CS3 and ESUG, Essen, August 25th - August 31st, 2001

Finding the conference was easier than last year (fortunately so as it was a very hot day), thanks to clear instructions and the University of Essen's garish but useful colour coding of their buildings. The principal hotel was value-for-money and pleasant enough. The walk between the venue and the two hotels (unfortunately situated in opposite directions from each other) ensured one got at least some exercise during the day. There were two social events.

- A Cincom-sponsored boat trip on Baldeneysee Lake: drinks and snacks, intelligent company and good conversations, a brief glimpse from the lake of the notorious Krupp Villa Heugel, and the rain - well, I'm British, who cares about rain (but it was suggested that as this was the one time in the whole conference when I was *not* carrying my umbrella, clearly I was responsible :-)
- A tour of the Zeche Zollverein complex: when one's host takes you on a tour of a disused bauhaus-style coal-extraction plant, the sceptical reader may form the impression that Essen is ill-supplied with cultural sites. Not so: the Museum of Design, now housed there, was a chance to imbibe German Kultur at its most characteristic. When I reached the sixth story and started admiring the (presumably well-designed) desk, bookshelves and chair I found there, only to have someone come in and tell me this was her office, not another wing of the museum, I decided I had imbibed a little too much Kultur and needed an evening in a beer cellar to recover :-). Seriously, the bauhaus style was surprisingly un-ugly (for a coal mine) and the chance to see how the mine developed over the last two centuries was interesting.

Style

In the text below, 'I' or 'my' refers to Niall Ross; speakers are referred to by name or in the third person. A question asked in or after a talk is prefaced by 'Q.' (occasionally I identify the questioner if it seems relevant). A question not prefaced by 'Q.' is a rhetorical question asked by the speaker (or is just my way of summarising their meaning).

Authors' Disclaimer and Acknowledgements

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

As there were two programme tracks and numerous additional discussions, I could not attend, still less report on, everything. I apologise to those whose talks are covered by a brief summary or 'missed it; see their slides'.

Likewise, there were several Camp Smalltalk projects, and I spent my time there working, only writing up what I recalled in the evenings, so my CS3 notes treat the projects I worked on in much more detail than the others. I apologies for any inaccuracies, and to those participants who are covered by the abbreviation 'et al.' because I failed to catch their names. If anyone who spots an omission emails me (or Stephane), corrections may be made.

John McIntosh also wrote daily reports on both CS3 and ESUG, available from: http://www.smalltalkconsulting.com/html/CSEssen2001_1.html. I recommend reading his text as well as mine. By a fortunate coincidence, we did not overlap much in which talks and projects we attended and where we did, our joint reports complement each other. My thanks to John, to those others who told me of things I missed, to Michele who leant me his laptop when the one I was using was needed for a demo, and above all to the speakers and participants whose work gave me something to report. Thanks also to the conference sponsors, Cincom, Roots, Georg Heeg and Daedalos, and to Professor Unland of Essen University for hosting us.

Summary of Projects and Talks

This year's ESUG began with a Camp Smalltalk (called CS3, not to be confused with the third Camp Smalltalk event, which John McIntosh and others understandably called CS3; see the wiki for an explanation of CS event naming).

For the first time ever, the main ESUG conference had enough talks to require two parallel streams. This was good news for Smalltalk but bad news for me as I wanted to see all of them but had left my time machine at home :-/. Attendance was well up on last year, itself higher than the year before; more good news for Smalltalk.

There was also a Doctoral Symposium, where university students working in Smalltalk could present their ideas, and plenty of ad-hoc discussion.

Camp Smalltalk (CS3)

By the time I arrived on Saturday afternoon, various projects were well underway.

Standardizing Sockets

This was worked on by Reinout Heeck et al. Reinout had to leave before the end of the conference and so I missed my chance to get a report from him of what happened. Read John McIntosh' report and the wiki pages.

SOAP, renamed webservices

Magnus et al. created a service on VA machine, exported it to VW, invoked said service on VA from VW, and got the correct result. They now want to write SUnit tests, and to test from Squeak as well.

GLORP

Alan Knight, Martin Feldtman et al. worked steadily on this throughout CS3. GLORP was written in VW. They have now ported working versions to Dolphin and VA. GLORP needs definitions of the table structures to map to, and the definition of how the mapping is done. For large schemas, hand-typing these definitions is hard so they worked on a tool; the tool uses GLORP to do its stuff.

Having seen the complexities TopLink for Java got into through having nested shared caches, Alan plans to implement one cache per transaction as the simplest thing that could possibly work to extend GLORP to multi-user.

Dialect-independent Programs

Peter van Rooijen wants to write utilities in 'Lowest Common Denominator' Smalltalk guaranteed to run in all dialects. Alas, most useful utilities have to stray beyond LCD-ST. Peter has therefore written a dialect isolation object. The aim is to write portable programs in a combination of LCD smalltalk and invocations of services from the dialect isolation object, an instance of EnsuHost. This class supports a set of methods for essential non-LCD-ST things.

- To write a dialect-neutral utility, invoke a suitable method on the (singleton) EnsuHost object for all non-LCD-ST methods
- To run the utility on a given dialect, load the appropriate EnsuHost for that dialect. When the utility calls a method on it, it does whatever is appropriate for that dialect. For example,

```
EnsuHost>>exceptionError
  ^ExError | ExHalt
```

is a VA implementation; when I looked, the team had just started to see if

```
EnsuHost>>exceptionError
  ^GenericException
```

was the appropriate VisualWorks equivalent. Peter has created the framework in VisualAge. While Abigail Lee of Roots and others wrote VW bodies for the methods, Andrew McQuiggin and I helped Peter code Squeak ones and simultaneously debated the approach. Part of our debate was about scenarios for using the framework.

- Peter pointed out that when writing a utility you plan to post on the web, dialect neutrality is often an obviously desirable feature from the start.
- By contrast, Andrew and I expected to stumble across useful utilities in our specific work and later post them; we would not code them in the (less natural for a given dialect) Ensu style initially.

As good XPers, we therefore wanted to code in one dialect, perhaps setting our browser to warn us when we were using non-LCD-ST and/or non-Ensu-mapped methods to avoid unnecessary dialect-specificity. Then we could refactor to another dialect when dialect-neutrality became needed. Ideally the refactors would be automatic for those equivalences that Peter's framework identifies (e.g. equivalent methods might be method-wrapped and converted by running the utility's tests in the other dialect).

However that's just an interesting idea whereas Peter's approach is here today and is certainly a way both of writing dialect-neutral utilities and of supplementing LCD-ST with a set of abstract methods each with several dialect-specific implementations, asserted to be semantically equivalent. This information is what any dialect-conversion program would need.

(Another kind of dialect neutrality has been achieved by Joseph Pelrine. He has modified the refactoring browser in Squeak to parse code from any chosen dialect. The code can then be either just stored or also sent down a socket for execution on that dialect's VM. See his Modular Squeak talk.)

The Refactoring Browser

They worked on finishing the port to Squeak, completing the UI that was started by Bob Hartwig. They made it file in cleanly and got most things working. They still have to make ‘extract to component’ work. They found it tricky to stay compatible with the other browsers due to differences in handling dialogs, etc. A version will be released at end of the week. Porting to Smalltalk/X was begun.

Alan Knight brought the VW 5i.4 RB port, which he gave to those CS3ers who asked for it to speed their work. It is already being widely used for development in Cincom. Alan guestimated another two weeks at least before Cincom had it on their website for general download.

Andrew McQuiggin and I began a new CS project to make the RB’s rewrite editor more accessible. Andrew and I have found it an excellent way of creating custom refactorings for specific applications. However, to the novice Smalltalker, it can be a difficult tool. The average application developer, on being told to use it by their team leader, writes a pattern, makes a trivial typo, can’t see why it doesn’t match what they expect and gets frustrated. So Andrew argued, and when Alan Knight, not exactly a novice, admitted he had found the rewrite tool too intimidating to get to grips with, the point was proved :-). (There is an analogous issue with word processor find-replace utilities that take metatext. Users usually do ‘find, replace’ on one or two cases before hitting ‘replaceAll’ to check they have written the correct metatext.)

Why not let them write example code they want their pattern to match in the rewrite tool’s ‘search for’ pane and then have the RB show them what pattern(s) would match it?

- The RB knows how to build a parse tree from some code.
- The RB rewrite tool knows what patterns can match each node in that parse tree.

So the only issue is how to communicate that knowledge to the user. A simple tool might just replace each node in the example code with some default pattern for matching instances of that node class. For example,

```
myDictionary
  at: myKey
  ifAbsent: [myDictionary at: myKey put: myValue]
```

could become

```
`myDictionary
  at: `myKey
  ifAbsent: [`myDictionary at: `myKey put: `myValue]
```

It’s trivial, but it would speed things up in teams composed of application developers with finite time to master new tools. Better still, let the user select from a list of possible matching pattern types for each node of the parse-tree. There are two ways of doing this.

- Display the parse tree and let the user invoke a pop-up menu on each node of its possible match types.

- Sequentially prompt the user with each node and its list of match types.

The latter sounded easier (unless a graphical RB parse-tree displayer already exists) so we decided to start there and wrote an initial test. Once we'd explored the guts of the refactoring browser sufficiently, it was straightforward to start implementing. The rewrite tool UI is a front-end to

```
myParseTreeRewriter
  replace: searchForString
  with: replaceByString
```

(where each string is expected to contain a mix of Smalltalk code and RB metacode), but there are also lots of other methods:

```
myParseTreeRewriter
  replace: searchForString
  withValueOf: nodeArgReplaceBlock
  when: nodeArgValidationBlock
```

and many variants thereof (ParseTreeSearcher has match:do: methods instead of replace:with:). Note that these methods create the rules the tree *will* use and a single ParseTreeRewriter can be sent several such messages to prepare a complex multi-rule rewrite before being sent

```
myParseTreeRewriter
  executeTree;
  tree
```

which effects the rewrite and returns the rewritten parse tree. Using the Block-argument replace methods instead of the String-argument replace methods made writing rules that converted example Smalltalk code into example metacode that matched it straightforward (if a little mind-stretching at times: to replace code with metacode when run, the rule must replace metacode in the search string with 'metametacode' in the replace, which is why we had to construct it with a block; the required replace was impossible to write as a parsable string).

Implemented this way, the utility should be as dialect-neutral as the RB since it uses already-ported RB code and LCD smalltalk (except for the trivial dialect-specific implementations of the rewrite window button that invokes it and of the selection list from which the user chooses which of the possible match types they want for each node).

Pages on this will appear on the wiki as soon as I finish this report. Ideas for further development:

- Provide several examples (separated by '----' in searchFor pane, or on successive pages of a searchFor notebook pane, or ...) and show whether the pattern derived from the first matches the others
- As above but have the parser return the most specific pattern matching all of them (analogous to the way the squeak methodFinder maps sample input and output values to methods that could return them).
- Have a simple means of showing what the rewrite pattern does to the example(s) before running it generally.

Lastly, the rewrite tool should report when a replace meta-pattern is not in the search string; mis-typing between search pane and replace pane is a common source of new-user frustration.

Numerical Methods
This project converts Didier Bessier's Numerical Methods in Smalltalk code (see his talk in my ESUG99 report) into an open-source utility. Roger Whitney et al. ported the code from VisualAge to VW3.0 and VW5i.4 completely, and to Squeak almost (3-4 errors still to correct from 90 tests). A team member has promised to port it to GemStone and Envy. Claus Gittinger is porting it to Smalltalk/X.

Didier emailed them that legally he needs his publisher's permission to release it. He expects this to be no problem as the more people can use the methods, the more people have a motive to buy the book for guidance in how to use them.

SUnit Port
No explicit work was done at CS3 but as the SUnit 3.0 webpage says they want a VW5i+Envy port, I note that:

- Joseph Pelrine has one in his office.
- Caersten Harle and I created another on Sunday for the Frost work (but JP's is the more tested)
- The SUnit.dat (presumably SUnit2.7 or 2.8) distributed with VW5i.2 seems to be broken; it hangs forever while loading (I observed this on my configuration at home; Caersten saw the same on his at CS3).

Frost
The task is to port from VW3.0 to VW 5i and extend it. Caersten Harle got the current state from Kerry et al. as a VW 5i Envy file. Hard work by Caersten and Roel Wuyts on Saturday succeeded in loaded it (there were slashes in the application names, which standard Envy dislikes) and in demonstrating that 3 + 4 = 7 in a Java workspace.

On Sunday, Caersten and I ran this as an SUnit test but our next test, to create a Java class, failed. We were blocked by the classic gotcha of all who dare subclass the compiler; the class at the very top of the specialized hierarchy (JavaObject in Frost) must be compiled in Smalltalk but must return the specialized compiler class when its subclasses send

```
compilerClass
evaluatorClass
classEvaluatorClass
```

Any problems in this mechanism means you can't inspect the code you want to debug (calls wrong compiler to do so), you can't debug the mechanism itself (endless 'self halt' loops), etc. (Years ago, I was one of a team who added rules to Smalltalk by subclassing the compiler and I well recalled repeated failed Envy loads, etc., due to the compiler switch happening too soon or too late before we finally cracked the pattern. It should be documented as it's a powerful pattern for compiler extension. Needless to say, I didn't have my old notes on it to hand. Sometime I'll

write it up as an ST chronicles article if noone does it first.) Hours of debugging later we sorted this out, reinitialized java.lang.Object and rewrote the code to create Java classes in namespaces, 5i-style.

Last thing on Sunday, we encountered Envy login bugs; the code we had been given was using Envy to store and retrieve the meta-data once created (you don't want two java.lang.Object's floating around in the system). Rather than dive too deep into this, on Monday, Caersten et al. moved the current state to Store. He now has a parcel and Store version and (with help from Alan Knight) this will be made available for future work. Thus there will be a single version of the code, not the several versions that Caersten found and had to sort out.

Future work:

- write more tests to verify it is working in 5i
- extract the Java parser as a separate utility to analyse Java code
- upgrade to the latest JDK
- a Java-Smalltalk converter would be nice, albeit a bit harder

ESUG Programme

ESUG was sponsored this year by Cincom, Roots, Georg Heeg and Daedalos. Roots paid for attendees' copies of an ESUG CD containing several non-commercial Smalltalks and an impressive collection of tutorials, books and other material. Cincom handed out CDs of VW5i.4 Beta and ObjectStudio 6.5 Beta. Professor Unland welcomed us to Essen and explained his new system engineering course' interest in Smalltalk. We thanked them all and started.

I have sorted the talks into various categories:

- Writing Smalltalk: eXtreme Programming, Teleworking and other software development approaches and tools
- Configuring Smalltalk: configuration management and allied topics
- Using Smalltalk: Smalltalk Web Frameworks
- Using Smalltalk: Applications
- Extending Smalltalk: Languages, Types and Aspects
- Miscellaneous and Impromptu

followed by Other Discussions and my Conclusions.

Writing Smalltalk: XProgramming, Teleworking, Software Development

Peter van Rooijen, Jan ter Haag- Test Driven Development

A majority of those listening to this talk had used SUnit (only 5+ for almost all of their development, only 20+ for a sizable part of it). Peter wants a tool that integrates test-driven development with the IDE, not an add-on. With SUnit, the user drives switching between tools; Peter's system aims to guide you to the correct tool. It also aims to support XP by prompting the user only and always to write production code to pass broken tests.

The tool window has panes for TestClasses, testMethods, test defects, the current test script, the test batch (the current subset of all tests that you want to run to drive your current work) and a status bar (as in TestRunner). In the test defects pane, each line summarises the assertion and the types of objects in it. Menus let you work on selections in each of these panes and on production classes that are the subject of the test, e.g. by redefining a method from a superclass to it. (As they are changing it continually, the demo version also has an 'editMenu' item at the foot of each menu; I liked that and would like to see it as a toggleable feature of the released version, and also of some other systems I know.)

The tool's main feature is to prompt the user with obvious things to do next and to make it easy to track where they have been. Peter[1] demoed by building a small system to handle distances (writing them, calculating them, etc.). First test case: express 1 metre in Smalltalk code ('1 m'); tool prompts for test class and method names, test application with prerequisite application (test framework is default, usually one would also need production application being tested) and runs test case. All is set up and first failure displayed in the comprehensive window (except that the test succeeded to Peter's surprise; after he deleted some code left in the system it failed).

Debugging shows the debugger with extra options, one to provide the missing method. The user provides the method's application and category (tool offers suggested category guessed from method name and arguments); it's created with implementation 'self halt '(to let you carry on elsewhere if you want; the tool can toggle regarding halts as test exceptions or to resume them) and user rewrites. Peter reran the test; debugger prompted him to define missing 'Meter' class, choose superclass. Then he got a doesNotUnderstand and wrote the 'm' method, being prompted to define the instance variable required.

He then looked at 'recently modified methods' to see what the test-driven development of this test had caused him to create so far. A redefined comparison method was needed; the tool prompted him with the class (Meter) that needed this change and the default (= instead of ==) comparison code. As soon as he accepted the change, it ran the test again.

The tool aims to rerun the tests automatically whenever you accept code written in response to a test failure. The motivation is the XP theory that you should only write code to pass tests. Write a new test and the tool only runs that new test because you have not yet changed any production code so could not (yet) have broken existing tests.

Next Peter wrote test: 1 m not equal to 2 m. His first change to production code to pass this test reran the whole batch and broke them all. Writing the missing accessor, as the tool prompted him to do, made them all pass. Next test: 1m + 2 m equals 3m; it fails as there is no definition of + for these units. The tool threw him into the missing method which he easily rewrote and accepted, whereupon the test batch ran automatically, all passing.

1. Peter talked while Jan drove the tool; for simplicity, I shall just write, 'Peter did'

Next test: print these units; 1 m printString should equal ‘1 m’ but failed. Peter’s background knowledge of Smalltalk told him to redefine printOn: not printString. Peter deliberately made the method return ‘1 m’ (hey, it passes the test and it is the simple(minded)est thing that could possibly work :-)). Next he tested whether 3 m printString equalled ‘3 m’; it failed. A keyboard short-cut took him to the recently modified printOn: which he now fixed and accepted, whereupon the batch ran and passed.

Q: testPrinting (my name) v. test5 (Peter’s name) for a test. I asserted that sensibly-named tests were of benefit and wanted the tool not to offer test + <Integer> defaults; Peter debated this.

Next test: 1000 m equals 1 km; failed. Peter tried

```
km
  ^self * 1000 m
```

(first omitting the return, his most common Smalltalk coding error). Testing that km printString is ‘1 km’ fails as it returns ‘1000 m’ so we need to refactor. (Looking at code showed redefined methods hadn’t inherited the category of those they overrode; something to fix in the tool.) Peter added a break point, ran to it and replaced 1000 m with Kilometer new. The tests threw him into creating Kilometer, giving it number instvar, etc., but still failed. Peter invoked ‘redefine’ to create Kilometer printOn: method.

Looking at the code, a Distance superclass was plainly needed. Peter created it and drag-dropped (VA, not tool, feature) Meter, Kilometer to be its subclasses. The session became audience-interactive as he used the RB to pull up instvars, accessors, generalized creators. As it was time for coffee, Peter deactivated one test from the batch (a convenient feature :-)), made the other work and pushed up the methods.

Currently Ensu runs on VA and Dolphin; more ports are planned. They have yet to decide whether to sell it, offer it as a free goodie or what. One issue is the fact that they have diverged from the SUnit core. I very much hope that they will reunite with it (e.g. by becoming a skin on the core) after this period of experimentation. Meanwhile, they believe that SUnit TestCase classes can become EnsuTestCase classes and vice versa by just changing the superclass.

Till Schümmer - Tukan: distributed cooperative programming in VW
Till (Till.Schuemmer@darmstadt.gmd.de) works at the Technical University of Darmstadt and at Fraunhofer-IPSI. Till spoke and his brother Jan helped in this talk (Jan spoke and Till helped in their talk on the K-Infinity knowledge management application).

Tukan is a tool for planning, designing and programming in a distributed setting. They believe in eXtreme Programming. Standard XP expects its practitioners to be co-located, its customers to visit or be visited by the programmers, etc. In the real world, all these people may be far apart; getting them together costs time and money that could be better spent.

Till opened the planning tool, as did Jan on the other machine. He selected a story card and added his user representation to it, indicating that he had begun work on it. Till then invited Jan to discuss it with him; Jan typed some text and accepted it, making it appear in Till's card. Icons show who has been asked to do a task, who has confirmed, who is currently working on what, etc. There is also a voting mechanism to decide how important people think a story is. cards can be linked to other cards or to diagrams.

XP is hostile to formal design documents but is happy to use informal diagrams to help illustrate ideas. Till drew some class diagrams with Jan, and played a 'hide and seek the class' game with his brother :-) to demonstrate the excellent responsiveness and speed of transfer of control. Where each user was scrolled to was visible to the other user.

Tukan programmers use a collaboration-aware refactoring browser. There are two modes:

- Semi-synchronous: user is made aware of possible conflicts by icons (weather-symbols; sun for latest version, rain for conflict) shown prefacing class and method names. Till worked on a class while Jan ran some tests. Jan promptly saw that he was testing an old version of the class; additional menu items let him see what had changed and load that if he wished.
- Synchronous: if Jan decides he wants to pair-program, he can look at icons showing what methods and classes are being worked on. If he sees e.g. someone working in another method of his class, he may talk to them by sending a chat method. Till and Jan exchanged messages to say 'What are you doing?, Shall we pair on this work?'. (After notionally establishing voice communications by telephone) they shared their refactoring browser and pair-programmed, easily swapping control of the text they were editing on a word by word, even character by character basis, the watcher seeing what was typed as soon as the typer did, much as if they were in the same room viewing the same monitor. Light underline icons and outlines showed each where the other's cursor was, what area each was scrolled to view, etc.

There are various patterns they can enforce regarding who can accept code changes. A good audio connection (e.g. a headset phone, not a hands-free area phone that picks up all the background machine noise) is important.

After they had refactored the code, Till returned to the planning game and saw what had been pair-programmed against this story card (the code they'd just changed), by whom (him and Jan) and when (just now).

Usage experience: they used Tukan to implement parts of itself and found it easy and fun to use. The Envy features are bandwidth intensive; otherwise the tool works acceptably over low-bandwidth e.g. 56k.

The Tukan Software Space tool assists in representing a software project (so arguably belongs in the next section on CM as much as this one on XP). Nodes represent artefacts (classes, methods, etc.) and weighted edges

represent semantic relationships between them. The tool generates such graphs (by static analysis). Till browsed the code to compare with the graph, explaining the simple weighting metrics (direct call, indirect call) used. He then taught the system by telling it he was an expert user and browsing the code. The system learns which parts this expert user thinks are close together and changes their semantic weights appropriately (experimental uses of this have given interesting results). They use layout and colour to illustrate the 'nimbus' (current focus) of a user.

Q. Also use the output of Michel Tilman's dynamic analysis tool (see discussion sessions) to compute such graphs? They will think about it.

They built this system from Envy, the Refactoring Browser and their Coast framework. To this, they added two more elements, the Tukan Software Space tool and the Tukan UI and apps.

Coast works by capturing Controller change messages

```
state: oldState changedTo: newState in: aBrowser
```

to ApplicationModel, and ApplicationModel's messages thence to the domain Model

```
logApplicationNotice: aString
```

Tukan is built in VW 3.0 + Envy. They are porting it to VW5i.4 and Store. They want people to use it and give them experience reports. They also will increase the number of tests (they began working on the cooperative framework before they knew about XP). They are also looking for other uses. One idea arose from noticing that they could parse Amazon's 'who is browsing/buying what books' information into the same style to build e.g. a collaborative book buying tool; any market for this?

Q. also use for pair programming when co-located? Yes, now they have the telepointer tool, sitting side by side viewing two Tukan-connected machines is as good as (maybe better than) two people crammed in front of one machine.

Q. use the Stable Squeak design instead of Envy? They will think about it.

Q. Do you present this at XP conferences (it might make them see what Smalltalk can do)? Till presented a design paper at XP2000. He may demo at another XP conference; as yet it is unclear whether this can be organised. (Off-line, I agreed to work with them on a scenario that would convey the tool's power, using my former NetMeeting-based teleworking experience. From my knowledge both of the value teleworking can bring to XP and of the limitations of current tools, I was impressed by this presentation.)

Greg Hutchinson - Producing quality code: Supporting the process
Greg is a mainly VA-using consultant with much experience of Smalltalk's use in finance. His talk was an experience report on his use of XP on a project in Switzerland. He likes some things, had problems with others and implemented some processes to improve XP support.

At starting, they wanted:

- zero test cases failing: no release if any tests don't pass
- zero load warnings and packaging warnings
- zero EnvyQA and Smallint warnings: when someone looks at these and decides they're O.K., they should not appear again
- no category-name usage inconsistencies
- no 2-3 weeks testing-packaging hell

They liked SUnit-style test-driven-development, which they did in TestMentor (TestMentor was the site's testing tool, therefore they implemented an SUnit framework in TestMentor.) This gave them more confidence in their ability to complete the project. It also let the framework team enhance the framework, which they always wanted to do, without getting complaints from the application team, as previously routinely happened when the framework changes were only discovered later to break the application. However their tests grew until they took an hour. They moved them to a central machine but then that began to bottleneck.

They liked daily image building and packaging. They always had a stable product for demos, fixed build errors early, recovered from image crashes easily. The set up a 'theProduct' icon on the boss' desktop; he liked always seeing the up-to-date state. However image/packaging is time-consuming. They automated the process; developer versions their changes as a config map and submits to central machine which dispatches it to a free machine which loads it, runs test cases and sends email back with results. The same process ran a main test task at 04:00.

Accepting runs Smallint so you fix the warnings, or review and ignore them. Envy/QA has ignore sets (for reviewed, not-a-problem warnings) but they are not version controlled with the rest of the code. They implemented a means of saving ignored warnings with the method so the next developer did not see them. Part of the release process was for the reviewer to run a report of 'ignored warnings' and review any questionable ones.

Q. what happened when changing code made the warning irrelevant? There was no process for removing the 'ignore' save; had to be done manually.

Greg implemented selecting method category names from a predefined list (via browser dynamic submenu). He demoed how some of these names were updated automatically: (re)write an accessor and it gets put in 'accessing' category. Except for the calculated ones, they were mainly interested in whether a method was part of the class' public interface or a private method. They added a Smallint warning if a method that only occurred in a private category was called publicly. This warning could be ignored in a way that only applied to that class. In effect, this was a 'friend' implementation, recorded by changing the category name to hold the friendly class' name (but thus you could only have one friend, plus a change to the class' name would not change the method category name

until/unless the method was re-accepted). For overridden methods, one menu pick was ‘use superclass category’. They had some special basic category names that could be combined.

Q. Roel Wuyts has done similar work identifying template methods, hook methods and clusters, i.e. groups of methods that interwork. These could be added to this system? Greg will discuss with Roel.

They made it possible to run the tests from the coding browser, obviating the need to switch from it to TestMentor UI and back. The browser found the appropriate TestCase class for an application class, or the TestCase classes in a test application for an application, and ran them. Developers had private developer maps that tracked the deltas they had made to a given base. Whenever they editioned an application, the new edition was added to their current developer map edition automatically. When the developer’s map was versioned, a single menupick sent their work to the remote machine to run the test cases; typically in 25 minutes an email returned saying whether it passed and what failed if not, after which they could release their work to the main stream. Occasionally, another developer would complete and submit to the main image while they were testing; this was caught and warned (had to re-submit).

Greg’s system also had some hotkeys for rapidly adding method calls and their parameters to code, etc., which he demoed.

Q. could the system just queue any maps that were submitted while another was running until first completed? That behaviour could easily be added but it was too rare a problem for us to bother.

Q. Greg had a special generic TestCase class for window tests: setUp opens window, tearDown closes, and for the tests he wrote scripts just as in other test case. I asked about SUnit-type UI testing versus using the more elaborate TestMentor ‘user action recording’ utilities and, to my surprise, Greg said he found he made little use of the latter. He found his pattern for writing raw SUnit tests adequate for most cases. I urged him to put examples on the web as many people were unsure how to write UI tests in SUnit. Greg said he would. I also asked about getting this toolkit as a VA goodie; Greg said he thought it could be arranged (I expect to contact him in October to follow up on this.)

Piotr Palacz - Modeling and roundtrip for Smalltalk in Simply Objects
Simply Objects is a meta-modelling tool that supports reverse engineering. I missed this talk and so did John; see Piotr’s slides/website.

Andy Berry - Making Software Simpler
I lunched with Andy on the conference’ last day but completely forgot to ask him for details of his talk. John missed it too. See his slides/website.

Alfred Wullschleger - Fractal design and development techniques in Smalltalk
Another talk I missed. Read John McIntosh’ summary.

Niall Ross, XP-rience: confessions of a (not that) extreme programmer
If I dumped all my detailed slide notes into this document, I would wildly unbalance my ESUG write-up. Hence I will restrict myself to summarising some lessons learned. To see how well I grasped XP earlier in the project whose experience this talk reports on, read my ESUG99 report summary of Kent Beck's talk and my ESUG2000 summary of Joseph Pelrine's talks.

Much experience is learning for yourself the truth of what you were told. The need for speed in running the tests, preached by Kent Beck, is one lesson we relearned ourselves. Don't wait (as we did) till the tests take an hour to run. You will put off running the whole suite till the end of the day, then wonder which of the many things you've done today caused the error you now see in someone else's feature. Refactoring is for tests, not just for code (but see Greg's talk for another approach to handling this problem.)

Beware necessary but not sufficient. It's easy to write a test the system must pass, harder to write 'enough' tests. After discovering the problem, we used partition analysis: list the axes of the problem domain and the important partitions along each axis; gradually, add tests to cover the cross-product

- services: virtual leased line, virtual private network, protected ...
- configurations: star network, ring network, ...
- lifecycle states: request, creation, fault, reroute, deletion, ...
  - faults: cut, fade, corruption, ...

We used inheritance and delegation to combine the test cases in the ways we needed. Only by formalizing the pattern for writing test cases that is (I believe) implicit in Kent's original paper could we easily understand, reuse, inherit from and delegate to each others' tests:

- A test case implements a generally useful scenario. Create scenario in test case' setUp method, e.g. setUp multi-layer network configuration.
- If you need another scenario, create another test case. If new scenario is a variant of old, subclass old, e.g. write a subclass to provision a Virtual Private Network over nodes of the multi-layer configuration.
- Each test method of test case verifies a different aspect; it may change scenario's state but not its nature, e.g. one test method of the subclass verifies that the VPN has sufficient bandwidth reserved on lower layers. Another cuts a link it used (i.e. changed scenario state) and verifies that the VPN's state changes to being alarmed, etc.

Equally important is completing the assertion set in each tests. Two wrongs don't make a right ... but may easily make a test appear to pass. Early on, we wrote tests to realize pipes set up in one telecoms layer by seizing resources from connections in a lower layer - but forgot to assert they ran in the same directions, until we met puzzling errors later on.

More generally, all tests are partial constraints on the system, used by XP to drive coding. In some cases it became obvious that what we were writing were formal-method-like constraints: commutativity, transitivity and other requirements that had to hold for all instances of certain generic classes.

For example, the far-end IP address that my service' IP address is talking to had better be the same as the IP address my service' far-end is using, otherwise, though we both have the right kind and number of resources, no communication will occur. This is simply an instance of a general commutativity requirement in our domain:

```
item supporter farEnd == item farEnd supporter
```

We pushed such assertions up to tests in generic TestClass classes, which held the objects on which they were invoked in instVars, and arranged for them to be invoked on said objects as part of all subclasses' tests. I suspect a better solution could be found.

Another rule we grew into was to write tests, not design documents and not task lists; make tests the way you capture your thoughts about what to do. We were a teleworking team. To fully involve remote team members, we met at our desks, wearing our phone headsets, our monitors NetMeeted together. Thus we always had a shared IDE to hand. At first we wrote tasks into meeting minutes and Excel spreadsheets. Then we realized that the best way to define a task was to write a test for it. A method name and comment is enough in a meeting, though sometimes some rough code would help understanding. And if it was sometimes slower to find and/or agree a test class and method name than it would have been to add a task name to the meeting minutes and move on, that was a benefit too; fewer tasks! (Side effect: Envy + method comments were our process tool; when was this test/task raised? It was O.K.; will Tukan make it, or Store, better?)

I remarked that if Joseph's talk on test resources and evolving hard-to-compute test results had been made at ESUG99 instead of ESUG2000, I could have told my team, "This is what we should do.", instead of "This is what we should have done." Lack of a test resource pattern (mainly in tearDown, not setUp) was the reasons our test suite time grew to an hour. We blundered into the evolve-results pattern; it's better to plan it. Following (a defensible paraphrase of) Kent Beck, we ignored UI testing, and suffered when our meta-data-driven UI was broken by refactors not long before demos. We needed TestMentor or Greg's SUnit UI-test pattern.

XP says all coding is refactoring. At first we used refactor-to-pass-tests to drive our main work, but casually fixed bugs if we saw them in the code. Soon, we learned not to fix bugs without first writing a test to elicit them (maybe it's not a bug, maybe you didn't fix it), and to at least think about whether to do an elegance refactoring without a test to prompt it. We also learnt that refactoring needs the process-support of scheduled incremental development. You must break to remake, so must discipline customers and managers not to demand deliveries within increments; *agree* an increment cycle and then (make them) stick to it.

As our project involved heavy analysis modelling to understand the domain, the whole team were very aware of architecture and design methods and this made us think about their role in XP. Our conclusions:

- While pair-programmed, we often drew diagrams to explain ideas to each other. XP shifts the paradigm, making tests configure-controlled items you must have, while designs are just optional quickly-sketched 'what I thought that day' extras. If diagrams communicate, use them.
- XP tests demand a kind of abstraction; to ask the question without implying an answer. This can be hard and an up-front design (or, one could say, problem analysis) can help you find what tests to write, both in starting and major refactors. The smoothest major refactor I ever did was delayed for some days, during which I sketched my ideas for it.
- Refactoring needs the ability to move behaviour in small units (see e.g. 'Smalltalk Best Practice Patterns'). Any barrier that imposes a tax on refactoring across it puts you back in the pre-XP world of high cost-of-change, therefore up-front design, so justifies an architecture. When we coupled our Smalltalk system via CORBA to Java UIs to do demos, these UIs being written by another team across the Atlantic, we could not casually move small units of behaviour between the two. When problems with data caching emerged, it was obvious more up-front work to define the interface would have helped. In an XP project, identify any irremovable barriers that tax refactoring (administratively or technically) and define their interfaces; that's your architecture.

Except as above, we confirmed XP. XP says designs will be wrong / out-of-date; ours were (even one written after the work it described by the one who did it :-)). Uniting our electronically-sketched diagrams with the code versions they related to might have helped their reusability (Envy can do this but you need Joseph's book or some spare time to work out how). Just as code refactoring needs a tool that understands the metamodel of code, i.e. the refactoring browser, so model refactoring needed a tool that understands the metamodel of our models, and we didn't have one. Visio was just a drawing tool, like coding in a text editor. Rational Rose was hopeless, its hard-coded metamodel at war with what we wanted. Meta-modelling tools like DOME and MetaEdit are the only ones which can bring *any* modelling value to XP; I'm unsure about how much.

Pair-programming was the last part of XP we used, for several interesting reasons. Firstly, we were sure we didn't have time to pair-program. We slowly learnt we didn't have time not to; too many misunderstandings and mistakes were made that pairs needed to refactor. Secondly (being both British and computer nerds :-)), we disliked the idea of 'excessive' time with each other instead of alone with our machines. Exacerbating this was the team's awareness of what such time was like previously. We went into a meeting room and discussed problems, away from our machines, so away from the means of making any progress towards solving them. By the end of the meeting, we were tired and depressed from discussion without action. Surfeited with each others' company, we retired to our machines and worked with minimal interaction, until the lack of same caused divergence that necessitated another meeting. When I finally and gradually compelled the team to pair, all this changed. Pair-programming replaced 'meetings, the substitute for work' with meetings at which work got done. Increasingly our time alone was spent browsing code, having ideas, trying

things out; anything *except* writing production code. We came to pair sessions (sometimes triplets or quartets) eager to code and show our ideas, not resentful at being interrupted in our work. Thirdly, there was what I called 'shy arrogance': "I can't pair till I've worked out how to do this task, or I'll look stupid." I defeated this by getting the team to talk about it.

Both our experience and research we commissioned (from Prof. Laurie Williams of NC State University) gave similar results re pair-compatibility. Two experienced people will storm ahead. Two novices will help each other. Don't pair an extreme introvert with an extrovert but with a less-introverted colleague; they'll be socialised by learning that others can make mistakes too. Two extroverts will be happy in their work, but slow. The least satisfactory pairing is between an expert and a novice. When using this to train new team members, we sometimes found it best to pair-program the test only, letting the novice write code to pass it in their own time (with occasional help). When the novice is 4 times slower than the expert, the latter must give them time to catch up; long pairing sessions simply lose them. Generally, pairs took the same net effort (research stats, excluding 2 extrovert-extrovert outlier pairs) or less (our experience) than un-paired to complete tasks, and their code was of higher quality.

I shall skip my disquisition on teleworking; read my Tukan talk write-up (including of my own questions) to get the key points. With adequate tools, teleworking and XP enable each other, making a potent combination. It saved us more than once when those who had to pair to achieve a task could not afford the travel time to co-locate before the deadline. As a happy side-effect, it also made us much more able to work with remote customers.

I also gave instances when testing, refactoring and pairing had interacted to support each other, described administrative problems our XP team had faced in a non-XP company and how we (partially) solved them, and said how my process of growing an XP team differed from Kent's prescription (offered as my experience, not claiming it as better). Lastly, I talked of XP scaling up and handover procedures. See my slides and notes for details.

Configuring Smalltalk: configuration management and allied topics

Roel Wuyts - Lightweight Classifications

Smalltalk IDEs are good coding environments but have mainly static ways of grouping; surely this is contrary to Smalltalk's style. The classification browser aims to unite static and arbitrary dynamic classifications of objects, source code, documentation and the classifications themselves. The original implementation of Koen de Hondt (see my ESUG99 report on it) was heavyweight. Roel and Stephane have re-implemented it in a lightweight (5 basic classes for framework) form in VW5i.4 + Envy.

Roel launched it and demoed, starting from the root classification. The tool has a tree in its left pane, displaying each item with text and icon (you can choose your own icons). Selected classifications show their classes, methods, subclassifications, etc., in middle and right-hand panes. Drag and drop moves them / adds them to your classification. Classifications are:

- extensional: contain whatever user has dropped into them

- intensional: find and contain all objects satisfying chosen criteria (criteria expressed as a block)

Classifications have editable context-sensitive menus. You can choose what editor to invoke on items, and have several choices in its menu. Roel can invoke either Envy and RB editors on classes, for example. When invoked, an editor appears in a pane within the overall window, meeting another aim of the classification browser, to avoid creating vast numbers of windows while editing code.

The lightweight classification model is very simple. Item has subclass Classification (contains Item) which has subclasses Intensional and Extensional. An Item has an ItemVisitor. ItemVisitor has subclasses Children, Deeplabel, Editor (which has subclass EditorRB), Iconizer, MenuBuilder and Shortlabel. In an Envy extension, they added just one method Object>>itemActionFor: to connect to the framework.

Classification is also a kind of Collection. It uses the doesNotUnderstand trick to get collection functionality (isCollectionMethod => handle, else super doesNotUnderstand).

The browser is one class (as it reuses all existing browser code). The leftpane TreeView is Koen De Hondt's RedRobin widgets (better than Aragon's tree viewer and much better than ICC's in Roel's opinion). Other panes use existing tools, with drag and drop between all panes.

Koen's thesis describes ways of using this tool to understand complicated systems that are new to you. You may create an 'I was here' classification to recall where you've looked, rough classifications for things that help you understand one aspect or seem related, etc.

Get it from www.iam.unibe.ch/~wuyts

Joseph Pelrine, MetaProg GmbH - Stable Squeak Explained
Squeak is fun but not designed for high-performance production coding. It's not cleanly coded and some parts are not clean.

- There is a compiler method that invokes Morphic; can't ship headless compiler
- File-out change-sets; you can find that what you thought you filed out was not what others filed in.

Squeak was written by visionaries who were above mundane production details. It's free but, as of today, the cost of a commercial Smalltalk licence you thus save simply returns on you in the extra effort of handling Squeak's poor configuration.

The Squeak world tour is about building a professional multi-developer stable clean version of Squeak. The participants (Joseph Pelrine, John Sarkela et al.) have worked on Smalltalk configuration tools before and see Stable Squeak as a chance to answer the question 'what would we do

different if we did it again today?'. They decided to go back to X3J20 (ANSI Smalltalk standard) despite it's being as notable for what it doesn't define as what it does. It gives a declarative semantics for Smalltalk. Joseph aims to add to this a flexible Smalltalk structure.

Smalltalk declarative semantics: in Smalltalk, we send messages whose side effects create new objects that implement program elements; thus Smalltalk creates classes, globals, etc. We do this in browsers, workspaces, batched via file-in, SIF or XML format. The execution mechanism is imperative (e.g. draw rectangle is 'move pen 5 right, 5 down, 5 left, 5 up). Imperative tells you how to do it, not just what to do. It can be implementation-dependent and may require a suitable initial state. This affects how we reproducibly regenerate the program from our configuration management system. The typical Smalltalk image contains base classes, tool classes, experimental or old but undeleted classes, and application classes. Package strippers aim to remove the classes we don't need in all these categories.

As against this, X3J20 declarative Smalltalk defines language elements, syntax, and static and runtime semantics. It says what the properties are (e.g. square is 5 units in length) not how to achieve them. This is implementation independent and has no initial state requirement (may declaratively define initial state requirements). With SUnit, Joseph et al. have written ANSI compliance easily (main problem is variations in the various vendors meta-protocol). A Smalltalk program is

- class and method definitions: class is a unique global name bound to the class object (N.B. declarative syntax does not support changing runtime binding).
- global definitions: name and optional initializer
- pool definitions: contents and their initializers
- an initialization sequence to start the program
- (There are no namespaces in X3J20. At CS1, all the VM architects agreed a declarative namespace syntax in a meeting in Joseph Pelrine's hotel room from which he was excluded!! :-))

The program is a sequence of these elements, their order determining initialization order. X3J20 also almost defines an interchange format; CS work is working on a portable binary interchange format.

All Smalltalks to date also make certain implementation assumptions:

- globals, pools stored in dictionaries
- methods are objects
- methods stored in dictionaries
- objects behaviour implemented by class object
- each class has associated unique metaclass

X3J20's declarative smalltalk does not *require* the above. It describes how to build a program before it executes, not the run-time behaviour. It neither requires nor prevents meta-protocol for reflection. (Stephane Ducasse remarked that SUN says they will make Anamorphic Smalltalk available soon, 'end of summer'; it offers a form of declarative reflection protocol.)

Kent Beck: "Learning about reflection is important to a Smalltalker. Even more important is learning when to use it and when not to."

The aim is to provide a rich hierarchy of loading exceptions (e.g. class or superclass binding clash throws exceptions on loading) allowing roll-back or other handling (e.g. maybe someone writes a way to move clashing class or method definitions to different namespaces as they load).

Q. initialization expressions could re-insert state dependence? Yes, you can't validate the initialization expression: it's just a doIt that could have side effects. You assume the programmer will only do sensible things in them.

Joseph launched Squeak and browsed the implementation of these declarative constructs. AnnotatedObject and Definition; latter knows module to which it belongs and has reference to the object defined. Subclasses (GlobalInitializers, etc.) have pre-load instVar to ensure sane load order. ComponentSpec is a lightweight way to reference other definitions.

Traditional Smalltalk tools install changes immediately as the user accepts them (thus you can crash your system). Joseph prefers to manipulate the definitions and then install the changes. Because the process is now broken down into a number of steps you can have browsers on different definitions. You can protect yourself from sudden irrecoverable image breaking. Above all, you can plug in different engines for validation. Joseph has brought the refactoring browser into a private namespace and use it to validate code. For example, Joseph can write and validate VW code in his Squeak image. You can likewise install onto another image, e.g. install the VW code into a VW image.

In this model, you can edit all (even invalid) programs. These programs are completely specified, reproducible from source code and require no distinct packaging. Digitalk's Team/V Firewall prototype, lost in the merger, had many of these ideas. It allowed extremely small runtime images ('3 + 4' 10k image, full GUI apps in 0.5 - 2MB.)

Q. how does this handle incremental compilation (e.g. see a halt, add a comma to method and accept)? Joseph is doing things one step at a time; he thinks a remote debugging process, suitably defined, can be implemented on this.

Semantic leakage occurs when you manipulate objects in the image using tools that are not synchronized with the semantic model of the image (e.g. programmatic creation of method in workspace; you didn't create a

Definition instance as definition creation is tied to the browser). You can have tools to search your image for undefined methods, etc., but you will never trap all the ways users can programmatically get round your modularisation approach.

Definitions are contained in module, called Package. Packages only contain definitions. ScopedNodes (Joseph admitted it was an ugly name) contain ScopedNodes or ScopedModules. There is one predefined hierarchy (cluster) and you can define many orthogonal nestable clusters. From any cluster root there is a unique path to a contained definition. A definition may have many paths to many cluster roots. All clusters can be tested for visibility and closure.

Jinsu is Joseph's name for the project of slicing up the current Squeak image. He showed slide of the proposed slices: many boxes and one 'here be dragons' annotation.

Thanks to Alan Kay, Dan others at Squeak central, Allen Wirfs-Brock and Dave Thomas, Craig Latta and John McIntosh (flow framework), Goren Hultgren (Stable Squeak wiki), and Stephane Ducasse and Roel Wuyts (ESUG). Joseph is contactable at jpelrine@acm.org.

John McIntosh and Joseph Pelrine - Stable Squeak Demoed
They have not divided the whole system yet; there are still a couple of sections to work on. The first cluster is Kernel, with five subpackages Common Language Data Types, Core, Event Handling, Exceptions and Processes. As in Envy, classes are defined in one package and may be extended in others. Packages have Prerequisites and dependants. These are all displayed in the ModuleManager window from which other class browsers can be launched. More unusually, you can also launch browsers for Globals, Pools and Initializers; these are modified inspectors. Working with declarative definitions means that much work goes on behind the screens to calculate things, making the UI slower (looked O.K. to me in demo); they will optimise things later. A multiScope Browser browses a group of modules.

In e.g. Envy, the base is partitioned for you into maps and apps when you start up but OTI did not publish how they did that. Stable Squeak provides the class ModuleBuilder. ModuleBuilder>>modules and various other methods let you build modules. ModuleBuilder will move all your classes and methods into the right packages. Things it doesn't handle are left in unmanaged (but it will do its best to leave as little as possible). Editing ModuleBuilder>>coreClasses, >>kernelClasses, etc., lets you slice Squeak differently from the way the Stable Squeak group have done if you want a different image.

They have ported ModuleBuilder et al. to VisualAge quickly. The only delay to putting it into VW is that VW and their system both have a class called MethodBuilder. (NameSpaces are the way to solve such clashes.

Joseph had tried this but met a problem. Alan Knight assured him that namespaces would solve the problem; he agreed that doubtless he can fix it as soon as he has a moment to work with a VW guy.)

He then launched VW, VA and sent 'Hello from VisualWorks' as SIF to Squeak as global initializer, then 'Hello from VisualAge'. As the definition hierarchy is now in VA, he then told the definition to SIF itself and send itself to Squeak. The same process (telling the definition to SIF and send)

```
myClass asDefinition sifToSqueak
```

makes the class appear in Squeak, but in an unnamed scope because the class' category was not there (VA/Squeak mismatch; Joseph could have sent the category as another definition). Then he sent a method asDefinition sifToSqueak (appeared in 'as yet unclassified' protocol for same reason as above). Next he put a callback in VA so every time he edited the class in VA it was sent to Squeak. He added instVars and saw them appear in Squeak, edited the method and saw the changes appear in Squeak, added a new method and it appeared.

At this point, John McIntosh took over the presentation. Slang is a subset of Squeak Smalltalk which can be translated to C.

```
self translate: filename doInLining: inLineFlag
forBrowserPlugin: false
```

puts it into files, which include platform-specific stuff. Source code for all the primitives used to live in the image, which made change sets heavy. The next solution was class VMMaker. For plugIns, you have Smalltalk and code in the virtual machine. As Linux can have a monolithic kernel or a lightweight kernel and load kernel modules, so the plugIns can be present or absent. Typically, Squeak ships with most plugIns installed because breaking the original Mac VM into plugIns was not done; John has been doing it over the past month.

They got the source code out of the image and created a source tree separating cross-platform and platform-specific; John showed the folder-structure. VMMaker works out what is internal and what is external, finds the files and builds the appropriate VM (200k min., 600k typical). These files can now be distributed via CVS, etc.

Georg Heeg - Porting VW code to VW5i
The main issues are Store and NameSpaces. See John McIntosh summary, plus Georg's slides are available from http://www.heeg.de/esug2001.

Andreas Kuckartz - A Flexible Configuration Tool Implemented In Squeak
I missed it. See John's summary.

Roland Wagener - VW 5i.4 MacOS X

I missed this; see John McIntosh' report.

Michele Lanza - CodeCrawler
See my report on this in my ESUG 2000 report for background. Sadly, I missed it this time round. See the slides on the ESUG webpage.

Using Smalltalk: Smalltalk Web Frameworks

Tutorial: Mark Weitzel - VisualAge Smalltalk's Server SmallTalk
Mark works in VAS in Research Triangle park, N.C. Mark is responsible for stuff built on VAS. There is now no server runtime fee (they had one but they found out that noone knew and so noone paid it :-).

What is Server Smalltalk: communication, remote invocation, distributed object model and programming model. To use SST, you need to be happy with smalltalk processes, strategies and pluggable behaviour (so not for the beginner). Building a server application is not like building traditional client-server. You need to think about concurrency, global variables (can be a bad idea), caching, etc.

The problem is to manage a complex application that needs to be flexible and have separation of concerns and enable everyone to do things while requiring noone to know the architecture's guts. Distributed identity and distributed garbage collection are key issues. Key components:

- Dispatcher: sends request to Invocation Handler Policies for dispatch: immediate, threaded, logical (view of other machine), pooled (most often used, for specific object pool)
- Object Space: gives object location, manages identity across virtual machines, maintains import/export sets. Object handles are a (space, id) pair
- Transport: SST is transport-independent: TCP, MQ, HTTP, etc.
- Marshaller: sender/receiver agree on marshalling scheme which flattens request plus args to bytes, sends to remote machine which reconstructs request plus args, thence executes operation and returns results. similarly. Different types: native SST (one for bulk objects, one lightweight typically for reference), RMI for Java (needs user to do a little more than in Java), etc.
- all connected to the Invocation Handler: this is the interaction framework that ties everything together

Simple patterns of locally generated and remotely generated invocation use these to get stuff off the wire and into objects as soon as possible, and vice versa as late as possible. Messages can be asynchronous and synchronous.

The whole system is very pluggable: choices are not bound together.

Q NameSpaces? No; one can have two class definitions in the system and have them marshalled correctly (a la Java) but you shouldn't want to.

Distribution/Interaction semantics:

- Bytewise
- Value-based: robust
- Referential: SST (robust), General RMI or IIOP-based (not *as* robust due to RMI limitations)

Mark then launched SST, inspected the invocation handler and displayed a servlet in NetScape. Their HTTP model is based on servlets. They have and will release a JSP goodie, but it merely replicates what SST can already do. They can distribute objects ST - ST and ST - Java. SST 6.0 will comply with HTTP 1.1 origin server spec (i.e. can be origin or relayer of info, etc.).

WebSphere integration: SST can talk to WebSphere (or any similar HTTP understander generating the right kind of cookies) via the OSE plugIn, giving you load balancing, sever affinity, etc.(but not persistent sessions). The plugIn inspects the cookie from the server to decide where to return the reply; if the server has gone away, it finds another server in the server group. Services are defined by WSDL and exposed via SOAP. (OSE plugIn also gives access to all kinds of routing.)

```
(server := SstHttpServer at: url path: directory)
  sessionConfiguration: SstSessionConfigurationmanager
    new .....
```

It is very easy to write servlets. Contracts is oneArgBlock taking output stream, mostly simply provided in basicProcess: method, if you can use deferred content (write at last possible moment). If you need better than this (e.g. for 1:1 comms) it is a little more complex but still easy.

Smalltalk JSPs: you can write them in your favourite HTTP editor (from NotePad to WebSphere) and they look very JSP-like. Mark demoed one that was an interface to browse Smalltalk pool dictionaries.

Q. Can servers take down the server and/or step on each others' memory? Yes, as in Java, since a servlet that hogs memory will run; the server will listen, put it on another machine and do the best it can but the servlet mechanism is just another doorway into the image.

Smalltalk-to-Smalltalk Communication: configurations are meta-information for controlling instances at run time (and sometimes instance factories as well). SetInvocationConfiguration is the configuration factory. You copy the configuration you get from it so changing it doesn't affect whoever else is using it. URLs tell SST which invocation scheme to use; all images involved in an interaction must use the same scheme.

Mark set up two images, made one the server and exported the transcript and some other stuff. Server specifies port, client can discover the space dynamically from first contact. He then 'inspected' a remote message call to show its returned result.

```
self invoke: (DirectedMessage receiver: . selector: .)
```

He then demoed remote 'Transcript show: ...'; marshalling the transcript by value is not a good idea. So one can send the message asynchronously (displays on Transcript, ignores return). If you need the return value, you can use replacementWith: to get another remote object returned your local transcript (e.g. nil salary in returned person object).

Application contexts allow you to organise things in a distributed application so things you import and export maintain themselves sensibly. Typically an application has one or two application contexts. Object spaces are simple or complex: SstComplexObjectSpace manages the object identities and import/export sets (i,e, it is the master, its slaves being SstSimpleObjectSpaces). Mark then demoed defining his configuration in the two images, starting the server and making a block object global in the server image available and sending it value from the other image. These symbolic references have no notion of identity (I noted this is a standard feature of distributed systems, proved at ANSA in '80s). To clarify this, he inspected the remote transcript (inspect shows transcript, basicInspect shows remote reference). This was not a symbolic reference but an integer key into the import/export set of the ApplicationContext (was number 11; first 10 are reserved). When doing a 3+ image environment, the sets need to be coherent; Virtual Private Networks are a case in point - you don't want to set up A to see B to see C who can't see A. This can cause the bogus receiver error: invoking on something you're not allowed to see.

Usage example: major car company uses SST to control production line e.g. checker talks to line to see that bolts are torqued correctly; if it were to fail that would be $10,000 per minute or more; SST is robust.

They have naming services: export your Smalltalk objects to refer to them remotely without giving image addresses. They have distributed garbage collection. When being a Java RMI, distributed garbage collection needs some management (typing needs management). They advise not to build full RMI (not a Smalltalk problem nor a Java problem, it's just difficult - creates spagetti); instead create a facade in Java or Smalltalk and pass objects by value between Java and Smalltalk. Their RMI supports JavaSecurityPolicy file (J1.2 and later).

They provide a remote SST administration server and a logical debugger (when debugging SST itself always turn on debugging inspector when doing so to see actual objects not what they proxy, make debugger launch in offending image, etc.). Don't block the UI (i.e. send #fork) as it complicates debugging. (There is also XD debugger - you can debug on one platform for packaging to another platform - don't confuse this with the distributed debugger.)

Mark explained that the PingPong example they provide is meant to show *all* of SST's capabilities; get the first testcase working and then you can do all the others and should do only those that are relevant to what you need to learn. Don't assume you need to understand all of it before starting.

Platforms in 6.0 will include Linux. Neat concepts include callbacks, futures (proxy for value you're waiting for; simple way to handle blocking), tunnelling (debug by email communication between your two debuggers, bypass firewalls).

There are several large clients (a car company, JWARS), several smaller ones, and also several they don't know about as business partners sell it.

Q. Interoperate with VW OpenTalk? We are interested in doing that (maybe Camp Smalltalk project; VA will cooperate).

Q. Security? Currently they like to run behind WebSphere. In a few weeks they will release SSL. Smalltalk does not come with a security model, unlike Java, so you have to add one of these (but you can therefore control your security model better).

Tutorial: Mark Weitzel - Web Services
I missed this, as did John. See Mark's slides.

Alan Knight - The Cincom Smalltalk e-Business Platform
VisualWave was one of the first web IDEs and very good when it first appeared. ParcPlace management issues meant that at a crucial time it did not evolve, whereas the web was evolving a lot, and now it is a niche tool. The Cincom eBusiness platform is VW's general purpose web enabler. With its release, the 'i' in 5i, which stood for 'internet' is now fully justified (in 5i.1, wits suggested the 'i' stood for 'imaginary number' :-)). Alan's detailed slides will soon be available on the web. Below I just note some points not mentioned in his slides; read the slides for details.

Most of what Alan's demoed is on the Cincom web pages; go to them and you too can select the debug page, do remote debugging, examine the SubscribeServlet (look at Smalltalk Chronicles web page for an example of its use), etc. Alan demoed with the 5i.4 Refactoring Browser Beta, which looked good. Using the OpenTalk remote debugger (plus Martin's work) he showed how you could self halt, debug and resume a servlet, with the debug showing your code interspersed with single lines of other stuff instead of lost inside 500 lines of generated stuff ending with a less than helpful '`; or ) expected`', a phenomenon all too familiar to users of current industry servlet debuggers. WebLogic's hotloading in theory lets a Java servlet do something like halt-debug but WebLogic say 'don't use it', and they are a very advanced Java Server. Most JSP books' debugging chapters start, "Here is how to insert a print statement in your page", which shows the level of their debugging ideas. (Alan worked on Sun's EJB 2.0 spec. EJB is the reason he left to move back to Smalltalk.)

An organisational problem in using these tools effectively is that the web designers one works with don't think like programmers. Once and once only is not an idea they have. Current HTML editors are primitive. (A utility from Georg Heeg, to generate VW-browser-readable stuff from web pages, is now in 5i.4 and helps.)

Tediously, Microsoft have just dropped support for the NetScape API in IE6. This means the VW plugIn no longer works there. VW will fix this. FYI, Fast CGI plus Apache plus Windows has a bug; perhaps the relevant companies will get round to fixing it sometime.

Q. Smalltalk-friendly ISP? The only specifically Smalltalk-friendly ISP Alan knows is Peter Leut in Vancouver, Zouku-net, but most ISP's don't care; Smalltalk is fine.

Q. Why use a gateway, not a pure Smalltalk system. Firstly, EzBoard uses pure Smalltalk. EzBoard started with three months work, funded by a gambling win. Last year it had 6 million hits daily on 3 linux servers. This year it has 400 million hits daily on 600 linux servers. It is now the 30th website in the world in terms of volume. So pure Smalltalk allows a full scalable solution. However, you may want a gateway for corporate coexistence with other content (e.g. your site support people already know Apache) or to do authentication and SSL at the front-end so Smalltalk doesn't have to. Alan noted that Smalltalk can now do SSL (not *complete* SSL; they implemented most algorithms, not all) and that VW's SSL was as fast as the fastest C implementation and much faster than the slower C ones. This is because while small integers are faster in C than Smalltalk, encryption needs large integers and Smalltalk has these for free.

Q. use as a proxy server? Alan had an off-line discussion with the questioner. I think the answer was yes, with some technical commentary.

Q. examples of use? The www.cincom.com/smalltalk site uses it. Look at Smalltalk Chronicles for a simple servlet example (the subscribe widget). Log in to https://www-koethen.heeg.de/cgi-bin/tcm/Web-TCM.ssp using username studentin and password koethen or amalt (Georg couldn't quite remember which) to see a professional translator test application.

By U.S. law, car insurers must make certain information available, which they now find convenient to do by putting it on the web. One of Cincom's customers, Progressive Info (a specialist high-risk insurer), realized that most were putting so much information on the web that one could reverse engineer their rating engine, although the law did not strictly require this. PI did so and launched an 'if we can't give you the best deal, we'll find the best deal' service on their web page, thus getting 'first refusal' on much business, to their profit. This is the kind of eBusiness activity where Smalltalk's flexibility gives you an edge.

Michel Bany - Web-enabling Existing ObjectStudio Applications Using Cincom e-Business Platform
See John McIntosh' summary.

Using Smalltalk: Applications

Ernest Micklei, Philemon, 3D Models by Programming
The title ('by programming') stressed the difference between what Ernest (emicklei@philemonworks.com) was doing and other 3D tools; he can script (program) much better in Smalltalk but does not want to view in

Smalltalk as others have already written good 3D engines. He therefore supports other engines by adding new SceneBuilders (for VRML, Quake, Scol, Shockwave, Povray, Atmosphere). Using Scol as rendering engine he demoed creating 3D worlds suitable for multi-user use on the internet.

The procedure to make a scene is:

- have an idea
- create some 3D objects (e.g. from existing library)
- apply an appearance to each object
- add the objects to the scene
- write the scene to file(s)
- view files

His tool is implemented in VisualAge, but uses none of VA's visual stuff:

```
builder := SceneBuilder m3d (could be vrml or whatever)
WallPen new height: .. ; width: ..; texture: ... ...
builder addAlll: pen objects. ...
```

and thence he wrote a file and loaded the external viewer. The demo had created a room with an entrance. Editing the code, he added floor, ceiling, etc., and a polyhedral shape in a display case. Other tools create these objects drag-and-drop but in Smalltalk he has access to the commands and so can generate parameters, etc., so write complex programs to create complex scenes (as usual in Smalltalk, access is the key value).

He then brought up examples of his commercial work:

- a CenterParc model customers could walk through; trees, buildings, wooden staircases, complex accommodation block, etc. He created this in two hours.
- A law company client wanted a model of their building. He built this, saving himself time by exploiting symmetries in their building.

Voyager is their application for building multi-user environments where the clients communicate with each other and the world via avatars. He launched two web browser on an example from their website (uses 1MB plugIn), logged into the world, made an avatar in one, and looked at it in the other. The default avatar is just the company logo; he changed it to have hands, face, etc. For the future, he wants to attach physics to his objects.

Jana Hintze - Wompland
She has written a tool which lets models (generated outside Squeak in another tool) be imported into Squeak and then played with by scripting. See John McIntosh' description.

Alexander Lazarevic - Pooh
Squeak stuff; see John McIntosh' description.

Christian Haider - Fractalizing a Dodecahedron
I missed this, as did John. See the slides.

Christophe Roche, Univ of Savoy - Ontological Knowledge
The speaker has two enthusiasms: ontology and Smalltalk. Ontology is important; whenever you communicate and share knowledge you use ontology. Ontology is about definition; he presented examples, problems and their Ontological Knowledge system (OK Model and OK Station). The OK Model uses 60kloc Smalltalk; they chose Smalltalk because they had to in order to build the system, not just because they liked it.

Cooperation between multi-disciplinary people is a growing feature of e.g. virtual manufacturing enterprises, eBusinesses, etc. Knowledge sharing is key to this. The design of a product by a multi-disciplinary group is either a knowledge-intensive activity, or a new Tower of Babel ('If your team are talking about apples and oranges, your software could be a lemon').

The first problem to solve is terminology (I recalled the Oxford philosophy tutors' insistence on, 'First define your terms.'). The first term to define is ontology: what it means has caused heated debate in the field. Ontology can be regarded as

- a knowledge base, a set of concepts with their interrelationships
- a vocabulary of basic terms with precise meanings
- an enabler of interoperability, a means whereby agents can communicate
- a terminology, plus a meaning system that gives some guarantee that the terms will be used consistently within the domain

The speaker emphasised using what ontology was for, i.e. knowledge sharing, to guide you to a definition: ontology is a defined terminology and a commitment to use it precisely. Most ontologies reply on very fuzzy foundations, e.g. they do not distinguish between concept and set, between attribute and relation, etc. You need logic, language and meaning to communicate.

The Ontological Knowledge (OK) model uses:

- linguistics: meanings depend on other meanings and are differential, i.e. defined via binary opposites
- epistemology: concepts are defined by specific differentiation, i.e. a new concept is defined from existing ones by identifying its specific difference from the nearest other(s); differences are pairs of opposites, sets are intensional, attributes have domain, range, cardinality, and default value. One property of their epistemology is to have a single hierarchy of terms, not multiple.

Cristophe then showed screenshot slides of the system, selecting modules, mapping from other ontologies (easy for them to map from others, reverse is much harder and noone has), defining elements, etc. The next screenshot showed a workspace-like OK tool window in which the Smalltalk code

used to invoke definitional activities was reproduced as a log (some of the code had a Lisp-like look, influenced by earlier use of Lisp in ontological research). Graphical tools show the concept-relationship graph.

Their tool, written in 70,000 lines of Smalltalk, is being commercialized by Ontologos, a French start-up company. It has been applied as a classifier, an isomorphic engine, a document browser and a mail manager. The tool has many interfaces to assist in the many different ways of representing knowledge. The OK Station compiles an ontology (defined by user, provided in LOK or LRK files; these are existing ontology languages) to its internal representation. It can reproduce it as XML.

For implementing complex artificial intelligence (i.e. knowledge-based) applications, Lisp and Smalltalk are the only two languages worth considering. Their system also had complex software engineering requirements for which Smalltalk was best (noting that Eiffel, Objective-C, Java, C++, etc., could also have met them with varying degrees of ease). The combination mandated use of Smalltalk. The tool was built over many years by a multi-person team which at first suggested they use Envy. However they also had an interest in multi-agent systems. There are *very* many definitions of agent in the literature. They used FIPA's design (agent research project) to implement a publish/subscribe means of distributing parcels. They really want a new system browser to separate hidden and exported methods, and to have pre and post conditions to each method. They implemented in VW 3.0; they are about to port to VW5i.

Their customers include a prestigious French cosmetics laboratory (a definite feather in their cap for a French project, this :-) and an aerospace research unit. Natural language translation, using the tool's ability to identify meanings in terms of related meanings, is another area where they believe they can contribute (I noted that a major U.S. natural language project also uses Smalltalk).

Jan Schümmer - Knowledge Management in VW: K-Infinity
COAST is an open source Smalltalk framework for building synchronous collaborative applications. The Coast team have built their first commercial product, the K-Infinity knowledge management system. The lead customer (a German web portal) needed to perform complex searches of the web. Implementing a solution in Smalltalk proved a powerful and cost-effective way to meet their needs and adding Coast to it to enable non-co-located pairs or groups to search collaboratively was a pleasant bonus for them. Other customers for K-Infinity are appearing.

The Coast team are justly proud of their collaboration layer, which is distinctly superior to NetMeeting's in my opinion. Their underlying communication layer, although it also seems to perform better than NetMeeting's on lesser bandwidth, is simply something they wrote because they needed it, not the focus of their efforts. They therefore plan to replace that layer with OpenTalk, thus making their collaborative framework goodie even better integrated with VW.

This summary is brief because my XP-rience talk occupied the same slot in the other programme track. For more information, see John's report, my report of this team's talk on the Tukan XP tool, and my description of the Coast tele-collaboration framework in my ESUG 2000 report.

Cosmocows don't mooh, they Squeak!, Mathieu van Echtelt, Cosmocows, Holland
A hard deadline forced the speaker to cancel. (I filled the gap, with a description of my XP-rience.) See my ESUG 2000 report on Cosmocows for background information. Their website http://www.cosmocows.com/ will be updated soon (within two months) to describe their InternetInvolver work and related topics.

Thomas Gagne - XML
Sadly, the .com crash has forced Thomas' company to economise by cancelling luxuries like transatlantic flights to ESUG.

Extending Smalltalk: Languages, Types and Aspects

Tutorial: David Simmons - SmallScript
Alas, Dave's RAID disk crashed last week losing 90GB of data. Backup recovery and reconstruction of the week's work he lost swallowed time Dave could not spare, given that he had a contractual requirement to deliver certain things for mid-October. Hence he could not be here. This was a keen disappointment to me and to others; we were eager to discuss Smallscript and .Net with him. However, we all agreed we were even more eager to see Dave get it released than to see him at ESUG talking about it.

(Meanwhile the time was not wasted; we had the worthy replacement of Joseph Pelrine talking about Stable Squeak.)

Frank Lesser - Smalltalk's need of Operating-System Synchronization
See John McIntosh' summary.

Claus Gittinger - Liberating Objects from Language
Get away from thinking of Smalltalk as a language; think of it as an environment. Smalltalk is not always the best language for all problems; some languages are better at some problems. Java is not better at any problem but if a user could write Java script within a Smalltalk environment, maybe they could gradually learn to use Smalltalk. (Plus I know that the fact that one *could* extend it in Java can be key to persuading a manager to let you write a system in Smalltalk.) Another reason is that languages evolve. Java and C# are not the last word in OO languages; even Smalltalk is not the last word. You need the ability to explore new languages in the context of reusing code written in old languages. Hence, Claus wants an environment in which one can always choose the right language for the problem:

- OO: ST, self, Java, C#
- Functional: Lisp, Scheme, ML
- Logic: Prolog

The implementation language should be transparent to the sender of a message. Each class, and sometimes each method within a class, should be able to have its own implementation language. For this to work, we need a common object model (no primitive types !!!). so all objects of different languages can interact. NB common object model is NOT Soap, XML, ... and not a class-based compiled language system, e.g. .Net where IDE is unintegrated and CLR integration is, he believes, per class, not per method. (he was one of those keen to discuss things with Dave Simmons).

Claus then showed a Smalltalk browser with Java classes. He uses the package preface name (e.g. Java.lang) as the category name in the system browser. He uses :: notation (e.g. Java::java::util::StringTokeniser) for nameSpaces. He showed the Java abs(number) method and showed Smalltalk code calling it (i.e. written abs: number). He put a breakpoint in the Java (here occurred the usual demo hiccough :-)). He then implemented a pop method (written in Smalltalk) in a Java class. (Java code can also be written in Smalltalk classes; current IDE mis-displays some constructs.)

Next he showed Prolog solving the 'towers of hanoi' problem. One wants to solve in Prolog and do UI in Smalltalk (writing UIs in Prolog is silly). He does this by writing a Prolog moveFrom:To: method. (His Prolog is implemented in Smalltalk. His Java is currently implemented as a distinct Java VM; he would like to implement it in Smalltalk).

Implementation details: Smalltalk already has VM orthogonality via evaluatorClass, compilerClass. Smalltalk doesn't have an orthogonal integrated binary object model. We can implement this in various ways:

- emulation
- common bytecode
  - — cross compile (e.g. Bistro): smalltalk contexts and blocks not in Java, arbitrary precision arithmetic not there, exceptions in Java are poor (re-raised exceptions tromp stack), no generic proxies in Java (no #become), poor generic containers in Java, poor metaclass protocol in Java. Hence cross-compile solutions are very slow.
  - — extended (e.g. .Net): this is a real competitor to his approach; he will be interested to see where this goes.
- multiple bytecode set (his ideal solution): every method has a flag to say which set its bytecodes are in.

VisualAge for Java hides the Smalltalk layer. (Why? Well, one presumes politics plus it has not been migrated to the latest version of VAS but is still running on VAS 4.5.) Smalltalk/X reveals it but they are a small company and it is not really a product. Smalltalk/X gets rid of primitive types (fakes them for Java) and JavaClass is subclass of Smalltalk's Class, the Java.lang.object class being a subclass of JavaClass, etc. (I noted that VW's Java utility, Frost, had a similar structure.)

The next step is MetaMethods: let the method itself interpret the bytecode; which interpreter a method uses is part of MetaMethod's protocol. Let subclasses of MetaMethod redefine this behaviour. Two hooks are needed

in the VM. Send valueWithReceiver:args: to MetaMethod if not understood. He wants this not to be limited to bytecode: also use lists for Lisp, graph representations for state machines, etc. MetaMethod will dispatch to the appropriate interpreter or JIT compiler and install generated code via a VM primitive. The jitter can now be implemented in a high-level language and debugged under a high-level IDE, with advantages like those reported by those who port Squeak. The prerequisites for this are just a common object model and the two hooks into the VM. The interpreter is only used for testing and validation (to interpret jitter while jitting itself) so can be slow. The dynamic machine code generator (jitter) should be reasonably fast.

His demo is available, with test cases, but not very usable just now.

Q: MetaMethod could implement the Self language? It depends on where MetaMethod lookup is done. If method lookup was also done by the receiver, not by the VM, then it could do Self.

Q. Could you extend a class from a language with a very exotic dispatch mechanism (e.g. Beta)? Maybe not.

Q. Current commercial uses? Claus has a current network management utility project where he is using Prolog to solve parts of the problem.

Francisco Garau, University of Buenos-Aires, Argentina - Concrete Type Inference in Squeak
By concrete type he means implementation types, the opposite of interfaces. Concrete types reveal the internal layout of objects, what instvars they have, how they are laid out in memory, etc. They are used not by humans but by compilers and other tools.

To analyse Smalltalk code statically is hard due to late binding. Concrete type is class of object plus Concrete Type of each instVar plus Concrete Type union of all indexed variables:

3 concreteType is <SmallInteger>

The Concrete Type of an expression is a set of its possible result types:

a = b concreteType is {<True>, <False>}

His aim is to build a static type flow network based on the possible runtime flow of code. Assignment, method invocation, etc., all have rules mapping the involved types to the result types. He builds a network of such types for each analysed method, not for the whole program, using the parse tree. Method analysis begins by seeding each initial slot with a monomorphic type (i.e. one of its possibles).

He then demoed in Squeak. 'InferType' on a Display boundingBox returned type Rectangle. His AnalysedMethod, created by this analysis, is like a CompiledMethod: both have behaviour. These exist within his overall type inference system, which replicates Smalltalk: his TiSystem contains TiClass with methodDictionary of TiAnalysedMethod, also TiCompiler, TiInterpreter, etc. It also has typeMemory (c.f. objectMemory) which holds the type. The correspondence is:

- compiled method: analysed method
- bytecodes: type flow network (i.e. within a method)
- compile time: connection time (i.e. time when you build the connections between the types)
- run time: flow time

There is no way to connect the slots of a message-send expression meaningfully (just as Smalltalk does not evaluate at compile time). Instead he seeds types and evaluates how they combine. For multiple argument methods, they analyse the cross-product of all the possible types on all the possible arguments.

Variable assignment changes the variable's type. He models this by a state machine for each variable where each state is another slot for that variable. Messages can also have side effects on objects, changing their internal structure; this creates new types. Thus while several objects (ST-side) can be of one type (Ti-side) it is also the case that one object (ST-side) can be of more than one type (Ti-side). His example was Point; if you write a method to assign 'hello' to the x variable, this method changes the type of a Point from Point x: <Float> y: <Float> to Point x: <String> y: <Float>.

Primitives on the ST-side are analysed with primitive analysis (instance of TiPrimitive). Just as VM implementer must write each primitive routine, they must write a specific implementation of each primitive analysis. Some are easy (e.g. primitive =). Some are hard (e.g. primitive perform:). They have only done three primitives so far.

Blocks are objects, so have types, but every block has a different type, with a reference to the block definition analysis. AnalysedBlocks are thus like AnalysedMethods. Blocks can refer to outer scope variables; his system cannot handle this correctly (so at present he converts them to block pseudo-arguments). Thus the tool is incomplete.

He then demoed analysing a simple (23 methods) program and displaying the results. (These were displayed textually. He has a visual display system for the structure of individual types but must draw their connections by hand. I suggested to him a 'show one type, show second plus connections to first, show third plus connections to first two, etc.' system. It would be easy to implement over HotDraw given what he already had. Presumably Squeak has HotDraw or an equivalent).

Q. I encouraged the speaker to examine Michel Tilman's Analysis Browser (se Discussion Sessions section below. Would the Analysis Browser, given a sufficiently large test set, not correspond to the output of his tool, given that it seeds types rather than trying all possible types? The speaker remarked that his tool would analyse all branches whereas the tests might fail to exercise a given branch. I agreed but suggested that

- comparing the concrete type analysis with run-time type analysis would be valuable
- given his 'mimic smalltalk runtime' approach, Michel's framework might prove an ideal way to progress his implementation and solve e.g. his block reference problem

He had a long demo from Michel before the conference ended and seemed most interested.

Johan Brichau - QSOUL/AOP: AOP using Logic Meta Programming in Squeak
I caught only the very end of this talk but got Johan to give me a demo in the break, albeit not in circumstances where I could take notes. The text following is what I recall.

Johan's system statically combines aspects onto a base system. Thus, for example, if your base application models the filling and emptying of tanks in a hydraulics model, you may want to specify how these operations can be combined; the tank cannot be emptied before it is filled and you cannot have two successive fills or empties. This behaviour can be wrapped around the base methods ('weaved around' as Johan puts it) as an aspect.

You may then want to log the operations; another aspect weaved onto the base methods, inside the 'order' weave, logs what hydraulic operations are performed without logging the 'order' behaviour.

Johan has implemented his system using anonymous metaclasses, taking this design from other Aspect people who aim at Java (or other languages) and so cannot follow a Smalltalk-specific design. I and others remarked the close relationship between aspects and method wrappers. Johan agreed and said he intended to rewrite his system to use method wrappers, since these were more granular, more lightweight and above all more dynamic.

Gilles Vanwormhoudt (vanwormhoudt@ensm-douai.fr): VA Typing
I missed this, as did John. See the slides.

Miscellaneous and Impromptu

Craig Latta - Handheld Squeak
Craig has been getting Squeak 3.1 to run on a handheld machine. He wanted to work on Squeak anywhere. His demo was on an iPAC 3670 (206MHz StrongARM 1110, 12-bit colour, 64Mb RAM, 640x480, 2GB disk card, VGA card a6 under WinCE 3) showing the graphics in Morphic. (It was slow to update the conference room's projector, somewhat faster when he turned off some morphic page effects during the talk and carried on, and much faster when it was just driving the handheld.)

Effort-wise, the Squeak port took him 2 weekends, 3 days of which were taken up with finding and enduring a bug in WinCE memory management. This was because the Win32 port already existed, plus there were lots of people to ask (particularly Hans-Martin Mosner who helped him with Morphic). Elapsed time was nearer a year of waiting for items to prepare the host OS, the hardware (didn't have floating point support) and cards (no driver for VGA card).

He did remote debugging via embedded Visual C environment on laptop while Squeak was running on the handheld, communicating instruction pointer etc., so allowing source-level debugging (also Squeak snapshot could be hot-swapped between the two). He did much work on sound.

Craig has a concern for ruggedization; he remarked that the hardware seems pretty fragile outside its case, e.g. it wouldn't survive being dropped on floor. An audience member remarked that laptops don't like this either.

Any Windows CE machine with at least 2 MB free for Squeak can run Squeak, but may need a smaller image than the one from Squeak Central which he is running. He thinks Squeak also runs on Synar. Squeak exists on Palm but only if you put Linux on it and the implementation is not generally available. Craig wants a much better UI for Squeak on handheld; luckily current is good enough to let you improve it directly. Handheld systems need all the Morphic optimizations and improvements they can get. Standard Squeak in many places assumes you are using a mouse; some action pairs (e.g. mouse down, mouse up) are impossible on handheld.

Q. Keyboards? He has a compact folding keyboard. Squeak, WinCE and Linux have virtual keyboards for stylus; the squeak handwriting recogniser (Genie) was too slow for him to use to code.

Tutorial: John McIntosh - Garbage Collection
John's detailed and well-illustrated slides are available from the Smalltalk Solutions 2001 website. John plans to provide them with yet more text to Stephane in September for inclusion in the Camp Smalltalk smalltalk book. I shall not compete with this here but just note a few points not mentioned (or not clarified) in the slides. Use what follows as notes to John's slides.

John wrote a server Smalltalk application 6 years ago that ran 24 x 7. He instrumented it to check its robustness, collected data for 6 months and noted the image size varied. He asked why, got no answer, so investigated himself. Now GC has become a mainstream technology.

John has seen some very silly GC remarks in comp.lang.smalltalk, e.g. "Do aCollection makeEmpty before deleting aCollection; this is easier on the garbage collector." A GC can be written in 12 lines of code. An optimal bug-free one is slightly harder. John found 2 GC bugs in Squeak last year. Eliot rewrote allInstancesWeakly in a 5i version and broke the GC.

John's slides cover almost all the garbage collectors known to Smalltalk (Claus Gittinger has implemented a treadmill collector). PhDs have been granted on mark and sweep GC theory: "It's a pity all the easy stuff has been taken" :-). GC bugs can be tricky. Swirling the mouse was a classic way of eliciting GC bugs in the good old days; it could GC the mouse as an object popped from the stack was briefly unreferenced. A recently-fixed Squeak GC bug could have the GC collect a class whose definition changed during GC; this led to a much later core dump while accessing an instance.

GC strategies interact with OS. Many host operating systems put unused pages out to disk. Then GC demands them and it's slow (and host classifies your program as a bad citizen). In a fixed memory allocation system, GC asks for a new page and the host gives one but tries to take back another in lieu; this can turn the lights on on all your pages; slow.

Slide 20: VW issue (now fixed, I think): an initialize created 10 million floats which were then freed. The allocator's default strategy for the next (larger than float) object was to scan the 10 million floats on the free-chain looking for something larger. Reviewing this with Eliot, John was impressed how fast it scanned the chain, but 10 million is a large number.

No Smalltalk vendor uses copy's ability to place objects cleverly in toSpace. It's three machine instructions in Smalltalk/X, more in Squeak, and any bug in the algorithm shows itself much faster than other methods. However, all Smalltalk's except Squeak have distinct object headers and bodies; they move the header, not the body. Squeak makes move a primitive to prevent interruptions while moving the whole object.

Slides 29, 30: John has never heard of an alternative Java GC plugIn; you take what you're given. Java claims the train algorithm (not used in any Smalltalk) is the neatest ever but John is very sceptical, plus he has looked at VAJ's separate implementation in Smalltalk (overlapping with Smalltalk GC). Those who test Java GCs tell him that Java does not GC well. Is it the fault of the theory or the implementation? (Plus be aware, published papers have bugs. You cannot implement from them.)

Slide 32: Alan Knight remarked that ParcPlace once implemented a GC that absolutely relied on the 2%, 98% values quoted here. It died on the first real task it was given: a file-in. (Could this be a rare example of a ParcPlace error that was *not* due to poor management. :-)) John remarked that SmallScript did not need InterGenerationalPointers due to clever tricks.

Slide 36: in this VM failure example, there were 30 million (re)creations of an object with over 1000 pointers from oldSpace. The server grew the remember table too large and quit. Eliot fixed this in 5i.4 by ensuring that any such objects would be tenured into oldSpace. Meanwhile, they found the offending object, a timestamp which the system was creating at hourly intervals and stuffing into 1000 by X arrays, referenced all over the place, and put it instead into 24 class variables, one for each hour of the day. :-)

Slide 38: Claus is researching how GCs work with today's programs and machines. He is looking at whether the 'only 2-3 generations' rule still holds today; he suspects not. John is making the Squeak GC easier to experiment with. The VW GC is changeable on the fly because Ungar and Jackson developed the theory in VW, so made experimenting with it easy. The VA GC is not. (However in VA you can turn the GC off, for as long as you have memory. :-))

Slide 41: StackSpace takes advantage of hardware. It is good for method-send contexts as they *are* a stack. He knew someone whose application nested method calls 10,000 deep, whereupon it did noticeably slow down, but overflowing stackSpace is very rarely a problem.

If you load MPEG in Squeak, grow the CHeap size to e.g. 500k. Squeak needs to give better warnings. If youngStart gets near end of memory, it does a GC on every allocation. If you need 10,000 allocations to complete before the current method ends and you can reset GC values, you may have no choice but to abandon your image. Slide 53: in 64k the 66% threshold leaves 20k, in 1MB it leaves far too much. As the overall available grows, reduce the percentage.

John suspects the VW profiler ignores garbage collection time by default; he has seen a 10s (VW profiler) task take longer by the wall clock. John's slides describe VW 5i.2. 5i.4 makes some non-trivial changes (e.g. slide 55 changing default value gives 1% performance gain; default fixed in 5i.3).

LargeSpace takes Bytes and Strings above 1025k but not large collections (separation between header and body would slow access). He only once saw an application that needed to optimise its use of LargeSpace. Often, you can shrink it if worried about footprint. (On slide 57, the X-axis points are multiples 1, 2, 3, etc. The rapid rise is at x 3.)

Slides 60 - 69: Soft limits are wake ups to GC to ensure it will be busy collecting before disaster happens. On a server, idleTime is rare but the GC may reply on it for housekeeping, so one must either force idleLoopAction to run or (better) control lowSpaceAction assuming you will never see idleLoopAction. Before 5i, there were very rare bizarre bugs when values grew past 512k (2^{29} or 2^{30}); suddenly they were not SmallIntegers but the primitives thought they had to be. A GrowthRefused notifier is a bug if a headless image tries to raise one, and when changing it to write a log, it's important to ensure that the last 2k of the log, containing the useful debug information, will be written in such an event. The VW IGC idle behaviour default numbers (slide 66) are far too small for servers. Switching off automatic shrink (i.e. hand back to OS) can be a good idea if you have repeated memory spikes; the OS can take longer than you think to return it.

Slide 68: for windows 3.X, 11MB was an important limit motivating this.

Slide 72: (misprint: VAS semiSpaces are the same size, 6, not 5.) John thinks noone actually uses the VAS dumper. He notes that what is actually there in VAS differs from, but presumably maps to, what is described (e.g. there are actually 231 segments, not 1).

Slides 90-91: the hole is where the FCGC ran, causing all their real-time users to drop off the net. The solution was to do more idle GC.

Stephane Ducasse - How to do a sexy Squeak Demo!
Stephane's ideal audience for a Squeak demo is second year students who have met Java and experienced its difficulties. 5th year students have learned how to live with Java; their minds are narrower. Explain that Squeak has some amazing stuff and also some proof of concept stuff. Scamper is proving you can write a web browser easily in Squeak, not trying to be the world's best web browser. Show a Squeak book (e.g. Mark Guzdial's) and stress there are others.

Prepare your image a week beforehand. When loading projects from the web, always copy your image; some projects can break your image (roll on Stable Squeak). Stephane uses Squeak 2.8; Squeak 3.0 had bugs, so he's waiting for 3.1 before upgrading. Use the Squeak new look, not the default look. Check appearance>displayDepth: default was 8, use 32 (Mac sleeps for 100ms at wrong depth).

Then start up, open project after project and blow your audience away with the shear range of what's available in Squeak; so many platforms, so much visual power, so much audio power, the network stuff, mathmorphs, etc., etc. Then start combining things. Stress that all the code is there. Mention the Refactoring Browser and XP's test-before-code style. Mention all the things that came first in Smalltalk: bit-mapped screens, GUI frameworks, VMs, refactoring, XP, etc.

Stephane's demo order is:

- Platforms: show frontispieces of Squeak projects on Hyperstone, Zaurus, EPOC32, Postlet and all the obvious ones. Explain that Squeak is written in itself, so exceptionally easy to port but (thanks to core C mapping) fast enough for real-time
- Projects: show all the projects submitted on Bob's superswiki.
- Simple apps: mouse tracker, star wars effect, switch to inboard scrollbars, etc.
- Standard demos: explain Morphic is new paradigm, Morphs know about time passing (have cube rotating when not selected). Show bouncing atom morph, infection scenario and bunny, then drag-drop atoms inside bunny's head, etc. Use the Alice bunny commands to show the readability of Smalltalk syntax.
- Network stuff: browse socket protocol of Commanche and all the network categories to show there is lots of network stuff. Show HTML object browser then Explorer (hierarchic instance browser) and the plugIn (e.g. Opera with Squeak Browser).

- Flash player: load flash file, select morph, pull out of flash, grow (menu>compressedSize impresses some people)
- Show Nebraska for exchanging objects
- Sound: demo John McIntosh' MPEG player, recording sound and playing, spectrum analysis of your voice, voices saying text. Then do music (midi file) switching on instruments, changing them, altering wave profiles, etc. (Stephane has also seen a demo of film and music synchronised but does not have the project.)
- Scripting: Panda3D scripting architecture
- More projects: WebPad project / Exobox project, Cosmocows (from their webpage), Microseeker (Squeak-controlled submarine).
- Avatar: adjust face parameters (beware when loading Avatar not to damage your image)
- Pooh
- Image Processing Framework
- Cassowary: incremental hierarchical constraint solving tool.
- Ray-tracing
- Pheromone (ants seeking food) problem solving example (beware when loading this not to damage your image).
- Create morph, assign sound, create joystick, make joystick control morph (don't use car drive standard demo example as its code is inappropriate to user expectations; Stephane explained the changes needed to make the car sanely steerable but I forgot to note them).
- Infrastructure: Stable Squeak

Lastly mention the Squeak mailing list, and that you need to filter it unless you want 50 emails daily. See ESUG website for the above demo image.

Q. Use recorder to capture demo and save with image so audience could start image and rerun demo slowly afterwards, recalling what they saw and understanding it better? There is no recorder in this image, maybe in GToys or Omni.

Roel Wuyts - MagicKeys: Assigning keybindings graphically in VW
MagicKeys lets you graphically display and change your keybindings in VisualWorks. Roel opened VW5i.4, loaded the MagicKeys parcel and showed the settings screen, and scrolled to the magic keys tab. The VW UI settings tool shows only the initial list (and is poor; VW plan to change it); hit 'edit' to get the MagicKeys tool and see the current list. (Reset resets your keybindings to Roel and Stephane's choices.)

To edit, select a binding, popup menu and it just asks you to type:

- the key (and any modifiers) you plan to use, or select it from a list (dynamically recreated every time you open this widget from a heuristic check of one-argument protocols)

- ESC and two keys

then accept it to change the list. Once your list is changed as you wish, apply and VW uses your new list of key bindings. Roel amused himself defining bizarre combinations to invoke keyboard print, etc., and other combinations for the interrupt key to dismiss the ‘no printer’ exception. (There are a few short-cuts that don’t seem to be implemented on keyboard events; Roel does not know how these are done and Magickeys does not provide them.)

VW implementation: a DispatchTable and the class KeyBindingsEntry mediate between e.g. workspaces and the KeyboardMappingtable. The method addToDispatchtable: maps keyBindingsEntry to aDispatchTable; use the MagicFeelPolicyConverter to map in the other direction.

- dispatchTable: is made of two parts (looking through them is not fun)
  - — baseTable: $a -> #(nil #action1 nil nil ... #action2 ...) Reading this is not fun: #action1 is at position 2, #action2 at position 8 (so means CTL+8)
  - — composeTable: ESC -> #($(#action $) #action2.
- KeyBindings = char -> event

There are some hairy aspects of the above. UIFeelPolicy’s way of handling ALT and META keystrokes are handled by the system as actions performed on class ParagraphEditor. Thus ALT and META are rerouted, not handled like CTL as in basetable above (Roger Whitney remarked he knew why this was, leading to off-line discussion; the VM maps ALT and META to the same keystroke for windows compatibility). Roel browsed UIFeelPolicy (using a keyboard short-cut to pop up a text-match to find the class) to show how ALT + META short-cuts were redirected by ParagraphEditor to the local feel policy, giving an extra layer of redirection. Roel did not see the need for this so bypasses it in MagicKeys. Alan Kay agreed that it was a redundant feature and should not be used.

On Mac, ALT and META are swapped due to mis-back-translation of the ALT +a -> single ACSII value -> decompose back into ALT + a; the setting you ask for is the setting you get, you just don’t see what you expect in a MagicKeys list on Mac.

Unloading does not always work; manually set FeelPolicy to something else after unloading to make sure it has. Only CTL-?? can be interrupt key, not ALT-?? or META-??

Versions for VW 3 and 5i, available as parcels, as Store or as Envy, can be downloaded from www.iam.unibe.ch/~wuyts

Q. UI to show current bindings? Also to show text description of short-cut, not name of method invoked? Keeping the edit list open is the only UI at present; yes a text field would be useful. CTL-V = ‘Paste’ is better than CTL-V = #methodName

Michel Tilman - Argo Framework Demo

VW users may know Michel's work from his Analysis Browser (get it from http://users.pandora.be/michel.tilman/smalltalk/ noting that the 5i parcel already contains the method wrappers application; for the earlier version, you must get these from John Brant's web page and load them separately.)

However that was just a spin-off from his Argo framework. The framework is best compared with tools like MetaEdit or DOME but it's not that like them. It was built as a means for creating web-enabled tools to manage documents, processes, and other administrative support needs of large organisations. Because Argo works on meta-data, it is particularly strong when the organisation needs its tools to be built quickly and changed frequently (a common situation today).

Michel demoed it to a group of us on Friday afternoon. Form views can be quickly constructed that view data obtained by navigating through long chains from a root to related objects, based on logical queries. As it is meta-circularly-defined, these forms are built by tools that are themselves defined by meta-data. See my ESUG 2000 report on it for more information. The meta-data of the current examples is in Dutch, but all the (VW Smalltalk) code is written in English; what I saw of it read well.

So, how to market it? In the '90s, the department where I used to work spent 100 times the effort Michel's tool would have taken in setting up web-enabled administrative tools for documents, contacts, projects, etc. As we were repeatedly reorganised, these tools aged and died, replaced by others. I'm sure many organisations are in the same position. Could one be a lead customer? Another idea is to create a website where people could upload a UML model and generate a web application.

Also, how to progress it? Should Michel make it a commercial product and start recruiting a team, or make it open source and aim to earn money as one of the finite number of people with experience building Smalltalk meta-programming applications who could be consultants to it? (As another of this finite number, I would be delighted if opportunities to work on it or with it emerged.)

I will be most interested to see where this goes.

Other Discussions

Smalltalk in Germany

Two years ago, German companies who hired Smalltalk contractors usually remarked that they would have phased out Smalltalk by now, replacing it with Java. They don't say that today. On the contrary, one firm told me they were turning work away daily and could employ 20 additional Smalltalk consultants with good client-interfacing skills tomorrow.

Relatively few new Smalltalk projects are being started in Germany as yet; Smalltalk is surviving because Smalltalk projects are surviving. However German consultants I talked to were moderately hopeful that the Smalltalk squeeze was over.

Some of the shine has come off Java in Germany. A great deal of money has been simply burned away in projects where Java's limitations (usually much helped and often outweighed by mismanagement and other factors) have prevented any profit being made. The .com crash has also diminished an area where Java's web-focus appeared to argue for it, whereas financials (more accustomed to Smalltalk) are still doing well. It is now remarked that 'dot' is 'tod' (the German word for 'death') spelt backwards, and that 'rapid development' means you can go bankrupt in one year, not several.

I heard some funny stories of German projects implemented in Smalltalk as proof of concept and then re-implemented in Java with immense growth of cost, delay of time and loss of functionality. In one case, 18 months after the Smalltalk utility was demonstrated, the rewritten Java utility was likewise demonstrated - by a team who had a echo console connected to the one in the demo booth, from which someone copied what the client had typed into the original Smalltalk version, got the answer and sent it back for display in the booth's Java facade (using that word in both its pattern and its colloquial senses :-)). In another, a three month quote to productise a demonstrated 'easy web information' utility was rejected because the product, like the demo, would have been in Smalltalk. The Java version *was* delivered - by several people, who worked on it for years, and could not get it to do as much.

Joseph Pelrine has left Daedalos to found his own company, MetaProg GmbH. He's finding no shortage of work (indeed, he talked of wanting six subordinates to help him do it all). Next week he talks to the chaos computer club (German telecom trials group living in a totally wired-up apartment block); he may turn up in a suit and tie just to startle them. :-)

General Conversations

Jun has a Prolog and a Lisp interpreter 'in case of emergency' (making yet another Smalltalk system with Prolog in it). Although billed as Smalltalk in 3D, apparently it grew out of research that had other aspects; hence the interpreters for other languages.

Martin Feldtman recently began to use VisualAge's XML framework in a product managing documents. Objects to XML was easy, XML to objects harder; only after querying VisualAge support did he learn there were no default collection class XML methods. Users must subclass the collection classes and apply their preferred mapping; VisualAge says there are too many alternative mappings for them to provide an acceptable default.

The many sources of Smalltalk goodies are ill-integrated. In particular, the UIUC archive web pages would benefit from an update. The search engine is extremely fussy (e.g. MethodWrappers not found by typing 'wrappers', CustomDeepCopy not found by anything less than the exact full phrase, etc.), its submission form's version list is so old it omits 5i, etc. Better upload and searching is available; might a CS project give the archive a flashier Smalltalk-powered upload, download and search front-end?

It would help novices if the ‘how to write wiki pages’ help added ‘how to indent code’ to its explanation of lists, tables, etc.

Conclusions

My third ESUG and a lot of fun. I look forward to next year.

- Teleworking and XP combine naturally. Could Tukan be the next tool to empower Smalltalkers as the Refactoring browser did (not as much, no doubt, but having led a teleworking team I see the potential)?
  - — As XP is popular, maybe it could make people notice us - but there again, maybe not; refactoring is popular and we’ve had the world’s only Refactoring Browser for years without being noticed, whereas there are (less capable) telework tools rivalling Tukan.
  - — Other relevant XP work is going on. Greg Hutchinson’s talk showed the benefit of wrapping effective process support round XP. Peter van Rooijen’s work is also in this area. (These are both in VA while Tukan is in VW but it’s all generating and trialling ideas.)
- The shine is coming off Java, to Smalltalk’s benefit. But there is still much to do to start new Smalltalk projects and bring newcomers to it.
- Smalltalk web tools are now powerful. Both VW and VA offer impressive frameworks. The .com crash means less money around but it also means a climate of scepticism towards hype and more desire for real evidence of benefit. We will see whether this works to Smalltalk’s relative advantage.
- Michel’s Argo meta-programming framework has potential. Meta-programming is one area where Smalltalk can really show its power and Argo’s original application area is one most modern organisations struggle with. Its problem is it’s *too* widely applicable; where to aim it, how to explain it?
- Lots of Squeak work is going on. Listening to the Stable Squeak talk, I was struck again by the truths of XP; do the ‘wrong’ thing (i.e. the simplest thing that could possibly work) and so learn how to do it right, but only if your system lets you experiment. Joseph’s past experience with Envy, Team/V, etc., gave him the idea of a better way to modularise software; the openness of Squeak let him try his idea.