Persistent Information Space Architecture

PISA Club Rules

Malcolm P. Atkinson, James R. Lucking,
Ronald Morrison, and Graham D. Pratten

University of Glasgow, 14 Lilybank Gardens, Glasgow, Scotland G12 8QQ.,
University of St. Andrews, North Haugh, St Andrews, Scotland KY16 9SS.
and
STC Technology Ltd., Copthall House, Nelson Place, Newcastle-under-Lyme,
England ST5 1EZ.

Abstract

The work of the PISA group is briefly reported to present our views on the essential elements of the PISA club rules, these being the guidelines for using PISA and for enhancing its base functionality. The long term and large scale issues lead us to consider software architecture as important and to propose four domains of influence as a basis for the PISA club rules. The domains of influence are briefly described, applications of the architecture are briefly outlined and the club rules are introduced.

Edition 1

August 20, 1987

1. PISA - objectives and summary

Examination of existing computer systems, particularly their software, exposes many dichotomies and discontinuities in their design and in the design of their user and administrator interfaces. Some of these were deliberately introduced to satisfy reasonable performance constraints themselves imposed by the early and emerging processing and storage technologies of the 1960's and 1970's. Others reflect the order in which the basic design concepts of computer systems have emerged over the last 3 decades. Yet other dichotomies and discontinuities have been caused by bad design or by the structural decay
[39] typical of many large software systems as they are evolved to meet new requirements.

We contend that the present dependence on the plethora of mechanisms such as command languages, editors, file systems, compilers and interpreters, linkage editors and binders, debuggers, performance monitors, service control languages, DBMS sublanguages and graphics libraries increases the cost of understanding and maintaining software, and of training programmers, and decreases the quality of software, even for the simplest activities. We observe also many systems suppliers now need to develop systems constructed from components from several disparate sources or conforming to different external standards; but such systems' users want to be presented with coherent interfaces thus providing a motivation for research into providing coherence.

The primary technical objective of the research by the PISA (Persistent Information Space Architecture) group is to exploit the opportunity provided by the dramatic shift in the cost of hardware relative to software to remove the incoherence between the various programming mechanisms above and thus provide a better environment for exploiting new computer systems. Specific technical objectives thus include:

a. Controlling complexity - by establishing consistent rules which apply throughout the
b. Introducing persistent data and separating the issue of what data structures are best for a program from the issues of identifying and preserving the data;
thereby allowing most file and database data to be processed using the same language
c. Controlling system evolution even though the nature of data including program is tha
d. Protecting data, particularly large bodies of data which are inherently valuable, from
e. Providing for concurrent computation and shared use of data which may be
geographically dispersed.

We have designed and implemented the language PS-algol as a testbed for these architectural concepts and objectives. The language Napier is being implemented to provide additionally a rigorous polymorphic type system.

The new architectures resulting from this research will replace complexity with simple consistent rules and may also permit higher performance execution and storage engines to be built. It is becoming widely agreed that persistence should be used as the basis for building operating systems and knowledge bases [15]. Further objectives include speeding up the development of integrated project support environments (IPSEs) and inteligent knowledge based systems (IKBSs).

Many papers are referenced in this note, but the note's primary predecessors are [8], [9] and [10].

Some changes are expected from this first edition of the Club Rules. They include:

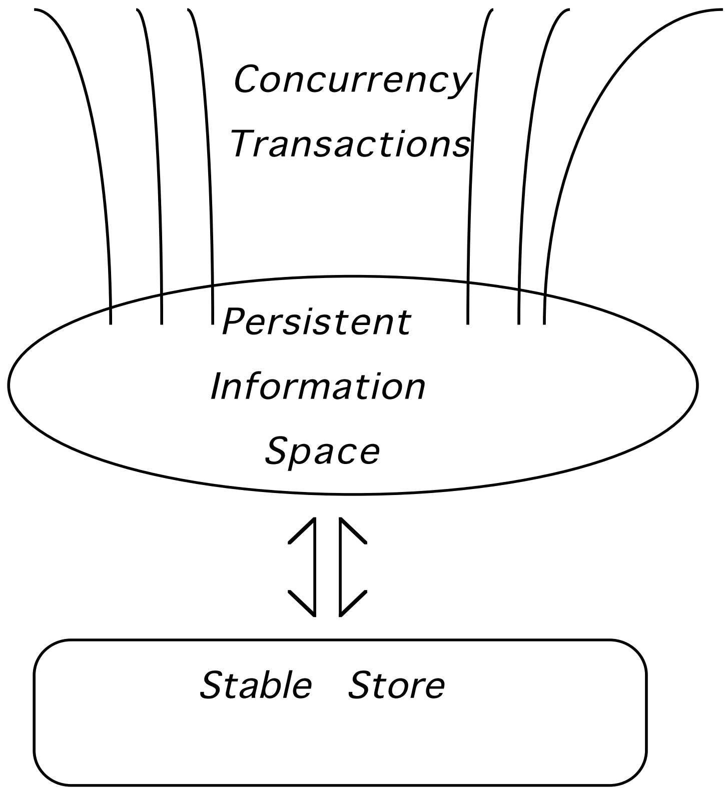
a. Definition of the support roles and requirements within the four PISA architectural d
b. Further definition of the intereaction between persistent systems and external standa

c. The text concerning the proposed "persistent academy" may be expanded with detailed Terms of Reference for persistent systems' configuration
management.

2. PISA Architecture and its Four Domains

2.1 Introduction

The PISA group introduced the concept of persistence to simplify the programmer's world. Any data structure of whatever type may exist within persistent systems for any length of time without explicit coding effort by the programmer [1,2,5]. This concept has proved to have far reaching consequences; for example data and program may be treated in precisely the same way in our systems, with very significant conceptual and implementation savings [6]. Persistence facilitates the reuse of data and program over long periods, but, as longer term usage is considered, new techniques have to be developed [11].



The diagram above abstracts the essential run-time view of the architecture showing a persistent information space supported by a stable store mechanism for reliability. The lines entering the persistent space indicate that several application processes may be sharing that persistent space, using different protocols only if the processes do not share data. Another interpretation of the lines entering the persistent space could be that they demonstrate a multitude of application systems and application systems development methodologies.

When constructing an application system, the support of program and data is divided between hardware and software ultimately on the basis of cost, performance and available technology; thus we envisage a substantial proportion of the applications system environment being provided by software. This software has to conform to an architecture if the programming environment is to remain under control. We take this further. The division between systems software and applications software is an inappropriate artifact of past habits. The architectural discipline and techniques that apply to one domain apply to the other domains, and, as improved software composition technology allows us to consider one person's applications suite as another person's supporting software so will the effects of the discipline and techniques spread. Thus the software architectures being developed will influence all applications programming [30,32].

We therefore divide the total architecture into four mutually supporting domains but we are not using the term domain with its strict mathematical meaning. Rather a PISA architectural domain is a slightly portmanteau concept characterised as a group of users and uses, a set of technologies (programming ones normally) and facilities available to that user group, and a requirement for continued support and maintenance of the technologies and facilities. The four domains are:

The Application Domain,
The Language Domain,
The System Building Domain and
The Store Domain.

2.2 The Application Domain

Most software effort is expended writing application software, and we consider it important to improve the organisation of this work and to support it with suitable methods, tools and components. Users within the applications domain build, taylor and specialise new systems from existing ones. Users in the applications domain may have access to all the tools provided by the architecture and implementation including the persistent stores, system construction tools, and language building components as well as programming languages and existing application systems. Products developed for use within the applications domain may also have access all the tools provided by the architecture.

Specific facilities for applications building include generic tool sets, which can work over a range of types and sizes of data, and adaptive components, that modify their behaviour to suit new types of data, types that may not have been declared when the component was coded and compiled. We expect all system users performing programming tasks to perform those tasks within this application domain, though the products created, particularly if they provide facilities within other domains, may not themselves use some of the application domain's facilities. (Note an important objective of the architecture is to reduce this set of facilities.)

2.3 The Language Domain

Users within the language domain provide applications systems implemented in a persistent language. The language domain provides the facilities to support persistent application systems building in the applications domain. The languages currently supported are PS-algol, Napier and Hope$^+$.

We expect the language domain to cater for all programming activity including the control of the programmer's environment. The languages must support the easy development of data models, generic tool sets, object stores and plug in componentry for system building. Specific responsibilities of the language domain is that it maps the persistent information space characteristic of our languages onto a stable store by providing suitable binding mechanisms, suitable type systems and a contextual naming scheme.

2.4 The System Building Domain

The system building domain supports the construction of the persistent language and the persistent environment. The major uses of the system building domain are the persistent programming language systems themselves though the facilities provided within these domain will normally be programmed in the applications domain.

The major system building tools provided in this domain are

(a) compiler componentry
(b) support for compilation and execution merging
(c) abstract program graphs
(d) abstract machine design
(e) demand driven optimisation

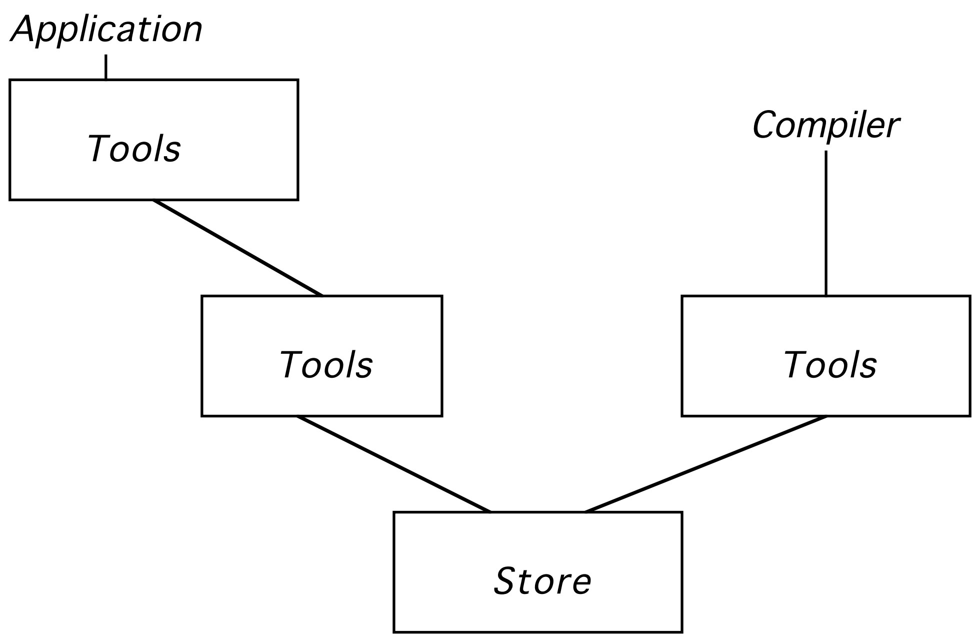
One difficulty exposed within this domain, typical of its essentially systems programming rather than application programming flavour, is that these tools sometimes require facilities whose use needs to be strictly controlled. We have endevoured to keep such inconsistencies to a minimum.

2.5 The Store Domain

One of the principal uses of computers is as a storage device - much system building is concerned with constructing reliable repositories for data with a variety of data capture, data retrieval and organisation mechanisms. A large proportion of programmers therefore need to use the architecture and languages to visualise a store and specialise it to their needs. The PISA approach has developed a uniform view of an arbitrarily large reliable and distributed store populated by strongly typed objects. The programmer defines new types of store by defining storage types and instances of those types become objects. Thus the power of the type system to define the range of stores required is of paramount importance, and the integrity of the store is achieved via type checking. Type is again important as a means of communicating to the store information about objects, essential when data is to be transmitted between hetrageneous, distributed components while preserving semantics. The type information can also be used to activate special treatment of some objects, for example: the use of special purpose hardware, data compression, encryption and protection.

3. PISA Club Rules

3.1 Use of Architecture



The diagram above illustrates that users within the language domain and system construction domain, or rather the products that these users produce, have access to a different set of underlying facilities. We urge most ardently, however, that implementors of these products should use the facilities provided by the application domain during their products' development.

3.2 General Rules

The most important rule, which possibly cannot be phrased accurately enough for a strict interpretation is that exploiters, users and implementors of persistent systems should conform to the spirit of this persistent architecture and use the architecture sympathetically. We have included some of implications of this statement in the text of this document. One important implication is that designers of new applications which require new features from a persistent system should implement those features and build them into the appropriate domain rather than exploiting some quirk of an existing persistent facility.

We consider that only by basing the overall architecture on the well defined semantics of a language such as PS-algol or Napier can the architecture be kept consistent and understandable.

We consider that the language PAIL (Persistent Architecture Intermediate Language) [33] provides an appropriate description for the abstract program graphs which describe compiled programs.

While we will continue to minimise the number of "priveleged" facilities needed specifically for systems programming purposes, exploiters and users of persistent systems should, if possible, eschew use of these facilities.

3.3 The External Influences

Persistent systems should not needlessly deviate from established International and Formal Standards [38]. We believe also that appropriate attention must also be paid to those standards being developed to facilitate Open Systems Intercommunication.

Further persistent systems will be used to impose a veneer of coherence upon a set of systems which themselves are alien. Hence persistent systems themselves should not needlessly diverge from state-of-the-art practices, particularly those associated with alien data systems.

3.4 The Persistent Academy

It is probably inevitable that several different persistent systems will be produced particularly for research purposes while others may be used to support commercial or (otherwise) valuable applications. The facilities provided within the various domains may change as experience and further research demonstrates the desirability and feasibility of enhancements. We therefore propose that a persistent academy be created to apply a modicum of configuration management to persistent systems.

Note that subjecting proposals for new and improved persistent systems to some form of scrutiny, if not inspection, as implied by normal committee work should be beneficial to all concerned with persistent systems. The persistent academy could also serve as a mechanism for interchanging ideas as well as a design control forum. Appendix 2 of this report lists some of the desirable objectives and functions of such a forum.

4. PISA Developments

In this section we introduce and annotate some more detailed "club rules" by commenting upon the development of persistent systems themselves. A list of references to interface definitions is included where appropriate. Note that the papers referenced are essentially snapshots indicating the results of our research at publication date. Longer term research into many technical areas including the issues of system evolution, of heterogeneity and of algorithms applicable in distributed stores will, however, be needed.

4.1 General

An important goal of a software architecture is to facilitate the reuse of program and data. This requires a sufficiently good component description system to allow components to be found, and to check on their composition, and a sufficiently powerful binding system to allow choice of the rebuilding necessary to replace components with improved components. As it is not only the components themselves, but also their definition which sometimes requires evolution, the binding mechanisms must incorporate support for meta data access and evolution [13].

Many of the approaches to software, data description and organisation which work quite well for small collections of data and program become intractable when the system is large or long lived. For example, appealing to a universal name space or a single name allocation authority. These issues become more significant as distribution is attempted. Their solution depends on a recognition of the need for independence and an architecture based on a federation of independent agents interacting via negotiated protocols and descriptions [8].

4.2 The Application Domain

The basic principles of persistence and their implications for type systems and bindings are well understood and are ripe for development. The effort needed here is the investment in good implementation technology including support for object addressing and type checking based on structural equivalence. Significant development is now possible in building systems using persistence: operating systems, data models, data base systems, AI systems, object stores, generic tool sets and a wide range of application suites [30,32].

During our implementation of applications, we have worked with a class of adaptive components which use the compiler at execution time to produce new subcomponents which match specific given data - gaining both flexibility and efficiency [9,13]. An application of this technique [30] has allowed us to rapidly implement a variety of data models and data manipulation languages. This will have benefits in the construction of new data base models, the verification of the utility of existing proposals, and the production of data and object oriented application systems.

The adaptive components can allow high performance algorithms for large scale data to be encapsulated in a manner similar to numerical algorithms. Developing such algorithms is a continuing line of research. It has become clear that no one definition of transaction suits all applications, so the applications architecture must provide the means for defining appropriate transactions [51,40,62]. Without the discipline of a well defined architecture, work within this application domain soon deteriorates into a morass of complexity.

As the application programmer has to interact with the environment, a consistent model, independent of machine and installation must be available [16,46,48,49,57].

List of interfaces used by Application Domain products

As stated earlier all the facilities of the architecture are available for use by application domain products; the following list identifies the more significant programming technologies. The references quoted mostly indicate papers which outline the nature and scope of the facility and which could also be used to guide exploitation of the facility.

PAIL [33]	Compiler Componentry [34]	
PS-algol [59]	Hope$^+$ [55]	Napier [52]

4.3 The Language Domain

At present we have two demonstrator languages PS-algol and Napier on which we have based our various experiments. The major issues investigated during the development of these language include:

Type System Design
Binding mechanisms
A contextual naming scheme.

4.3.1 Type System Design

When the environment is included in our remit this includes types to describe the data models and the data bases we use. Ideally we would like a set of types and a type algebra so that by a succession of operations of the algebra and the provision of parameters we can define a data type equivalent to any data model or conceptual data model [7,14,20,21,22,30,41,50,54,56].

In traditional terms we wish to merge the concepts of type systems with data base schemata to provide a type secure environment. One result we have already achieved is that the type equivalence rule for persistent systems is structural equivalence. This allows independently prepared types or schema to be matched [11]. This is only one step away from the facilities required of the schema to support a distributed persistent store. That is, the schema should be flexible enough to be held in parts, in distributed dictionaries rather than in a centralised one.

When describing and using large scale and long term data, one method of system modelling is to encapsulate the data in a system of co-operating concurrent processes with an access protocol. Support for concurrency must be included in the type system to allow for this kind of modelling [51,62].

With long lived data we cannot predict its use, we require data type and a schema that adapt to changed requirements. For example, it is often necessary to add attributes to the data without destroying the old data or bindings to it.

Our persistent language Napier contains a type system based on types as sets [7,24,50]. This allows polymorphic procedures of universally quantified types [35,43,44], abstract data types of existentially quantified types [45] and extensible data types [7]. The schema may be distributed with structural equivalence as the matching rule for types. Such a type system may be used to develop data models, generic tool sets and plug in component system building.

4.3.2 Binding mechanisms suitable for coping with adaptive data.

We have identified the inconsistencies of the binding and naming mechanisms present in current programming languages, operating systems, file systems and database management systems as a major source of distracting complexity in system construction. A binding consists of a name, a value, a type and an indication of whether the object is mutable or not and there are 16 different categories occuring commonly in programming languages, file systems, operating systems and database management systems.

With static binding there can be no change in the system. That is, since everything is static we cannot accommodate any alteration to the data. To allow system evolution in a controlled manner binding mechanisms are required within the language that accommodate change. Such binding mechanisms are dynamic or at least incremental.

We expect some bindings to be static for safety and others to be dynamic for flexibility and we have proposed the notion of flexible incremental binding sets in persistent systems to deal with this [12]. Some of our success in building software systems out of components result from deciding correctly which of the components are statically bound for safety and which are dynamically bound for flexibility. Our mixture of binding mechanisms allows for this, and our experience shows it can be understood and exploited by applications programmers and builders.

4.3.3 A contextual naming scheme.

With persistent systems the naming of objects in a conceptually flat store becomes a major problem. If the name space were also flat then name clashes and identifying components would become a major problem to the user. Programming languages (block structure), file systems (file directories) and operating systems (segments) have proposed solutions to this problem by imposing a contextual naming system in the form of a tree. In the most general case, the persistent information will form a graph and we have developed a scheme based on extensible environment constructors that allow contextual naming on the graph. In general, the environment constructors allow us to dynamically construct contexts in a manner that models both block structure and directories.

Names also have an important role to play in the description of data and in bindings. For this reason we have invented a method of abstracting over the names in a set of bindings in a type secure manner. This allows us to discover the names in bindings and to write algorithms which depend upon the meta-data without loosing the normal type and value system of the language [13].

List of interfaces used by Language Domain products

As stated earlier all the facilities of the architecture are available for use when developing language domain products; the following list identifies which of these facilities may be used by language domain products themselves. The references quoted mostly indicate papers which outline the nature and scope of the facility and which could also be used to guide exploitation of the facility.

PAIL [33] Compiler Componentry [34] Stores [17,18,19]

4.4 The System Building Domain

We have investigated various systems programming facilities with the intention of making them more generally, but also more rigourously and more safely, available. These include:

Compiler componentry
Support for compilation and execution merging
Abstract program graphs
Abstract machine design
Demand driven optimisation

4.4.1 Compiler componentry

_____Plug in tool sets are a very important method of system construction and the early UNIX systems were a good example of this. The essence of the technique is provision of the framework for storing individual components together with a method of using them in the construction of larger systems.

In order to support the rapid prototyping of compiler systems we have developed a technology based on plug in componentry. The compilation system is composed of separately compiled modules which are plugged in to the compiler itself as it is dynamically loaded. This technique has been used to develop compilers for PS-algol [26,59], Napier [7], SASL [61], STAPLE [31] and Hope$^+$ [42]. Each compiler module when run, plugs itself in. When the compiler runs, it constructs itself from the latest versions of these components in the persistent store.

We will continue to use the plug in componentry architecture to both aid the construction of systems from generic tool sets and also to replace the generic tools set method when it is not appropriate. These methods of system construction can also be used in the applications domain.

4.4.2 Support for the merging of compilation and execution

In persistent systems the boundary between compilation time and execution time is blurred. It is further blurred when we consider interactive compilers which may work against objects in the persistent store. To support this we require a compiler that is callable from within the system, in a type secure manner. This raises issues of where the type security is controlled in the system that are the same as the issues of capability control in machine architectures.

Other support facilities, such as browsers [32,47] and debuggers, must also be provided without endangering the type security of the persistent store.

4.4.3 Abstract program graphs

The traditional method of producing files from compilers, often losing information for compactness, is not necessary in a persistent system. The tables and graphs produced by the compiler can be kept until execution without explicitly altering their form. This allows better system diagnostics to be generated including browsers and debuggers.

An abstract description is also required of the abstract graphs. We have invented such a description called PAIL (Persistent Architecture Intermediate Language) [33] which we are using as the standard for code generators to work against. We are at present discussing with other groups the canonical form of such graphs.

4.4.4 Abstract machine design

As with any architecture there is an abstract model of a machine on which it will best execute. This method of compiler design has been common in defining an abstract machine to execute the language. Both PS-algol and Napier have well defined abstract machines [58,61]. These are block retention abstract machines to support both higher order procedures and abstract data types. We are discussing with other persistent language implementors (Albano at Pisa) on the ideal abstract machine [25].

4.4.5 Demand driven optimisation

With abstract descriptions of program graphs, abstract machine descriptions and program source available throughout the lifetime of the program and data it is possible to optimise the object according to its use. The optimisation is merely another view of the object and we can use it for greater efficiency [27].

We expect lazy demand driven optimizers, based on usage statistics, to be developed in persistent systems.

List of interfaces used by System Building Domain products

The following list identifies the facilities of the architecture which may be used by System Building Domain products. The references quoted mostly indicate papers which outline the nature and scope of the facility and which could also be used to guide exploitation of the facility.

Stores [17,18,19]

4.5 The Store Domain

Many systems programmers will have had first hand experience not only of the dichotomy between virtual store processing and file store proceesing but also of the very real problems in choosing a store interface which provides for both virtual store based and file store based applications. We now believe that we have developed an appropriate level of object typing for use in a persistent object store. The reliability of the store is also an issue for persistent systems and we have developed several, often complementary, mechanisms for supporting stable stores [3,4,17,18,23,28,29,36,37,60]. The overall performance of the store is critical to the total computational process, and this and the reliability requirement requires further development of adaptive algorithms and the introduction of hardware support for this kind of storage architecture.

5. Acknowledgement

We acknowledge the support of ICL, Alvey and the S.E.R.C. in enabling us to have brought the research to its present state.

Appendix 1 Document Control

A1.1 Table of Contents

A1.2 Document History

Edition 1

See section 1 for this documents predecessors. The first draft version of this PISA Club Rules document, identified as Draft 1 and dated July 3, 1987, was circulated only to MPA, JKB, NC, RM, GDP, by its editor JRL. A pre-publication edition 1, dated July 31, 1987 received the same circulation and included as the only major change a new appendix, numbered 2. This publication version of edition 1, dated 20 August, 1987 reflects some clarifications and further references.

Appendix 2 The Persistent Academy

The creation of a "persistent academy" to apply a modicum of configuration management to persistent systems was proposed in section 3.4 and this appendix briefly outlines its intended functions though initially we attempt to justify its creation.

There will always be a tension between basic research into software systems production and the use of the results of that research particularly when, as for persistent systems, these results are themselves significant (and significantly useful) software systems which can be used both as a basis of and for developing other research and further software systems. The requirements of systems users are for well validated and supported systems exhibiting some stability in interface specifications whereas research into systems requires repeated experiments with slightly different systems. One basic objective for the persistent academy is to provide a constructive forum within which the energy generated by these tensions may be exploited. Experience with other software systems and their standardisation does indicate that the earlier these tensions are recognised and solved via some form of communal action, then better future developments occur without necessitating a too rigid change control process which stifles innovation.

The persistent academy will incorpate at least two "parts", namely its forum and its secretariat/administration. Please note that we make no attempt yet to allocate functions and responsibilities to these parts.

Functions to be performed by, or under the technical direction of, the persistent academy include

marketing persistent systems, that is discussing/inventing their application/exploitation

discussion forum for all persistent systems issues and ideas

design control forum for persistent systems

supplier of "standard persistent systems" for exploitation

support of standard persistent systems

One important aspect of the proposed academy is that it must involve regular meetings and will thus require funding or sponsorship. In this respect, a suitable paradigm for the academy may be a formal, or informal, standards development committee and this comparison underlines other requirements, for example the existence of a formally defined and operative persistent system is an important pre-requisite for developing a standard. But, the proposed academy should include much more innovation and exploitation than the normal standards body. Perhaps the Standards Promotion and Application Group founded for IT standards in general and OSI standards in particular is a more relevant paradigm.

Appendix 3 Bibliography

1. Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-420.

2. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.18,
No. 7, (July 1981) 24-31.

3. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272

4. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience,
Vol.13, No.3

5. Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983,
Vol.26, No.4, 360-365.

6. Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct.
1985).

7. Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of
Data Types and Persistence Workshop, Appin, August 1985, 1-24.

8. Atkinson, M.P., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Pr

9 Atkinson, M.P., Morrison, R. & Pratten G.D.
"PISA Club Rules Position Paper" PISA Project Report PISA/1 Issue 1, 6
Dec '86

10 Atkinson, M.P., Morrison, R. & Pratten G.D.
"Persistent Information Architectures", Universities of Glasgow and St
Andrews
PPRR-36-87

11. Atkinson, M.P., Buneman, O.P. & Morrison, R.
"Delayed Binding and Type Checking in Database Programming Languages",
in preparation.

12. Atkinson, M.P. & Morrison, R.
"Flexible Incremental Bindings in a Persistent Object Store", Universities of Glasgow and S

13. Atkinson, M.P. & Morrison, R.
"Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store", proceedings of Persistent Object Systems: their design, implementatio

14. Atkinson, M.P., & Buneman, O.P.
"Database Programming Language Design", Universities of Glasgow and St Andrews PPRI

15. Balzer, R.
"Living with the Next Generation of Operating Systems", proceedings of Information Pr

16. Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", University Computing, 8, 2 (Summer 1986

17. Brown, A.L. and Cockshott, W.P.
"CPOMS - A Revised Version of The Persistent Object Management System in C". Universities of Glasgow and St Andrews PPRR-13-85.

18. Brown, A.L.
"The PINT POT (Persistent INformation space archiTecture Persistent Object sTore)
- in preparation - University of St Andrews.

19 Brown, A.L.
"A Distributed Stable Store", proceedings of Persistent Object Systems: their design, implementation and use Workshop, Appin, August 1987, pp461-468, Universities of Glasgow and St Andrews PPRR-44-87.

20. Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303.

21. Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986.

22. Buneman, O.P. & Ochari, A
"A Domain Theoretic Approach to Higher-Order Relations", Universities of Glasgow and S

23. Campin, J. & Atkinson, M.P.
"A Persistent Store Garbage Collector with Statistical Facilities", Universities of Glasgow and St Andrews PPRR-29-86.

24. Cardelli, L. & Wegner, P.

"On understanding types, data abstraction and polymorphism", ACM.Computing Surveys 17, 4 (December 1985), 531-523.

25. Cardelli, L.
"The Functional Abstract Machine", Polymorphism, VOL 1, NO 1 (1983).

26. Carrick, R., Cole, A.J. & Morrison, R.
"An Introduction to PS-algol Programming", Universities of Glasgow and St Andrews PPRI

27 Carrick, R. & Munro, D.
"Execution Strategies in Persistent Systems", proceedings of Persistent Object Systems: their

28. Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

29. Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383.

30. Cooper, R.L., Adberrahmane, D., Atkinson, M.P. & Dearle A.
"Constructing Database Systems in a Persistent Environment", Proc VLDB Brighton 1987.

31. Davie, A.J.T., Munro, D. & McNally, D.J.
"An Informal Description of the STAPLE Language Version 1.4", University of St Andrews CS/86/3.

32. Dearle, A. & Brown, A.L.
"Safe Browsing in a Strongly Typed Persistent Environment", Universities of Glasgow and S

33. Dearle, A.
"A Persistent Architecture Intermediate Language", Universities of Glasgow and St Andrews PPRR-35-87.

34 Dearle, A.
"Constructing Compilers in a Persistent Environment", proceedings of Persistent Object Systems: their design, implementation and use Workshop, Appin, August 1987, pp443-455, Universities of Glasgow and St Andrews PPRR-44-87.
.

35. Demers, A. & Donahue, J. Revised report on Russell.
Technical report TR79-389, (1979), Cornell University.

36 Guy, M. R.
"Persistent Store - Successor to Virtual Store" proceedings of Persistent Object System

37. Hurst, A.J.

"A Context Sensitive Addressing Model", Universities of Glasgow and St Andrews PPRR-27-87.

38. ICL
"The ICL/STC Standards Directory" Document reference 12918/001 from ICL at 60, Portman Rd, Reading.

39. Lehman, M. M.
"Programs, life-cycles and Laws of Software Evolution", Proc IEEE Vol 68 No 9
Sept'80.

40. Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117.

41. Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

42. McNally, D.J,
"Implementation in the STAPLE Project", University of St. Andrews STAPLE Report St.A/87

43. Matthews, D.
"An Overview of the Poly Programming Language", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 265-274.

44. Milner, R.
"A theory of type polymorphism in programming", JACM 26(4), 792-818.

45. Mitchell, J.C. & Plotkin, G.D.
"Abstract types have existential type", Proc POPL 1985.

46. Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

47. Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments"

48. Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, V

49. Morrison R., Dearle, A., Brown, A. & Atkinson M.P.

"An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 153-157.

50. Morrison, R, Brown, A, Connor, R and Dearle, A.
"Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Envir

51. Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J. &
"Polymorphic Persistent Processes", Universities of Glasgow and St Andrews PPRR-39-87.

52. Morrison, R.
"Napier Reference Manual" in preparation.

53 Morrison, R, Brown, A.L, Carrick, R, Connor, R and Dearle, A..
"Napier Abstract Machine Manual" in preparation

54. Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438.

55. Perry, N.
"Hope$^+$" Imperial College report IC/FER/LANG/2.5.1/7.

56 Philbrow, P.
"Associative Storage and Retrieval: Some Language Design Issues", proceedings of Persistent Object Systems: their design, implementation and use Workshop, Appin, August 1987, pp226-2

57. Philbrow, P & Atkinson M.P.
"Machine Independent print Facilities for Bitmapped Screens", in preparation.

58. PS-algol Abstract Machine Manual
Universities of Glasgow and St Andrews PPRR-11

59. PS-algol Reference Manual - 4th edition
Universities of Glasgow and St Andrews PPRR-12

60. Rosenberg, J. & Abramson, D.A.
"The Monads Architecture: Motivation and Implementation", Proc First Pan Pacific Conference, Melbourne 1985, 4/10-4/23.

61. Turner, D.
"SASL Language Reference Manual", University of St Andrews, CS79/3 (1979).

62. Wai, F.
"Distribution and Persistence", proceedings of Persistent Object Systems: their design, implem