A SMALLTALK-BASED SYSTEM FOR DYNAMIC MULTI-CONTEXT INFORMATION PROCESSING

Adriaan van Os, Soops *adriaan@soops.nl*

Tim Verwaart, LEI *tim.verwaart@wur.nl*

12th ESUG Conference, K then, September 8, 2004

Adriaan van Os, SOOPS (*adriaan@soops.nl*)

Tim Verwaart, LEI (*tim.verwaart@wur.nl*)

A SMALLTALK-BASED SYSTEM FOR DYNAMIC MULTI-CONTEXT INFORMATION PROCESSING

ESUG 2004, K then, September 8, 2004

Soops

- Smalltalk software company founded in 1992
- Based in Amsterdam (NL) and W rzburg (D)
- Specialized in real-time commodity exchanges (power, gas)
- Developed new software for the dynamic information need of the FADN

www.soops.nl

Soops is founded in 1992 and has always been a Smalltalk company.

The main office is based in a nice historic building in the center of Amsterdam. We have a daughter company in Germany. We have customers in Amsterdam Power eXchange (NL + B), in the UK UKPX, Delta Lloyd, Mammoet Shipping/BigLift and of course LEI.

LEI and the FADN

- LEI: agricultural economics research institute
- A major activity of LEI is the FADN:
- Farm Accountancy Data Network
- 60 employees collect huge amounts of data
- 20 employees prepare statistical reports
- DSS for politicians and business managers

www.lei.nl

LEI – the economics research institute of Wageningen University and Research Centre – operates an information network (Farm Accountancy Data Network - FADN) for monitoring economic and technical performance of Dutch agriculture. The core business of LEI is to fulfil information needs of agricultural policy makers in government and agribusiness. The FADN is one of the means to do this. In the FADN, 60 full-time employees collect data to be applied in a vast range of research projects and statistical reports. Some 20 part-time users prepare data reports for research and statistics from the collected data. The main goal of the FADN is to monitor the performance of the agriculture industry.

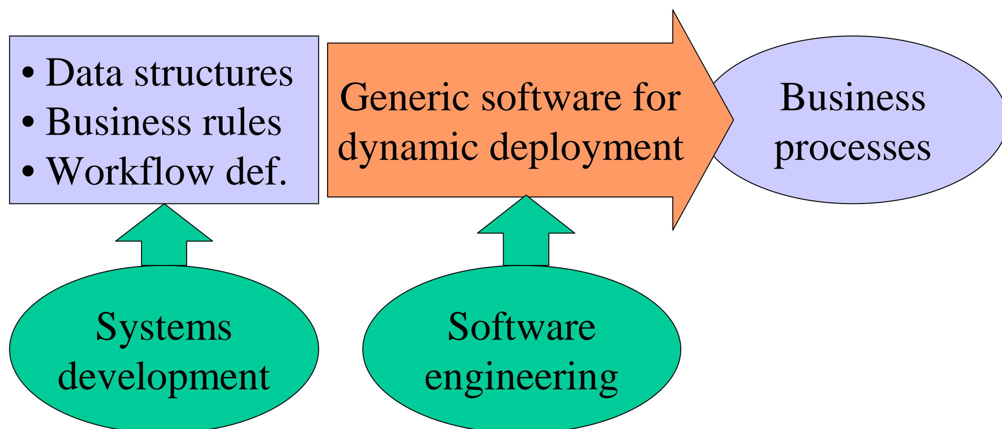
Dynamic information need

- Rapidly changing policy issues require rapid adaptation of complex data collection systems
- Many research projects in many contexts: e.g.: financial, technical, ecological, hydrological
  - Require different viewpoints on common data
  - Reuse data because of high data collection cost

The information need is dynamic. For new policy issues new data have to be collected. Complex data structures and data collection procedures must be rapidly adaptable to changing demands. Because of the high cost of data collection, data has to be reusable for many projects in several contexts, also future projects with yet unknown questions. The same data are being used for financial, technical, ecological and hydrological research, as well as research on social issues like animal welfare and food safety.

Model driven system behaviour

Software-changes put continuity at risk, so:
separate software/information system development



In spite of the dynamic character of information need, software systems should preferably be stable. Software changes are expensive and are a risk to system stability. This is why we want to separate the software engineering process from the information systems development activities. For this purpose LEI developed a system for model-based information processing. The models include data structure descriptions, business rules and workflow descriptions.

A team of four employees who do not have a software engineering background but a thorough knowledge of the business processes maintain the models and adapt them to changing information needs. System behaviour is dynamically being generated from the models by system components for workload scheduling, manual data entry, electronic data exchange, reporting, financial accounting, aggregation of statistics, and data quality monitoring. In this way the system can be changed without changing the software. Developing new software releases is separated from developing the information system. Both processes can run in their own pace. The system has been engineered in VisualWorks and runs on a GemStone application server.

In this presentation we introduce:

- Principles of the multi-context data model with business rules
- Principles of business rule driven workflow enactment
- Implementation of model driven system behaviour in VisualWorks
- Implementation of persistency with GemStone

In this presentation we introduce:

•Principles of the multi-context data model with business rules;

•Principles of business rule driven workflow enactment;

•Implementation of model driven system behaviour in VisualWorks;

•Implementation of persistency with GemStone.

This system demonstrates that Smalltalk can effectively be used for developing complex business-critical applications.

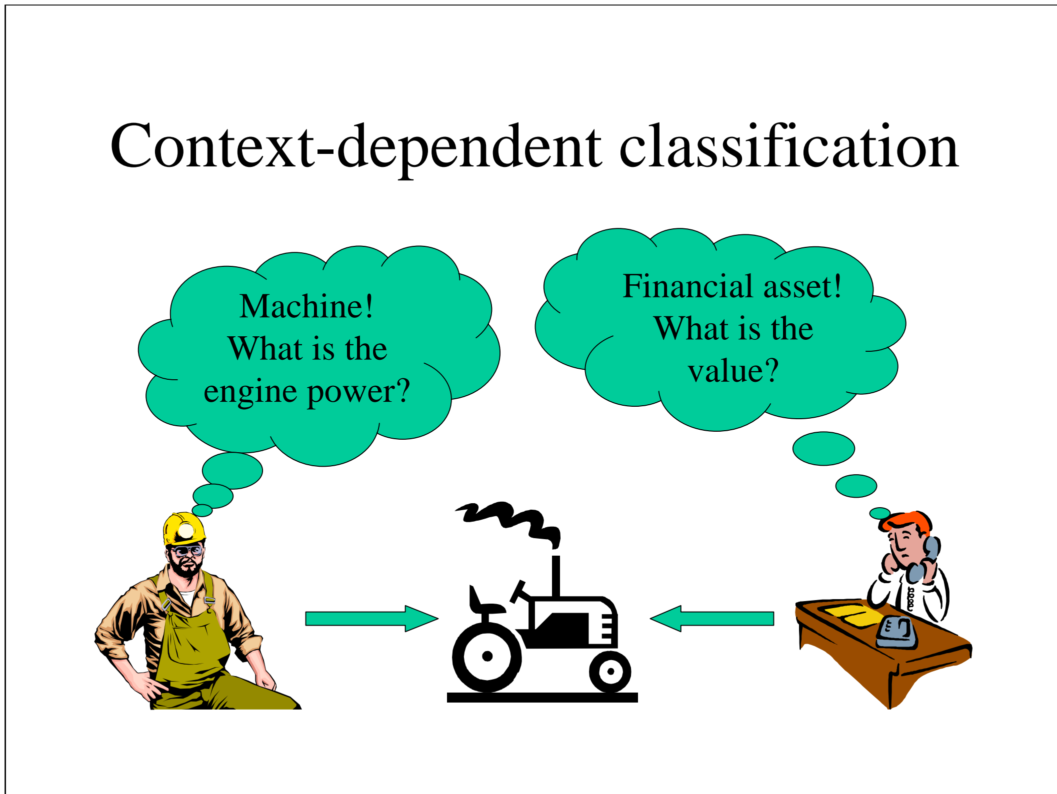
Part 2
Multi-context data model

2 MULTI-CONTEXT DATA MODEL

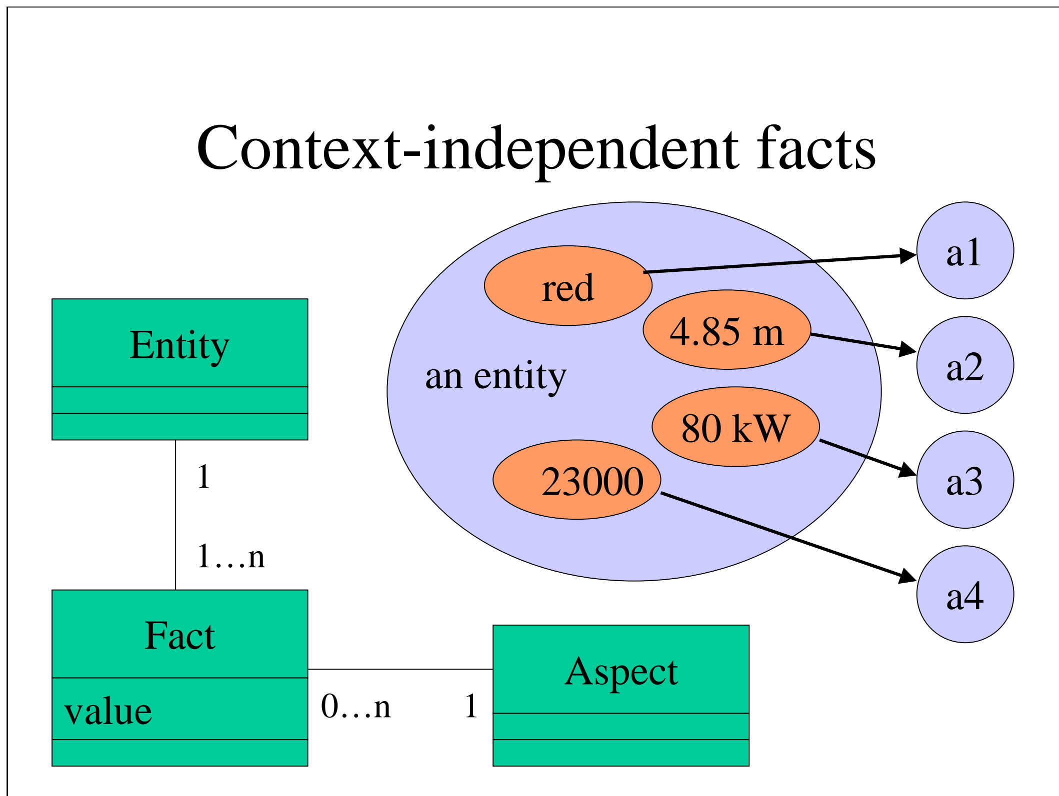
Why multi-context?

- Reuse data
- Data collection is expensive
- Data for different purposes are being collected in a single workflow
- Data must be reusable for future projects
- So: record facts, not interpretations
- The interpretation is context-dependent

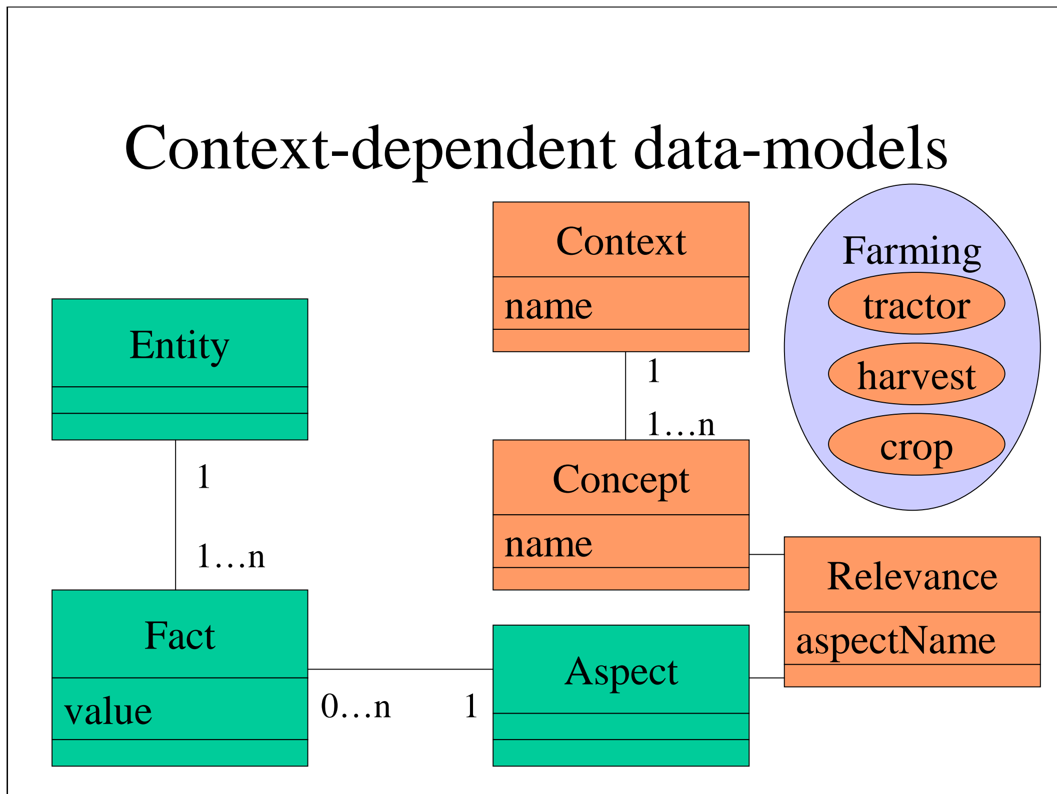
As has been stated in the introduction, the high cost of data collection urges to reuse data in a broad range of projects. Data are collected for different purposes in a single workflow. Apart from the purposes known at the time of data collection, the data must also be reusable for future projects. This will only be possible if elementary facts are recorded, not if interpretations of data in the context of known applications are recorded. The interpretation of facts and the classification of objects are context-dependent.



Example: A farmer uses a barn and two tractors. He owns the barn and one of the tractors. The other tractor is leased. For the farmer both tractors are relevant entities of type “machine”, and the engine power is a relevant aspect. For the farmer the barn is a relevant entity of type “building” and the capacity for potato storage is a relevant aspect. For the accountant the owned tractor and the barn are both relevant objects of type “asset” with relevant aspect “value”. The accountant is not interested in the leased tractor as an entity. He will enter the lease fee in the books under the running costs.

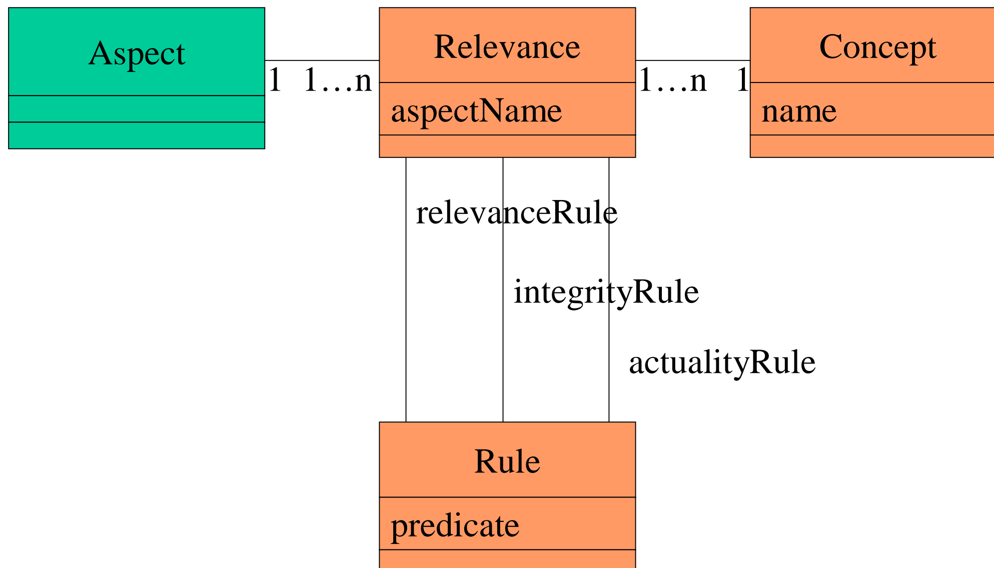


Context-dependent interpretation is enabled by context-independent representation of facts. Data values are recorded in the repository as collections of facts about an entity. An entity represents some individual object instance. An entity as such is not typed as a member of some class of objects. It is just a bunch of facts about something. Every fact represents the value of some aspect of the entity, e.g. color, length, engine power or financial value.



The typing or classification of an entity is only possible in a context. A context is defined as a set of concepts for the interpretation of data, for instance the concepts of tractor, building and crop in the farming context. A concept represents a class of entities. The relevance of aspects (or attributes) of the entities depends on the context, and aspects may be named differently in different contexts. In our data model "relevance"-objects link concepts with aspects, thus representing that in the concept's context the aspect may be relevant as an attribute of the concept.

The extension of concepts with entities as class members (the population) can only occur within a context, using the values of facts and/or the type of entities in another context (*Example*: In context *X* we classify an entity as an arable farm if it is classified as a farm in context *Y* and if it has at least *N* acres of arable land). In this presentation we will not go into more detail about the extension of concepts with entities, because of time limits. We will focus on the relevance of aspects.



The relevance of aspects as attributes of a concept may be conditional. For instance we may only be interested in the quality of a lot of potatoes if the quantity is at least ten tons. For this purpose we introduce relevance rules. Relevance rules are attached to a relevance object and hold a predicate, expressed as a logical formula. A relevance rule expresses that the current aspect is relevant for the current concept if the predicate is true. Furthermore we define integrity rules and actuality rules. Integrity rules and actuality rules have an important role in the workflow scheduling. Integrity rules specify the conditions to be met in order to consider the work as done. Actuality rules specify when these conditions have to be met.

Temporal data model

- A temporal data model is required for the application of actuality rules
- Facts have “valid time” labels: instances of fact are labeled with start-date and end-date of validity
- Time labels can be used in actuality rules:

```
daysafter(enddate(asset.value)) < 90
```

A temporal data model is necessary for the implementation of actuality rules. All facts are labeled with a period of validity. This enables the system to detect for which period most recent data are known. The actuality rule might specify, for instance, that the time lag between system date and most recent fact value of an aspect may not be more than three months. If the system detects a violation of the rule, it will try to enact a workflow that can result in a more recent value of the fact.

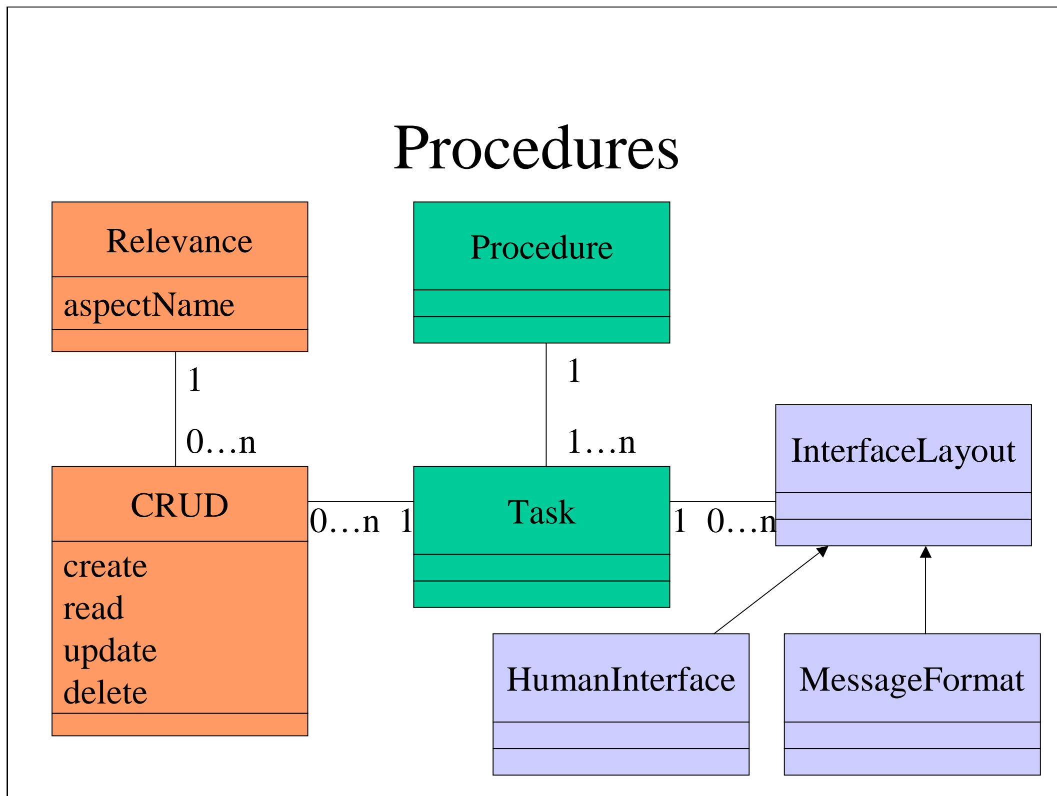
Part 3
Workflow enactment

3 WORKFLOW ENACTMENT

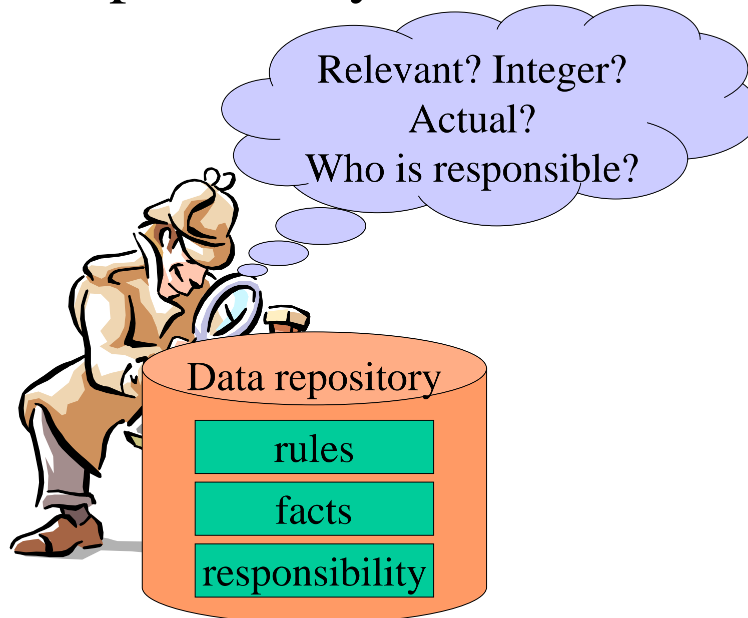
How to enact workflow?

- The rules specify **when** action must be taken
- We also need to know **how** it can be done:
  - Procedure descriptions (workflow definitions)
  - Interface descriptions (human + electronic)
- And **who** has to solve rule violations:
  - Responsibility of employees for entities
- And then start a procedure

The rules as discussed in the previous section completely specify the information need. Workflow enactment however requires more than just the rules that specify the information need. The rules do not specify how the need can be fulfilled and who should do it. For that purpose we need descriptions of procedures, interfaces and responsibility. And of course system components that actually enact the procedures (the workflow engine).



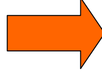
Procedures are modeled as sequences of tasks. The tasks are linked with the description of a context through CRUD objects that specify possible fact modifications, by referring to relevant aspects and specifying as logical (binary) variables the right to create, read, update, or delete facts. If a task may modify the database, it refers to one or more interface layout specifications for either human-computer interaction (data entry forms) or computer-computer interaction (message formats). This information and the rules are used for procedure execution.

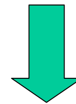


A software component called “attendant” actually checks the rules, bring them under the attention of responsible employees and propose - to the user - a procedure to be invoked. If a violation is found, the attendant needs more information to invoke a procedure. This information is specified in responsibility objects and applicability objects. Responsibility objects link employees (system users) with entities in the database. Applicability objects link rules with procedures to be invoked in case of rule violation.

Enactment by the workflow engine

Message from the attendant





Start procedure X

After receipt of a message from the attendant the user can start the procedure suggested by the attendant or solve the problem in another way, e.g. by starting another procedure. If the procedure involves some manual data entry, the “zebra” software component will dynamically generate data entry forms, applying CRUD, relevance rules and layout specifications.

Part 4
Implementation of dynamics

4 IMPLEMENTATION OF DYNAMICS

No more programming

- The information need is dynamic and based on the context and the data
- Therefore it is impossible to hard code data entry screens
- Data entry screens must be generated
- Data entry screens must be structured
- Data entry screens must be ergonomic and efficient

No more programming was one of the **challenges** when we started to build the system. You can imagine that we have many data-entry screens. These are **too flexible** and are **changing too rapidly** to hard code them. No problem, because we didn't want to program all these data entry screens in the first place.

So if you don't hard code them, they must be generated. But you cannot just generate screens based on the models. It will drive the employees **crazy** if the system presents fields for whatever the system needs to know in an order nobody understands. Therefore data-entry screens must be **structured**. We call this **configurations**. This means that you have to tell the system what could be on one screen.

You also have to structure the data entry screen somehow to satisfy the ergonomic and economical (efficiency) requirements.

Configuring a data entry screen

- CRUDs tell the system what could be on the screen
- Layout-data tells the generator where things can be on the screen

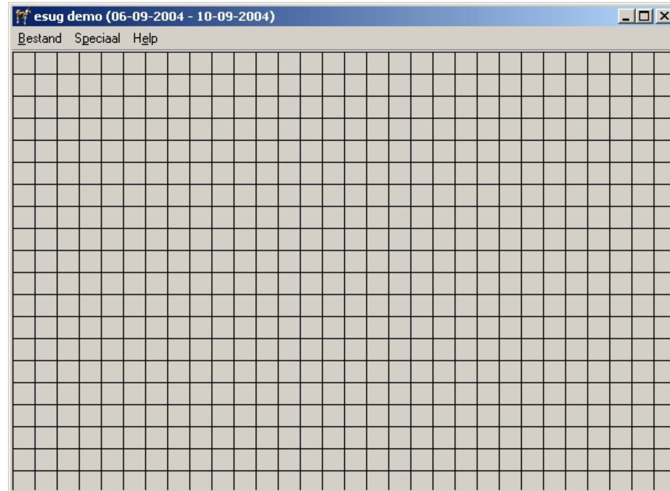
What do we have?

We have a procedure including those CRUDs representing a mask telling the application generator what data it can create, read, modify or delete. So for every aspect you want to manipulate data for, you define a CRUD. Without a CRUD you can do **nothing**, no matter what the rules say. With a CRUD, you still have to **comply** with the rules. This makes it possible to generate screens with only some of the aspects of a model and not all, because this usually doesn't make sense.

What is missing?

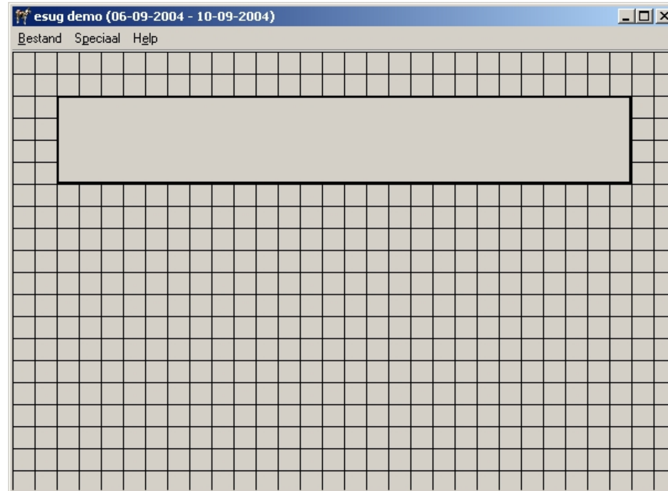
Next step is to tell where you want to see these aspects on a screen.

Screen editor: defining grid



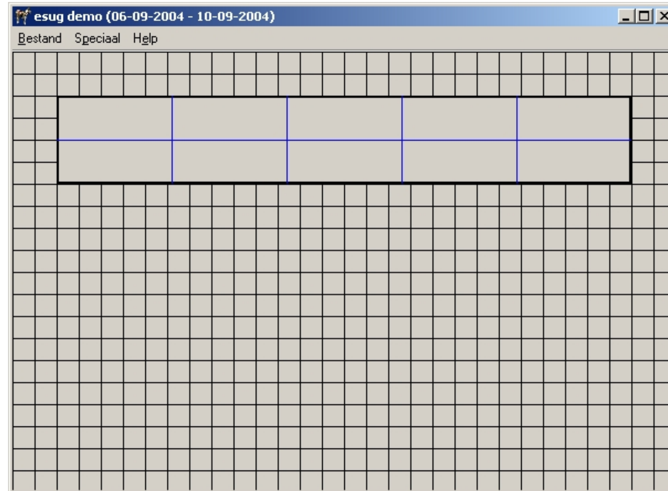
The grid is used to position blocks.

Screen editor: defining block



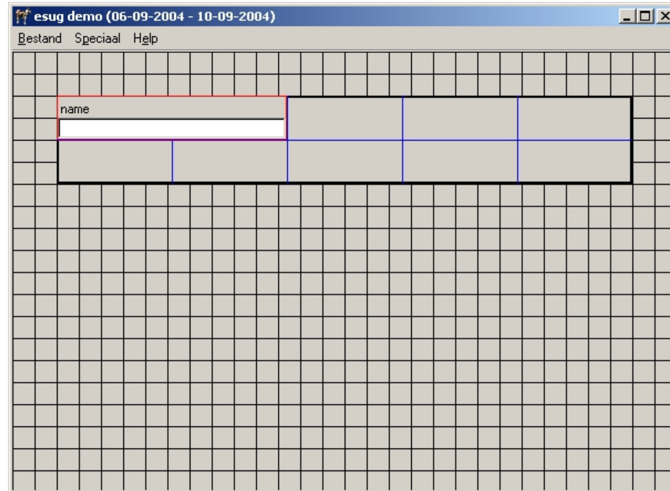
A block is bound to a concept in your model. This means you can only manipulate aspects known in this concept.

Screen editor: defining block grid



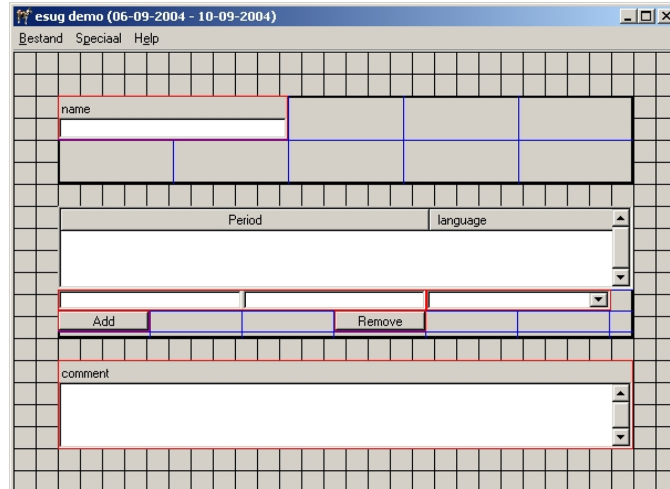
Inside the block a grid is used to position widgets.

Screen editor: adding a widget



A widget using up two cells of the grid. Within a block all widgets will manipulate the same one entity.

Screen editor: adding more



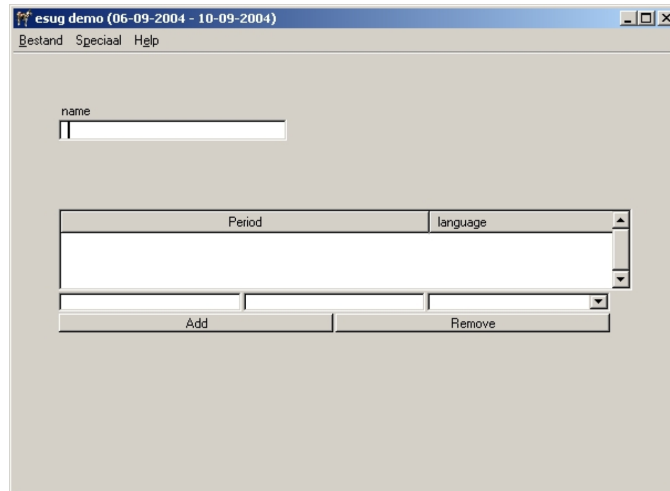
To speed things up we just add some more blocks and widgets at once.

In the middle you see a different kind of block. It's a collection block. Why? Let me tell first how blocks depend on each other. The block represent different concepts. You can only add more blocks if you have a association aspect between them. An association aspect is a one different types of aspects the system uses. The others are of course number and string.

To add a block you have to point an association aspect of the upper block. If this is a one-to-many association is the added block will a collection block. This also defines the concept of the added block.

Note the lower block with one widget. This is a detail block. A special block that belongs to the same concept as it's parent and manipulates the same entity. These blocks are useful when you have widgets that don't conveniently in a collection block like a text widget. Also it gives some mere possibilities for the layout.

Final result data entry screen



When executing the screen looks like this. You see less widgets then in the preview. The widget at the bottom is not there. It's apparently not relevant in the current state, but may become relevant during data entry.

I can say that debugging of these screens need good communication skills because a bug can be in the model and rules, in the configuration, some unexpected data somewhere, or in the application generator. This all can be built by different persons, and guessing where to start looking needs some experience.

The hard part: Performance

- Every change means evaluating an integrity rule
- Every change might trigger the recalculation of other values (recursively)
- Every change means that widgets can appear or disappear according to the relevance rules

Even without **Pollock**, getting widgets on the screen was really the easiest part of the generator.

It's harder to get the widgets on and off the screen in an **acceptable** time.

Users expect they can enter data just as fast as in any static screen. However, in a dynamic screen things are different. Every change of value might mean a lot of work to do for the system.

The integrity rules are a minor issue. This is just one rule to be evaluated when a value changes. And before being made persistent all values for one concept are evaluated to catch integrity violations based on rules and data for more then one CRUD.

The triggers between widgets can cause recalculations of other values on the screen (recursively). This is already a bit worse in terms of performance.

But the real pain comes when we look at the relevance rule. If you define a very general screen, where hundreds of widgets could be relevant, every change will cause a re-evaluation of hundreds of relevance rules. Those rules may need a serious amount of data to evaluate.

Part 5
Implementation of persistency

5 IMPLEMENTATION OF PERSISTENCY

GemStone Application Server

- Smalltalk Application Server
- Object Database

- Mapping comes for ‘free’
- Our ‘domain’ level code runs both in GemStone and VisualWorks

Application Server

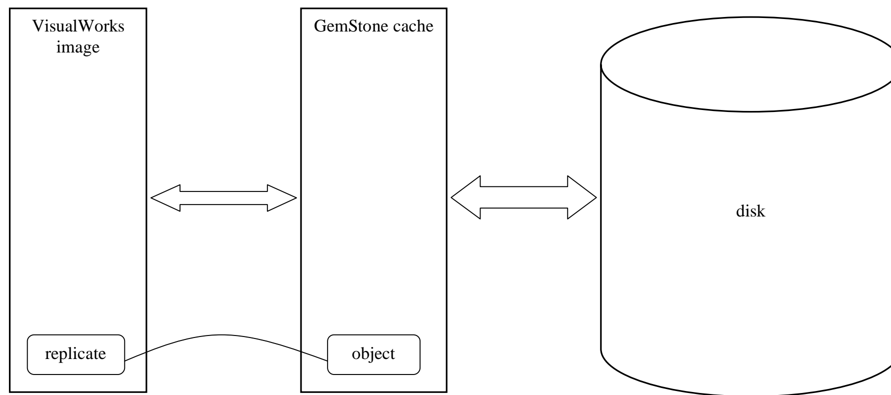
The system is implemented in VisualWorks 3. It uses GemStone as an **object database** and as a **Smalltalk application server**.

Using GemStone is great because the **mappings** come for free and it’s application server capabilities make it possible to do as much **processing on the server** as you like. Our domain object model including behaviour is built in a way that it **works fast in GemStone**, but it **also runs on VisualWorks**. I have a version without GemStone on my laptop, which uses the same code. I’m willing to demo the application for those who are interested later on.

But.

Code that normally just performs **carefree** in a stand-alone image, suddenly can become noticeably slow when you execute the same code in GemStone, or partially in GemStone and partially in VisualWorks. For example, when the data needs to be read from disk first.

Replication and Partitioning



How does the this all look like?

1. First you have your VisualWorks image.
2. Then there is GemStone server with objects on disk and a big cache. This means we have **two object spaces** now.
3. These must be **synchronised** when needed.

When? This synchronisation is implicitly done when you change the locus of execution somewhere in your program. During this synchronisation the changed objects that live in both spaces are **replicated** from one space to another.

Replication is **time consuming** for many reasons. The encoding, network transport and the decoding all taking their time. Also, both the client and the server session maintain caches and/or state of what is send to client. It's therefore recommendable to **reduce** both the number of synchronisations (network **roundtrips**) and the number of replicated objects.

Because of the **nature** of the system we do most work on the server. To much data involved to get it to the client.

GemStone has a couple ways to manipulate the **locus of execution**. The 'locus of execution' is something you have to be **aware** of while **designing** and **building** your application to make it performing.

How to enhance Performance

- Reduce the number of objects
- Cluster the data
- Reduce the number of roundtrips
- Reduce replication

- Cache results

- Buy more or faster hardware

One of the benefits of reducing the objects is that you don't have to wait for those objects being read from the hard disk, because they are no longer there.

Why would you want to cluster the data? GemStone uses 8kb pieces of disk space, called a page, to store data. A page is read and written at once. You can reduce disk i/o if you manage to have the objects you need on a single page. Or at least on as few pages as possible.

Replication. For example it's much more efficient to send an **xml** document as a string to the client and parse it into a DOM again, then sending the DOM as objects from the GemStone server session to the client. On of the reasons for this is that you are bloating a cache on the client side and registry on the server side. No surprise if you ever heard of 'James Foster strings'.

Our system often needs a lot of data to do its job. It applies the rules to many entities and facts. Al these calculations are done at the **server side**. These calculations are done by a class called **evaluator**, which is able to cache results. This **caching** introduces new problems, because you had to decide what to cache and what not and for how long. GemStone doesn't like **temporary objects** too much, because of its attempts to garbage collect them. We figured that out that caching works best if the calculation took 5ms CPU time as a threshold. At least in the cases we looked at.

Reducing the number of objects

- Make the facts reusable (flyweight pattern)
- Make the default date implicit
- Simplify numbers (1.0 becomes 1)
- Eliminate single item collections

The institute wants to know a lot and hence collects lots of data. There are millions of entities in the database and, every entity consists of at least 3/4 objects. All those entities can have many facts so we had hundreds of millions of objects after a year or two. GemStone only stores **2 billion** objects so we had to reduce the number of objects.

If we didn't do anything we would run out of objects before the estimated retirement of the application.

Also many facts are the **same**, they are only attached to different entities for different aspects. If we cut those references to entity and aspect, it will be a reusable over many entities and aspects.

The third reduction is based on making things **implicit**. When a facts doesn't have a period, we don't store it. For fact being a **SmallInteger** we then no longer use an object pointer.

So we started studying the data and found that we have many collections with only one element. Because we don't have **time series** everywhere.

Questions?

So this is all we can tell in 25 minutes. Are there any questions?