins can be modeled.

Example 9. An arbitrary ercaton indexing name and price for later retrieval.

```
<article>
  <erc:id>~sample/learjet</erc:id>
  <name erc:index="~sample/goods">Learjet</name>
  <erc:idref erc:index="~sample/goods#price">~prices/lj</erc:idref>
</article>
```

Arbitrary XML nodes may be indexed this way. Normally, such an index annotation is not marked explicitly but is inheriteded. Index operation does not need any XML schema to work. If an XML schema does exist for a given ercaton, it may of course be used to manage its index annotations automatically. However, we do not recommend to rely on a schema in a programming task.

Example 10. Retrieving all articles more expensive than 1799 €/$ from the `~sample/items` index ercaton which acts as a view joining `goods` and `prices`.

```
/bin/simplequery (index="~sample/items" name="price" value=">1799")
```

The programmer is not exposed to SQL, EjbQL, XQL, and the like. The result is a list ercaton holding a list of ercato-ids satisfying the constraint and additional indexed information. Information about ercatons which the user has no browse capability for will not be found. Support exists to manage hierarchical catalogs and to index cumulative data for commutative operations (such as the sum of values).

When index annotations or indexed data within an ercaton is inherited from a clonebase which is being changed, index information does change accordingly and the result of queries will change as well. This change is allowed to happen asynchronously with implementation-defined worst-case delay.

The ercato engine specification can be updated to include support for forthcoming dataspace structuring techniques such as the semantic web in such a way that existing ercatons can profit.

9 Kernel vs. user space

The ercato engine may be regarded an operating system managing ercatons as a resource. If this point of view is used, the ercato engine represents the kernel space of the operating system and everything said so far applies to the kernel.

There is a constant battle to minimize kernel functionality in favor of user space and the ercato programming model is no exception. Therefore and not unlike Unix, a large set of functionality is found in standard actions within service ercatons owned by `~root`, the so-called system ercatons.

Example 11. A standard action 'cp'.

```
/bin/cp (from="~sample/count" to="~sample/count4")
```

The ercaton `/bin/cp` has a main action able to copy ercatons. And `~root` can do so for all owners.

Many actions named similar to the Unix `/bin` directory exist. `/bin/ls` may be used, for instance, to browse catalogs such like the catalog of ercato-ids. Much of the power comes from the many standard actions in system ercatons available, and their number keeps growing. As has been said before, ercatons are *not* separate executable files executing in a separate address space or script files.

An ercaton containing algorithms referred by system ercatons plays a role similar to a shared library. The resource ercaton containing most system actions' code is `/lib/erxlib.jar`. Actions in system ercatons are additionally aware of annotations in the erx-namespace (ErcatoExtensions). This paper cannot describe the erx-layer here due to space limitations. A last figure may visualize how the components fit together.

In the case of the ercatoJ engine (the rightmost pillar), every ercaton is represented by an Enterprise JavaBean (EJB) entity bean instance during runtime. Note that deployment of business logic is independent of EJB deployment, though.

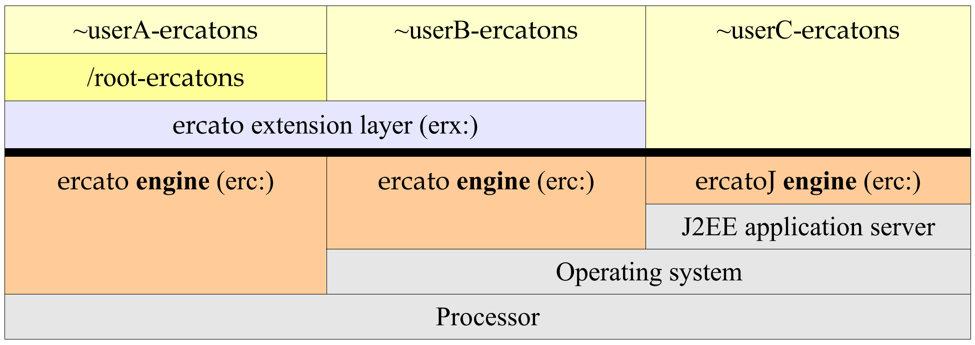


Fig. 3. Different components of an ercato programming environment. The three engine-pillars represent three options to implement the engine. The ercatoJ engine uses the right-most option. The lower half represents the kernel as described in this paper while the upper half represents the user space.

All ercatons depend on the kernel while many ercatons additionally depend on system ercatons (the erx-layer) to facilitate implementation.

An address ercaton being part of an address management application may serve as an example how both, the erc- and erx-layers interact.

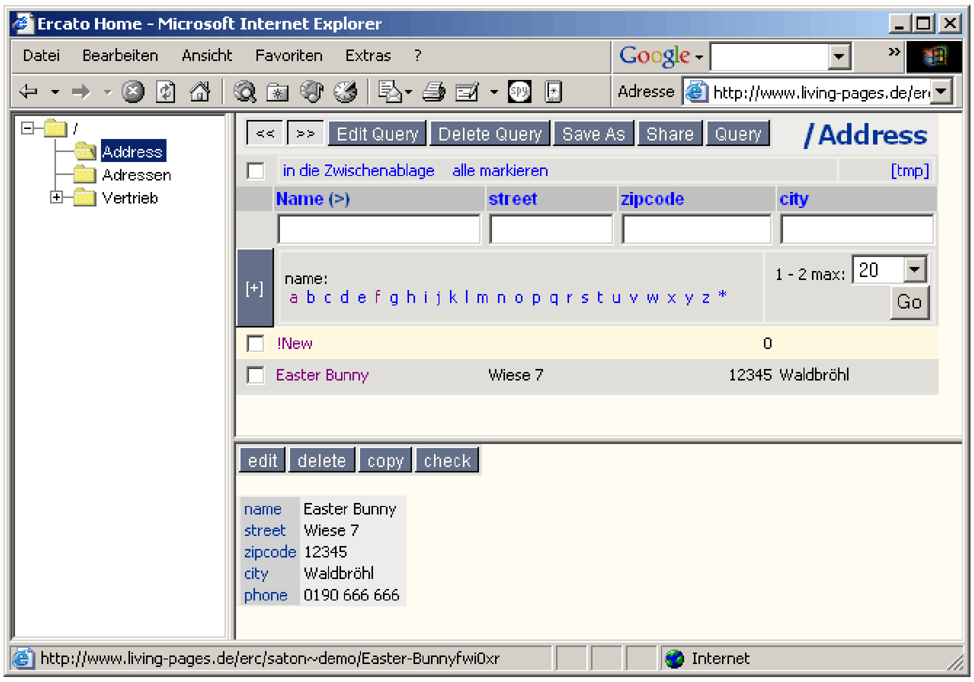
Example 12. The address of Easter Bunny.

```
<address>
   <erc:id>~sample/adr/bunny</erc:id>
   <erc:clone>~sample/adr/base</erc:clone>
   <name>Easter Bunny</name>
   <street>Wiese 7</street>
   <zipcode>12345</zipcode>
   <phone>0190 666 666</phone>
</address>
```

Assume that we want address searching to be available and offered in a navigation as well as administration operations such as edit, copy, delete and verification of address data. Additionally, we want other users to be able to find and use this address without necessarily the right to alter it. A corresponding address manager application containing Easter Bunnies address may roughly look like Fig. 4.

In order to achieve this, we specify a suitable clonebase as listed in Example 13. It contains annotation attributes of elements like `erx:field-ref="string"`. It contains an action reference to the system ercaton `/bin/edit`, too. The annotation is used by the system ercaton to provide a suitable implementation of an edit cycle and it is used by the implicit system view target ercaton to provide a suitable view of the ercaton.

This interaction between ercatons and annotation is a typical pattern within the erx extension layer. The ercato engine requires no notion about view or edit or any Thing.


Ercato Home - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Google
Adresse http://www.living-pages.de/er
/
Address
Adressen
Vertrieb
<< >> Edit Query Delete Query Save As Share Query
/Address
in die Zwischenablage alle markieren [tmp]
[+]
name:
a b c d e f g h i j k l m n o p q r s t u v w x y z *
1 - 2 max: 20
Go
edit delete copy check
http://www.living-pages.de/erc/saton~demo/Easter-BunnyfwiOxr
Internet

Name (>)	street	zipcode	city
!New		0	
Easter Bunny	Wiese 7	12345	Waldbröhl

name	Easter Bunny
street	Wiese 7
zipcode	12345
city	Waldbröhl
phone	0190 666 666

Fig. 4. The Easter Bunny ercaton in the lower right frame of the resulting address manager application. Navigation and search facility are provided by a navigation and a query system ercaton.

Example 13. The address ercaton used as Easter Bunnies clonebase.

```
<address xmlns:erc="..." xmlns:erx="http://ercato.com/xmlns/ErcatoExtensions">
   <erc:id>~sample/adr/base</erc:id>
   <erc:type>prototype</erc:type>
   <erc:catalog category="/Address" id-ref="~sample/catalog"/>

   <name    erx:field-ref="string" erc:index="~sample/catalog"/>
   <street  erx:field-ref="string" erc:index="~sample/catalog"/>
   <zipcode erx:field-ref="int"    erc:index="~sample/catalog"/>
   <city    erx:field-ref="string" erc:index="~sample/catalog"/>
   <phone   erx:field-ref="string"/>

   <erc:action name="edit">   /bin/edit       </erc:action>
   <erc:action name="delete"> /bin/rm!wizard  </erc:action>
   <erc:action name="copy">   /bin/cp!forEdit </erc:action>
   <erc:action name="check">  ~sample/check.xsl
      <erc:arg name="default">Waldbröhl</erc:arg>
   </erc:action>
   <erc:trigger name="on-change">!check</erc:trigger>
</address>
```

10 Patterns – or lessons learned

We have now been working with ercatons for years and we are amazed how much our thinking about software engineering has changed since. In particular, we and our partners have successfully used the J2EE-based ercatoJ engine in a number of projects.

Like a migration from procedural to object-oriented, the migration from object-oriented to Thing-oriented has led us to develop a number of patterns and best practices which are worth mentioning here.

The firewall pattern. Because an ercaton may contain both code, even source code, and business data, it may be difficult to understand how the two could be kept separate. First of all: there is no need for it. An ercato engine serves as a perfect repository for source and/or binary code. However, with possibly millions of ercatons in a system it may be reassuring to know which ercatons are structurally most important. The firewall pattern provides a way to achieve this.

A firewall is composed of empty proxy ercatons where each proxy *p* statically inherits from a base *b* and serves as clonebase for other ercatons *a*. Now *b* may change without triggering a change of *a*. Ercaton *a* will only change when the empty proxy *p* is recreated. Ercaton *a* is said to be protected by a firewall.

The pool of ercatons not protected by a firewall shall be small and mostly contain prototypes, resources, service and test ercatons. This pool is called “the software” and may be stored into CVS or replicated. Whenever the software is modified, all modified ercatons are copied into the ercato engine and tested. After all tests succeeded, the firewall proxies are copied. This way, no harm is caused to business data should the tests fail.

Formally, the firewall pattern recreates the distinction between “data” and “text”. However, an ercaton may be moved between both sides of the firewall (by inheriting from the proxy or not) without altering its semantics. Thing-oriented programming therefore provides a unification of both.

The face pattern. Ercatons and their actions always have a user interface. Ercatons specify targets to provide arbitrary user-interfaces. Customization of a target typically involves changing a stylesheet. The face pattern is used to create alternate user interfaces without the need to create or customize targets.

An alternate user interface may be provided by an alternate ercaton but same target. Such an ercaton is called an alternate “face” of the original ercaton and may be returned by an action or another target. Faces may be used to display a subset of information, to create a flow of displays (such as a wizard) or to create parametrization frontends to actions (such as forms). Faces are particularly powerful when they inherit from the ercaton.

The doctor pattern. Ercatons may modify each other. Ercatons which modify other ercatons in order to improve them in a rather general way or to remove a defect are called doctor ercatons.

A particular doctor ercaton is an ercaton which queries ercatons for cross-cutting properties and modifies their closures or action delegation chains. The most elegant way to do so for closures is by inserting a clonebase into the inheritance chain. In this particular case, the doctor pattern is known as aspect-oriented programming. However, we so far have only used this for insertion and removal of a logging aspect.

A garbage collector removing unreferenced ercatons which mark themselves as volatile is another example of the doctor pattern. Unreferenced ercatons are not removed by an ercato engine.

The workflow pattern. Ercatons gather information during lifetime. This makes them good candidates to be passed along the edges of a workflow. Such ercatons are work ercatons.

To represent the next possible states reachable in the workflow, actions represent each possible transition and permissions are set such that only legal transitions are executable at a given state. Actually, different rôles may see different legal transitions which is just fine.

Because an ercaton has a unique id while traveling along the workflow, a catalog entry which may change over time is used for navigation.

The workflow pattern is also useful when invoking a series of actions on a single ercaton.

Aggregation. An ercaton is used as a container. A car with four wheels may be *five* objects in Java but is only *one* ercaton. Multiple list elements may inherit from a single element in a clonebase.

Association. An ercaton may refer to another associated ercaton using an `<idref>`-tag. A target may eliminate the difference between aggregation and association by using an `<expand>`-attribute.

The use of such a pseudo aggregation is required for a many-to-one relation or a tighter permission for the associated part.

The structure constraint pattern. In the XML world, structure constraints are often implemented by an XML schema. In an OO language, a class serves this purpose. While both may be used with ercatons (by use of a schema or object tag), the structure constraint pattern provides an alternative.

The ercaton contains constraint markup which is used to validate an ercaton against its constraints within a trigger fired by the on-change event. The trigger may raise an exception to veto change. The trigger is synchronized with the transaction.

Besides structure and types of values, structure constraints may check referential constraints and dynamic conditions and may normalize inner structure. Structure constraints may be refined by inheritance.

Lessons learned

The ercatoJ engine is and was successfully used in a number of J2EE-based enterprise projects, by us and our partners. Most notably, it now implements a large software system at the chemical corporation Henkel KGaA, Düsseldorf, Germany. The system is used for the development and partly also production of new chemical recipes. It interacts with a farm of SAP/R3 systems and replaces a host system with about a million lines of code. The system went productive with great success [14]. Fig. 5 shows a screen shot for a part of the application, released for publication.

The users of this application run demanding operations involving recursive retrieval of ingredient information, updating lists or doing complex searches. The system, on average, serves about thousand transactions per minute & processor and has good reserve for peak loads. Our analysis shows that certain optimizations in current DOM tree implementations could yield at least one order of magnitude overall performance improvement. Performance on complex retrieval operations already is on equal with a native SQL implementation.

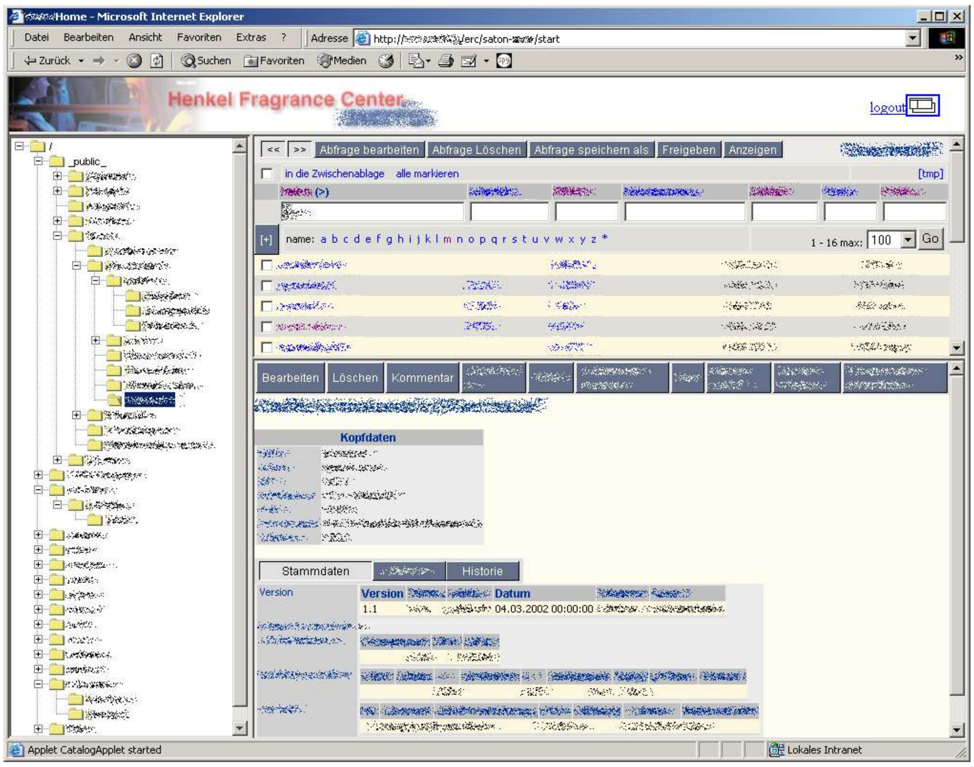


Fig. 5. Ercato-based application in production at Henkel. Five ercatons are seen (i.e., their user interfaces as specified by their declared target pipelines).

The left hand frame contains a catalog ercaton. The lower right frame shows a recipe ercaton with some action buttons. The upper right frame shows a query ercaton with result for its current query.

A general observation we have made is that development really focusses on the business logic. Despite the fact that a special ercato development environment does not exist, we observe an increase in developer productivity by a factor of three or better (compared to J2EE/CMP plus Struts). Altogether, much less programming language source code is required.

However, we have noticed as well that a good style guide or a best practice example is essential for success in a larger team. There are simply too many ways to work with ercatons and a common sense about best practice is yet to emerge.

11 Conclusion

Things in general or ercatons in particular provide a concrete way to bridge the gap between our real world and programs. Transforming a real-world entity into a software entity can be a non-mathematical task and “authoring”, “building” or “growing” rather than “programming” or “modeling” would be the right wording. Still, ercatons provide enough expressive power to express knowledge about similarities or inheritance relations, behavior, structural constraints etc. Of course, the formulation of algorithms remains a mathematical task.

We found that this copies traditional engineering methods and is able to dramatically reduce the size of software projects where traditional object-oriented systems may yield too complex solutions.

The current implementation certainly falls short with respect to language elegance, performance and tool (IDE) support when compared with an object-oriented language like Java, Smalltalk or Self. The concept does not. Especially when comparing the approach to EJB- or .NET-based alternatives, or an XML-based persistence approach [17], then ercatons have some very interesting features.

The current mainstream heads for more and more abstract language technologies, combined with graphical tools to hide them from ordinary programmers. We believe this trend will reverse.

Think Thing.

12 References

1. Alan Borning: "*ThingLab – A Constraint-Oriented Simulation Laborator*". XEROX PARC report SSL-79-3, July 1979.
2. David Ungar and Randall B. Smith: "*Self: The Power of Simplicity*". OOPSLA '87 Conference Proceedings, pp. 227-241, Orlando, FL, October, 1987. http://research.sun.com/research/self/
3. Walter Smith: "*SELF and the Origins of NewtonScript*". PIE Developers magazine, July 1994. http://wsmith.best.vwh.net/Self-intro.html
4. Netscape: "*Core JavaScript Reference*". http://devedge.netscape.com/library/manuals/2000/javascript/1.5/reference/
5. Mike Plusch: "*Water: Simplified Web Services and XML Programming*". (2002) John Wiley & Sons. ISBN: 0764525360. http://www.waterlang.org/
6. Richard Pawson and Robert Matthews: "*Naked Objects*". (2002) John Wiley & Sons, Ltd. ISBN: 0470844205. http://www.nakedobjects.org/book.html
7. Azad Bolour: "*Notes on the Eclipse Plug-in Architecture*". http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html
8. Kent Beck: "*Extreme Programming Explained: Embrace Change*". (1999) Addison-Wesley ISBN: 0201616416.
9. "*The Agile Manifesto*". http://agilemanifesto.org/
10. Jürgen Diercks, Falk Langhammer: "Bauen statt modellieren". iX-Magazin 2/ 2004, p. 100-103. Heise Verlag. http://www.heise.de/kiosk/archiv/ix/2004/2/100
11. Gerald J. Sussman and Guy L. Steele, Jr. "*Scheme: An Interpreter for Extended Lambda Calculus*". MIT AI Lab. AI Lab Memo AIM-349. December 1975. ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-349.pdf
12. W3C: "Simple Object Access Protocol (SOAP) 1.1". W3C Note 08 May 2000. http://www.w3.org/TR/SOAP/
13. Bill Shannon: "*Java 2 Platform Enterprise Edition Specification, v1.4*". (2003) Sun microsystems. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf
14. Joachim Buth and Falk Langhammer: "*Ercatons – XML-based J2EE project at Henkel*". iX-Konferenz 2003, Heidelberg, Germany. Proceedings http://www.heise.de/newsticker/meldung/43691
15. W3C: "*XML Path Language (Xpath) Version 1.0*". Recommendation 16 November 1999. http://www.w3.org/TR/xpath
16. XOperator evaluation kit. http://www.living-pages.de/de/projects/xop/
17. Tamino XML server home page. http://www.softwareag.com/tamino/