ESUG, Southampton, August 28th - September 1st, 2000

This conference had several very interesting talks and several very interesting people attending. Venue and accommodation were connected by a walk through the parks so, for a city-centre site, things were reasonably pleasant. (However, as Southampton was heavily bombed in WWII and so, except for the parks, has a ugly '60's-style centre, I think Cincom judged wisely in treating us to a tour, cream tea and pub evening in adjacent historic Winchester :-). Accommodation and meals were plain but good value for money. The only problem in the arrangements was finding the accommodation venue in Southampton on first arriving; some people spent 2 hours between reaching the vicinity and finding not-overly-signposted accommodation block (the Southampton resident who should have signposted it for us fell ill or forgot or something).

Authors' Disclaimer

This report was written by Niall Ross (nfr@bigwig.net) and edited by Stephane Ducasse (ducasse@iam.unibe.ch) and Roel Wuyts (roel.wuyts@iam.unibe.ch). It presents our personal view. No view of either author's employer is expressed or implied.

Style

The occasional 'I', 'my', 'we', 'our' refers to Niall Ross and his team.

Summary of Talks

I have sorted the talks into five categories:

- Meta-Programming and Adaptive Programming
- Testing and Experience Reports
- New Product Directions
- Web, Multimedia and Cooperative Working
- Discussion Sessions and Miscellaneous

Michel Tilman's talks (first and fourth categories) were of most immediate relevance to our meta-programming work. Eliot Miranda's talk in the first category was fascinating; of little immediate relevance to us but strongly suggesting where VMs and programming may go in future (as is supported by other indications). Joseph Pelrine's talks in the second section were useful examples of well-run projects and their eXtreme Programming experience. Dave Simmon's talk in the third category sheds an interesting light on Microsoft's rival to Java and its VM. The discussions were good.

I found several of the other talks useful and/or interesting. In the fourth category, I wondered about adding COAST to our work or to DoME, and about using Ernest Micklei's domino-toppling application to give lively title slides to my talks, while CosmoCows sounded such fun I was strongly tempted to ask if they were hiring. It was also good to hear talks about ST web work. In the third category, Dolphin's MVP framework looked good, the squeak reports were encouraging, and TotallyObject's market niche was new to me. Piotr Palacz talk (last category) generated heat and light.

Meta-Programming and Adaptive Programming

Co-evolution, Roel Wuyts, Programming Technology Lab, Brussels

Analysis, then Design then (forever stuck in) implementation is the usual way of software projects. Co-evolution aims to keep design and implementation synchronised (e.g. how is this composite pattern implemented here?). By making the design information explicit, it can be extracted from the code. He uses Smalltalk Object Unification Language (SOUL: see last year's ESUG talk) to make the design information explicit. SOUL is a reflective logic meta-language designed to reason about code structure. The meta-language reasons about the base language by reasoning about the parse tree of the base language. For example:

```
Rule class(?c) if
  constant(?c),
  [Smalltalk includes: ?c name].
```

is one of the predicates at the very top of SOUL that connects it to Smalltalk. Other predicates define inheritance and then build up SOUL reasoning from the top-level ones. Roel uses a prolog-like syntax for these predicates since to use Smalltalk syntax for both the reasoning language and the language being reasoned over is confusing (he's tried it).

He has predicates to support elaborate nested queries e.g. the chain of methods from a selected method, the method in the class hierarchy of class X that uses a temporary variable named Y, etc. Other predicates capture design patterns, e.g. composite pattern, factory pattern. His predicates work for all VW and HotDraw composite patterns. For example,

```
q Query
  compositePattern([VisualPart],[CompositePart],?msg)
```

finds the composite pattern between VisualPart and CompositePart.

```
instVarTypes{[Point],[#x],?possibleTypes),
stripHierarchyClasses()
```

finds types based on messages sent, assignments and instvar accessors to compute a list of possible types and then reduces it to get its common sufficiently specific superclasses, thus giving the list of most general possible types.

He used this type information to construct UML diagrams from code. It extracted a UML diagram for HotDraw (taking 2.5 hours), which was very helpful in understanding HotDraw. He would like to unite this with the COAST UML editor. He also wrote a predicate, extending Class, that makes classes dependents, so tracking changes to them. Hooking this up to constraints lets him enforce HotDraw design patterns on all subclasses of HotDraw framework classes; necessary methods are generated as the pattern detects that the user's work requires them.

He built all this on VW and Envy and has ported it to Squeak. He is currently working on QSoul, an extension that has a constraint solver.

Building run-time analysis tools using pluggable interpreters, Michel Tilman, Senior System Architect, Unisys, Belgium
The speaker has been experimenting with building analysis tools using two techniques:

- method wrappers (see downloadable code and papers by John Brant, reachable from http://whysmalltalk.com/method.htm and/or John's webpage): you can use method wrappers to do before/after methods, to build interaction diagrams and, in his case, to build an interpreter. Method wrappers wrap the compiled code of the method with the wrapper and insert the wrapped code into the method dictionary as the value of the method selector key; thus whenever that method is invoked, the wrapped code is executed. The developer wraps methods by selecting classes or individual methods in a wrapping utility.
- a smalltalk interpreter written in smalltalk

A VW2.5 plus ENVY version of this is on his home page. His demo was on 5i, to which he is porting it (looked fairly complete; John's published wrappers and wrapper-based tools run on VW3.0; I have ported them to VW 3.0 + ENVY 4.0). His eval method handles the abstract parse tree; smalltalk does the run-time engine, garbage collection, etc. He has a clean basic expression tree (see slides) e.g.

```
Assignment (variable, value)
MessageExpression(receiver selector arguments
methodCache)
```

etc. (his slides were slightly wrong for clarity; the returning block must be defined in the outer method-wrapper defining method in order to return to it), plus self and super (but not thisContext because little used, would be easy to add), Contexts, BlockClosures, etc.

An appropriate call of #eval: iterating over statements then allows him to invoke appropriate implementations of eval: defined on appropriate nodes in his expression tree. He uses a #contextAt: method to extract the appropriate nested context from the eval: argument. For blocks, evaluating the block is not the same as executing the block; #eval: just creates the BlockClosure (his BlockClosure in his abstract grammar, not the VW BlockClosure class) so he had to wrap his BlockClosures in a real native VW BlockClosure using #asNativeBlockClosure (tedious as he needed a different line of code for each possible number of arguments to the block; he uses a recursive trick for this) and put it in the right place.

He uses a methodCache to track all links between receiver class and implementation e.g. to query system for all cases where a method call was received and handled by two different implementations during the test run, (i.e. call truly polymorphic, or could reveal error if this were not intended). The tool is not protected from evaluating its own code but can loop or fail in that case.

Michel then demoed a coverage tool and a tool for gathering run-time typing information. (In the latter, he treats disjoint occurrences of the same variable as different to see if different types are appearing in them, to detect errors.) He began by selecting some classes, installing method wrappers on them (i.e. all their methods) and running a scenario. Colour-coding indicates what has been activated not-at-all, partially, completely. Classes and protocols show what percentage of their methods were activated. Methods show which of their code lines have been activated. Selecting a variable shows what types it contained and how often (this is not available for pseudo-variables today, but it would be easy to add). His browser allows easy one-window navigation up and down sender and receiver/implementor chains.

It runs 6-7 times slower than the actual code, which is impressive given that there is no optimisation (but noting that only the selected classes, not all classes in the system, are being interpreted).

Method Wrapping is very useful for debugging parts of the code the debugger won't work on. They can be used to optimise code, e.g. to find #or: calls that more often take the right branch than the left and so might be switched round for better performance.

A Tragedy of PIC Proportions, Eliot Miranda, Cincom, U.S.
This talk described how the VisualWorks virtual machine's message sends have been made more efficient by the use of adaptive programming techniques. Polymorphic Inline Caches (PICs) achieve easy performance gains, leaving a hard task remaining.

After a very quick reverse Polish tutorial, illustrated by a quickly written code example, Eliot brought up a method browser and showed its byte codes (which are in reverse Polish) and their relationship to the source.

```
self class name
```

in reverse Polish is

```
push self
send class (a primitive)
send name (not a primitive)
```

This puts self on the stack, pops it and sends class to it and pushes the result on stack, then pops that result, sends name to it and pushes that on the stack, etc. To see this kind of thing, look at 'bytecodes' of any inspector on a method. (One can reconstruct the source from the bytecodes. Squeak also caches parameter names efficiently, by assuming that most parameter names are variants of anInteger, a<ClassName>, etc., and so, XP-wise, you can work without any source at all; an 800k Squeak image can hold 900k of inferred source.)

This is easy to understand but, simply implemented, a pig to make go fast. The VM finds the bytecode at current position and switches (256 possibilities - push receiver, pop stack, push an argument, etc.). Having got the selector and receiver class from this, you linear search the method dictionary (as fast as hash for most sizes of classes). This is not very fast so, since 1976, Smalltalk has had a hash table, keying on selector and receiver. If a lookup fails on this hash table, the original lookup (i.e. described above) is done and its result put in the table, keyed on class and selector. 29 times in 30, the hash hits on a 1024 table and a larger table does not give significantly better odds.

Then a better idea surfaced (via Deutch, Alan Shipman et al.). To save time decoding bytecodes, generate as machine code for specific cases the lookup procedure the interpreter does. To lookup bytecodes, load registers with class, receiver, selector. To see this kind of thing, look at 'translation' of any inspector on a method. (On a windows machine, a 640k heap called the Nzone, corresponding to roughly 3000 methods, is gradually built up by this at run-time; c.f. the VW method cache is 4096k - it was 2048k for many years.)

Most selectors always link to the same method of the same class. The first time the send occurs, the VM finds the method's compiled code and stores it in the selector register (i.e. replace the register's contents with the compiled code). The second time, the VM checks if it is a small object (e.g. small integer) that is encoded specially (the Java primitive types discussion on the web summarises how Smalltalk handles these), in which case it gets the class appropriately for it, or it is an ordinary object, in which case it compares the class register (which holds the class which provided the method you linked to the first time) with the current receiver class. If they are the same, the linked method is valid to use. Thus several lookups and comparisons have been converted to a single comparison (of classes) leading immediately to the compiled code for the method. There is a factor of 6 gain in performance from this. (If a method is recompiled, every entry for that selector in the heap must be flushed as it is too hard to work out just the entry that corresponds to that method on that class.)

The downside is that this is expensive if the cache misses; the classes are not the same. The new class for that method is put into the cache and the old flushed. If your program has many such truly polymorphic swaps then 20% of runtime can be spent (and in VW until recently was spent) in such cache swaps. Eliot computed the profile and showed 10% of time in cache construction and more in class hierarchy searching to find a new method. Analysis shows that

- 40% of sends do nothing (e.g. a primitive fail branch is never traversed as the primitive always succeeds)
- 55% of sends always bind to one class
- 4.5% linked to no more than 8 classes
- 0.5% linked to more than 8 classes

The Self language (Dave Unger et al.) had a higher proportion of the last category. They evolved the monomorphic lookup cache described above to a Polymorphic Inline Cache. Whenever a linkSend fails, it is likely that the class to which the selector binds will vary, so allocate a small collection (8 elements max) of possible classes for that selector in the lookup cache. If the class does not equal the last lookup found, compare it to the second-last lookup, then the third last and so on. Every lookup on failure to match becomes the new last lookup found. This removes the 10% cache construction spike but leaves the 12% method lookup spike.

When a PIC overflows beyond its eighth case, construct code to do a lookup every time and put that in the cache; for these 0.5% of very polymorphic methods, this is the sensible thing to do. This approach runs faster than before *but* spends more time in the VM than before; 12%+ in findMethod. The problem is that the 0.5% of truly polymorphic methods tend to be defined near the top of the hierarchy but invoked in leaf classes. The process is: look in cache, fail to find, walk from leaf to root, put it in cache, next time do the same thing. The base classes have 2700 methods of which 300 are closed (i.e. <=8) PICs and 30 are open (i.e. >8) PICs.

The solution is to cache all the superclass selector pairs that are tried during the walk from leaf receiver to root. Hence the second time that selector is encountered, the walk is only from the leaf to the nearest shared superclass on the path to root. This reduces the findMethod spike to less than the array copying spike. Real tasks (e.g. the packager) now run twice as fast!!!

A secondary effect is that the VM has done type analysis on your code and so has the information needed by an optimising compiler (the HotSpot Java compiler and the self compiler use this type analysis information). This can free the programmer from the need (in other languages) to provide constraining type information. In this approach, the programmer need provide no type information. A questioner mentioned Anamorphic. They developed subtle techniques in their Smalltalk VM, then tried to move them to Java. There is the problem that these techniques on web servers can make things run slower as you are often in start-up mode, not getting the cache benefits; the solution is to do pre-jittered code. Anamorphic's smalltalk engine did this in 97 but their Java equivalent does not yet do so. (Java has primitive types and sealed methods which both informs the compiler and restricts the coder, so it was hard for them to show an immediate benefit relative to other Java compilers.) This is optimised run-time code generation. The Anamorphic engine is deferring the decision of when to generate case-specific optimised code as late as possible (if 30,000 occurrences of a send is encountered then inline that send).

Typed Smalltalk was an approach of providing concrete type information. It put a load on the programmer and was too general to be useful. Gilad Bracha did the StrongTalk type-expression (more Smalltalk level) type system. This was good for programmers but not useful to optimising compilers because the more abstract expressions it used had to be turned into concrete types. Eliot's hope is that this approach can be combined with using the VM to deduce concrete type information so that the programmer and compiler can both be happy.

Cache CPUs, e.g. TransMeta, are an implementation at the hardware level of these ideas and, for that reason, does not work well with a software implementation of them. An HP paper claims only a 14% increase from heavy hardware use of these techniques so Eliot suspects software is the way to go. Smalltalk has the most advanced work in dynamic type theory and VM research.

We're on the verge of a revolution in how we develop. Cincom will put Store and OpenTalk (which is like a NetMeeting to support remote pair programming; see last year's ESUG talk) on a website and let people use it like a Netmeet server specifically for pair-programming Smalltalk code. The semantic model behind Store will allow storing other Smalltalk dialects. VW is a year from having the ability to point a toolset at itself when the aim is to evolve the tools, and point it at an image when the aim is to use the tools on another dialect.

Dave Simmons remarked that most static type theorists confuse physical structure (and its use in e.g. compilers) with behavioural type. Therefore he warmly agreed with Eliot's separation of the two into distinct complementary modes and epochs. SmallScript (Smalltalk plus some functionally-oriented optional features, see Dave's talk on it below) has an optional type system, mixins, classes are types, classes can uninherit behaviour, type expressions can be reasoned about (e.g. type of first arg == type of second arg therefore use this method implementation), much type inference (little must be declared by the programmer).

In discussion, it was remarked that many people in type research are unhappy with Sun who are not putting their work into Java (ECOOP feedback), Gilad Bracha for one; he still has Strongtalk on his laptop.

Testing and Experience Reports

An eXtreme Programming Project on Infection Scenarios, Joseph Pelrine, Daedalos Consulting

A company wanted to know how much they should charge for a vaccine. They therefore wanted to model infection scenarios. How much will vaccinating a given population save that society in terms of doctors visits, insurance claims, etc.

Infection spreads through a population via infected agents (human, insect, food, ...). Varicella (chicken pox) has human infectious agents; children get it, are infectious for 10 days then are non-seriously ill for 5 days and are thereafter immune. Adults (and very rarely children) who get it may have serious complications. Risk of infection depends on age: infants and adults are less at risk than schoolchildren. As humans are the agents, the more people get infected, the more thereafter get infected until a saturation level is reached, causing peaks and troughs of infection (epidemics). So the normal lifecycle is possible maternal protection -> susceptible -> infectious -> immune. Vaccination moves susceptible people to either complete or partial immunity. Some of the partially immune vaccinatees get 'silent infection'; they don't get the disease but they get 'boosted' by an encounter with the disease to the completely immune state. Others gradually lose their protection over time.

Joseph’s program applies a sequence of operations to his base population to move it on one iteration; change it for age, new arrivals, economics, etc., then recalculate the infectious scenario. Once infected, a human runs through a series of typical symptoms and side effects, expressed as a weighted tree (some visit doctor, some don’t; some get one treatment, some another, ...). Joseph only needs to sum the percentage outcomes those nodes that have effects relevant to his model (hospitalization cost, work days lost through illness cost, ...).

After pre-aging the simulation to get to a reasonably stable state, the vaccination program can be switched on. Vaccination has

- coverage rate: not all of the population are covered
- efficacy: how many get complete protection, how many get partial protection
- catch-up: getting it into the population as fast as possible by vaccinating a range of age groups (e.g. all 5-year-olds and all 13-year-olds for 7 years)
- waning rate: a poor waning rate is a real risk; you’ve protected a child against getting the disease when it’s harmless, so exposed them to the risk of catching it as an adult when it can be much more serious

The mathematics representing the above was simple quantised iterative difference equations with quadratic terms. Over reasonable iterations, these can generate chaotic behaviour that well represents real world effects.

The system was built as a fixed price, fixed duration eXtreme Programming project. The planning game is an early XP stage where developers and customers discuss scope of project: customers tell user stories which the developer estimates ‘cost’ to implement in units. Customer has fixed unit pile and must allocate value to stories. Joseph started with a simple planning game in which he annotated an estimate cost on every story card with a degree of certainty (1-4): certainty 1 means use a load factor of 1.5 times estimate, 2 means 2.2 load factor, 3 means 3.0 load factor, 4 means must do spike solution[1] in one day to better estimate. They had one 4-rated story. After 2 weeks of set up time, Joseph estimated 6 weeks of time, with 2 week iterations (Joseph uses 2 week iterations on all his projects, sometimes even shorter iterations). They made such good progress they had to take a vacation in July while waiting for the customer to produce more data.

- Sometimes Joseph couldn’t manage to pair program with his co-workers. He dislikes not pair programming (sometimes he puts his cat beside him - paw programming) so he pair programmed with the customer, who was smart but knew no Smalltalk; Joseph learned to write very simple clear code and the customer learned to read smalltalk to the point where Joseph could send methods to the customer to explain things.

1. Spike solution example: if must connect to unknown database, implement solution to send one string and one integer to and from that database

- Joseph decided to export the raw results data into Excel for display. This pleased the customer as they had an earlier program for creating reports from data using Excel.
- The customer talked about having it set up for other countries and Joseph said, 'I'll do it when we need it', but the customer went on talking about 'must do it'. So Joseph made a combo box with one drop-down entry for Germany. This kept the customer quiet and saved Joseph much time that would have been wasted because he now realizes that what he has to do for internationalization is quite different from what he thought when it was first mentioned.

The message is: do XP in a way that keeps the customer happy.

Three clinical studies are ongoing to validate the model the system implements. Customer quote: "Let them think we're working in Java. Smalltalk is our secret weapon for competitive advantage."

There is nothing in XP yet about how to coach the customer to be a good XP customer. (This customer was good but some customers have had bad experiences with other software developers and are unwilling to change their ways; sometimes this is so much the case that Joseph decides to refuse their bid.) Joseph specified that he would meet with the customer once a week, which surprised but pleased him. He worked with the customer on the unit tests.

Although it's easy to write a test whose failure will indicate something wrong with the system, it is hard to calculate in detail all the correct results a test should verify when the reason for writing the system is that such data is hard to compute. Joseph started with a test on a simple system with one parameter, computed the correct value by hand and kept the result. Then he added a second parameter, e.g. added effects of death, passed that test and redid first test, which now broke because of the new parameter. Then he calculated the delta for the new parameter's effect on that test (easier than computing the whole) and changed the test to verify the new values. Proceeding in this way as several parameters were added, he built up a suite of detailed result verification tests to complement his other tests.

Smalltalk projects are usually successful. This one (done in VisualAge on Envy) this summer, performs quite well in comparison with an earlier cray simulation; 4-5 years simulated per second.

Advanced eXtreme Programming Testing Techniques in Smalltalk, Joseph Pelrine, Daedalos Consulting AG, Switzerland

Joseph's talk was a prerun of his OOPLSA talk (at OOPSLA it will also have examples - get these from Joseph). Joseph works mostly in VAST, sometimes VW, almost always ENVY. All tools in his talk are available to whoever wants to play with them or port them.

SUnit status: the original article was reprinted in the 'best of Beck' book: 'a sorted collection' (Joseph claimed the original title was 'a sordid collection' :-). SUnit is now being tested in itself and has been taken over by Camp Smalltalk (Sames Schuster). The SUnit engine is now stable on all dialects and can be used for run-time assertions, etc. (Jeff Odell has very good VAST UI for SUnit.) SUnit works well but not well enough.

Model-level vs. view-level: GUI testing is easier now the internet has accustomed people to sub-optimal rapidly-changing interfaces. Any non-trivial application will have significant code in the UI. He used TestMentor on some 'XP-complete' projects and discovered that only a little over half of the code was actually exercised by these XP tests. SUnit is not good at testing window geometry and behaviour (e.g. move lower-left corner and check all contained widgets stay same size and relative location) tests, at testing interwidget synchronisation (e.g. only enable O.K. button if something selected in list widget - is this model code or view code?), and at testing model-view communication (is the stuff being entered really reaching the model and is the model really being shown in the view?). We also need to test validation of user input data (e.g. 30foo99 is not a date, someone born yesterday cannot be the parent of someone born 20 years ago, etc.). Some rules of thumb are

- if there is latency between view and model, test model
- if there are multiple views, test model
- if a high-degree of control needed, test model
- if the model and view are highly interdependent, test the view
- for deep object verification, test the model

Question: is the controller where these test-GUI-or-test-model issues belongs? Answer: perhaps, depending on how you define the purpose of the controller. You could simply define the controller as the place where this stuff lives (c.f Rebecca Wurfs-Brock coordinator versus controller definitions).

Test Resources: instantiating test objects can be expensive in performance; database connections, building complex objects. We don't want to build these up and tear them down for each test case; this violates XP which requires that the feedback loop be short. However keeping a test object alive over multiple test cases breaks one of the primary rules of testing, but is nevertheless desirable. Hence Joseph et al. have developed TestResources. The TestResource class (to be released in a week or so as part of SUnit 3.0) is an optional singleton (you can have more than one if you manage them yourself). It's polymorphic with setUp and tearDown. All resources are initialized before TestSuite runs (and saves you running 60 sure-to-fail tests if they don't initialize). TestCases optionally (preferably) define required resources (TestCase class>>resources).

When do you release a TestResource? There is no correct time so the user decides. The default is that TestRunner resets each resource. Alternates are TimedReleaseTestResource and ManualReleaseTestResource. (If you're in the debugger you must proceed, not quit, for SUnit to catch the releasing of resources.)

In future, this test resources mechanism will also handle initialization order, and conditional initialization dependent upon a pre-run TestCase (e.g. run TestCase to check database connection before database connection resource initialized).

Performance testing: the simplest thing that can possibly work is

```
self assert: ((Time millisecondsToRun: [...]) < limit)
```

but most Smalltalk dialects have profilers which can be called from SUnit.

Testing the stripped, packaged image: you have to test deliverability of software as well as functionality. Joseph insists his developers load their code into a fresh image every day. The rule is: package early, package often. Experience the process and its bottlenecks early. For ENVY, SUnit can include class-based prerequisite checks and Joseph has produced an extension to SUnit that does this (PrereqCheckTestCase). There is a problem with detecting method-level dependencies when extending a class defined in appA to appB and appC. B and C have A as prereq but if C calls a method defined in B, the need for C to prerequisite B cannot be caught. Refactoring prerequisites in Envy can be difficult since it is not simple to remove an unneeded prereq if one if *its* prereqs *is* needed. (Joseph demoed how to do this in the Application Editions Browser.)

SmallInt has many code quality tests: methods sent but not implemented, methods sent but not implemented in application. Joseph's SUnit extension tests for these. SUnit can also be extended to run the VA packager but he does not recommend this routinely as it is too slow (5 minutes) so only do it when integrating. (He has a set package timer utility to make the packager take however long he wants to step out of the office :-)

The hardest thing in XP is writing a test case. Joseph's heuristic is: only write a test case that you know will fail. However, it can also capture domain knowledge, e.g. of relationships between projects. When working with other people, writing tests before coding is interesting because sometimes, thanks to what someone else wrote last night, the test works!!!

Some test cases are hard to write because it's hard to think about the simplest thing that can possibly work; the Sherlock Homes principle: eliminate everything that is impossible.

Silvermark's TestMentor is a big tool with a big learning curve but it is the best tool for many tasks. It's totally integrated with the environment so one can get events or other test data at any level. It also allows reuse of test steps. It handles external resources well, e.g. compare file to massaged data from program. It lets you do regression testing - compare current and past results of tests (N.B. this can cause ENVY repository bloat). Other useful tools are Refactoring Browser's SmallInt, VAAssistPro (VA only) and Joseph et al.'s personal tools (see the Mastering/Envy Developer book).

People asked about patterns for writing tests; someone did some work in EuroPloP (see EuroPlop webpage).

New Product Directions

The Smallscript Language and Platforms, Dave Simmons, QKS, U.S.

This talk was an introduction and overview to the QKS Smalltalk-2000 dialect and its support of the Microsoft .NET Platform. Dave started by reading an excerpt about binding time from ‘Machine language and machine organisations’ pub 1968. One may bind fixed attributes at structure creation time and bind varying attributes at their moment of computation; bind early and bind late strategies. The book discussed whether templates (classes) should themselves be bound early or late. These ideas are where Smalltalk came from.

Dave trained as an electrical engineer, worked on what later became the internet, real-time data acquisition in labs, fortran compiler, etc. The book on Objective-C changed his whole concept of computing. Dave founded Quasar Knowledge Systems, surveyed OO and chose Smalltalk. He was not deeply acquainted with existing smalltalk implementations and therefore focused on different issues. QKS produced Smalltalk Agents for Mac but Mac had problems in 1995/6 so they changed direction.

Dave became unsatisfied with the large monolithic nature of Smalltalk (minimum app 5MB). So he revectored his work in 1998 to a small, always-bootstrappable scripting language. There are three styles of scripting:‘

- unix shell
- mac
- web

Dave focused on how to rearchitect Smalltalk as a web scripting language for people who were not Smalltalk programmers (or programmers at all).

Dave wants Smalltalk to be able to be used as a classic language. Popular languages let you write 30 lines of code in a text editor and evaluate it. They let you call other languages easily. So we need text-based development with XML source and transparent cross-language linkages.

Dave needs it to have a small footprint when put inside larger applications, with declarative grammar, syntax ‘sugar’ extensions and transparency to COM/CORBA. He wants an optional typing system that supported multi-method dispatch. Smalltalk uses double-dispatch e.g. for numerical methods but this becomes complicated when users start adding their own types. Multi-method dispatch and mixins are better.

In late 1999, Microsoft contacted QKS about COM+2.0, now known as MS.NET. MS.NET is a cross-language VM; VC++, VB, JScript (echmaScript), C# (Cool), scheme, etc., (no comment on Java). It supports multiple CPU and OS targets including embedded devices. It has a standard ‘common’ object model, a rich package-loader and metadata system and a detailed security model. It aims to replace COM, not be a layer on top of it.

Its distribution is based on XML/SOAP. Demos speak of powerful applications built with little coding. All this became public in July (Florida PDC). QKS wrote the part that handles dynamic languages.

Dave demoed VisualStudio.NET (alpha release of latest version): C# interoperating with Smalltalk under a common language model and VM. The demo freely mixed C# code and Smalltalk code, the latter using Smalltalk reflection. Using dynamic technology, Dave can add methods to their 'sealed' class libraries (to the surprise of the Microsoft people). Behind the scenes, the demo took the C# file to a .exe file, the Smallscript file to a .dll. The former runs on the .NET runtime core library, while the Smallscript runs on their dynamic language support library, both of which run in the .NET platform VM. The Smallscript can also run on the QKS' AOS (Agents Object System) VM. (Smallscript will also target the SUN JVM platform, since Dave has already solved the similar, albeit easier, problems of doing this in the .NET platform.)

.NET runs now on X86 win32, and soon on WinCE and embedded devices. There is no official story for Linux, Mac, PalmOS, Solaris. A Mac version will appear if MS Office needs it (and not otherwise). The AOS platform runs on X86 win32, Linux, freeBSD, BeOS. It will run on Power PC Mac OS-X. They are interested in WinCE, Sony Playstation, PalmOS and Solaris. The basic features will be free.

AOS version 4 features: no image file is required. It has

- an intrinsic loader and XML parser
- first class dynamic type system. This is very important for language inter-operation and rearchitecting frameworks to make them smaller and more versatile. A key idea is to separate behavioural type (important for human beings) from structural type (important to the compiler, not to humans)
- rich meta-object services (read, write, bind delegation - e.g. doesNotUnderstand proxy pattern) and object management protocol (set object readNotification: aBool, bindNotification: aBool, ...)
- pluggable architecture for VM extension (via C++ or AOS methods; you can write smalltalk classes as C++ objects)
- intrinsic class library: minimal set to support Smalltalk adequately, with a modular package/link/load/export system, so this Smalltalk brings very little compulsory baggage with it; you only get what you need
- multi-threaded: a long-standing QKS concern
- high-performance FFI and namespace-scoped multi-method dispatch: this allows very performant numeric processing
- Selector aliasing: for example, C# will not let you write the colon of a selector (e.g. myMethod:) so you show it as e.g. myMethod_ in C#. Another need is when different class libraries use the same selector name: you need to alias these names to use both in the same code.

The AOS platform lets you write Smalltalk in the usual dynamic way. (Currently .NET is somewhat resistant to this. This, and almost every problem Dave has met relates to the design of the garbage collector, whose designer, clever though he is, was unfamiliar with dynamic languages.)

AOS has a unified object model. All objects have extensible properties sets, and a resizable number of by-reference named slots and indexed slots, by-value byte storage and (for full international support) unicode character elements. The class system is built on top of an instance-specific behaviour system. The byte storage is just structured memory access, so anything (e.g. C++ objects, native heap, virtual, etc.) can be referenced by a Smalltalk object and can be morphed (e.g. from a C++ object to the heap to Smalltalk) in real-time. Objects are fully resizable and restructurable and the VM is optimised for heavy use of this feature (thus, for example, a collection just adds slots as it has to grow, instead of being created at a given size, getting full and then copying to new collection with empty space, plus any object can be viewed a stream of bytes, strings or whatever). Weak refs, initialization, finalization and full lifecycle management are intrinsically supported so programmers do not have to worry about them. The standard tag bits mechanism combines this general picture with more compactly-stored SmallIntegers, Characters and WeakPointers to Objects.

The AOS IDE (available free from their web site) has a web-like browser (with web support). The compiler is fully XML-text-aware (makes it easy to author web pages) and has a large graphics library. Scripts are like workspaces. It will be refactored to be leaner and less monolithic, in line with Dave's evolving philosophy. (They have licence to ship Virtual Studio with their IDE so can support writing C++ and linking to Smalltalk e.g. for heavy numeric processing.)

Most of their version 4 VM was written in or generated from Smalltalk, with some non-Smalltalk core elements, using their IDE. Dave is determined to keep the VM and bootstrapped core image below 500k. After failing to type in the correct command (the usual demo hiccough - Eliot: 'that's why we don't use command lines'), Dave just changed the virtual machine to invoke computation of 10,000 factorial, to show their impressive speed of numeric calculation. Their V4 is a JIT compiler with few further optimisations currently. It is much faster than the .NET platform; they will use typing optimisations (c.f. ideas in Eliot's talk) to avoid costly calls in .NET.

To offer something comprehensible to non-Smalltalkers, they start with

```
self initializePlatform.
self main
```

All Smalltalk methods can be called in C++ syntax as well as Smalltalk syntax. Dave sees this initial accessibility to those unfamiliar with Smalltalk dialect as important to make the out-of-the-box experience easy. Smalltalk offers too many new things to learn at once; those who learn one of Smalltalk's many advantages will be ready to learn others.

Smallscript uses an XML compiler to parse the code separately from compiling the methods, thus supporting other languages. Dave then demoed a C# class subclassing a Smalltalk class. (The current .NET platform loads slowly, like many other windows facilities, because .dll loading is inefficient; Office 2000 cheats and loads many of its .dlls at boot time to load fast.)

In Smallscript, a class is also a namespace and so can have functions defined in that namespace. The XML defines various standard Smalltalk and specific properties of a class. Specific concepts include modules (for deployment) and assemblies, StyledStrings with implicit concatenation, method signatures (types). There are various things that let you write Smalltalk in other-language style:

```
() has the same effect as value
(aParam) has the same effect as value: aParam
:binarySelector= allows i+=, i-=, i*=
```

etc. Ideally the IDE would be able to parse such things away (but of course if a user has laid out comments interspersed with code they may not be correctly laid out in the re-parsed syntax).

Smallscript provides many annotations for explicit other-language support. These are not Smalltalk (but of course they can be done in Smalltalk, e.g. break to label can be effected by thisContext pc: label).

Smallscript VM and documentation may be available after November.

Dolphin Overview, Blair McGlashan and Andy Bower, Object Arts
Dolphin aims to improve on Visual Smalltalk. It began focused on PC and small size and will still run on small machines. Dolphin 4.0 supports ActiveX and better code export.

Dolphin has a very visual style for the launcher, class hierarchy browser, visual composer, etc., but with good linkage to workspaces to mix visual and programmatic control. (Subject to a minor hiccough, as per the usual law of demos) Andy quickly edited the class browser to use a different view. They have good drag and drop (e.g. drag methods to workspaces to edit). The protocol browser (protocols are hierarchic) can be used to polymorph e.g. collection protocol on other classes, plus it will detect mismatches and/or generate stubs. You can also test conformance to protocols at runtime.

They use a Model-View-Presenter variant of MVC with much support for building and managing a range of views on a model. Unlike e.g. Visual Basic’s flat hierarchy and dialogs for every component type, they support a deep hierarchy of visual components and being able to control these components from workspaces provided for them.

The package format is now source, not binary, so editable and can be put in a configuration control system such as CVS. They find this a usable scheme for up to 6 developers. Line-based control systems work much better for Smalltalk than people think, provided you preserve your file-out order. (There is also an ENVY-like utility using an OODB someone wrote.)

Dolphin Lagoon is their deployment image. In Dolphin v3 their smallest image was 1.5MB (from 4MB development). In Dolphin v4, they can get the same image down to 540k. (The stripping progress bar measures the number of objects left in the system, thus completes before tracking all across the screen).

Until last year, most Dolphin sales were to hobbyists. Since then, they've seen a lot more companies buying the full system. In Florida, it is used to administer medication to monitored patients. The Zagreb stock exchange is run using a Smalltalk/X server and Dolphin clients. A medical system in Japan uses it. A petroleum distribution system is using it. Etc.

Andy then demonstrated registering ActiveX components, generating classes for accessing the underlying COM objects, examining ActiveX control components, etc. You can use an ActiveX component either by generating wrappers or via an i-dispatch (doesNotUnderstand trick) mechanism (less performant, quicker to code).

Lastly, Andy showed their plugIn. Binary packages can be used for dynamic update of the system (as is done in the Zagreb stock exchange) or in the plugIn. He demoed interactive applets, communicating applets, certificates, applets communicating with other users, applets driving avatars, etc. The dolphin class browser, etc., will run inside a net browser thus making debugging much easier.

Dolphin is an interpreted solution, not a JIT. It is probably 2.5 times slower than VW for most things (though not for arithmetic) but it has good overall performance because it is not doing all the widget emulation of the other Smalltalks. They're thinking about how to work with .NET. They don't use the VM as a portability mechanism and are very tied into windows. However they do want to do something with .NET, and would like to support the Windows CE machines, so are looking to make the VM somewhat more portable. SmalltalkMT is much faster than Dolphin but has a very barebones framework.

The number of Java vendors has dropped significantly in the last couple of years because their all playing in the same space. Smalltalk is diverse enough to support 6 vendors, plus the Smalltalk-on-Java systems, plus a massive open-source activity (Squeak).

Model-View-Presenter: twisting the triad, Blair McGlashan and Andy Bower, Object Arts

In 1984, Andy wrote Intuitive Solution, written in C. After 3 months of finding every new function broke an earlier one, they read the blue book, implemented an OO framework from it on C and thus completed the project. In 1988. they rewrote it in C++. By 1995, they needed a new solution to confront emerging rivals. Their major customers were insurance companies doing fact-finds on desktops. Andy had often tried to learn Smalltalk and failed (because he never used it for real but just did the Digitalk tutorial). When Blair joined the company, Andy told him to learn it. After two days Blair said, 'This Smalltalk is <deleted>'. After a week, he said, 'This Smalltalk, it's got some good bits.' After two weeks, he said, 'This Smalltalk, it's the <deleted> !!!' So then Andy learned Smalltalk. Dolphins have been around for a long time but we've only recently discovered how intelligent they are and this is Andy's opinion about Smalltalk, hence the name: Dolphin Smalltalk.

They started by making it widget-based, influenced by people's seeming to like visual basic, windows dialogs and MFC. However, all these were not hierarchical; the could not compose and reuse widgets (or not at all easily). Thus Dolphin's first widgets were UI components where data, display and behaviour were combined in a single entity, with a workspace for smalltalk commands, and visual composition of widgets with the tool writing the required subcomposition commands. The result was an easy to draw UI first time but poor reuse. Their workspace widget was an example; they used text edit then decided they wanted a rich text edit and had to refactor (wanted multiple inheritance).

After 3 months, they were refactoring all over the place and dissatisfied, so they bought VisualWorks and studied model-view-controller. MVC is good for pluggable widgets but poor for composite widgets. Hence VisualWorks introduced ApplicationModel to coordinate composition. Another 3 months were spent building the MVC framework (tricky to add new UI framework to Smalltalk as your browsers must be migrated from the old framework to the new as you rewrite it). They found there were anomalies because it breaks the observer pattern, leading to the componentAt: problem; if you want to change something in one view, you either generate 'spurious' updates to influence the appropriate view, or have the ApplicationModel know which view it must address. There was also the issue that the plug-and-play controllers were inappropriate for modern event-driven OS. Then Dave Simmons showed them the Taligent Model-View-Presenter pattern (written in C++). It's a triad, like MVC, but the Presenter and View are hardcoded to know about each other, while the model knows nothing about its presenter. In MVC, controllers did low level handling (e.g. convert mouse movement into mouse tracking).

[Dave Simmons worked on Taligent. Its origins were in Mac and Smalltalk. Its designer had worked with Smalltalk and took his problems with MVC to this work; he used C++ because they thought they needed its performance at that time. The Mac's of that time already had event-driven OS. Thus the Talogent design was motivated by just those considerations that led ObjectArts to it, compromised to fit C++.]

MVP components are hierarchical and based around presenters. Views are the presenters' skins; they are hierarchical independently of the presenter hierarchy and one can mutate views on a presenter. Typically, Models and

Presenters are class-based but views are instances built inside the ViewComposer and saved as resources. Sometime people wish Views were class-based, so they could be source controlled, etc., but Andy felt that composing views was not really adding new behaviour. Only when a view truly needs behaviour do you need a view class; usually, specific UI behaviour is attached to the presenter. (Each class can publish aspects, i.e. properties, on its class and instance side; unskilled users can then configure these.) One can replace views at run-time without shutting down the Presenter.

MVC uses dependency to implement the observer pattern but they preferred to use events (SelfAddressedStampedEnvelope pattern, #when:send:to:), not least because you then don't have to release dependents; you can rely on finalization to release. This is particularly useful because Dolphin uses native widgets so unreleased windows are tying up windows resources. (They still have MVC in Dolphin Smalltalk; they just don't use it in their GUI framework.)

VisualBasic has VBX, Com has ActiveX, Java has Beans, they're looking for a name for what Dolphin has ('Balls' is their current working name; alternatives solicited :-). Dolphin has publishing and instance streaming because windows widgets are not designed to be dynamic (some of their properties can only be set on creation) so an apparent Smalltalk edit may conceal a windows recreate.

Their recommended process is first to build the model, then the UI. Is the model mutable or immutable? Immutable models need value components (e.g. if you display a string, will an edit replace it with a new string or change the internals of the existing string?). Is the component basic (implement view class and presenter class then install view instance as named resource on presenter) or composite, the more usual case (implement presenter class, draw composite view, install on presenter)? Composites are slightly heavier-weight so you use basic components when they are adequate; you could build all components as composites down to a very basic level if one wanted.

Andy demoed a basic component: the scribble pad. Its model is a list of inkstrokes (lines). The Inkstroke class can draw itself on a canvas, provide its data, etc. View subclasses to ScribbleView to draw these inkstrokes, repainting the entire view whenever the whole list is changed but only what is changed when an item is added to the list. Then he installed a new view for the scribble view using the visual composer; designed view is attached to class. Suitable methods (e.g. onLeftButtonPressed:) were implemented to create inkstrokes in response to user events (these methods used MouseTracking and other classes implemented to provide a high-level abstractions of windows components).

He then demoed building a composite component: Etch-a-Sketch.

- The model was a list of InkStrokes as before

- a #createComponents method added the two knobs and scribble pad that will make the Etch-a-Sketch
- #createSchematicWiring maps the component events to overall behaviour methods
- model: builds the composite's model (easy as it's just the same model as the scribble pad).

He then graphically arranged the components in the view composer, saved the result and installed it on the presenter. View components are connected to the presenter by name (stubbed out views can be given 'deaf objects' that just return themselves on doesNotUnderstand to ignore events). Then he attached an additional view to the model (one that displayed an avatar commenting on what was drawn :-).

A typical refactoring is to build a composite shell component and then refactor it as an embeddable component as well as a shell because you realize you want to reuse it. Views are rectangular albeit one may attach them to presenters in ways that achieve non-rectangular events.

Dolphin has the parent manage layout of children when it resizes, subject to each child's arrangement policy. (Dave Simmon's system, by contrast, delegates re-layout to each child, controlled by that child's policy. It also uses the latency of finalization to cache resources; cache discarded resource so if user requests same resource soon after, you may be able to recover an instance of it that has not yet been finalized from the cache.)

Menus are somewhat irregular under windows. They have a static menu editor; dynamic menus must be code-built.

Components can be exported as binary and run in their web plugIn. Unlike Java, where one must inherit from Applet to be an applet, any component will work in the plugIn. They are comparable in size to Java components and fast to start up (but of course, only work on windows machines).

Future directions:

- MVP could display a schematic component logic presenter, wiring the events between presenters, to let developers understand the logic of their applications.
- Models and Presenters are written in ANSI Smalltalk, so portable to other Smalltalks; if the relevant views were implemented on other platforms, an application could be ported (e.g. the refactoring browser). This could point towards a general Smalltalk use, possibly open-source; ObjectArts have to think about this.

In summary, they believe MVP is the ideal modern Observer framework, It improves MVC, keeping its virtues and losing its disadvantages. Particularly, it is more suitable for event-driven OS.

Squeak Demos, Jeff Sarkela, Exobox, U.S.
Morphs are shapes. One may choose any one of a very large number as a basis for work, then rotate, scale, translate, associate sounds, compose them transitively (e.g. text flows from morph to morph), etc. A very large selection of Grab handles let you control them in many ways. Morphs are prototype-based; make new ones by cloning existing morphs from lists, from thumbnails in tools drawer (turn on handles to see tools drawer), etc.

One may put any morph on the plane of another morph, e.g. put a browser on the screen of a PC in a 3D picture and play with the browser (slow but cool). Phoneme recognisers can make lips move as words are spoken, etc. (John showed a basic one and described an excellent one written in Argentina.) Tools let you record sound, synthesize and edit it, analyse it, etc.

Stephane Ducasse has a long web page about what grab handles do, etc., which he will post.

Lastly, a Squeak peculiarity is that the backarrow <- can be used for assignment := by anyone who lacks the appropriate keyset.

Stable Squeak: the Squeak World Tour, John Sarkela, Exobox, U.S.
Ralph Johnston taught his OO design course using XP and Squeak for the first time this year. Jeff Gardela used it in a start-up this year and found it was superb but not production-ready yet. The Squeak world tour aims to fix this; they want a production quality Smalltalk system. There are far bigger enemies out there than other Smalltalks, so we must extend the spirit of Camp Smalltalk. Squeak is a self-hosting VM, written in Smalltalk with a Smalltalk-to-C translator, direct object pointers, an incremental garbage collector and dynamically-loaded named primitives. The base image includes a web server, browser email client, chat, ftp, telnet, etc. It has very rich sound and graphics support: sound synthesis, KLATT speech, music description, ..., plus 3D engine, VRML, Morphic., 3D scripting, ...

In Disney, they've built a game OceanBlast and are advertising in Disney through portal go.com, plus there is a plugIn for NetScape and Internet Explorer. The Disney VR group have a 3D game engine with avatars, pluggable physics, etc., and Squeak is plugged into this.

Ralph is teaching XP using Squeak. Squeak is good for education (free, runs on all platforms, has Freecell) and great for embedded devices because it's written in itself and runtimes can be made very small. It has lots of reusable functionality for developers. The UIUC summer OO design course used Squeak and XP to build a functional object swiki in four weeks with 6 programmers who also learned Smalltalk at the same time.

However, so many things in Squeak 'almost work' in the sense that they work but they are not production quality. The Squeak world tour is avoiding being Disney-centric because Disney, like Squeak's inventors, are experimental and the Squeak world tour is about sorting this out.

Squeak needs a production quality base library. The core team is more interested in experimentation. Squeak may be many people's first encounter with Smalltalk. Most of Smalltalk's problems are not technical and not lack of success stories; they're about the out-of-the-box experience being too hard to let people explore Smalltalk further. (Someone sees something superb and get motivated to look. They download Squeak and fire up. They spend a few minutes looking and can't make progress. They get demotivated again.)

The solution is to use Camp Smalltalk style development, bringing the camp to developers wherever they may live, working with Squeak central (who are very keen to see this done) to incorporate all refinements back into the base system. The current plan is to create a minimal development image and refactor it until all methods can be compiled from source code, with no undeclared references, etc. Then factor the remaining functionality into modules with no method or class redefinitions, a well-defined module dependency lattice and as many unit tests as possible. Then refactor the base into a headless image able to bind image segments (no dependence on compiler) with a set of bindable UIs (plus text-based stdin, stdout, stderr). One goal is to create something usable by all those skilled programmers that want to use emacs, c.f. Dave Simmons talk. Another is to decouple code from VM so people can pull stuff out of Store and put it on whatever platform has the VM support wanted.

Steve Wessels has a skins framework that loads on top of Squeak that makes it look like motif, windows, or whatever (some issues of compatibility). This should be made available.

The ANSI process is too slow, and doesn't specify the important parts of Smalltalk (which is how to specify frameworks that are reusable by inheritance) albeit it does provide useful conformance tests. ANSI is not a medium for defining something like Smalltalk which is a seamless integration of a language and an environment (class library, etc.). What is needed is a proper and faster standardisation process that can standardise optional things (*if* you include them *then* they must be done like this) and which ANSI just eventually rubberstamps. Camp Smalltalk, with its XP testing style, may show the way forward. So you can have a Pocket Smalltalk without reflection or exceptions but when you move to an environment that has them then they are done in a standard way. A grand semantic model of the whole language is not the goal, albeit a model of how these optional elements can be configured might be useful.

The first camp smalltalk had 50 - 60 attendees, invitation only. Leadership is important, leaders should be known and they should be able to manage their time. A little leadership early gets a project off the ground (some Camp Smalltalk projects did much better than others for this reason.)

Start-up and shutdown patterns needed to register many cross-references that are currently hard-coded in methods. Many refs to isMorphic need to become refs to 'the currently loaded display mechanism' and so on.

Some issues are not part of stable Squeak's immediate needs but are of interest. Roel argued for full block closures; it's been said to be only a few days work. Andy mentioned internationalisation; Oshimason (Squeak on hand-held) has done work on this.

Smalltalk scripting language fo r VisualAge, Matt Sims, Totally Objects, U.K.

Matt Sims works for British Telecom as an e-commerce security consultant. However his passion is Smalltalk. Totally Objects is a 2 part-time person company (Matt and his father-in-law) providing Smalltalk consultancy, building end-user systems and frameworks/tools. Their main strengths are in banking and internet.

They had to build tools for persistence, printing, etc., where they could not afford or could not find them:

- Registry gives you smalltalk access to your windows registry
- Toolbar does windows 95 look and feel
- Socket set extends VAST sockets, etc.

They often make more money selling these cheap tools than from the projects that originally prompted their construction.

Their product 'Readability' formats Smalltalk code and gave him parsing experience, including the knowledge that he needed to learn how to write a parser. The speaker therefore wrote another better parser. Until recently (v5.5), VAST did not let you ship the compiler, so he used his parser to execute code in run-time images (run-time execution of code is his definition of a script).

Another issue is that the VAST serializer cannot serialize blocks and other executable objects (e.g. SortedCollection contains a block). So he converted his parser to an interpreter (one hour's work). He then separated the map between the source code and the executables in the image so that he could vary the mapping.

If you let people run scripts, how do you prevent the customer writing harmful scripts as they have access to the whole image. The quick and dirty solution is to use cryptography so the image checked that the script came from a trusted source.

A better solution is sandboxes. He experimented with a changed mapping between source code and executables that let him dynamically specify whether a class is visible, which of its methods are visible and any aliases of its method names, with protocol between the script writer and the interpreter so that only permitted methods get invoked. His first approach was very hard to configure but he managed to simplify it.

He then walked through using his scripter to patch a method in a running image.

His end-users cannot write Smalltalk but they can write familiar BASIC statements. Hence he is writing a BASIC syntax parser that accesses Smalltalk objects using BASIC syntax. (He had to provide a default stream for printing, restrict ‘open’ statements, etc.)

[Matt Sims knew of someone called David at the Open University who wrote something to show what Smalltalk code looked like in other languages such as PASCAL. David is nothing to do with the m206 course.]

Web, Multimedia and Cooperative Working

Internet application development using a meta-repository, Michel Tilman, Senior System Architect, Unisys, Belgium

Michel’s internet framework piggybacks on the meta-repository he has been working on for the last five years (see last year’s talk). Their requirement was for a configurable flexible adaptable and end-user programmable[1] system with no hard-coding of model and business rules. Instead it has high-level domain-specific languages captured as meta-data.

In client-server mode, the framework has a dynamic object model; when you change the model, the system changes its behaviour. The model defines objects, states, events and conditions under which an object changes state. Their goal is to support databases serving electronic documents and workflow over the internet. Their client is the Belgian public school system. Various applications share a common business model. The meta-model is modelled, stored and edited in terms of itself. A single repository contains the data and the meta-data.

- An application defines or extends an object model with classes, associations and basic constraints.
- The application environment is defined as a set of views on the shared model.

These two tasks done, you immediately have a fully operational application with default screens, etc. You then add business rules (authorisations, user-defined constraints, etc.)

Michel then demoed what was involved in configuring an application (he did not actually do the full configuration for time reasons). They have a form-based model editor (he said it was not yet complete but, to me, the demo seemed to show a good deal). Form views can navigate through long chains constructing a form that views data from a root and from related objects, even distant ones.

A class has a list of properties, not instvars (in principal a dictionary, like our properties dictionary in our meta-programming system, but Michel actually implements it as an array). Specific smalltalk code links to these properties by clever use of doesNotUnderstand (see Michel’s other talk; as it happens, doesNotUnderstand is now very optimised in VW5i.3 via doesNotUnderstand caching - see Eliot Miranda’s talk).

1. This assumes the end-user has appropriate domain knowledge and skill.

Reusing this for the internet, he (for example) can represent this data as a structure (e.g a hierarchic structure) and attach to each node a web-form-associated query that returns another web-form-associated query. Michel demonstrated on a VisualWave server (works on others). The structure was shown in one pane and selecting a node showed its query in another. Running the query returns another query (includes just returning data) which was displayed in a window.

Lastly, Michel reviewed the design. There is a 4-tier architecture

- partial use of VisualWave: session management and HTTP/CGI interface (did not use all as disagreed with some VisualWave aspects)
- Session resolver: identifies application id and message (registered with typed arguments).
- Application model: there is a main application id (root) and subcomponents with their navigational access paths
- View: generates the web page; HTML / Javascript plus an XML document builder

He achieved in 5 months what a Java group took 40 months to fail to achieve. In future, he will look at bookmarks (currently he regenerates session context so you can't), at using SOAP-compatible message protocol, at Applets and at true XML documents which merge the results of a query with a description of the document within which the result will be displayed. XML is about key-value pairs (which is what meta-data is) and style sheets which say how to display particular types.

Smalltalk needs to formalize the metamodel the way CLOS has done. In CLOS, you can choose a dictionary representation of how the class' instvars are held and then the compiler maps them to

```
instvarDict at: #instvarname
```

invisibly, without the user having to worry about it when referring to the instvar in their code. Michel overrides doesNotUnderstand both to achieve a similar effect (accessor method not understood so look for its selector in properties dictionary keys) and to construct SQL for queries to their relational database.

Squeak MultiMedia Projects, Mathieu van Echtelt, Cosmocows, Holland

The speaker got involved in Smalltalk in 1995 in university and then in Soops (20 person Dutch company using VW). Then he studied Squeak while doing a Ph.D. He made a prototype in Squeak and showed it to a Dutch T.V. company that was experimenting with shows that the audience could manipulate over the internet as the show was being broadcast.

He then showed a film about the Typeotoons show (for children aged 8 - 15). Scenarios are draw up by children interacting with the author (fill in blanks style) and translated into 2.5D OR 3D. It is managed as a game in which the winners are those children whose proposed words and plot ideas are most often accepted by the author. Animated graphics, based on ideas supplied by the children, are a key feature of the programme. The short production times mean they must have a very productive environment. There are three platforms; that used by the initial group of children (50 children), that used in the studio (4 winning children from last week who must find last word of story) and that used by the broadcast group (show programme and select word for next week's story). The letters are given behaviours - Y can jump high, Z is strong, X must have sound, etc. - so the tasks influence which letters are chosen.

The speaker proposed a project for an InternetInvolver. The Typeotoons current environment is not well integrated, the children's environment is not as well animated as the TV show, and they took 3 years to reach their current point. The InternetInvolver will be a morphic browser instead of an HTML browser. After preliminary discussions, the project took off three months ago and Cosmocows was set up. Their job will be to implement as much functionality in Squeak as they can. Other partners are Dutch state TV, Gronigen University, Ijsfontein. They have designed a tool tower of all the tools they need and how they relate:

- squeak VM and class libraries
- coding tools (class and refactoring browser)
- factory tools: teamwork tool, packaging tools, sunit, modelling tools, performance analysis tools
- work environments: client-server, financial, games, web-server, database, connectivity
- user products: internet involver

CosmoCows will work primarily in the last two items, trusting the Squeak community to produce the first three. Perhaps Squeak can be made more accessible by identifying roles: visitor, player/user, tweaker, programmer. The less educated roles need appropriate help and instructions. Cosmocows will have to train non-Smalltalk programmers, and train non-programmers in Alice (the Squeak scripting language), etc., so may do something about documentation.

Jeff Sarkela and Roel Wuyts recommended that the speaker talk to and visit Squeak central, and offered 'letters of introduction'; education is one of Alan Kay's interests. Jeff recommended codifying the morph logics as XP tests and thus approaching discovery of what the InternetInvolver was to be.

Cosmocows built a demo in squeak, found it was too slow (took 9 seconds) and asked the mailing list. Within three hours they had several responses which gave them an order-of-magnitude speed-up. This example helped them persuade the TV company that open source was the way to go.

COAST, Jan Shümmer, Till Schümmer, Christian Schuckmann, GMD-IPSI / Intelligent Views, Germany

COAST is an open source Smalltalk framework for building synchronous collaborative applications. Its developers want to support building complex OO applications, including hypermedia. There are many problems that can arise (and that NetMeeting often encounters) in collaborative applications. There are requirements both on the application (group awareness, coupling control, session management, floor control) and the framework (usability, right level of abstraction, uniform approach, reusable). COAST provides a reference architecture, ready-to-use components, a domain model template and an implementation that hides as much of the dirty details as possible.

The speakers demoed (on two screens) a UML editing application that displayed whatever Smalltalk classes you gave it. Simultaneously (in contrast to NetMeeting), one user could expand a class to show subclasses while another could make the same class show its attributes. Windows on the canvas showed which areas were visible to which users. Smalltalk code in inspectors let them move elements, open notifiers, etc.

Single-user applications use Model-View-Controller. The V and C are tightly coupled to the M and so one often separates the M into the ApplicationModel or ValueModel holding the application logic, and the domain Model holding the business logic. One collaboration approach would be to share the domain model and attach multiple application models to it. Here, the applications are distinct but their data is the same; there is no collaboration awareness. Another approach is to share the application model as well, leaving only the VC parts distinct. Application state can be accessed from each application instance. COAST uses this second approach. (In the demo, colour coding distinguished M, AM and VC items; they are thinking of using this generally.)

Session management: COAST maintains a relationship between the shared application models and the users sharing it. When a user is added to an application model, it creates a VC pair on their machine for that application. You can programmatically control which users are interested in which applications. Applications are hierarchically oriented so users can share different sub-parts of an overall application. COAST also supports different coupling modes, e.g. letting users share the same scrolling position or have different scrolling positions. Labelled screens tell other users where a given user is in these less-coupled modes.

The above is achieved by an elaborate layered architecture of clients connected over IP to mediators and stores. The COAST server can run from a client but it is better to run it in its own image (more reliable). Shared objects are modelled as slots and frames. Concurrency control is based on units called cells; cells are assembled to build more complex units. COAST has its own marshaller which converts cells to bytestreams. Model building starts from a prototype class. Actual classes acquire slots as they are filled. They also model inverse relationships and other constraints such as typing (there is some type checking). At present, these are captured in elegant smalltalk initialization scripts. Later, their UML editor will able to input these by drawing them; at present, it just displays them.

Shared objects are bundled in clusters; an unloaded object is loaded when referenced, along with all others in its cluster. Clustering can be done by the application programmer or be controlled by a clustering policy (at present, there is just one policy; put object into cluster of first referring object). At present, they have a simple persistence mechanism of files; later they may use a database. Changes to shared objects are done in transactions (#transactionDo: and similar methods). Transactions are pessimistic or optimistic (may rollback). Transactions are processed locally and then globally. Only changes that use the framework elements are captured (so operating on a string directly instead of via MVC might not be caught).

Views have virtual slots that trigger redisplay (e.g. of a graphic object's bounds). Virtual slots are computed on demand (lazy) or on invalidation (eager); for example, if the bounds of an object depend on the length of its name then the bounds computation may depend on its domain model object(s) as well as on application model objects. This constraint mechanism keeps the display consistent and is integrated into the transaction scheme (you don't want to update the display within a transaction). If a slot is changed, a dependent virtual slot is invalidated which in turn invalidates the view, thus ensuring update on transaction close.

They have built several applications:

- learning environment
- process support system: simulate processes and play through them
- the Beach project is about building computers into walls and furniture; COAST's less coupled modes lets them suit different furniture types
- entertainment; cooperative puzzles (can give people the same jigsaw but with a different picture :-)
- Tukan: this combines a UML editor with enhanced ENVY browsers that show which users are working on which methods (rain icon shows other user has later version of a method - load their version and you get sunshine icon)

Their experience is

- performance: multi-user applications are not noticeably slower than single-user. Tukan performed adequately with 30,000 shared objects and 12 users; a larger number would have the problem of how to display so many on one screen, plus they've seen a server crashing at 15 connections (maybe a sockets problem?) Slow machines do not slow down fast machines as the fast machine processes updating its views independently of the slow machine processing updating its views.

- network connection: they have functioned over 28k; there is an initial replication effort and then only low bandwidth is needed for replication; this is much better than NetMeeting (but in a test between Germany and Argentina, NetScape and COAST were comparable because the main bottleneck was network latency).
- development effort: experienced users took a week to build a COAST version of the UML editor and less for the cooperative puzzle application

GMD-IPSI is a German government institute for information research. The COAST framework was begun there in 1996 and probably took 3 years of effort to build (very rough estimate). There are people in GMD who think COAST has nothing beyond Java RMI; it is easy to demo an effect in the latter in five minutes, hard to get the two hours of concentrated attention needed to show COAST's power. The authors are now moving to a small private company to continue their work. COAST is going open source; see www.openCoast.org. It is in VW3.0 and Envy, being ported to VW5i.

A Smalltalk Web Server Application, Janko Mivsek, EraNova, Slovenia
The speaker's company began building Aida web four years ago. It now uses it for all its web applications. AIDA/Web is a web server (i.e. can serve HTML web pages) plus a framework for dynamic web applications, i.e. dynamically-generated pages from the smalltalk server. The smalltalk server pages let you make calls to smalltalk from the HTML to construct the served page dynamically (like Microsoft and Java ASP, JSP).

Aida is open source and currently runs on VW and Dolphin Smalltalk (VAST port being worked on); available from http://www.eranova.si/aida. They believe Aida can compete very effectively with Java (e.g. IBM WebSphere) and Microsoft offerings. They chose Smalltalk to have complete control of their webserver. The speaker demoed basic Aida functions such as dynamically-calculated statistics (most servers need to do statistics as a batch activity).

The company is doing some internet things (web sites, portals, e-shops) but make most of their money on intranet and extranet work (web-based business apps such as billing apps for Slovenian gas, business to business apps relying on electronic signatures, customer relationship management).

The key design idea was to have a web of objects connected by references paralleled by a web of pages connected by URLs. Each object should be able to represent itself as a web page, with object references being mapped automatically to URLs and vice versa. This is to solve the broken link problem that arises from the lack of referential integrity on the web, since Smalltalk has referential integrity (garbage collector respects it).

MVC for the Web: the aim is to separate the representation(s) from the domain model via an observer pattern. The web is just another observing representation. A WebApplication observes a domain object. There is a WebApp per domain object per session. A user's session state is held in the

WebApp. You can have many views of the same object (e.g. hierarchical or chronological views of newsgroup discussion thread). Web forms are hard to program from scratch so the web server simplifies this; connecting the fields to the domain object by pluggable adapters is in principal easy but in fact there are many gotchas on the web. Domain models are updated by aspect adaptors and by userAction methods.

Web Elements are text, fields, URLs, images, tables. They have Smalltalk equivalents of these elements with composition constructs - nesting, reuse. The domain object is observed by the WebApp which produces the web page which is composed of WebElements and of WebWidgets composed of WebElements. The WebApp persists for the session but the WebPage is regenerated (in fresh Smalltalk process - but these are very lightweight) for each request to that object (no caching).

Sessions separate users from each other. The web is stateless so they use cookies to set a session id but this is not a private (secure) solution so they can instead set the session id as a parameter in a URL. Sessions are permanent and persistent; a user can reopen their browser next day to resume a session.

Security: no-one has broken their server (yet; ESUG attendees are *not* requested to try :-). They have access rights per domain object, per view and to update. URL links to prohibited objects are inactivated automatically (becomes just text or greyed instead of link). They have authentication. Smalltalk previously has been lacking secure sockets and advanced cryptography. Work is being done on this but they took Apache's (undocumented) SSL C library and did CConnect to it; they have just released this secure socket layer on the web. They are now working on seamlessly integrating this to SWAZoo and will also integrate it with AIDA/Web.

Persistence: they use GemStone (started with Versant but had problems with their server-side smalltalk interface so switched) to store almost everything; requests, sessions, security information. Their architecture allows many users to one GemStone licence (a money-saver). They have no plans to use a relational database; discussion noted that the Camp Smalltalk Glorp project and three or four commercial products interface Smalltalk to relational databases and so should allow easy adoption of RDBs.

Benchmark: Aida can sustain 35,000 hits per hour on a 40 element web page stored in GemStone served by pentium machine. This is O.K. for intranets needing complex applications but not for mega-internet applications. Sockets are the main bottleneck. Setting up a socket takes much more than creating a process. Every hit must open a connection. Eliot Miranda described how soon they will move the VW socket handling up into the virtual machine, thus multi-threading I/O (only), to allow a flyweight pattern to conserve sockets (and VAST should keep pace). The speaker noted they cannot pass 50 hits per second until this is done and need to pass 200 hits per second to be competitive. Because the Smalltalk process model is so open, it's very easy to write simple schedulers that prevent starvation e.g. of connected people by would-be-new-connectors, which is a common failing of standard servers that can collapse in this scenario. These simple schedulers can in turn be manipulated e.g. dynamically.

That native threads allow scaling is a common argument for Java against Smalltalk. The speaker felt that clustering was a better way of addressing scalability than threading. Aida is a stateful server, unlike many Java servers, so clustering is not quite trivial. Eliot remarked that one Smalltalk web company (EzBoard, c.f. www.ezboard.com) has managed 6 million hits per day on 3 linux servers (they keep the data in Smalltalk memory and are wholly I/O bound and memory-bound) so the native thread issue is a slander.

The speaker then toured the code of AIDA.The WebMediator waits for a request and finds its session from the WebSessionManager. The WebElement hierarchy is a typical composition hierarchy paralleling the elements of a web page in HTML, with suitable construction protocol, including easy handling of web tables (in HTML, tables are not trivial). WebText permits many HTML text decorations (not every one is replicated as a Smalltalk method on the Smalltalk side; one can just supply HTML strings in Smalltalk calls). More complex components (e.g. a newsgroup) are WebWidgets. The methods for constructing these are long and hard to refactor. Ideally they would be done in a visual editor but the speaker doubted whether that was the best solution due to the difficulty of keeping the visual editor up-to-date and exposing all the power of Smalltalk through it.

The usual process is that a page is written by a web designer (e.g. in MacroMedia DreamWeaver). A Smalltalk programmer then uses this as a template for manually recoding the page in Smalltalk. An automatic generator would produce too low-level code (as it would be mapping from a lower medium to a higher) but they could add a construct to simply invoke raw HTML, for the static parts of page, into a method to save having to recode everything. A member of the audience (Piotr, I think) had used an XML parser to translate HTML into Java (Smalltalk should be simpler) and found it fairly easy when the input was sensible. This could be easily added as the XML parser exists.

The method registerViews registers different views for an application. Eliot Miranda noted that pragmas (VW3.0 and since) would be a better method, and could easily be extended to other Smalltalk dialects by finding the right class in the compiler and adding an instance variable to it. (There was some audience scepticism about Eliot's saying it was easy to dive into the compiler. I backed Eliot up; we subclassed elements of the compiler to mix rules and Smalltalk 4 years ago; it was a surprisingly easy and powerful technique.)

SWAZoo is a Camp Smalltalk open-source web application server (Smalltalk Web Application Zoo) that aims to merge Commanche, Hydrogen, ByteSmith's toolkit, etc. There are four web-related Camp Smalltalk projects. The others are

- SOAP (simple object access protocol) is supported by Microsoft and may be the way of doing business-to-business commerce in future.
- XML
- Internet Client/Server framework

Eliot: Cincom have written some SSL code in Smalltalk. Smalltalk does large integer arithmetic in primitives therefore expect it to outperform C. The only question is convincing people that it is secure.

Smalltalk and the Web, Juan Carlos Cruz, Valtech AG, Zurich
The speaker described one way of use Smalltalk for web solutions that he knew (VisualWorks plug-in), compared it with another (ClassicBlend), and summarised others.

The plug-in DLL manages the interface between the browser and the VM. It maps the VW UI to the Net Browser's UI so that work in one is displayed on the other. The plugIn DLL is 104k, plugIn image is 3.9Mb. The plugIn development environment helps you specify parcels that can be loaded as applets (not every parcel can be). It also helps you generate a customised plugIn image (customised plugIn images can contain code that does not change between your applets, to aid performance). Lastly, it provides debug tools. There is only a windows version today but a unix version is being worked on. It is now included in the VW non-commercial distribution.

There are two ways to create an applet.

- Subclass AppletModel in a parcel you then save as an applet which is invoked by a simple HTML call with applet parcel name, width/height and mime type ('application\x-visualworks-parcel'). The applet also provides protocol for operations on the status bar of the browser. Nothing changes between the VW protocol and the HTML; you can open and run it in VW, etc.
- Instead of subclassing AppletModel, you can adapt an existing application by duplicating a subset of the AppletModel protocol in your application class (3 methods: isAppletModel, plugInConnection, ... are essential; many others are useful for e.g. web communication but not essential).

You switch the PlugIn debugger on and off as needed. The speaker put a halt in his demo applet and (after a brief hiccough in obedience to the usual law of demonstrations :-) saw the halt from the web page. (Most applet debugging can be done in VW but web comms must be debugged from the web applet.)

To avoid the cost of loading a large application every time you launch your applet, either the standard image can be customised or the VWPRELOAD attribute in the EMBED tag can list additional parcels to load.

VWPLUGIN.INI initialization file sets various configuration and security parameters. Currently, they use a trusted site strategy; either all but a denied list are allowed to download parcels, or all but an allowed list are unable to download parcels. If you trust the source, they can do anything on your computer - no equivalent of the Java sandbox (but a skilled and determined hacker can break their way out of the sandbox so the difference is more one of perception).

PlugIn comms can be by HTTP get and post URLs, by sockets, and by VW communications approaches such as Database Connect, Distributed SmallTalk (Corba) and VisualWave. The speaker demoed generating classes from one image running applet to another (server) connected to it via DST (N.B. there is a useful parcel that auto-configures ORB on applet image when starting). After the usual demo hiccough, the speaker added an attribute to a class on the server image using a class-displaying applet on the client image.

Question: how useable is this given that the client must install the plugIn before they can browse your applet? Users who click on the URL will be prompted to download the whole thing (zipped file is 1.2Mb, expanding to 3.9MB and 104k), walk through a simple installation script and can then use VW-based applets. Thus while fine for an intranet, it is a demotivator on the public web (analogy with steps up to a shop - somewhat deters customers).

Classic Blend 2.0 uses client and server lightweight performance-tuned ORBs to connect a Java client to a VW (only) Smalltalk server through proxies generated by ClassicBlend from the applet smalltalk class' window spec. (Thus in Model-View-Controller, the M remains pure smalltalk while the VC is split into Java and Smalltalk proxies communicating with each other.) Tools help you generate the HTML that sets this up. There are several VW widget behaviours not included - drag and drop, entry/exit notification, dynamic focus control (#hasFocus etc.), ... You can develop new widgets and/or behaviours yourself but this is work.

ClassicBlend 3.0 handles VW and VisualAge Smalltalk and is bean-based. Client GUIs can be constructed in any bean-compliant GUI builder but some GUI work is now required to develop applets in contrast to the 2.0 all-generated approach (but possibly more flexible). ClassicBlend 3.2 includes Swing but the speaker has not used it.

Lastly, the speaker listed some other solutions: VisualAge Ultra-light client (see last year's conference talk on ULC), Corba (Netscape includes an ORB but there are many drawbacks to using it) and straight use of RMI-IIOP. An audience speaker also mentioned Smalltalk-X, which can run both Java and Smalltalk so that a mixed mainly-Smalltalk application can be written with key client behaviour in Java.

Making the virtual domino toppling application, Ernest Micklei, PhilemonWorks, The Netherlands

People go to great lengths to break the domino toppling record (now at 2 million!!!). The speaker has a SmallScript3D programming framework in which he decided to animate this activity in Smalltalk. He maps SmallScript's output to an ASCII format that can be read by a VRML program. A touch sensor is attached to the 'first' domino in the VRML so the user can start it toppling.

Firstly, he had to understand the physics of the domino toppling process. At any moment, only one stone (the furthest forward in motion) controls the process, with the angle of earlier still-falling stones being determined by back-propagation from this stone. He can make one domino hit more than one so the 'front' is in fact a collection of 'fronts'.An elegant high-level script controls the domain behaviour.

Next he had to add time. A clock tick controls the speed of the animation. It is mapped to a rotation angle for each stone which maps to position and behaviour. Next he had to visualize this domain model. A primitive, the stone cube, is wrapped by various wrappers to handle its rotation angle, yaw angle, etc., to build a fully-behaved stone object. A special swivelling stone (like a mediaeval tiltyard quintain) can reverse the direction of toppling.

Impressively, animations on the speaker's slides demonstrated the behaviours they talked about, showing simple linear domino rows, circles, spirals, wedges (multiple fronts example), swivelling flow example, the ESUG logo in dominos and a huge multiple streams toppling in parallel finale. Perhaps most impressive to those who do domino toppling in the real world would be the way everything could be set up again in a moment.

As the processing is sequential, symmetric multi-front examples show slight asymmetries in their behaviour; this could be fixed by computing the state update over the fronts collection and then updating the display. Also, the physical model is simplistic; it does not handle momentum, weight, rotation outside the toppling path, collision of toppling rows, propagation speed in the material, uneven terrain, friction of touching stones.

Lastly, the speaker defined a simple concise text language (Domino Text Format) for defining domino scenes and wrote a reader for it. The language uses 2D ASCII character arrangement to show the location of the dominos. The character at a location defines what angle etc. the domino at that location has. A spreadsheet program is good for writing in this language (and the speaker has prepared an Excel file with useful notes on how to write them). The speaker wrote a scenario, exported it to a text file, ran it through his program to generate VRML, showed the result in NetScape and toppled it.

There is a website

- http://www.philemonworks.com/domino/uk with tutorial, etc., and he is setting up a domino webring - as soon as last stone topples, the browser moves to new URL.

The speaker's wishlist includes having sound, 3 dimensions (domino stairs, bridges, etc.), adding editors to simplify scene building, and building the domino scenes inside a virtual world.

A Framework for Heterogeneous Coordination, Thomas Hofmann, University of Bern, Switzerland

Coordination is about managing dependencies between activities. RPC and RMI style communications have the disadvantage of tight coupling; they must know the identity and address of participants and must wait for participants to be communicative. By contrast, Linda is a blackboard architecture where agents can exchange tuples which contain any information. The blackboard paradigm uncouples the agents. Linda's basic ontology is write, read (copy) and take for tuples. In Linda, tuples are vectors (fixed size) with associative retrieval and can be searched for by masking template tuples.

Since the original Linda, many extensions have been introduced, e.g. towards OO data structures on the blackboard, but it still lacks configurability. Open distributed systems need to use multiple protocols, set access and concurrency controls, etc. Linda also lacks heterogeneity; open systems need to run on several platforms and be usable by multiple programming languages.

The core of the framework is the OpenSpace class and its associate classes. Class Entry contains the data to be exchanged on the blackboard. Entry subclasses have matching algorithms (to let agents find data) held in associated ConfigurationPolicy objects held by Entry subclasses (and so able to be dynamically changed). Class OpenSpace supports the blackboard ontology

```
write: anEntry
read: aTemplate
take: aTemplate
```

SpaceAgent represents clients of the OpenSpace. The SpaceServer provides SpaceAgents with references to the OpenSpaces they ask for (by name, by IOR, ...). It can also act as a factory to create spaces. Templates are able to match entries if they are an instance of the same class (hierarchy) as the entry of if its has bindings for all the keys of the entry.

The speaker outlined a simple trading example: buyer request product, sellers make offers, buyer accepts best deal, seller accepts, seller delivers product, buyer collects product. He demonstrated it from scratch (on VW3.0 and DST; it runs on any platform that VW runs on and in any language that has an ORB.): create trading space (for real estate), input buyer and sellers of houses (from different images), request house (in France, at a given maximum price), make two offers, accept best.

Security has been thought of but is not to product quality (e.g. sellers don't see their rivals in this example but that's up to the programmer). Rollback of transactions in the event of server crash would also need work for a product.

The demo indicated additional ontology. To append an offer, a MarketAgent must take the TailEntry instance for the request, increase its index and write it, then write the offer. (This will be done by the OpenSpace in the next version.) ConfigurationPolicy can enforce before and after consistency checks on accesses. The framework currently has three types: RequestPolicy, OfferPolicy and DealPolicy.

After some time, the space will clutter with forgotten entries; garbage collection is needed, e.g. by timestamping entries. (I suggested what he needed was an ability to configure the high-level ontology of discourse over the low-level read, write, take ontology. In the example, the ontology is request, offer, deal, and the garbage-collectable entries are obviously those other offers relating to a request that has progressed to a deal. The deal request and offer become garbage-collectable after product pick-up.)

Discussion Sessions

Panel Discussion

Smalltalk needs to publicise its success stories (e.g. MetaEdit, Penn State university information system, ...), and to have books and industry recognition of its contribution. David Simmons recently had argument in Florida with someone who didn't know that Smalltalk originated eXtreme Programming and key VM work. MicroSoft MSDM publications is willing to publish Smalltalk articles. Their .NET platform is a rival to Sun's Java.

In the past, Smalltalk vendors have been a big problem with Smalltalk adoption. Until recently, much has been proprietary (Eliot gave JIT as an example; he read a Java paper talking about operand handling which was solved in Smalltalk years ago but kept proprietary). Another problem is being ahead of the curve. A major customer of Xerox in Smalltalk's early days was the CIA (internally they were called 'the customer'). When ParcPlace spun off, it was accustomed to large well-funded customers and very unused to handling the kind of marketplace that evolved. Digitalk had a turboSmalltalk model but this was killed by the merger.

Smalltalk has not been in the 'always there' mode (c.f. unix, visual basic, etc.). Eliot recommended the book 'The Revenge of the Command Line Interface'. Real value lies in software that is solving hard problems and is not yet a commodity. When software becomes a commodity it must become ubiquitous and then it must become free, otherwise it will fossilize. Over-engineering is not commercially viable: an example is MicroSoft versus Apple. When Eliot started, he wanted to produce solutions to hard problems up-front, but you can't sell it because its too hard. You must solve immediate problems or broad problems. People make money on consulting fees or on a product-upgrade cycle. Apple took things too far: they solved the 5 year hence problem without releasing solutions to the immediate problem quickly enough.

In adaptive programming, Smalltalk is the sleeping giant. The web world needs to change web servers without shutting them down; Smalltalk is uniquely suited to that. The market is beginning to understand that this is a problem for them. The Java people are beginning to talk about this. David Simmons reviewed several languages when studying this. Eliot stressed that dynamic programming is not just about the execution environment (dynamic binding, etc.); in a server environment, he could bring up a new server and serialize his objects across (albeit perhaps with massive latency and concurrency problems). The key value of Smalltalk is being able to be so expressive without types, flattening the design into the language.

In the Smalltalk community, it's cool to push the envelope, rather than to commercially providing-a-solution-for-time-X-cost-Y-to-company-Z. Perhaps we need an easy adoption path for outsiders; ability to write code in emacs and suchlike. No Smalltalk VM has an open architecture that other languages can target (Visual Age has it but it's proprietary).

Companies are changing themselves into fossils overnight. For example, the company Integral (exotic financial products) used Smalltalk because it was able to keep up with their market's rate of change. Two years ago, purely to keep their venture capitalists quiet, they switched to Java. They have released nothing since.

Joseph Pelrine did some number crunching in VAST and found it was slow. So he ported the number crunching code to SmalltalkMT, converted it to a DLL and called it from VAST. He got a five times performance increase. Smalltalk MT, like Smalltalk QKS, does not have a traditional smalltalk heritage. Thus it has different strengths and weaknesses.

Smalltalk and Corporate Culture, Piotr Palacz, Australia and U.S.A
Piotr has moved from Australia to U.S., and has occupied many roles (manager, architect, team leader, etc.) in many confused projects (and some unconfused ones) in both places (15 years of working in many companies and consultant in many others), so he has had to think hard about corporate culture issues. His preferred tool is Smalltalk. He works in corporate reality; how can he recognise environments that are congenial to sensible usage of sensible tools?

Cultural viewpoint is often thought unimportant or dull by techies (and the speaker is not a trained anthropologist). After a review of possible meanings of the word culture, the speaker described Smalltalk culture. Using and sharing a preferred tool creates a culture, mostly based on tacit knowledge. Managers have a dream of mass-produced software by replaceable 'resources'; in this dream, most programmers are like soldiers in soviet army; replacable, using cheap tools which are tedious but deterministic to use. Some programmers are special forces types; skilled, motivated, expensive. (In both cases, casualty rates are high.)

However these resources are in fact people who create things. Programmers are like musicians, who are attracted to different instruments and styles of music, e.g. oboe players are said to be irascible as their instruments are difficult (My sister, who plays the French horn, confirms that brass players are reputed extrovert and viola players are the subject of ‘stupid’ jokes, etc. - Niall). Does this apply to programmers’ language choices.

Some people play finite games towards a specific end. Some play infinite games for the sake of playing. Some people do both; play to an end but also enjoy it. (Chess is a good metaphor for Russian culture; we win wars, we lose pieces. Poker is a good metaphor for U.S. culture; big gains and losses, bluff and risk. Go is a good metaphor for Japan; long-term strategic control of terrain.)

Two elements characterise different corporate cultures; their control mechanisms, and their attitude towards knowledge (its creation, growing and dissemination).

Dan Ingalls’ ‘Design Principles of Smalltalk’ paper said Smalltalk was to support the creative spirit of programmers. He was influenced by the idea of amplifying human abilities through encouraging learning and ease of use. This contrasts with using tools merely as means to increase productivity (i.e. maximise profit). It involves changing the form of discourse away from the textual.

Smalltalk likes simplicity, expressiveness, transparency (the tool is not the problem, the tool lets you see the problem c.f. clean window vs. dirty window), ‘habitability’ (you feel at home in the tool). Contrasting programming cultures include:

- efficiency (the esoteric, the arcane) e.g. C++; Piotr thought this, not Java, was the worst thing that happened to programming this century. It creates so many problems by itself that it creates a special breed of experts who can live solving ‘artificial’ C++ problems, not real problems.
- correctness: formal methods are a good example; correctness for its own sake can be excessive
- marketing in programming: Java is the perfect example. It was not marketed so much to those who would use it as to others who were convinced by its business case and were ill-equipped to judge if it would meet that case

Ingalls believed that technical excellence was the main factor that makes languages survive; alas, this is not so. Before the speaker left Australia for the U.S., he had worked in mainly small companies, and he expected to find post-industrialism, technopolis, economic rationality, technical aggressiveness and large budgets in the U.S. He had various surprises.

- U.S. corporations have authoritarian, sometimes directly military, command values
- factory pattern: staff are replaceable (and therefore cheap) components

- death marches: victory is always declared (by managers who want bonuses) but people fall sick and product is not really valuable
- mimicry: culture was not to stand out from the crowd
- degenerated pragmatist: typical expression ‘if you are so smart, how come you’re not a millionaire yet’
- functional illiteracy: attachment to talking rather than reading (this one I have also noticed, though I would say attachment to slides without notes as against documents)
- distorted and controlled discourse: very poor distinction between logical and other (e.g. ad hominem, from authority, ...) arguments
- manufactured consent: semblance of real discussion but the authority halts discussion if outcome not what is desired
- omnipresent amnesia: failure to learn from mistakes
- absence of management theory: speaker noted managers didn’t recognise the big names (but I would not see this alone as diagnostic)

The speaker linked these features to the educational system, the corporate culture and the mass culture.

Corporate cultures divide into sales-centred and people-centred. The speaker linked these to the attitude to knowledge, including, he felt, whether the culture itself is consciously discussed or not (but I disagreed; explicit discussion of culture in a corporation can be a means whereby it refuses to recognise, let alone correct, its vices, e.g. when sticking to the plan, or even remembering it, is characterised as rigidity and lack of responsiveness to change). Recruitment in such conscious cultures often focuses on whether the recruitee fits into the culture, as opposed to the undiscriminating ‘cheap resources’ recruitment of the unconscious cultures. A key feature is whether the key asset is perceived as the people or as the deal, the contract, the delivery date and the bonuses.

Piotr identified oral exchanges as a feature to the authoritarian culture and email/written exchanges as more typical of the democratic kind. I queried this (it’s contrary to my experience) and the speaker said he was thinking less about how the decision was made than about how decisions were communicated to staff (I agree) and whether they were adhered to (I disagree).

Generally, the review contrasted hierarchic and ‘open’ cultures to the great advantage of the latter, the former showing the (conventionally assumed?) features of less comfortable working environment, more petty direction of work, hypocrisy instead of self-honesty, etc.

There was a query about those banks in the U.S. that have adopted Smalltalk (and are now the ones that are taking over all the others). The speaker commented that the need to support new financial products in short deadlines created an exceptionally strong demand for Smalltalk-specific capabilities, liberating these organisations from the usual authoritarian constraints.

Sprint has announced it is ‘dropping Smalltalk from 1 January 2001’. The speaker had visibility of this process. A technical steering committee tried to asses the arguments for and against Smalltalk by collecting an unstructured set of statements from people involved in Smalltalk. Two external consultants (one the speaker) were asked to review them but this review had no impact. The main action of the committee was phone calls to people to verify (and/or ensure) that there would be no serious emotional opposition to the dropping of Smalltalk. There was no rational evaluation.

There was much audience discussion. Jason (Cincom) has seen customers do a range of things: some are very driven by people who are keen to do the latest fashion (e.g. Java). Some organisations make quite rational choices, e.g. for commercial positioning. Many look at the cost model re retraining people and hiring new people. Overall, some make very rational decisions, both to stay with Smalltalk and not to, while others make very emotional decisions to abandon it (and maybe sometimes not to). A Japanese bank who rewrote their system in Java, taking 12 months during which no new functionality was added, were a particular example of an irrational, high-risk decision.

I argued that organisations that were ‘open’ by the speaker’s definition (opposed to hierarchy, consciously discussing their culture, eager to embrace change, etc.) nevertheless could be irrationally opposed to ‘sensible use of sensible tools’ for reasons intimately related to their ‘open’ nature. Responsiveness to change can mean taking the latest, not the best. and forgetting experience rather than reasoning about it. A democratic anti-authoritarian environment is by its nature a very *overtly* political one in which continual politicking effort is necessary to hold any position, and so can be one in which it is very hard to acquire and retain authority by virtue of technical skill. Hence while I shared the speaker’s goal, to recognise environments that are congenial to sensible use of sensible tools, I disagreed with his criteria. An element of hierarchic leadership can be very useful in achieving and adhering to sensible choices. Much pre-existing discussion of culture can be an obstacle to rational discussion of culture.

Lastly, the speaker touched on extreme programming. Methodologies can be regarded as cultural utopias that prescribe tacit value systems without going too deeply into the real preconditions of their success. XP by contrast is minimalist and identifies its constraints/capabilities: small or medium teams, requirements subject to change, managing risk and learning from failure (there was some discussion of this; the above is an edit of the speaker’s first version), etc.

The speaker dissented from Kent Beck’s opinion that risk management was the key problem and asserted that frequent change was (always, he seemed to be claiming) an effect of troubled information flow. He contended that XP’s stress on oral communication, on refactoring instead of (re)design, etc. were defensive measures. From this position, he deduced a legitimate role for XP as a means by which developers could defend themselves against hierarchic culture.

All this was very much debated by several in the audience. Requirements can change for rational reasons: customer has to learn what they really need by experiment, genuine ignorance of the problem, wicked problems (‘wicked’ here is the technical term for a class of problems that cannot be solved correctly without first building an exploratory solution, e.g. first explorer landing on the shore of an undiscovered continent cannot *deduce* the best route inland; only some exploration will let them find it), etc. I argued that hierarchic cultures were the ones that could resist change. The more open cultures favoured by the speaker were the ones where requirements would change rapidly for organisational reasons. Andy Bower argued that it was in accord with the logic of the above talk that the programmers be allowed to act as business analysts, by participating in XP for requirement elicitation, contrary to the speaker’s contention that this was a misuse of them. Jeff Sarkela said that XP’s emphasis on early delivery was to enter maintenance, the ordinary mode of software, as soon as possible. Bringing a piece of software out is like having a child; nothing is the same again.

ESUG Goals, Stephane Ducasse and Roel Wuyts
ESUG’s goals are to connect Smalltalk users, to organise advanced seminars (the summer schools) and to promote Smalltalk.

The two first points are being addressed by ESUG’s current activities. The last one is not really active for now. The ESUG board is looking for volunteers who would like to participate in organizing lectures (they have a lot of teaching material, on CD). They would like to organize refactoring sessions around Squeak and XP days.

Smalltalk can be promoted by CDs, etc. People argued for a wiki as well as a CD. People can help by sponsoring, by teaching Smalltalk, by organising events, by marketing (T-Shirt, logo, etc.).

There was enthusiastic discussion whether the wiki could be a COAST / OpenTalk server for remote Smalltalk mentoring / pair programming. We could start by a message on comp.lang.smalltalk and contact Ralph Johnston to get a community working on this. Ralph has been talking to Eliot and others about setting up a remote pair-programming site for business consultancy, etc. This would be an extension of Camp Smalltalk; Virtual Camp Smalltalk. A progression to this would be to unify the many Smalltalk sites to a few cooperating distributed portals. Eliot will write a proposal and put it on comp.lang.smalltalk.

Issue: must monitor use of this in both directions to avoid over-solicitation of key central Camp Smalltalkers by newbies, while allowing ‘looking over shoulders’ (perhaps capture discussions as tutorials).

ESUG needs sponsors and more attendees. One good thing this year is that 80% - 90% of the attendees (60 people) were attending ESUG for the first time. ESUG needs to make a profit, or at least not to lose money. A good way to sponsor ESUG is to send people to attend. Vendors can also help, e.g. by sending speakers for free.

For ESUG 2001, we could host a Camp Smalltalk 2001 and the Squeak world tour. This would need programme alignment e.g. by a session in which you pair programme or do a squeak tutorial as you wish.

When to hold ESUG; if ESUG stays at its current annual slot (and finding another without drawbacks or clashes is not easy), then perhaps ESUG should be described in the U.K. as a 4 day (Tuesday - Friday) conference plus a pre-day on Monday. At present, ESUG loses out on U.K. attendees because it clashes with the August bank holiday (but that does ensure that those who are determined to attend from the U.K. tend not to have competing commitments).

Lastly, we must form more national user groups:

- The Dutch OO Information Technology (DoIt) Smalltalk user Group is setting up a site: www.gosmalltalk.
- The Swiss Smalltalk User group (SSUG) is replacing their old web site[1] with a new one at http://kilana.unibe.ch:8080/SSUGWiki/ (contact ducasse@iam.unibe.ch)

Conclusions

Although the food and venue could not compare with Ghent last year, this was a fun conference with some very interesting talks. I look forward to next year.

- I understood Michel’s meta-programming work better than last year (he explained better and I listened better); I got real value for my work from the discussions.
- The separation of behavioural (for programmer) and structural (for compiler) types into separate defining mechanisms operating in separate epochs is a vital and powerful idea. While Java tries to pillage old Smalltalk work for its current VM designs, etc., does this work point to a Smalltalk future in which programmers will be freed from the constraints of single-epoch binding?
    - — Dave Simmon’s work with Smallscript on .NET is relevant (and a fascinating and hopeful development in general).
    - — Eliot’s adaptive VM work is relevant (and impressive).
- From my own experience with teleworkers in my team, I know that remote XP-driven pair-programming is a powerful technique. Can we find the tools and business model to make this the next great idea to be demonstrated first in Smalltalk?
- I felt confirmed in the opinion formed at ESUG99 (during which it was announced) that Cincom’s taking over VW is good news for Smalltalk.

Other discussions

I had several very useful meta-data discussions with Michel Tilman and other attendees. (I also spoke to several possible contractors who could work with us on our meta-programming system.)

1. http://www .iam.unibe.ch/~scg/cgi-bin/Smalltalk.cgi?SwissSmalltalkUserGroup

Andrew McQuiggin, Equitable Life gave me interesting information about his experience with Ultra-Light Client. For reasons that reflect the general concept of thin client rather than ULC's specific implementation of it, thin clients tend to get fat.

Eliot Miranda of Cincom, explained Didier Bessier's experience that performance profiling shows the slowness of Smalltalk number-crunching is concentrated in primitives (see Didier's talk in my last year's ESUG report). It's not a lack of number-crunching primitives as such but the way floating point types work in the VM.

I had an interesting discussion with Eliot about adaptive programming in telecomms. For example, imagine a mail application built to use a storeAndForward paradigm which, while running, must be extended (to meet a market window of opportunity) to handle a continuous media (e.g. audio). Then the relevant class (interface) must be extended e.g. by a method

```
isStoreAndForward
  ^true
```

which, if false, will require bandwidth reservation on the transmission connection (not needed in storeAndForward). Smalltalk can add this method to a running system. Java cannot; its compiler grabs the interface and uses it in references therefore, although you can dynamically load classes, a Java system must be shut down to let a revised interface be used.

Eliot's PC start screen shows Bill Gates in lederhosen leading a microsoft flag-waving rally. When he went through Frankfurt airport he was told to turn it on; fortunately the security guard had a good sense of humour - she nearly arrested him until she noticed that the symbol in the white centre of the red-bordered flags was the Microsoft logo, not a certain other symbol. (But if Smallscript on .NET becomes a powerful Smalltalk lever, Eliot may start thinking more kindly of Bill. Surely it's the Java people we should be portraying in this way. :-)