Ercatons and Organic Programming: Say Good-Bye to Planned Economy

Oliver Imbusch, Falk Langhammer and Guido von Walter

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

ABSTRACT

Organic programming (OP) is our proposed and already 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. In some respects, OP is less than a (new) programming language, in others, it is more. An "ercato machine" implements the ideas discussed and has been used to validate the concepts described here.

Organic programming is centered around the concept of a true "Thing". A thing in an executing software system is bound to behave the way an autonomous object does in our real world, or like a cell does in an organism. Software objects do not. Therefore, traditional software systems must be planned ahead like in a centrally planned economy while with OP, software systems grow. This fact is traced back to be the root why current software development often fails to meet our expectations when it comes to large-scale projects. OP should then be able to provide the means to make software development achieve what other engineering disciplines have achieved a long time ago: that project effort scales sub-linearly with size.

With OP we introduce a new term because we hope that the approach we are pursuing is radical enough to justify this.

Keywords

1. Motivation

We are going to present an alternative way to develop software. We followed this idea starting from some general observations all the way down to a reference implementation which was then used in projects of sufficient size. This makes us trust into the initial idea to the point that we may now be obsessed by it. Let us start by sharing some of the general observations.

We start with this trivial anecdote.

"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 with object-oriented programming (OOP)? We may see 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. We may then create appropriate objects on demand while tiling. Still, we probably would miss cases where we need a hole in the tile etc. And still, we would define classes which model a *factory* which produces tiles, not tiles themselves (this may even hold true with most prototype-based languages, cf. below).

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

Dagstuhl Seminar Proceedings 06031
Organic Computing - Controlled Emergence
http://drops.dagstuhl.de/opus/volltexte/2006/574

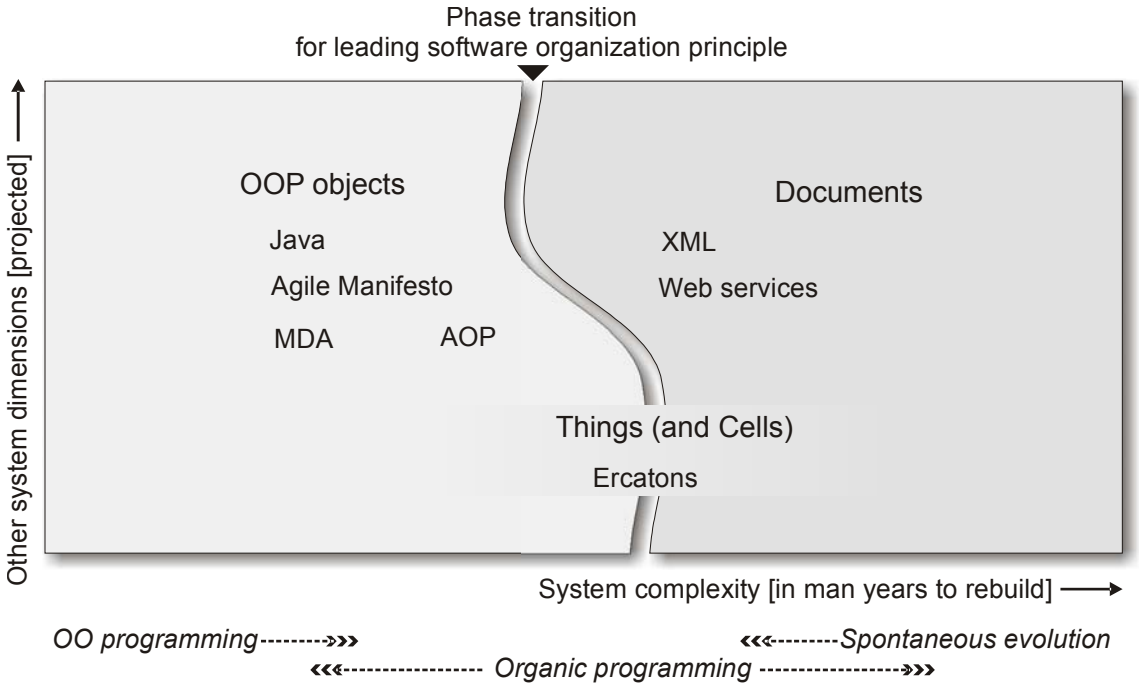

Figure 1. Beyond a given critical degree of complexity, a software system cannot be redeployed or rebuilt anymore. However, it can still be grown and evolve. This is like a single building which can be rebuilt where the city (normally) can't.

Organic computing tries to merge the successful paradigms of the two regimes (objects and documents) into a new programming model which is more appropriate for large-scale projects than traditional OOP methods.

produce tiles of all shapes we analyzed beforehand to be required.

On first inspection, there seems to be no point in this anecdote or 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. This is why software has been called "*soft*" in the first place.

At a second look however, this is not true anymore for large enough systems. For instance, there is one software system which *obviously* is too complex for this: the web (www). Changes to it are only possible incrementally and while it is running. Nobody would consider to redeploy the entire web just to apply a change (e.g., in one page of one site). Somewhere in between a pocket calculator application and the web there is this phase-transition of "too complex to redeploy" (cf. fig. 1). On one end of this transition software is modified-redeployed-tested until it fits, on the other end, it either is "cut-to-fit" while in production or it freezes when it does not support sufficient "cutting".

And this is the point of the anecdote: a software system with a tile factory class would not support sufficient cutting when redeployment is not an option anymore. This is bad news: traditional OOP appears to be confined to the "low-complexity-side" of the two regimes of software systems separated by before-mentioned phase-transition. And pushing against the transition point makes software very expensive. This observation may be expressed in a more dramatic way:

"Software is harder than hardware."

This summarizes the fact that, in a complex-enough software system, an object representing a tile would be harder to change than the physical tile itself.

Deep in our hearts we all know this: You may make a screw purchased in a do-it-yourself market shorter if it is too long for some hole. But you cannot (in some email clients) sort by recipient in a renamed "Sent" folder.

In this paper, we will therefore propose an alternative way to develop software which may better been described by "building" or "growing" software rather than "programming". The ultimate goal is to merge both sides of the before-mentioned phase-transition into a single paradigm. Its basic assumptions may be summarized by the following "manifesto".

The Manifesto of Organic 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, static 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 OP model.

2. The emerging Organic Programming paradigm

Recently, a number of organic computing initiatives have taken off, especially in the domains of systems design and computer vision. Notably, there is the autonomous computing initiative (AC) of IBM [7] which emphasizes on self-x properties of a system such as self-configuring, self-adapting, self-explaining, self-healing or self-protecting. The more biologically inspired initiatives [18] include self-organization and self-emergent complexity.

We do however, not specifically adopt the notions of Organic Programming found in the agent community [8] or some more general ideas not targeted at software engineering [19].

There is one common insight behind all of these activities: complex properties of large systems (in nature or civilization) *always* emerge from its constituents in a spontaneous manner without being centralized or planned. Large complex systems in nature (and after the failure of communism, civilization as well) are all self-organizing.

At a medium level, the largest *software* systems such as the web are, if not self-organized, at least evolving spontaneously which, for the time being, is all we ask them for.

The problem now is that a software system which was crafted according to common (object-oriented) wisdom not only is not self-organizing; it even cannot evolve without asking programmers (or in the case of MDA, architects) to go back and change the blueprints. Every bit of flexibility must be built into the blueprints and often still, is available to experts only.

Organic programming is to deliver software systems which may still be object-oriented but which, at the same time are able to evolve spontaneously *by construction*.

2.1 Approaching the complexity-transition from the low-complexity end

The low-complexity end in the phase diagram of fig. 1 is characterized by the traditional methods of software engineering such as OOP, model-driven architectures, generative methods and component models. However, even within the low-complexity end, we see a split-off of technologies which are closer to the transition point than the mainstream. In this paper, we summarize these split-off technologies as the "Thing-oriented" trend.

Mainstream. The mainstream trend (in software engineering) is currently dominated by generative methods such as model-driven architecture (MDA), aspect-oriented programming (AOP), generators built-in into IDEs and an increasing level of (formal) 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 (incl. executable models).

The evolution of programming languages has long been 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 of the model without escaping the limitations of an object-oriented language. Component-based software as well as Aspect-oriented programming are pushing those limitations a bit. Altogether, we still find it disappointing how different software objects are from real-world entities after a quarter century of research. They are poor when it comes to represent them while *using* or *growing* a system, as opposed to representing them while modeling (of course, this is why we classified OOP as the dominating low-complexity technology, contrary to common believe).

The mainstream seems to disagree, given the awareness for model-driven architectures (MDA) or executable 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 or change this model, as laid out in the motivation part.

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, or for a part of a problem; 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 large system is against the gen-

eral engineering principle of keeping things as simple as could possibly work. *Say Good-Bye to Planned Economy.*

Thing-orientation. The trend towards Thing-oriented programming (as we name it in this work) may first have emerged in 1979 with "ThingLab" [4]. The publication of the programming language "Self" [14] in 1987 and of prototype-based languages in general were important milestones. The language "NewtonScript" [12] was inspired by Self and led itself to "JavaScript" [16]. Both are prototype-based. More recently, XML-based language "Water" [10] and Java-based system "NakedObjects" [9] 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 plug-in architectures as well, as recently demonstrated by the popularity of Eclipse [3]. A plug-in has some characteristics of a Thing (as we are going to define it) which an object does not.

There also is strong theoretical evidence that the emergence of object oriented patterns (such as those of the gang-of-four) is related to the assumptions made by the Manifesto of Organic Programming. One may, for instance, read §0 of the Manifesto as "symmetries are broken" and this, in turn is a sign of the limitations inherent in traditional object-oriented languages [5].

Orthogonally, new engineering methods emerged, with Extreme Programming [1] and the Agile Manifesto [15] 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 OP Manifesto. Ercatons (as defined in this paper) are meant to address *all* aspects of the OP manifesto and to be fully Thing-oriented. First information about it was released in 2003 [6].

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 will present our definition of the notion of a Thing in the next section, after having described the other end of the complexity-transition.

2.2 Approaching the complexity-transition from the high-complexity end

Highly-complex software systems currently *are not* object-oriented. Rather, they *contain* components which *use* OOP.

Examples include commercial operating systems which include components from various sources and vendors, or the web which consists of millions of independently created parts. Features emerge which were never planned ahead of time. Organic Programming includes creating software which i*s able* to evolve[1] spontaneously, i.e., in unanticipated ways.

Technologically, the high-complexity end is dominated by a few simple, yet powerful mechanisms such as **documents** (web pages, executable files, xml exchange messages), **authorization** schemes to maintain system stability, **fault-tolerance** and **distribution** (clicking a broken link does *not* crash the web ;-), **re-usability** (ever wondered about the many similar web styles, or how wonderfully Unix-grep fits into many tasks? – as opposed to 'class Person's which are *never* re-used...), and **simplicity**. Today's web specification is actually missing some complex parts such as a mechanism to enforce referential link integrity for the sake of simplicity. Many believe that this was key for the success of this particular specification[2].

Complexity emerges from simplicity, as paradox as this may sound. Complexity does not emerge from abstractions such as MDA.

More recently, there is a push down to the complexity transition point from the high end in order to achieve a higher level of programmability and interconnection. XML schema definitions, web services, semantic web services and discovery registries such as UDDI [17] are the most prominent examples. Of course, this will impose a challenge to any attempt to re-factor such a complex system in the future (a link may be moved on a web page but a web service is supposed to stay). We do not request Organic Programming to support re-factorizations.

We are now going to propose a construction which invents and combines concepts from *both* complexity regimes into a single programming model. It is centered around the term "Thing".

3. Definitions

3.1 Definition of Thing-oriented Programming

The basic idea behind a "Thing" is the ability to represent a real-world entity without an 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

1 For the sake of our notion of Organic Programming, it is sufficient that software can be evolved by independent *humans* (or *their* software) which have not been instructed to do so; it is not mandatory that the software evolves by itself.

2 HTTP and HTML by the way, have been created out of pragmatic need rather than fundamental research.

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. A Thing is able to exist and evolve without a model.

Definition: A "Thing" is a software entity within a production software system with the following 7 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, e.g., slots.

   Its behavior is defined by reaction to received messages or events such as change.

   Inner structure is separate from algorithm.
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 self-contained and self-explaining, e.g., human-readable:

   There is an equivalent externalizable and human-readable form accessible by name.

   It has one owner controlling its visibility and encapsulation.

   It self-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).

   Also, it may determine more formal properties such as search index entries which it may contain.
7. **Deterministic** – It will only temporarily maintain inconsistent inner structure (transaction safety).

Definition: A "Cell" is a Thing with 1 additional property:

1. **Activity** – It shows activity or changes on its own.

   This behavior is defined by reaction to events such as elapse of time.

Note that a production software system will, in general not be able to maintain a thread for each Cell as there may be too many (millions).

Cells are not (necessarily) agents. They may not have a goal etc. Agents are meant to be built from (a federation of) Cells.

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.

Definition: "Organic Programming" is the art of growing, evolving and enhancing a software system which is made of Things and while it is alive, i.e., in full production.

Both activities actually resemble growing and building rather than modeling and programming. We therefore sometimes call Organic 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 (a production software system) only. To make this more explicit:

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

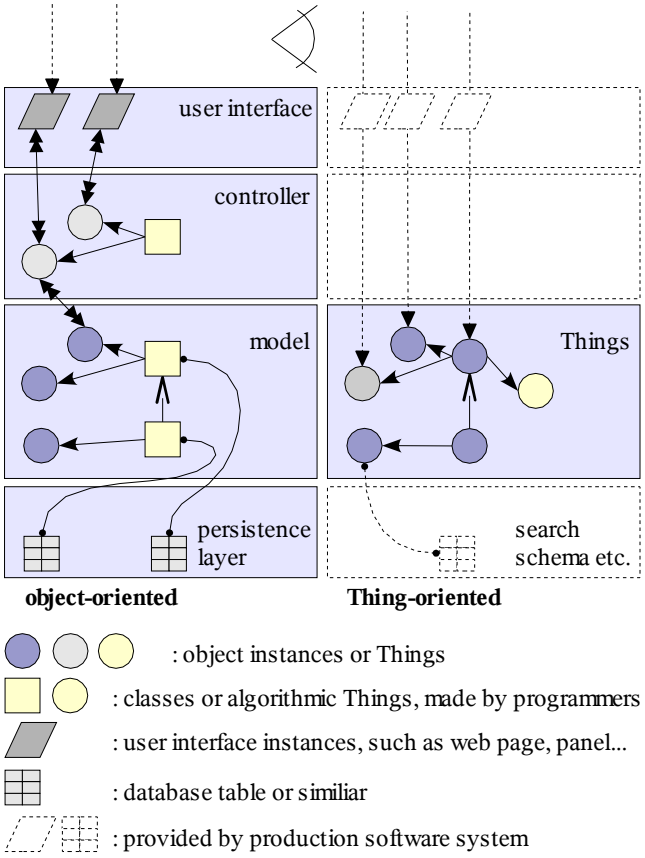

Figure 2. A Thing defines all relevant aspects within a software system, i.e., every single instance does. There is no intrinsic difference between a user's and a programmer's view.

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 Thing-oriented programming as prototype-based.

The discussion so far left open the question why we claim that a "Thing-oriented" approach solves the problems of Organic Programming. The answer is that Things may be the first concept unifying objects and documents into a single ... thing. This not only bridges the chasm between the two complexity regimes. Also, it immediately follows from the initial question:

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

(given the fact that both, documents and objects, appear as representing software entities in traditional approaches without a unifying concept).

In Figure 2 we conclude this section with an overview about how layering of a Thing-oriented software system differs from a traditional object-oriented one. This change in software layering has been called NakedObjects" [9].

It is difficult to praise the value of OOP without actually programming. The same way, we felt it be necessary to actually *do* Organic Programming[3].

3.2 Definition of Ercatons

In order to do Organic Programming one needs three more items: a precise specification of the exact software representation of a Thing to work with, a virtual machine to execute it, and a large-enough problem to solve. As we successively increase focus as we go, we will have to leave out detail.

Ercatons[4] provide a precise specification of one possible exact software representation of a Thing or Cell.

As mandated, 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 [13] as ercatons. However, this would not be in line with both clauses of §3 of the Manifesto. All typical OOP features (such as polymorphism, inheritance, encapsulation, method signatures etc.) are features of an ercaton *independent of algorithm*.

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. Some examples follow the definition.

Definition: An "ercaton" is a Cell (as previously defined) with following specifications (note that we have to specify technical properties of a Thing in order to actually build software – no new concepts get involved):

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.

3 This research work has *not* been funded. We just felt that it had to be done.

4 "Ercaton" is derived from (M)ercato + (I)on and literally means "elementary market particle".

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

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

   Do not compare Thing syntax with program code for a class – the syntax of an ercaton manifests its current internal structure and applies to a single instance. You won't find stuff like loops etc.

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

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

   plain an ordinary ercaton.
   role a user, owner, group, 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 prior structure 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 machine* is fully defined by all ercatons, independent of the order they have been manipulated in.

   This implies that an ercaton 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 machine is 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 machine must support at least a web-based interface.
   object specifies projection onto named API for an algorithm; an ercato machine 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 [21] using the externalized form. The transmission of authentication depends on the trust between two ercato machines.

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

Definition: An "ercato machine" is a production software system supporting ercatons as Things. It is a language-independent virtual machine hiding technical implementation details.

Definition: The "ercatoJ engine" is the ercato machine we have implemented on top of J2EE [11]. Note however, that ercatons are language-independent. The ercatoJ engine exists and is in mission-critical production use, e.g., at German chemical corporation Henkel KGaA [2].

All of the above 8 points are to be discussed in more detail elsewhere. This paper, however, aims at providing an overview and discussing the general idea with respect to Organic Programming. We will therefore only give some short examples to provide a better feeling.

3.3 Examples

Our first example introduces the syntax, the ercato markup.

```
<counter xmlns:erc="...">
  <erc:id> ~sample/count </erc:id>
  <count>  0 </count>             <!-- state -->
  <erc:action>
    /bin/increment (xpath="//count")
  </erc:action>
</counter>
```

Example 1. Simple counter ercaton able to update its state.

This ercaton contains the entire logic required to maintain a counter to be incremented, stored, viewed and used (in a transaction-safe way). It is identified as `~sample/count`. To increment it, the expression `~sample/count()` is evaluated. The example uses another ercaton, `/bin/increment`, to delegate the implementation.

```
<counter xmlns:erc="..." xmlns:xsl="...">
  <erc:id> ~sample/count </erc:id>
  <count> 0 </count>
  <erc:action name="main">
    <erc:arg name="amount"> 1 </erc:arg>
    <erc:native lang="Xslt">
      <xsl:template match="count">
        <count>
          <xsl:value-of select=". + $amount"/>
        </count>
      </xsl:template>
    </erc:native >
  </erc:action>
</counter>
```

Example 2. The counter ercaton rewritten to be stand-alone.

The rewritten ercaton uses a closure in order to be self-contained (not referring to any other ercaton). Its owner (or another ercaton action) may do this rewrite during production. Both forms behave equivalently, the `amount` argument is implicitly passed down the delegation chain in the first example. Java is supported as a language as well.

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

Figure 3. The counter ercaton of Example 1/2 in its default web browser look&feel (after three clicks onto 'main').

A click onto the "main"-button would increment the count to 4. *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. An ercaton may specify its own render pipeline.

4. The programming model

Rather than featuring a programming language, ercatons are programming-language independent. It is therefore necessary to show how ercato programs can be 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.

Ercatons feature OO inheritance and mapping to relational data w/o relying on a programming language. They are Cells too because they may contain triggers which fire upon elapse of time. They show autonomous behavior (or life), useful to implement self-x properties or agent patterns.

Ercatons are meant to directly implement entities of the real world also known as the problem domain or business logic. Most ercatons do so. 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. Invocation is cheap as it takes place within the local address space (e.g., the same Java virtual machine).

  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. They act like a user-space extension of the kernel.

- Preparation of state for a user interface or exchange: Implemented by service ercatons which are XSLT stylesheets used in render pipelines.

- Algorithmically non-trivial tasks: Implemented in an OO language (e.g., Java) after projecting the state of involved ercatons onto appropriate object instances.

  Within the ercatoJ engine, invocation of an action implemented in Java incurs the same overhead as invocation of an EJBean business method.

Example 3 shows an ercaton containing an action which is implemented in Java, together with complete and working Java code. It inherits from another, its so-called *clonebase*. It is kept synchronized with changes of its clonebase.

```
<counter>
  <erc:clone> ~sample/count </erc:clone>
  <erc:id> ~sample/count2 </erc:id>
  <erc:object lang="Java">
    <erc:archive> ~sample/lib.jar </erc:archive>
    <erc:class> sample.Counter </erc:class>
  </erc:object>
  <erc:action name="main">
    <erc:native lang="Java">
      <erc:method> increment </erc:method>
      <erc:parameter name="amount" type="int"/>
    </erc:native>
  </erc:action>
</counter>
```

Example 3. A derived counter ercaton, overriding the Xslt with a Java implementation. ~sample/count is its clonebase.

Both ercatons are in a relation like `~sample/count2` **extends** `~sample/count` and `~sample/count2` adds the `object` tag and redefines the `native` tag. The `count` and `amount` default values are inherited.

Inheritance may be the most exciting single aspect of ercatons. It is supported where syntax is as general as XML (the infoset tree) by inventing an algebra for XML [23].

```
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)) {
         counter = ec.getTextNode (false);
         count   = Integer.parseInt (
                     counter.getData ());
      }
   }
   protected void approve () {
      counter.setData (String.valueOf (count));
   }
   private Text counter;
   private int  count;
}
```

Example 3 cont'd. The counter's Java implementation w/o relying on any of the many Java/XML binding frameworks.

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:	OO language:
Signature to define usage of algorithms	**OO-class**	OO-interface
Signature to define usage of state	Permissions or XML-schema	**OO-class**

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

A long-lasting problem of object-oriented programming has been the tedious mapping of objects to relational databases and the insufficiency 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."

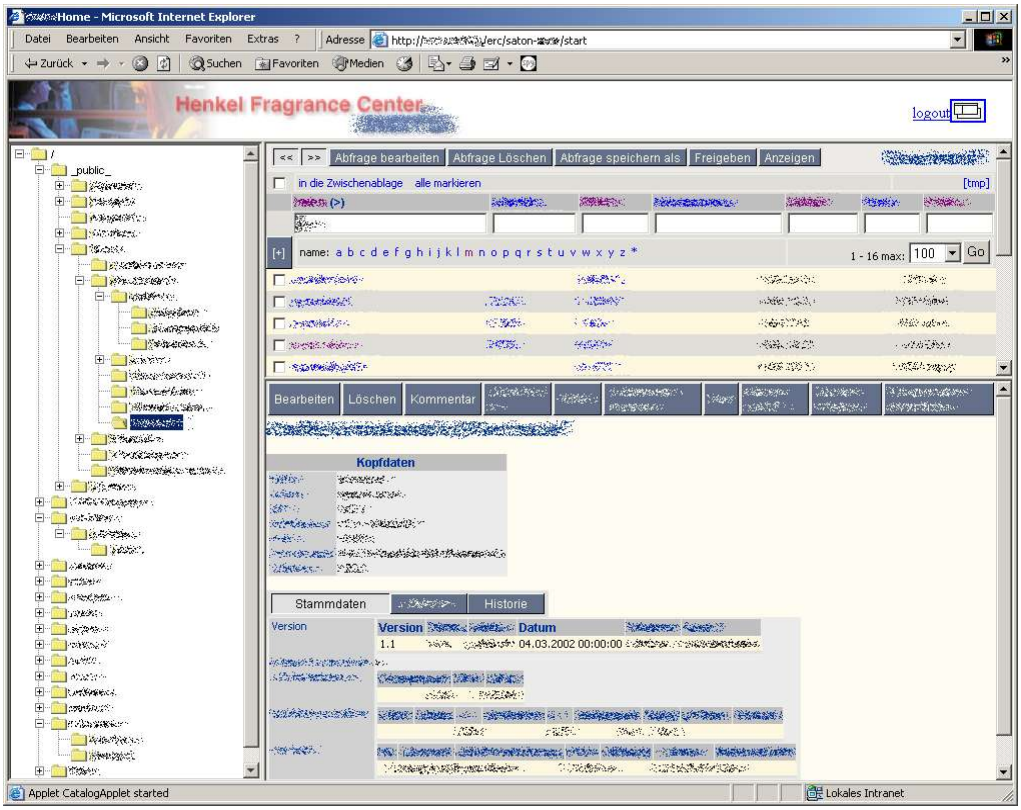

Figure 4. An ercaton-based application in mission-critical production at German chemical corporation Henkel. Fiveercatons are seen (i.e., their user interfaces as specified by their declared target pipelines).

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. The screen shot has been partially obscured upon request.

The internet is a good example where complex real-world software systems already obey this rule (a network of web-sites 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 which isolate from differences of schemata between arbitrary ercatons.
2. Index attributes in arbitrary ercatons.
3. Query ercatons for structured queries across relations plus API for supported OO-languages.

5. Lessons learned for Organic Programming

A growing number of early adopters including ourselves have now been working with ercatons for several years and we are amazed how much our thinking about software engineering has changed since. In particular, the J2EE-based ercatoJ engine is and has been used in a number of projects. Unfortunately, it it beyond the scope of this paper to describe the new patterns which have emerged while building and growing ercato-based systems.

Therefore, we may only quote one project where 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[5]. It interacts with a farm of SAP/R3 systems and replaces a host system which was coded with about a million lines of code. The system went productive with great success [2]. shows a screen shot for a part of the application, released for publication.

The system went productive 2 years ago. It was successfully and significantly grown by others as well as by us which

5The users of this application run demanding operations involving recursive retrieval of ingredient information 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 about one order of magnitude overall performance improvement. Performance on complex retrieval operations already is on equal with a native SQL implementation.

turned out to be more or less as easily as an enterprise is grown. The same way of thinking applies. After a while of operation, the system has evolved to a degree of complexity nobody would pay an architect to create a complete MDA model for. Nevertheless, it is rock-solid and is self-explanatory for everybody interested into it.

To summarize, while we advise to use traditional OOP methods to create algorithmic parts of a larger system, we encourage to drop those methods for entire systems in favor of more evolution-aware methods such as ercatons – but without giving up on OO principles.

6. Conclusion

Things and Cells in general or ercatons in particular provide a concrete way to bridge the chasm 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.

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 current EJB-, .NET- or XML-based persistence approaches [20].

7. References

[1] Kent Beck: "*Extreme Programming Explained: Embrace Change*". (1999) Addison-Wesley ISBN: 0201616416.

[2] 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

[3] Azad Bolour: "*Notes on the Eclipse Plug-in Architecture*". http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

[4] Alan Borning: "*ThingLab – A Constraint-Oriented Simulation Laborator*". XEROX PARC report SSL-79-3, July 1979.

[5] James O. Coplien and Liping Zhao: "*Symmetry Breaking in Software Patterns*". (2001) Springer Lecture Notes in Computer Science LNCS 2177. http://users.rcn.com/jcoplien/Patterns/Symmetry/Springer/SpringerSymmetry.html and private communication.

[6] Jürgen Diercks and Falk Langhammer: *"Bauen statt modellieren"*. iX-Magazin 2/ 2004, p. 100-103. Heise Verlag. http://www.heise.de/kiosk/archiv/ix/2004/2/100

[7] Jeffrey O. Kephart and David M. Chess: *"The Vision of Autonomic Computing"*. Computer pp.41-50. IEEE publication. January 2003. http://www.research.ibm.com/autonomic/research/papers/AC_Vision_Computer_Jan_2003.pdf and http://www.research.ibm.com/autonomic/manifesto/

[8] Hideyuki Nakashima: *"GAEA, an organic programming language"*. (2000). http://www.carc.aist.go.jp/gaea/

[9] Richard Pawson and Robert Matthews: "*Naked Objects*". (2002) John Wiley & Sons, Ltd. ISBN: 0470844205. http://www.nakedobjects.org/book.html

[10] Mike Plusch: "*Water: Simplified Web Services and XML Programming*". (2002) John Wiley & Sons. ISBN: 0764525360. http://www.waterlang.org/

[11] Bill Shannon: "*Java 2 Platform Enterprise Edition Specification, v1.4*". (2003) Sun microsystems. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

[12] Walter Smith: "*SELF and the Origins of NewtonScript*". PIE Developers magazine, July 1994. http://wsmith.best.vwh.net/Self-intro.html

[13] 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

[14] 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/

[15] "*The Agile Manifesto*". http://agilemanifesto.org/

[16] Netscape: "*Core JavaScript Reference*". http://devedge.netscape.com/library/manuals/2000/javascript/1.5/reference/

[17] OASIS: *"UDDI Spec Technical Committee Draft 3.0.2"*. OASIS Committee Draft. October 2004. http://uddi.org/pubs/uddi_v3.htm

[18] The Organic Computing Page of selected German Computer Society members. http://www.organic-computing.org/software/index.html (2004); and (German) http://www.gi-ev.de/download/VDE-ITG-GI-Positionspapier%20Organic%20Computing.pdf (2003).

[19] Organic Programming Tribe: *"Organic Programming"* (used as synonym to Organic Computing). http://organicprogramming.tribe.net/

[20] Tamino XML server home page. http://www.softwareag.com/tamino/

[21] W3C: *"Simple Object Access Protocol (SOAP) 1.1"*. W3C Note 08 May 2000. http://www.w3.org/TR/SOAP/

[22] W3C: "*XML Path Language (Xpath) Version 1.0*". Recommendation 16 November 1999. http://www.w3.org/TR/xpath

[23] XOperator evaluation kit. http://www.living-pages.de/de/projects/xop/