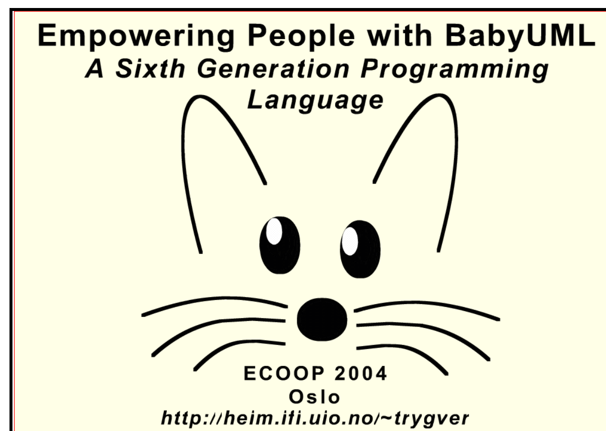
Empowering People with BabyUML
A Sixth Generation Programming Language

Opening talk,
ECOOP 2004, Oslo 2004.

Trygve Reenskaug
Department of informatics, University of Oslo.

http://heim.ifi.uio.no/~trygver



I regard BabyUML as a sixth generation language because it combines the algorithmic capabilities of the third with the semantic modelling of the fourth. The third generation includes languages such as FORTRAN, Algol, Java, and C#. They are all computation-centered languages and their metamodel is defined in terms of symbols composed of text strings. The fourth generation languages are storage-centered and include languages such as NIAM and E-R. (Languages for artificial intelligence are commonly known as fifth generation languages. We ignore them for the time being so the next generation becomes the sixth).

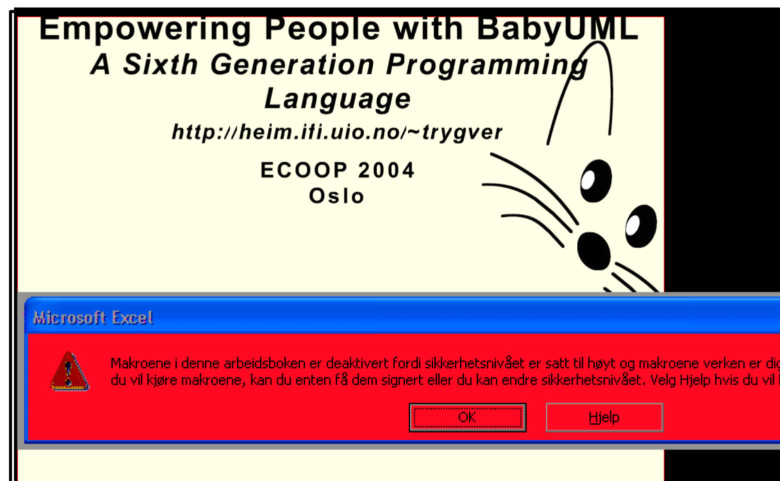
BabyUML is a communication-centered language based on communicating objects. It has a powerful metamodel based on the OMG Meta Object Facility (MOF) and exploits the strong input output facilities of current computers to support programs that can usefully be seen from multiple perspectives, using text, tables and graphics for effective communication between man and computer.

The foremost motivation for developing BabyUML is ethical; we need information systems that can be mastered by users and programmes alike. The systems should be transparent and tangible so that a person can explore them, understand them and even modify them. BabyUML also breaks with tradition by replacing the idea of a closed application with an open module that is created within a running context. Current programming technology calls for a four stage process: modelling, coding, loading, and execution. BabyUML merges them into one, making programming a question of dynamically modifying a running system.

My first experiments are based on Squeak, an open source variant of the Smalltalk information environment. I have chosen Squeak as my laboratory because it makes it easy to test out new dynamics, languages, tools, and architectures that I want to develop for BabyUML.

© 2004 Trygve Reenskaug

1. Introduction



I claim that our current systems are out of control. Take a simple example: I have been adding data to a spreadsheet almost every day for the past five years. It gives a warning about potential harmful macros every time I open it. I have never knowingly added a macro to this spreadsheet or imported one. Further, I cannot find any difference in my spreadsheet with macros acitvated or passivated. That's reasonable, because the relevant commands says that there are no macros there. *I can't see them. I don't use them. I can't get rid of them.*

I hate this program because it is clear that I can never master it. Also: I receive updates to my various programs almost monthly to close newly discovered security holes. This shows that the developers do not master the programs either. The programs are just too complicated to be mastered by humans, and no amount of patching and debugging can ever make them manageable.

What we need are programs that are simple and transparent to users and programmers alike. I can hear the objection: Our programs are inherently complex and cannot be made simple without sacrificing important functionality. I don't believe it. I believe the complexity is man made and avoidable. Consider a wood with its trees, flowers, squirrels, moss, paths and whatever. An extremely complex ecological system that provides scientists with any number of deep questions to ponder. Yet all of us can walk in the wood and enjoy its nature. We can follow the trodden paths or make our own. We can master it. Don't tell me the current information systems are too complex to be mastered by users or even programmers. It is merely a question of finding the right way of doing it.

I use Squeak to explore the nature of transparent, tangible, powerful and flexible information systems. This talk is about my first tentative steps when I try to find a path through the wilderness.

This talk is in 5 parts:

1. *Introduction*
2. *Classes: An example of transparent programming*. How the program with its models and metamodels can be transparently visible to the user, and how the program can evolve in real time under user control.
3. *Components: An example of a transparent architecture*. Demonstrating how the system architecture can be part of the program and transparent to the user.
4. *Genealogy: Sources of inspiration for BabyUML.* BabyUML will be a new mix of many ideas, concepts and programs. I discuss some of the inspirators to my current work.
5. *Conclusion: BabyUML - So What?* If what I have suggested appears desirable, what can we do *now*?

2. Classes: An example of transparent programming

How can the program with its models and metamodels be transparently visible to the user, and how canthe program can evolve in real time under user control

CTR123	
Cost	45700
Time	9.00
Resource	Dave Smith

We imagine an engineering company that is preparing a bid for a new project. The various disciplines are asked to give their input to the process. An important part of this input are the CTRs - Cost-Time-Resource estimates for all activities. The figure above shows a simplified tool for creating CTRs in the Piping discipline. Please imagine that this small box symbolizes the planning tool for Piping.

The user doesn't want to see time presented to two decimal places. He thinks it is silly in his case because the number is only an early guesstimate. There are two traditional and one new avenue open to fixing this problem:

1) *Ask the programmer to change the program for this field.*
   This takes a long time and ties up valuable resources.
   Further, may be other engineers like the two decimals.
2) *Ask the programmer to add a new feature, permitting users to change the presentation.*
   This solves the general problem, but takes even longer and adds to the program complexity.
   In the end, programs may and do get so many features that nobody can master them.
3) *Let the user examine and modify the program.*
   This is the BabyUML way. It saves time and lets the user master his own information. Let's see how it is done.

2.1. The Inspector

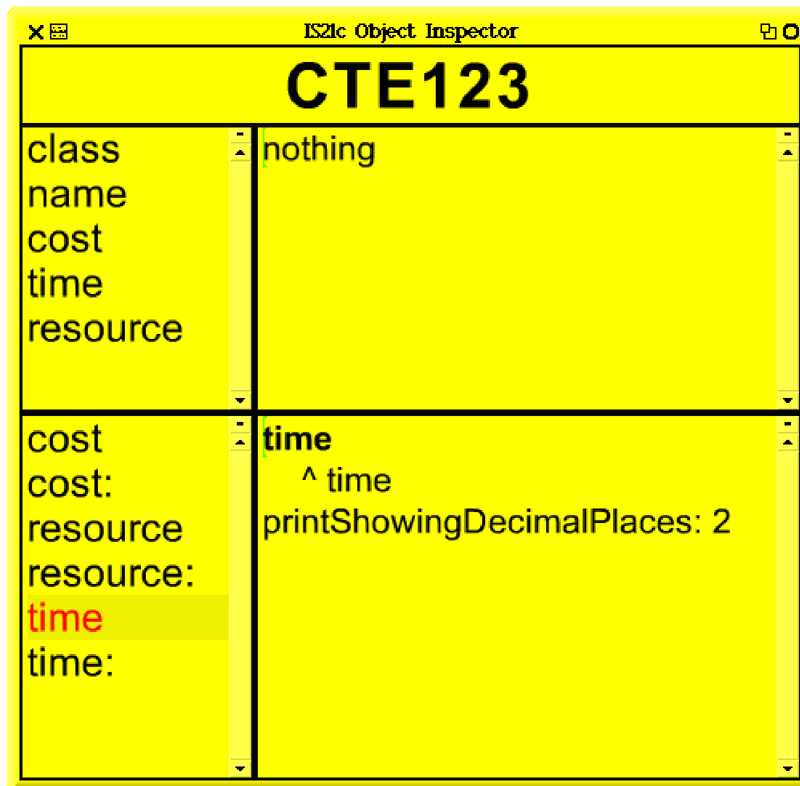
The user selects an information object for inspection. He sees a UML-like presentation of the object. There are four fields as shown below:

- The name of the object
- Object state as a list of object attributes
- Object behavior as a list of object methods.
- Value fields showing the value of the selected attribute or method.

We select an attribute, observe its value and change it in any view.

We select the *time* method and note that it is a number presented with two decimal places. Just change the number of decimal places to 0. The program is changed on the fly and so is the presentation. This is a consequence of our merging build time with run time.

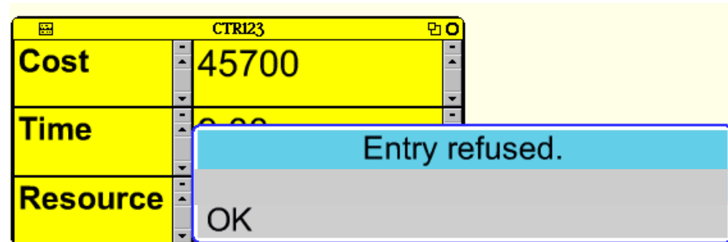
:



Programmers may accuse us of cheating because the methods are really not stored in the object itself, but in the class. But this is an implementation detail. Methods are executed in the context of the object. They are stored in the class only because it would be inefficient to let every object contain a copy of its methods.

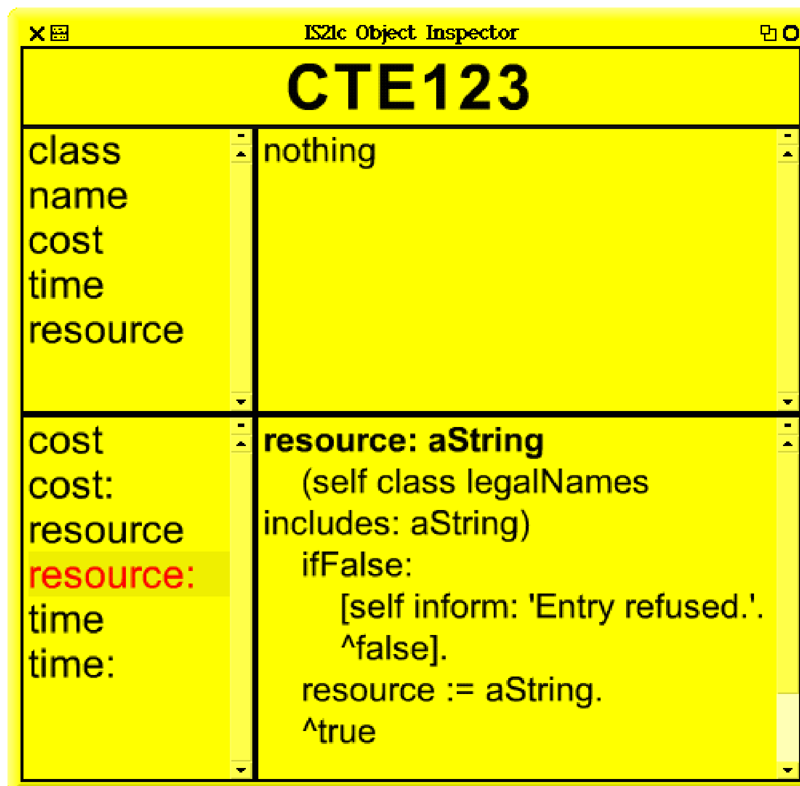
It is also interesting to a programmerto note that the class hierarchy is invisible. The class hierarchy is about code storage and code sharing. The arrangement of the class hierarchy does not in any way influence the behavior of the object, and it is more in the nature of a comment.

A more advanced example concerns the *resources* field. The user changes the name to Pete Smith and gets the following *warning:*



Why can't the user specify this name? We let the user inspect the program to read what's wrong (below). He sees that a name is rejected if the system doesn't know it as the name of an employee. May be Pete is being hired, but has not been entered in the system yet.

:



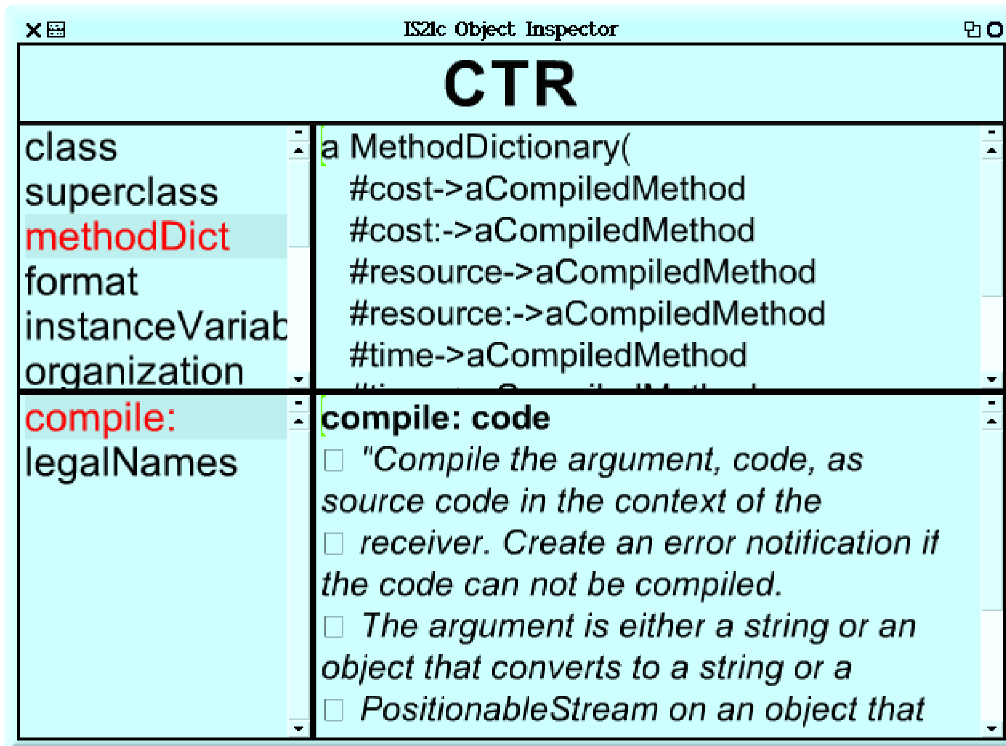
The user comments out the warning and the changed program permits him to enter the offending name. It will now be the user’s responsibility to ensure that all invariants are checked before the CTR is finalized.

There are clearly issues here about what various people can usefully and safely do, but these are details that should be settled by the enterprise and the people involved, not by the underlying infrastructure.

2.2. *The Class*

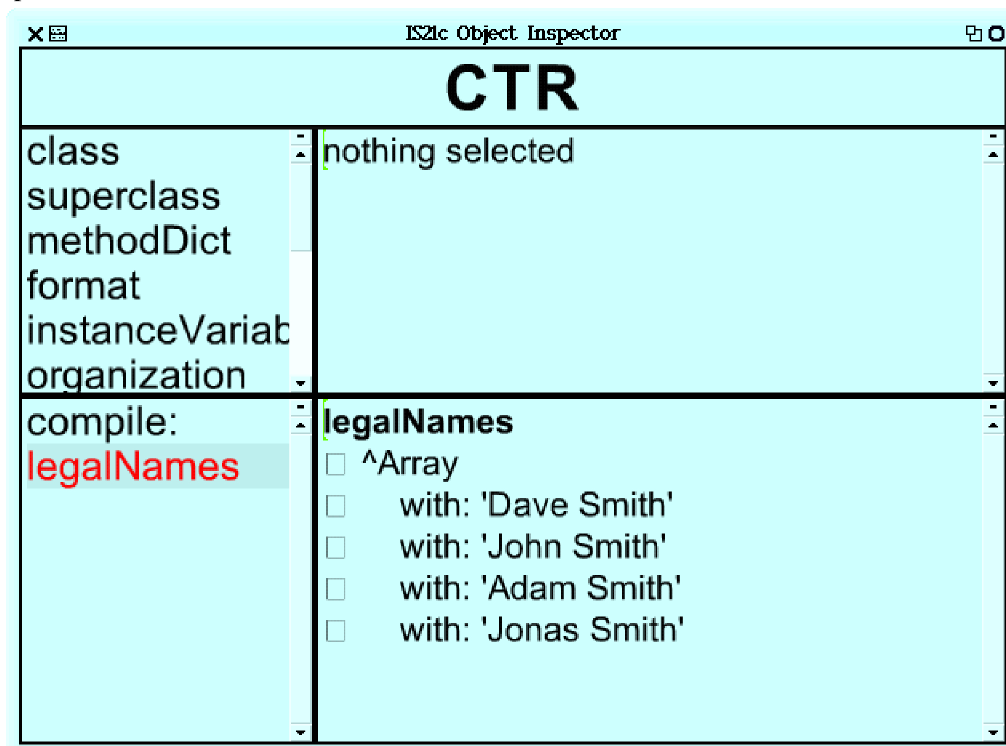
Every object is an instance of a class, and every object knows its class. The class exists as another object and we can inspect it by inspecting our object’s *class* field.

We get:



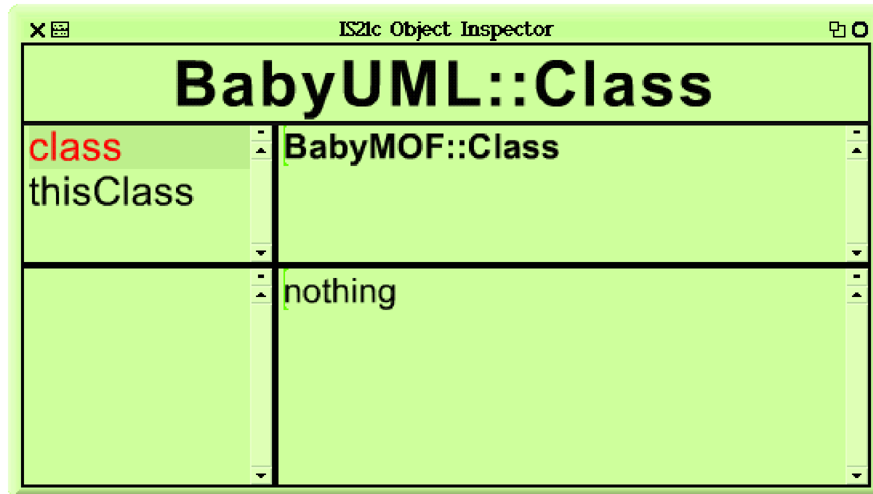
We see that the methods of our object are stored as an attribute in the class object.

We finally look at the *legalNames* method.We see that a static method in an object is a normal method in its class. (There is nothing special with static attributes and methods; they are attributes and methods of the class object). We also see that this method is clearly a hack for demo purposes. In real life, it would be a database operation.



2.3. *The metaclass*

The class is an ordinary object, so it is an instance of a class. What is the class of a class? In the demo, we get the following when we inspect the class of a class:



A class is an instance of a metaclass. This could be the metaclass defined in the UML package, the UML::Class. In an earlier experiment, I have implemented the objects defined in the UML specification. The objects were as follows:

- *MOF::Class*. The class in the OMG Meta Object facility. It is an instance of itself.
- *BabyUML::Class*. The metaclass object in the BabyUML image. It is an instance of *MOF::Class*.
- *Application*. An application class. It is an instance of *BabyUML::Class*.
- *Instance*. An application object. It is an instance of *Application*.

The conclusion is that UML can be seen as a programming language, but it needs to be both simplified and extended to make it useful in practical programming.

3. Components: An example of a transparent architecture

Demonstrating how the system architecture can be part of the program and transparent to the user.

3.1. *The application*

CTR123

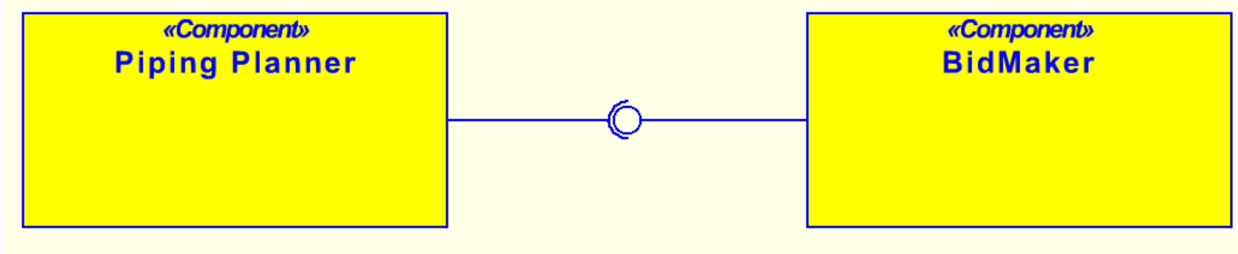
Cost	45700
Time	9.00
Resource	Dave Smith

We start from the same example tool. We now assume that the piping engineer is curious as to his working context. How does his tool fit into the overall system architecture and who uses his results?

3.2. Components

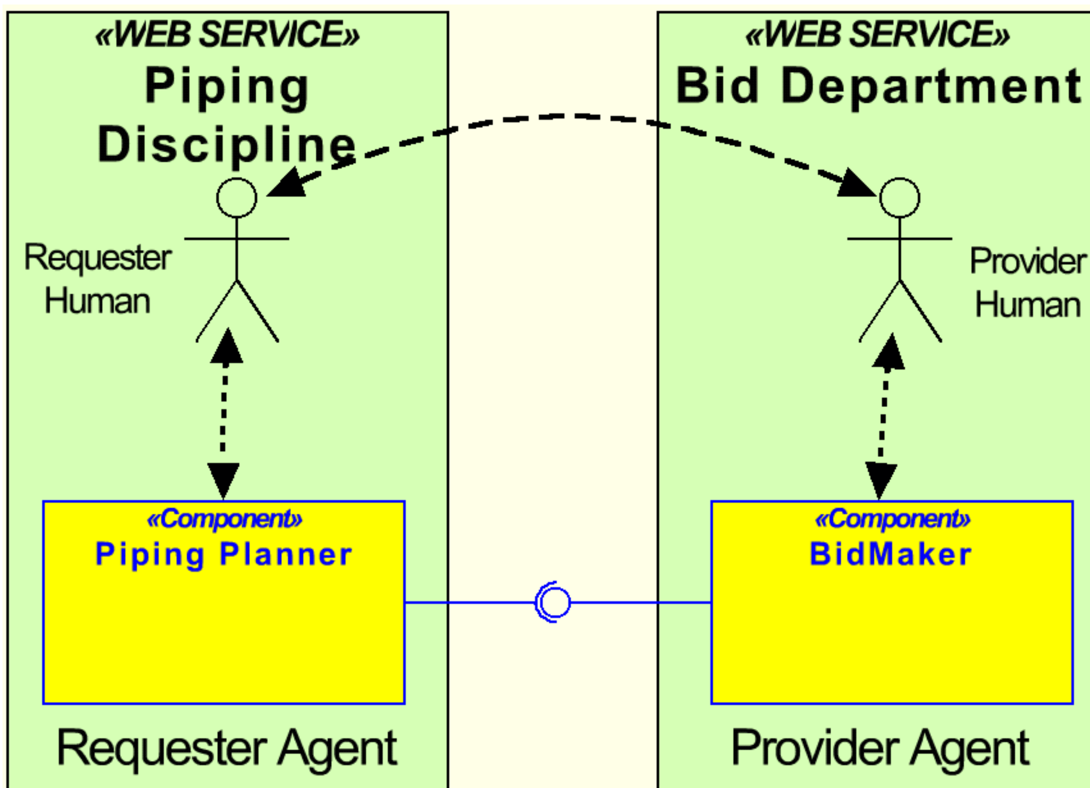
The small icon at the top left hides a menu that is used to move out of the context of the application. We choose zoom out and see that our input tool is a component with one required interface as shown below.

We next see that this component is currently linked to a BidMaker component.



3.3. Web Services

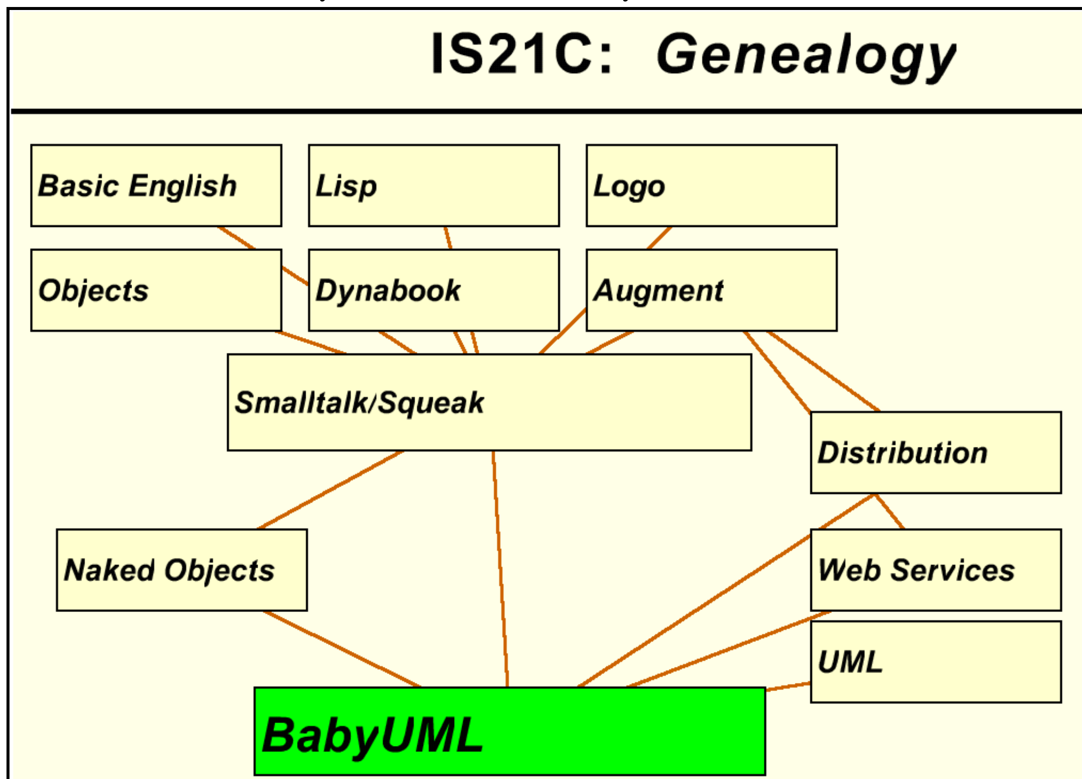
Who owns these components? Zooming further out, we see that we are actually providing data for the Bid Department.



So imagine, if you please, that we have designed the business processes, that we have decided which parts of the bid process are to be manual and which parts to be computer assisted. The agents are Components, and all models are part of the information system and can be explored on line. There is no question of maintaing the various models while the program changes. The model *is* the program and the user can explore it from any perspective of his choice.

4. Genealogy: Sources of inspiration for BabyUML

BabyUML will be a new mix of many ideas, concepts and programs. Let us look at some of the contributors to the vision of information systems for the 21st century:.



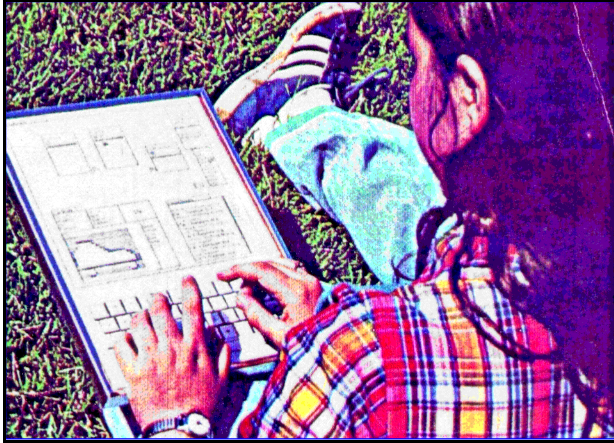
4.1. Augment.

Douglas C. Engelbart: *Augmenting Human Intellect: A Conceptual Framework.* Summary Report AFOSR-3223 under Contract AF 49(638)-1024, SRI Project 3578 for Air Force Office of Scientific Research, Stanford Research Institute, Menlo Park, Ca., October 1962:

> By “augmenting human intellect” we mean increasing the capability of a man to approach a complex problem situation, to gain comprehension to suit his particular needs, and to derive solutions to problems. Increased capability in this respect is taken to mean a mixture of the following: more-rapid comprehension, better comprehension, the possibility of gaining a useful degree of comprehension in a situation that previously was too complex, speedier solutions, better solutions, and the possibility of finding solutions to problems that before seemed insoluble. And by “complex situations” we include the professional problems of diplomats, executives, social scientists, life scientists, physical scientists, attorneys, designers--whether the problem situation exists for twenty minutes or twenty years. We do not speak of isolated clever tricks that help in particular situations. We refer to a way of life in an integrated domain where hunches, cut-and-try, intangibles, and the human “feel for a situation” usefully co-exist with powerful concepts, streamlined terminology and notation, sophisticated methods, and high-powered electronic aids.

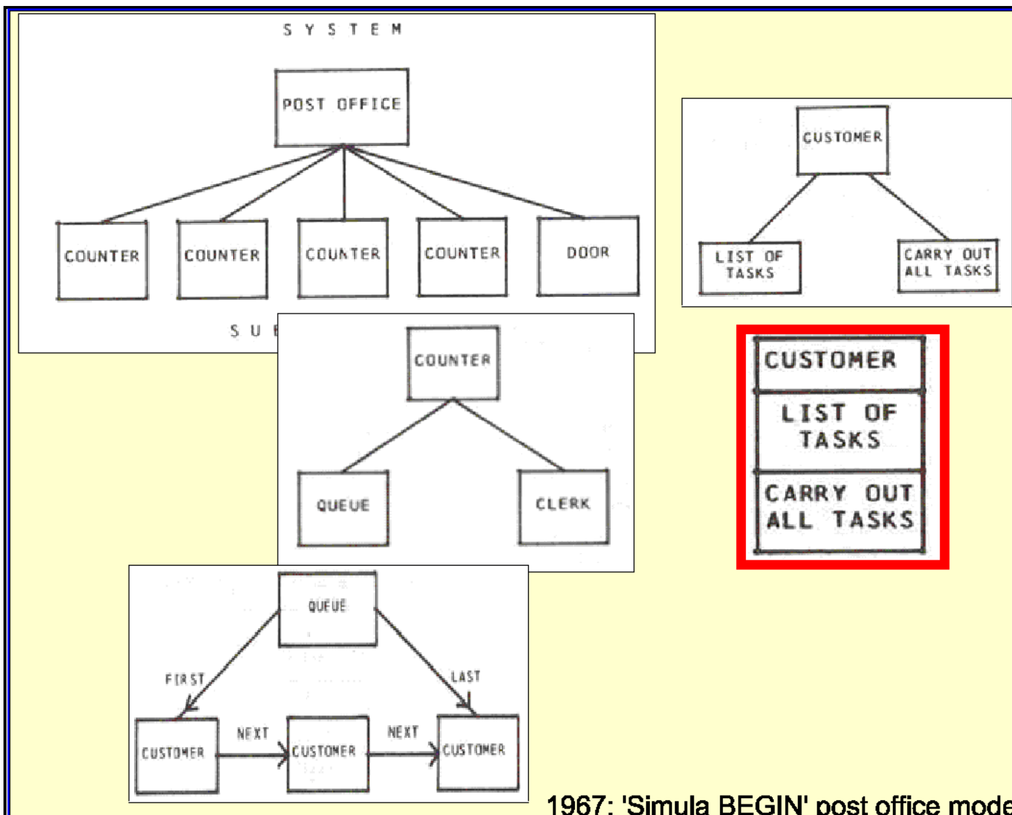
I want to master my information system so it can augment my mind and act as a resonance box for my thoughts.

4.2. Dynabook



Alan Kay’s dream of a “Dynabook”; a dynamic book that contains a person’s information and lets that person master it. An important part of this information is information about the information itself; i.e., the programs.

4.3. Objects



This classic example illustrates the notion of object orientation in a nutshell:

1) A postOffice is modelled as a system consisting of a number of COUNTERs.
2) A COUNTER has a QUEUE and a CLERK.
3) A QUEUE has a number of CUSTOMERs.

4) We are getting to the point: A CUSTOMER has a list of tasks, (i.e., *data*), and it carries out all these tasks (i.e., *behavior*).
5) Data and behavior are encapsulated in an *object* such that the external properties of the object are separated from it internal implementation.

This is a far reaching and fundamental invention done by the Norwegians Ole Johan Dahl and Kristen Nygaard in the sixties. It forms the foundation for all modern thinking about information and information processing.

4.4. Logo

Learning by doing. It is more important to understand the concepts than to memorize the textbook; the importance of process. These are concepts introduced by Simon Papert to the teching of children. We use his ideas to help making the information systems tangible and transparent.

4.5. Lisp

Lisp proves that it is possible to build very rich and powerful systems from very simple parts. In the case of Lisp, all parts are lists. Alan Kay used the same principle, replacing lists with objects.

4.6. Basic English

George Bernard Shaw was not satisfied with the English language, he thought it too arbitrary and hard to learn. So he invented “Basic English”, an English like language with an extensible vocabulary and a grammar with consistent and simple rules. Our language for describing information should be like this; extensible yet easy to learn.

4.7. Smalltalk/Squeak

Various Smalltalks have been cerated through the years: Smalltalk 72, 76, and 78. Smalltalk 80 was published and put in the public domain. There has later been several implementations, both commercial and free.

Everything of interest in Smalltalk is represented by objects. User domain information is, of course, described by objects. Elementary things such as characters and integers are objects. Programs exist in Smalltalk as class objects and method objects. This uniformity makes it feasible to make the information systems visible and tangible at all levels of abstraction. It also helps making the abstract concrete, since the corresponding objects are visible and tangible. Smalltalk combines state and behavior in a uniform way as was described in the Post Office example above. In particular, the execution stack and its frames are visible and tangible objects and Debugger is a regular Smalltalk class.

We are used to the sequence *model, write code, compile, load, execute,* and *stop*. In Smalltalk, this is no longer the case because build-time is merged into run-time. We no longer see application programming as a green-field exercise. Programming is now modifying and extending an existing system; we never start from scratch.

The statement *Smalltalk saveAs:a file name* can be inserted anywhere in any program. It copies a snapshot of all objects including the stack and current program pointer to file. When this file is read into memory, execution continues with the next statement.

The execution I am showing in the talk started in 1972. This is the way of the future. We will never start from scratch, we will always be modifying an existing system. There is an important side effect. A running application co-exists with its program, its architecture, and its programming and design environment. So it can be made visible and tangible for the user to peruse and even modify.

Originally, the focus of Smalltalk has been the individual and learning. We employ its technology to the requirements of the enterprise as a whole and the needs of the people working there taken individually and collectively.

4.8. *Naked Objects*

Richard Pawson has an important contribution to our quest. He lets developers and users together build an object model of the user domain. The developer simultanesously implements this model in Java. The user interface is automatically generated, showing the user information as tanglible objects on the screen. The user invokes object behavior by giving menu commands and/or by drag and drop. The objects are called naked objects because they are shown directly on the screen without any fancy dressing.

An important contribution is that the naked objects approach focuses on the user’s mental model and augments his problem-solving capabilities. This in contrast with current main-stream systems that treats the user as an imperfect input/output device to be controlled by a script.

The Naked Objects framework is an early solution to the need for merging the user’s mental model with the computer’s information model. This model appears as concrete, tangible, and behaviorally complete objects that can be manipulated freely by the user. This is a clear step on the road towards the information systems of the 21st century.

4.9. Distribution

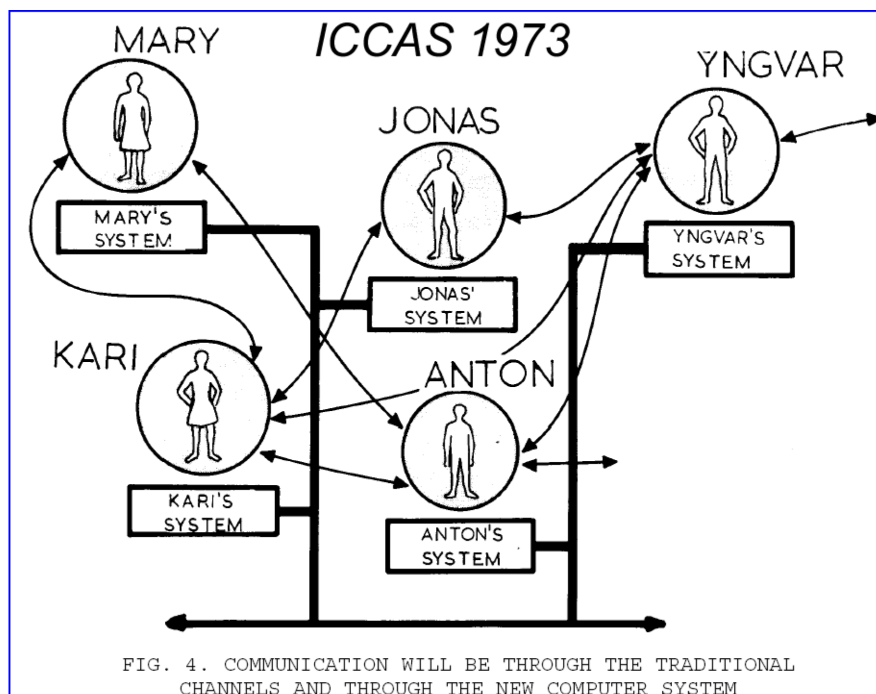


FIG. 4. COMMUNICATION WILL BE THROUGH THE TRADITIONAL CHANNELS AND THROUGH THE NEW COMPUTER SYSTEM

I worked with a system for the computer aided design and manufacture of ships all though the sixties.The system, called Autokon, first went into production in the Detailed Design office in 1963. It featured a central database with several programs around it. On program was for describing the ship's parts for automatic manufacture. This program included a facility for designers defining design rules so that they could generate sequences of similar parts.

Autokon was later extended to cover the Main Design office, using the generation facility to generate det ship's design drawings. An important insight was caused by a blunder. The Main and Detail Design offices shared a common database. Main Design produced the numerical equivalent of the old 1:50 drawings. A bright lad in Detail Design recognised that all information was now precisely represented in the database. Presto! Control tapes for the flame cutters could be pulled straight out of the database. They cut more than 300 tons of steel before discovering the difference between precision and accuracy. Yes, the data in the database had 40-bit precision. No, dimensions were still as approximate as they had been in the 1:50 drawings. 300 tons of scrap steel and some angry finger pointing was the result.

The blunder itself was of course easily fixed and never repeated. Many results from Main Design could be used as long as they were carefully checked, corrected and augmented in Detail Design. But the real problem was a deep one. In the manual system, the Main Design office had ownership and full control over their information. Similarly, Detail Design had control over *their* information. And most important, the transfer of information was formal and carefully controlled by both sender and receiver. I believed the required solution to be fundamental and general as reported to the ICCAS conference in 1973: The computer system must mirror the line organization and the line departments must remain in control of their information and their programs. *We need a system of distributed components.* The components should be interlinked in such a way that transfer of information can be computer assisted, but remains under human control.

I believe this is still a valid assertion, and a distributed information system that mirrors the line organization's distribution of authority and responsibility is part of the BabyUML foundation.

4.10. *Web Services*

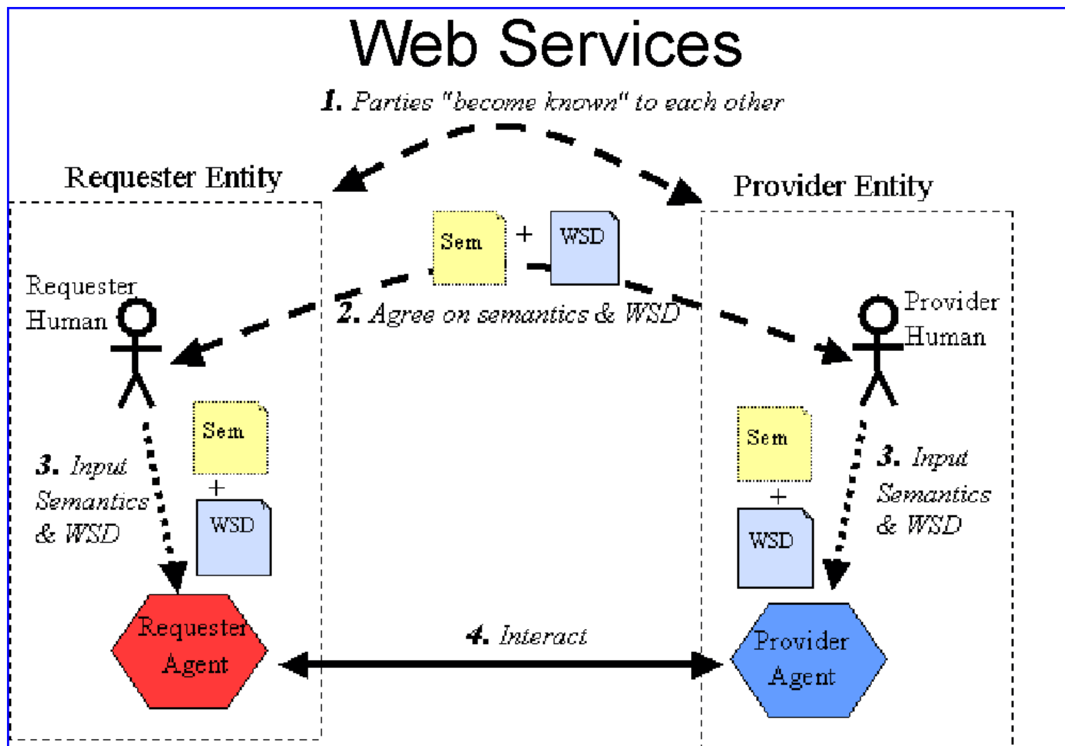


Figure 3-2. Discovery Process

Web services are being standardized by the W3C. Quote from
http://www.w3.org/TR/ws-arch/#id2260892

1.1 Purpose of the Web Service Architecture

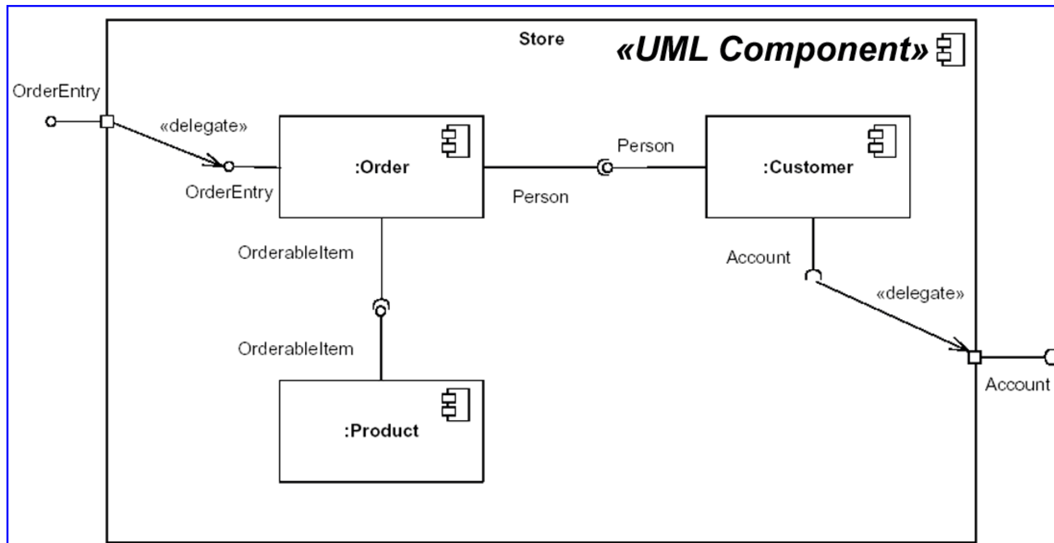
Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. This document (WSA) is intended to provide a common definition of a Web service, and define its place within a larger Web services framework to guide the community. The WSA provides a conceptual model and a context for understanding Web services and the relationships between the components of this model.

The architecture does not attempt to specify how Web services are implemented, and imposes no restriction on how Web services might be combined. The WSA describes both the minimal characteristics that are common to all Web services, and a number of characteristics that are needed by many, but not all, Web services.

The Web services architecture is an interoperability architecture: it identifies those global elements of the global Web services network that are required in order to ensure interoperability between Web services.

This looks very promising from our point of view. We see the people as well as the machines and consider the interplay between them. Web Services may be the answer to the needs foreseen in 1972. We must only hope that Web Services do not follow the path of Enterprise Java Beans: From the simple to the extremely complicated.

4.11. UML



In UML 2.0, a Component is an object that encapsulates a number of internal objects. Objects outside the Component access its operations through *Ports*. A port is characterized by one or more interfaces, each interface is either *required* or *provided*. Components are interlinked by matching required and provided interfaces. The internal objects of a Component can be other components or simple objects.

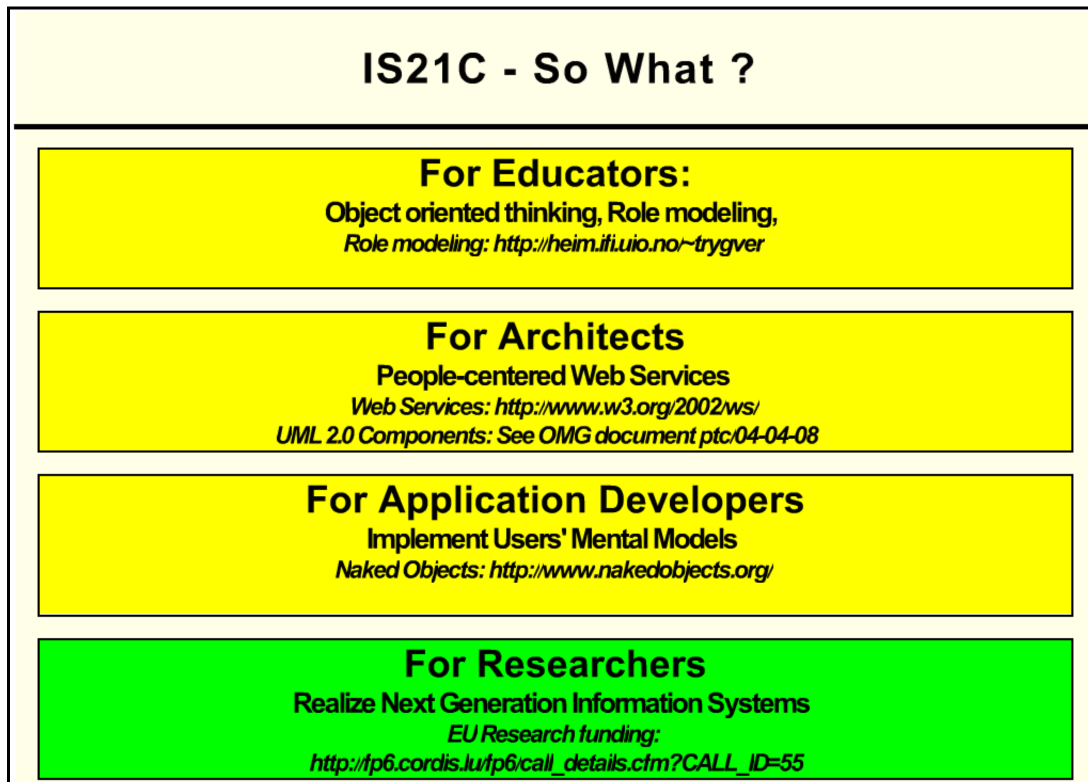
We need a powerful, yet simple way of creating modules in the object world. Such modules of objects should help clarifying the stystem structure, and they should also support security and privacy. Note that the well known notions of namespaces and packages do not address this need since they are concepts used to discipline the programmer at build time. We merge build time into run time and need run time modulatization of interacting objects. The notion of a Component in UML 2.0 seems a good starting point for further exploration.

4.12. BabyUML

The sum of it all. We earlier illustrated our current ideas by a couple of demonstration examples, but there is much more to it than these simple examples can show.

5. Conclusion: BabyUML - So What?

Mixing it all together will take many years. So a reasonable question is: What can we do *now*?



5.1. For Educators

The key is to teach students to think in systems of interacting objects even if they have to be implement in by classes. Models are built with objects and the Object is the key concept. Classes are auxiliary assistants; often useful but not essential.

Role modelling is the premier object modelling technique. My book “Working With Objects” is out of print. Its last draft can be freely downloaded from my web site. A primitive version is part of UML under the name of Collaborations.

5.2. For Architects

The key is to promote users to becoming first class members of the development team and partners in building the domain model. This model will become the users’ mental model and the application’s object model. It should be possible to go top-down from business process to individual application in rapidly repeated cycles. Web Services and modelling with UML 2 Components are useful aids. Also look at Model Driven Architecture, MDA. MDA will only be really interesting when the model becomes the program and MDA becomes MDD - Model Driven Development.

Also look at Douglas Engelbart’s Bootstrap idea to see if you could use it to bootstrap your organization. (http://www.bootstrap.org/)

5.3. For Application Developers

Promote the users from passive script followers to active problem solvers. Let the information system augment their intellect. Check if the Naked Objects approach is applicable to your requirements at http://www.nakedobjects.org/welcome.html#welcome

5.4. For Researchers

You now have the opportunity to introduce a fundamentally new way of working with information systems.

I wrote my first programs by putting bits directly into memory. Machine code was the first generation programming language.

We then got assemblers. The metamodel was still the hardware instruction set. This was the second generation programming language.

The third generation languages followed in the late fifties and early sixties. A program is a specification of a machine. The third generation languages model the computation independently of the hardware so that the programmer could think on a higher level (statements and blocks rather than bits). Fortran, Cobol, Algol were among the first of these languages. They are essentially textual, limited to punched card input and lineprinter output by the limitations of the early hardware. Current languages such as Simula, C++, Java are essentially third generation. It is about time they were replaced by something more powerful.

The fourth generation languages are the information modelling languages used for database design. They model information and relationships between information entities. Their metamodels are much richer and programmers typically combine text with graphics to master them.

It was most unfortunate that languages for artificial intelligence (AI) were heavily promoted as the fifth generation languages that should solve all our problems. This never materialized, of course, since artificial intelligence is always artificial, never intelligent. But the most important objection is that the fifth generation languages aimed at replacing people by automating their intellect, rather than augmenting it.

It is now time to launch *the sixth generation of programming languages*. Their scope will be much wider than the individual applications of the earlier languages. They should be languages for the modelling/programming of distributed systems, and will essentially be communication-oriented. The models will be very rich and people will work with them in different perspectives depending on viewpoint, interest and prior knowledge. I believe the metamodel should be something like the OMG Meta Object Facility (MOF). Different environments for different kinds of information systems can be created from this building block. A UML-like modelling language is but one example.

The first electronic digital computer was the Colossus. The first stored program computer was the Baby, and the first practical, modern computer was the EDSAC (October 1946). The Machester Autocode from 1951 is probably the world's first high level language. All this was created in England, while objects and with their classes were invented in Norway

Perhaps the sixth generation programming/modelling languages and environments will be European. We need to maintain and refine our tradition of relying on individual initiative and responsibility. We need to augment the capabilities of the individual to warrant our high labour costs. So I challenge the European research and development community to make the sixth generation programming language and information environment to become reality. May be that this time, we can make it a commercial as well as a technological success.

Several factors are essential to make this undertaking successful:

1) *Technological vision and leadership.* There are many interesting issues. It will be essential to have a clear goal and separate the important issues from the side issues.
2) *Scientific and technological excellence.* Many fields of expertise have to come together to create the required technlogy.
3) *High quality administrative leadership.* Managing a project of this size and importance takes a first class conductor.
4) *Business acumen.* A new business model is needed, a model that makes money by delivering high quality products rather than selling a stream of inferior versions. Part of this factor is the salesmanship needed to move the product into main stream computing.
5) *Finance.* Financing of the initial stages should be possible through the EU 6th Framework Programme. The development could start as a fairly small STREP and mature trough a much larger IP. (See below)

The most important of these factors will always be whichever is the weakest one because they are all essential for success.

Taken all together, this project should be very interesting, rewarding, and even fun.

The following information about EU funding has been extracted from
http://europa.eu.int/comm/research/fp6/pdf/fp6-in-brief_en.pdf

2.4 Specific Targeted Research Projects (STREP) and Specific Targeted Innovation Projects (STIP)
STREPs and STIPs are multipartner research, demonstration or innovation projects. They are an evolved form of the shared-cost RTD projects and demonstration projects used in FP5. Their purpose is to support research, technological development and demonstration or innovation activities of a more limited scope and ambition than IPs. The Community contribution may range from hundreds of thousands of Euros to a few millions of Euros and is paid as a grant to the budget (percentage of total costs of the project). There must be a minimum of three participants from three different Member States or Associated States of which at least two are from Member States or Associated Candidate States. Different minimum numbers may be specified in the calls for proposals. Special conditions for minimum numbers of participants apply for the “Specific international co-operation activities (INCO)” part of the programme (to be specified in the work programme).

2.2 Integrated Project (IP)
IPs are multipartner projects to support objective-driven research, where the primary deliverable is generating the knowledge required to implement the thematic priorities. IPs should bring together a critical mass of resources to reach ambitious goals aimed either at increasing Europe’s competitiveness or at addressing major societal needs. They must contain a research component and may contain technological development and demonstration components, as appropriate, as well as perhaps a training component. A project may be at any point in the research spectrum. A single project may indeed span large parts of the spectrum, i.e. from basic to applied research.