e l e m e n t s

The elements Object Model
Overview

Doug Palmer[1]
TARMS

Patric de Gentile-Williams[2]
TARMS

September 8, 2000

Typeset in LaTeX.

[1] `doug@tarms.com`
[2] `pdgw@tarms.com`

This document is an overview of the elements Object Model project. It covers a miscellany of areas: the justification and ideas behind the project, the scope of the project, modelling conventions, documentation conventions and general philosophy.

Those wishing to examine elements from a business perspective are urged to read sections 1, 2 and 3. Appendix A may prove useful if you are not familiar with the software engineering philosophy behind object modelling. Readers with a more technical bent will also find sections 4 and 5 of interest.

1 Introduction

The financial industry is faced with a period of dramatic change, in which competitive pressures are forcing violent industry consolidation. At the same time a significant increase in the operational complexity of these organisations, coupled with major increases in risk profiles of these businesses, are increasing the demands made of information technology (IT).

Regulators are confronting these issues by requiring greater and greater levels of disclosure and analysis of risks and processes.

One of the most immediate consequences of these market and regulatory trends is the need for financial institutions to invest substantially in IT. Unfortunately, the complex nature of the required systems and the lack of standards in the industry has lead to a great many failed development projects, to projects whose scope was dramatically curtailed, or simply to projects whose budgets and deadlines were massively extended in order to meet the original specification.

As a result many organisations operate with woefully inadequate systems, on the grounds that they either cannot afford the ideal system or they cannot afford the risk of trying to install the ideal system. In many cases these organisations do not have the expertise to know what is achievable. Even the most sophisticated institutions, those driving the theory of finance forward, tend to have little standardisation in their systems.

Today, most organisations have a plethora of disparate, unconnected systems. The operating costs of financial institutions are huge, as all of these disparate systems have to be individually supported and administered, as do their data sources. The consequences are inevitable: the cost of continued ownership is far higher than the purchase cost.

There are many financial software products available on the market, few of which adhere to any standards. Further, the risks of purchase of off-the-shelf software are massively increased by the risk of incompatibility with existing systems.

The obvious question arises as to whether anything can be done either by the financial industry itself, the regulators, the IT industry or academia to reduce the cost and risk of purchase, installation and ownership of these systems. To do so would both increase the efficiency of the industry and reduce the frequency of Barings, Daiwa and Nat West type incidents.

1.1 What are the Pieces of the Solution?

In IT as in many other industries, problems of productivity have often been solved by the creation and adoption of standards. More often than not these standards have emerged from a ruthless process of natural selection: eg., Windows or TCP/IP (the Internet's data transfer protocol).

In the construction of any large software system, there are two key areas to consider: the architecture of the system (particularly for distributed systems) and the modelling of the data, relationships and processes.

A number of basic solutions have emerged to address the architectural issues of software development.

- Open systems
- Object request architectures
- Copy based architectures
- Standard middleware products
- Web delivery strategies
- Software and architecture standards

The middleware industry has been making significant progress towards the creation of standard messaging and communications architectures. This has been achieved either through the promotion of message based architectures (eg. Rendezvous[26] or MQSeries[17]) or through the use of object request architectures (eg., CORBA[20], DCOM[4], etc.).

On the modelling of data and processes the finance industry is making concerted efforts to implement standards for Straight Through Processing, (via organisations such as GSTPC, ISITC[12], SWIFT, FIX[8] and the DTC[6]). In these efforts the modelling of inter-organisation data exchange is making significant progress. In addition, initiatives like RiskMetrics and CreditMetrics[25] are providing some standards in risk management methodologies — however the implementation of these methodologies is still prohibitively expensive due to the tailor made nature of their implementations.

On the middleware front, network data models, such as FpML[9] and FinXML[7] are endeavouring to provide standardised message formats for the description of complex trade data and instruments. Other standards, such as FIX[8], OFX[19] and BizTalk[1] offer common languages for transactions.

Unfortunately, the most basic of modelling issues are not addressed. Any financial software application development project will need to begin with basic issues such as the construction of date calculation routines, the definitions for financial instruments, the methodology for calculation of profit and loss — issues which should be covered by a common object model[3]. Some of the advantages flowing from a common object model would be:

- Multiple systems would be able to use common reference data, position and limit servers, a VAR server, etc.
- Proprietary option valuation models could be plugged seamlessly into off-the-shelf systems.
- Proprietary rate curve builders would produce results that are entirely intelligible to all other system components built with the object model.
- Data could be aggregated from various systems for positions, sensitivities and risk.

From the point of view of the financial industry, the particular area which seems to be crying out for standardisation is the production of of a good (or indeed any) standard object model. The standardised network data models, discussed above, provide some pieces of the solution and, indeed, demonstrate the need for standardisation and the general acceptance of such an approach in the financial markets. However, these formats essentially provide a standardised layout for data; there is still a need for a standard for interpreting the data and providing common behaviour and meaning.

1.2 How does TARMS Propose to Solve the Problem?

TARMS proposes to publish an *Object Model for Finance*, as well as an associated network data model[4] and various business process models, placing these in *Open Source* (see section 3). This will make the models accessible not only to financial institutions, but also to software development companies, data providers, consultants etc. The models will cover such areas as:

- Deal Management
- Instrument Modelling
- Risk Management
- Accounting
- Rates
- Dates
- Reference Data

[3]See appendix A for a discussion of modelling
[4]However, see section 2.2.

- Data Management

To address the problems described above, a common model needs general adoption. Unless a model is imposed by decree, the model needs to withstand peer review so that it can be voluntarily adopted. Peer review implies open publication of the model specification.

The Open Source model ensures that the product of this endeavour will be of the highest standards, both from a technical perspective and from a business analysis perspective.

1.3 What are the Benefits?

The presence of a standard object model provides benefits to a number of groups.

1.3.1 For Financial institutions

The concept of a ubiquitous object model for the development and integration of financial software applications and a product embodying this model will, if successful, deliver a number of crucial benefits to financial institutions. These are:

- Reduced cost of purchase of software packages. If much of the infrastructure of financial software is standardised, then development times are reduced, and consequentially costs are reduced. Indeed, this would also be true for any software that financial institutions develop internally.

- Reduced cost of ownership of software packages. The cost of installation of new packages would be reduced by standard interfaces. The cost of maintenance would also be reduced due to reduced data maintenance (data would be maintained centrally and distributed to subscribing applications).

- Lower risk of purchase. If the risks of incompatibility with legacy systems are reduced by the existence of standards, then the risk of adding a new system is reduced. Hence, the consequences of any individual decision will be smaller, thus leading to quicker decisions.

- Diversification of suppliers. If the above are true, then the risk of buying systems from lesser known suppliers, or introducing unfamiliar operating systems and programming languages is reduced by the lower purchase costs, lower installation time and cost.

- Integration of existing systems. This is made easier by the emergence of standard adapters, as well as by the existence of common network data models.

- Reduced costs of operations for the business as a whole. As all systems within the enterprise become integrated and new functions are added, control and settlement functions are reduced to exception processing, thus considerably improving efficiency.

- Greater control over the trading operations. The current mirage of a tightly controlled organisation becomes a reality, with all business processes reporting on their current status and all decisions made with relevant and timely information.
- Reduced time to market for new products. Once the benefits of standards make themselves felt in earnest, it should be possible either to acquire and install or to develop the infrastructure required to support a new business in a very short time
- Emergence of modular units. Software components will emerge which provide the building blocks of systems development.
- Emergence of a reservoir of skilled staff. The pool of relevantly skilled staff will be greater, so recruitment will be easier, and overall training costs will be reduced. (Or if they are maintained, then the level of competence will rise.)
- Reduced time and costs for mergers. Mergers of financial institutions will be easier, as their systems integration should be far less painful.
- Development of common reference data. Financial institutions will be able to have all their systems use common reference data.
- Coverage of present and future issues. A standard object model, because of its purpose and by incremental refinement, can be rich enough to cover the whole business problem. When new models are designed from the ground up they inevitably leave things out.
- Elimination of "reinvention of the wheel." A common object model will provide standard conventions for things like instrument descriptions and payment schedules.

1.3.2 For Financial Regulators

From a regulatory point of view a standard object model should create a similar list of benefits to those enumerated in section 1.3.1.

- Stronger players in the financial markets. The benefits described above will result in stronger players in the financial markets, either due to reduced costs and therefore higher profitability, or due to increased control within these organisations and therefore lower levels of exceptional losses, and probably both.

- Reduced regulatory burden for the regulator. The regulatory burden of inspecting systems and processes will be greatly reduced due to the existence of standards and benchmarks.
- Improved risk management. Lower costs will lead to a wider adoption of leading edge risk management.
- Improved risk management systems. Lower development costs will lead to more competition in the development of risk management systems, thus improving the accuracy, functionality and scope of available products.
- Reduced regulatory burden for the institution. Once a standard object model is established, regulatory reporting burdens would be considerably eased for reporting institutions. Conversely, regulators could introduce greater reporting requirements at relatively little additional cost.

1.3.3 For Software Development Houses

The benefits for Software development houses would be:

- Shorter development cycles and an easier implementation cycle. A large body of common analysis has already been provided and interoperable libraries of common function will be available.
- Support for specialisation. The ability to focus on the value added parts of their products, thus creating a trend towards specialisation.
- Reduced entry hurdles. The lower risk profile of software purchase will make it easier for small players to break into the business.
- The lower risk of purchase will shorten the sales cycle.
- An easier demonstration process. New products can be painlessly hooked into an existing infrastructure.

2 What elements Is

elements is the *specification* of an object model intended to allow the ready exchange of complex financial information.

The underlying theme running through elements is that the specification allows the modelling of complex financial instruments and conventions in terms of more primitive building blocks. elements is intended to allow the interpretation of the meaning of financial information, rather than treating financial data as a set of magical tag values which feed into equally mysterious valuation routines.

Modelling finance in such a way has several benefits. The most essential benefit is that disparate elements become comparable; eg., a variety of instruments can be reduced to future cashflows and compared directly. Additionally, by reducing financial instruments and conventions to simple building blocks, the elements can be made extensible; a bond with a non-standard cashflow structure can be easily added, for example, rather than being dependent on the introduction of an additional class which handles that particular cashflow structure.

It is, of course, important that additional classes can be added when genuinely necessary. Designing things in such a way as to allow other programmers to extend what we have with minimal pain is very important. We do, however, want to give the sense that most things can be built by configuration, rather than by code, placing control of the system in the hands of the users, rather than the programmers.

2.1 Scope

Table 1 gives an outline of the intended scope of elements. Not all of these features will be available in the first release. Rather, this list gives what we intend to cover in release 2.0.

Table 1: elements Packages

Utilities		A group of utility interfaces useful across many packages.
Object Management	Object Identity	How to recognise and version objects.
	Dynamic Data	Handling dynamically modifiable data.
	Reference Data	Handling reference data.
Dates	Dates	Basic and business date functions.
	Date Rolling	Date rolling conventions.
	Holidays and Weekends	When to roll a date.
	Periods	Business periods.
Reference Data	Locations	Languages and locations.
	Currencies	
	Organisations	Parties outside this institutions.
	Accounts	For back office and accounting use.

Table 1: elements Packages

	Books	Parties within the institution.
	Users	User identity and security.
Rate Infrastructure	Rates	Primitive point rates.
	Rate Curves	1-dimensional rate collections.
	Rate Surfaces	2-dimensional rate collections.
Instrument Infrastructure	Instruments	Basic instrument primitives.
	Deals	Basic deal modelling.
	Contingent Claims	Basic option-like modelling.
Position Infrastructure	Positions	Basic position modelling.
	Maturity Grids	Ways of laying out positions over time.
	Position Adjustments	Ways of directly altering positions.
Deal Modelling	Loans	
	Securities	
	ET Futures	
	FX	
	Contingent Claims	Options and suchlike.
	Equities	
	Repos	
	Indexes and Baskets	
	Complex Contingent Claims	
	Energy	
	Commodities	
Deal Management	Orders	
	Back Office	
	Accounting Events	
Risks	Market Risk	
	Credit Risk	
	Liquidity	
	Risk Factors	

Table 1: elements Packages

	Relative Performance	
	Partial Derivatives	
	Limits	Limits on positions or partial derivatives of positions.

elements concentrates on the areas common to all financial systems. There will be considerable variation between institutions on how certain things are computed and we do not wish to impose a common methodology. However, the presence of certain items can be modelled, even if we do not specify how these things are derived. Only the interfaces to the software elements that will provide the calculations for these models strictly need to be specified.

As an example, the procedures used to calculate and allocate profit and loss will vary from institution to institution. To a certain extent P&L calculations are part of the business model of the institution and part of the institution's intellectual property. It would not be sensible or useful for elements to impose a common set of procedures. The *presence* of P&L, however, can be assumed. A common, sufficiently flexible representation of P&L should, therefore, be part of the model. As a consequence of this, interfaces to P&L construction routines need to be provided.

As another example, the techniques for calculating analytics are likely to vary from institution to institution. Even something as apparently standard as a yield to maturity calculation carries hidden assumptions about accuracy and derivation. Again, elements should not impose a calculation methodology, but rather provide a means of identifying the analytic and necessary interface to the appropriate software.

Naturally, there are some calculations that are both truly common and necessary to allow systems to construct other elements using a common methodology. Interest rate calculations are an example of genuinely common calculations. elements should specify these kind of parts of the model.

In addition to truly common calculations, there are calculations that form an effective standard, even though these calculations could be varied by an institution. Mark to market P&L is an example of a near-universal standard. These calculations can supplied as part of the specification or as examples.

2.2 Auxiliaries

As well as the actual object model there are a number of additional items that need to be provided to make elements usable. Not all of these auxiliary elements can or will be provided with the initial release.

elements objects need to be exchanged between programs written in different languages on different platforms. For architectures which use a copying approach — in contrast to reference-based, CORBA-like architectures — a *network data model* (NDM) or "wire format" is needed to provide an implementation-neutral means of exchange.

There are existing efforts to produce standard NDMs: FinXML[7] and FpML[9]. elements does not, and need not, complete with these NDMs. Instead, if the use of one or other of these NDMs is needed, elements will need to be able to adapt to the models. The elements NDM should not attempt to supersede these existing NDMs, there are several reasons for avoiding such an approach:

- Use of the elements NDM assumes the elements model in all systems. The use of a standardised NDM allows exchange with components which are not elements-compliant.
- elements is intended to be complete and is, therefore, complex. Certain systems may wish to comply with the NDMs, while avoiding the full power of the elements approach.
- The additional complexity of an object model specification over a network data model specification tends to ensure that elements will lag behind the NDMs in terms of instrument coverage. Users may wish to use elements for infrastructure, but also use the standard NDMs for more complete exchange.
- Use of an existing standard NDM eases the path to employing elements for new applications and, therefore, widening the potential user-base for elements.

elements objects need to be storeable in some persistent storage. Although object-oriented databases, such as Gemstone, are available and would be ideal, most IT departments are built around relational database management systems (RDBMS). Persistent storage, therefore, essentially means a normal relational database system such as Oracle or Sybase. A *data model* is needed to allow the saving and loading of objects to and from an RDBMS. This data model may be *partial* in the sense that it is oriented towards what can be cleanly supported by an RDBMS, rather than covering all possible instances of the object model; for example, excessive use of polymorphism may be prohibited.

The data models — both network and relational — form a large auxiliary body of work, dependent on the structure of the object model. We intend to publish these models subsequent to the initial release of elements.

The elements object model needs to have a common representation, so that people can machine-read it. See section 4.2 for the adopted format.

elements also needs human-friendly documentation. See section 5 for a discussion of documentation. There is also a need for supplementary information, such as this document, discussing the philosophy and concepts behind elements.

Although elements is intended to be architecture-neutral, the need to provide interfaces for calculation objects and management objects means that an abstract architecture needs to be specified. This abstract architecture needs to be sufficiently general to allow different physical implementations[5]. A sample architecture, representing one possible physical architecture, also needs to be provided so that a concrete example illuminates the thinking behind the model.

2.3 What elements Is Not

It is important to emphasise what elements is *not*.

elements by itself does not use the information to do anything useful. For example, elements does not calculate analytics, produce position screens or even deal entry screens. An application using elements provides these components. The are, of course, points where elements must provide a common methodology — for example, position construction presupposes a methodology for rolling deals into positions at end of day. In addition, there are points where elements touches upon an *almost*-common methodology; these methodologies need to be specified without forcing the model to use them.

elements is not an architecture; the precise process of information routing and delivery is implementation dependent. We intend to publish a possible architectural implementation, as a guide to implementors. However, software that uses elements should not be restricted to this architectural model.

elements is not a mere transaction model. Most open financial models today are intended for the handling of transactional data. For example, "buy 10 units of commodity A," "transfer USD 200 between this account and that account." There is no sense of what A or USD actually means in financial terms; they are simply identifiers for the commodity being moved. In contrast, elements aims to provide a framework where the various items being traded can be meaningfully compared.

3 Open Source

Open Source[21] is a recently developed framework for giving software away on the Internet.

Commercial software companies face many challenges in developing their business in today's fast-moving and competitive industry environment. Recently many people have proposed the use of an Open Source development model as one possible way to address those challenges.

Moving to Open Source for a product can potentially provide better value to customers, including in particular the ability for customers or third parties to improve that product through bug fixes and product enhancements. In this way a company can create better and more reliable products that are likely to more truly reflect customers' requirements.

[5]In particular, it should not force a choice between message-oriented middleware, such as Rendezvous[26] and distributed object architectures, such as CORBA[20].

This is the model adopted by Netscape when they chose to give away their web server software[16], and is the model used for Linux[14].

Open Source requires all the source code and an executable version of the code to be available for inspection to all users. The Open Source model encourages the user community to make comments, fix bugs, suggest improvements etc. The result is a far more robust product which has not only survived, but actually been enhanced by peer scrutiny. This process is best described by Eric Raymond in his paper “The Cathedral and the Bazaar”[24].

From a standards point of view, Open Source provides obvious benefits. As discussed above, in section 1.2, a set of models will only be of use if they become a de-facto standard. This creates certain necessary conditions:

- The models need to be of sufficient technical quality to inspire confidence;
- The models need to be accessible to all interested parties;
- The investment required in evaluation and adoption of the models needs to be as small as possible;
- The general adoption of the models must not confer a competitive edge to any particular group of users;
- The benefits of adoption must be clear.

We believe that the Open Source model of software distribution and equally importantly the Open Source model of software development go a long way to achieving these objectives.

The peer scrutiny feature of Open Source ensures that there are no nasty hidden surprises, as all users become testers and sometimes fixers as well. This ensures that lessons need only be learnt once. These features have created, both in reputation and also in reality, some of the most stable and high quality pieces of software available. Open Source is now synonymous with high quality.

Open Source is by definition free to all. The standard licence types are very open. The Internet allows for the widest possible distribution in the shortest possible time.

By providing the model free of charge, there will be little in the way of obstacles to evaluation and adoption of the models.

As TARMS is neither a financial institution nor a large software vendor, consultant or data vendor, we cannot be seen as a threat to the financial community. This should result in fewer issues of internal politics preventing an honest evaluation of the models.

3.1 Open Source and elements

The elements model and model documentation are to be provided to the world using the Open Source model. From our point of view, Open Source provides some major benefits:

- Open Source reduces cost of ownership. Anyone using the object model is given the model in source form[6] so that they can examine it, follow the relationships between the pieces and even fix it if it is wrong. Contrast this with a traditional closed API, where you have to take it on faith that everything is OK under the hood. As a result, anyone using the object model is less vulnerable to the costs of *us* making a mistake — they can always fix it themselves.

- Open Source encourages collaboration and contribution. Since other people can examine the model, they can make enhancements and suggestions. From our point of view, this represents research that we don't have to do. We can incorporate any good enhancements into the next release of the model.

- Open Source allows a "Release early and release often" approach. Early releases can be presented to the public for examination, feedback and, for some brave souls, early use. This approach frees us from the need to present a complete package from the outset.

The Open Source appellation is normally used to describe actual programs, rather than the models that programs are built upon. Commonly, models are usually described as "open standards." However, there are a number of reasons for wanting to approach elements from an Open Source perspective:

- The choice of UML (section 4.1) means that the models are available in machine-readable form and can be used to generate structural code. This property tends to encourage a view of elements as a form of source code.

- elements specifications can be viewed as a form of declarative programming. Again, the nature of elements suggests source code, rather than a standards specification.

- Standards tend to be fixed. An underlying aim of elements is to encourage experimentation, modification, improvement and feedback — all aims of the Open Source philosophy. In particular, we wish to encourage the open publication of model extensions.

For these reasons, it seems more appropriate to place the elements models under an Open Source licensing structure. Since we wish to allow the use of the elements model for profit in the form of implementations, we do not wish to prevent proprietary use of elements; some care in the license language is needed, as the standard Open Source licences tend to be couched in terms of source code.

[6] Source form for a UML model being, at present, a Rose petal file or, in the future, an XMI document.

3.2 So How do We Make any Money?

Fair question. It looks like we have just given all our intellectual property rights away. However, note that we are only giving the specification away, along with documentation and some example code and class libraries. We can still make money in the following areas:

- Class libraries based on elements. For example, a Java implementation of elements. These can be sold to anyone who does not want to work through the specs themselves. The specifications and models we are producing describe correct, rather than efficient, behaviour; there is likely to be a range of implementations, with some implementations concentrating on breadth and some on depth.
- Testing and branding other vendor's libraries as "elements-Compliant."
- Components based on elements.
- Education and books.
- Consultancy.
- "Soft" benefits in terms of visibility, etc.

4 Modelling

The general aim of elements is to provide a consistent model for describing financial objects. The aim of the model is to provide a single, consistent structure for financial information, so that this information can be freely exchanged between diverse applications.

4.1 Modelling Language

The *Unified Modelling Language*[11, 2] (UML) has been chosen as the notation method. UML is a common object modelling language distilled from the various methodologies that were developed in the 90s: Booch, Rumbaugh, OMT, etc. The developers of these methodologies agreed to pool the features of the methodologies to provide a single consistent notation, hence the "Unified" in UML. No common development methodology was agreed upon by the various contributors to UML, so UML remains a notation, rather than a complete methodology. UML is now supported by the Object Management Group[20] (the OMG) as the standard object modelling notation, making UML a likely general standard.

4.2 Tools

Rational Rose[3] has been chosen as the basic modelling tool. The choice of Rose is largely a consequence of a trade-off between price, features and flexibility. There are richer offerings on the market in terms of features and general software engineering support. Rose, however, is cheap, fairly ubiquitous and a first choice for those developing object modelling tools. Rose can also be used economically by a single developer; an important consideration when adoption of elements by other developers — as opposed to our development of elements — is considered.

The models created by Rose are stored in *petal* files. Any Rose user can read in a petal file and examine the model. Many other modelling packages can also import petal files[10, 15, 18] making the format an obvious exchange medium.[7]

4.3 Modelling Conventions

UML allows considerable latitude in terms of modelling approach. In order to ensure consistency, some standard approaches need to be imposed. The modelling conventions used are discussed more completely in a separate document.[23]

4.3.1 Packages

The basic approach within the elements project is to break the model into packages containing well-defined areas of functionality. The aim here is to allow users of elements to partially implement the model — enough for their own purposes. There is no reason why a bond-trading system, for example, should need to know about equities or FX options, unless there is some purpose behind the knowledge. To support this approach, the packages of the elements model need to be broken into package-based functional units, each stored as a separate petal file. To use the model for development, the appropriate packages need to be chosen and integrated into a single model.

Functional packages need to introduce two new elements: new classes and additional attributes, operations and associations for existing classes. The additional elements are called “class extensions” as they extend pre-existing classes. The UML package model is essentially a system for partitioning name-spaces and does not support class extensions. To allow a full package model for elements, packages are implemented as full models with stub classes to take the extensions. A Rose script allows the merging of models to form a complete model.

[7] In general, the adoption of the closed petal format runs counter to the general spirit of Open Source. However, the increasing use of XMI[29] and Rational’s support of XMI promises a truly open model format in the future.

4.3.2 Interfaces and Classes

The modelling structure of `elements` has been designed with two major requirements in mind:

- `elements` should be easily translatable into single-inheritance languages, such as Java.
- `elements` should be readily extensible. In particular, modellers should not be locked into a class hierarchy chosen by the model's originators.

Extensive use of interfaces allows both support for single inheritance and extensibility. Interfaces are the basis for most of the `elements` model and are the primary focus for describing behaviour; operations are described entirely in terms of interfaces, as are navigable association ends. Classes provide *an* implementation of an interface. Those wishing to extend `elements` can add additional classes to implement interfaces and use them interchangeably with the supplied classes.

5 Documentation

The petal files discussed in section 4.2 provide the primary representation of the `elements` model. As much as possible, all information on the model is placed in the petal file. The general aim is to generate documentation from the model itself, eliminating maintenance problems.

The documentation spaces provided by Rose, unfortunately, only provides facilities for plain-text documentation. This limitation is likely to prove difficult when layout and emphasis needs to be specified. Since a large portion of `elements` will have mathematical leanings, the lack of math support is likely to prove very difficult. Rose's ability to associate files and URLs with parts of the model is useful, but likely to lead to fragmentation and administration problems.

The process by which documentation is produced is discussed in the documentation conventions document.[22] LaTeX[13, 27] is a typesetting package in common use in the academic community, noted for its flexibility and mathematics support. The input to LaTeX is plain text surrounded by layout and other typesetting commands; a format suitable for inclusion in Rose models.

LaTeX features very sophisticated mathematical typesetting, making it popular with those who have to handle elaborate equations: mathematicians, physicists and, potentially, responsible financial modellers. Mathematical expressions can be added directly into Rose documentation. As an example, the sequence: `$\int^\pi_0 e^{- \xi x^2} dx$` will generate $\int^\pi_0 e^{- \xi x^2} dx$ in any output documentation.

Automatic documentation generation using Rose models starts with a Rose script that traverses the model, generating a LaTeX source file. This source file, along with a package designed to interpret the model, can then be processed by LaTeX (and support tools) to generate Postscript or PDF files.

References

[1] *BizTalk*.
http://www.biztalk.org.

[2] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1999.

[3] Rational Corporation. *Rational Rose*.
http://www.rational.com/products/rose/index.jtmpl.

[4] *Distributed Object Component Model (DCOM)*.
http://www.microsoft.com/com/tech/DCOM.asp.

[5] Edsger Wijbe Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[6] *The Depository Trust Company*.
http://www.dtc.org.

[7] *FinXML, The Digital Language for Capital Markets*.
http://www.finxml.org.

[8] *The Financial Information Exchange Protocol*.
http://www.fixprotocol.org.

[9] *Financial Products Markup Language*.
http://www.fpml.org.

[10] *GDPro Object Modelling Tool*.
http://www.advancedsw.com/welcome.html.

[11] Object Management Group. *What is OMG-UML and Why is it Important?*
http://www.omg.org/news/pr97/umlprimer.html.

[12] *Industry Standardization for Institutional Trade Commission*.
http://www.isitc.org.

[13] Leslie Lamport. *LaTeX, a Document Preparation System*. Addison-Wesley, second edition, 1994.

[14] *The Linux Home Page*.
http://www.linux.org.

[15] *MagicDraw UML Object Modelling Tool*.
http://www.nomagic.com/magicdrawuml.

[16] *The Mozilla Organization*.
http://www.mozilla.org.

[17] *MQSeries Family*.
http://www.software.ibm.com/ts/mqseries.

[18] *ObjectDomain Object Modelling Tool*.
http://www.objectdomain.com/domain/index.htm.

[19] *Open Financial Exchange*.
http://www.ofx.net.

[20] *The Object Management Group*. The standards body for CORBA and UML.
http://www.omg.org.

[21] *The Open Source Page*.
http://www.opensource.org.

[22] Doug Palmer. *The Elements Object Model: Documentation Standards and Techniques*, July 1999.
http://www.afs.net.au/intinfo/ObjectModel/documentation.pdf.

[23] Doug Palmer and Danny Cron. *The Elements Object Model: Modelling Standards*, July 1999.
http://www.afs.net.au/intinfo/ObjectModel/modelling.pdf.

[24] Eric S. Raymond. *The Cathedral and the Bazaar*, 1998.
http://www.tuxedo.org/ esr/writings/cathedral-bazaar/cathedral-bazaar.html.

[25] *The RiskMetrics Group*.
http://www.riskmetrics.com.

[26] *TIB/Rendezvous*.
http://www.rv.tibco.com/index.html.

[27] *The TeX Users Group*.
http://www.tug.org.

[28] Nicklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[29] *XML Metadata Interchange (XMI)*.
http://www.software.ibm.com/ad/features/xmi.html.

A What is an Object Model?

Almost any attempt to understand something involves the building of models. Models are a description of some portion of reality, intended to provide insight into the workings of that part of the world. Examples of models are the climate models used to investigate predictions of global warming, the models of black holes from theoretical physics or the Black-Scholes model for options.

There are limits to the human ability to understand complexity. Models are always simplifications of the phenomena that they seek to describe. A model ship does not reproduce the ship in miniature, down to every nail and shipworm. A model describing a trading room is unlikely to describe the effects of a trader's lunch, or the romantic gossip of the office. Modelling narrows the problems being studied by focusing on only one aspect at a time. This is essentially the approach of *divide and conquer*[5]: attack a hard problem by dividing it into a series of smaller problems that you can solve.

Through modelling we achieve four aims:

- Models help us to visualise a system as it is or as we want it to be;
- Models permit us to specify the structure or behaviour of a system;
- Models give us a template that guides us in constructing a system;
- Models document the decisions we have made.

Modelling is a proven and well-accepted engineering technique, being particularly useful in the following areas:

- Models are built to communicate the desired structure and behaviour of a system;
- Models are built to visualise and control the system's architecture;
- Models are built to better understand the system being created, often highlighting opportunities for simplification and re-use;
- Models are built to manage risk.

Modelling is a central part of all activities that lead up to the deployment of good software. All software development involves modelling. A word processor models sentences, paragraphs and fonts. A flight simulator models the behaviour of an aircraft. A trading system models the flow of deals through a trading room.

Experience with modelling suggests four basic principles:

- The initial choice of model has a profound influence on how a problem is attacked and how a solution is shaped;
- Every model may be expressed at different levels of precision;
- The best models are connected to reality;
- No single model is sufficient. Every non-trivial system is best approached through a small set of nearly independent models.

Traditional structured analysis and programming separates the description of the world, *the data*, from the actions that manipulate the data, *the algorithm*, an approach that can be summed up by the title "Algorithms + Data Structures = Programs."[28] The thing that is being modelled (known as *the domain*) is captured in a *data model*, a description of the important entities in the domain, their attributes and the relationships between them.

This traditional approach is gradually being replaced by a more modern approach, *object modelling*. Object modelling ties the description of the domain and the actions that act upon the domain into a single unit. Rather than try and statically describe the domain, an emphasis is placed on the behaviour of the elements of the domain. Object modelling allows natural expressions of generalisation — "both individuals and companies have similar legal obligations, up to a point, and can be treated in similar ways" — and substitution — "as far as individual transport goes, I can substitute a bicycle for a car without it changing the nature of what I am doing."

Generalisation and substitution (known, technically, as *polymorphism*) can be used to make object models robust and extensible. Using polymorphism, substitutions and changes can be made without affecting other parts of the model.

An example financial model might model a deal by associating it with basic information such as the dealer, the counterparty, identification numbers and an associated financial instrument. The instrument can be any modelled instrument: loans, FX, options, repos. The objects representing the instruments form a hierarchy of financial instruments, with the elements common to all instruments at the top of the hierarchy and specialisations for securities, equities, derivatives, etc. forming subclasses below the common instrument model. A deal can query its associated instrument for information such as (for example) its mark to market value, with each instrument type responding in a different way.

Using the above model, a new instrument can be added without disturbing the overall structure of the model. Provided the model of the new instrument can answer the same sorts of questions as other models, no other part of the model needs to be changed.

Object modelling provides a powerful way of describing and designing software. Inheritance provides a method for analysing and grouping functionality. Polymorphism allows flexibility and robustness.