Environments for Cooperative Systems Development

Ian Sommerville

Introduction

The predictions made in the early-1980s about the role of software engineering environments and their positive influence on the software process have not come true. Integrated project support environments are not widely used and their benefits have not been demonstrated. Several experimental IPSEs have been evaluated and abandoned by industry.

There are a number of reasons for this situation:

1. International and national funding concentrated on environment architecture and infrastructure rather than tools. It was expected that independent tool vendors would build tools for these architectures. By and large, they haven't.
2. The environments provided control facilities for the software development process which ought to decrease overall life cycle costs for developed software. However, current environments don't necessarily reduce development costs; sometimes they may increase them. Given current accounting practices, there is no market for systems with a 10-year payback.
3. Environment vendors made unrealistic pricing decisions. Most software buyers simply won't pay several times the price of a hardware platform for a software product.
4. Where tools were available in an environment, their performance was often appalling. A cynic might suggest that this was because the designers of the environment infrastructure didn't understand tool requirements.
5. Environments were sometimes not usable in conjunction with other software which worked outside the environment. The worse example of this was PCTE which, until recently, needed its own special Unix kernel.
6. Environments are (fairly) monolithic. Buyers had to buy everything at once. It wasn't possible to buy environment components as they were required.

I believe that a bottom-up approach to environment construction which starts with an environment architecture is doomed to commercial failure. I believe that environments will only be successful if they reduce software development costs and if they are structured so that they may be purchased bit-by-bit according to need.

There are two obvious ways to reduce development costs:

1. Support those software process activities which consume most development costs.
2. Support a software development strategy which dramatically reduces the amount of software which has to be created.

Cooperative software development

Software engineering is about teamwork. Teams of people formulate system requirements, design and implement the system and carry out quality checks on the finished software. It is not unusual for several hundred people, organised into smaller cooperating teams, to be involved in the development of a software product.

CASE tools and software development environments recognise that software development is a team activity by providing shared information repositories and configuration management systems. These allow project information to be accessible to all team members and ensure that engineers can

Prof. I. Sommerville, Computing Department, Lancaster University, LANCASTER LA1 4YR
Phone: 0524 65201; Fax: 0524 381707; E-mail: is@uk.ac.lancs.comp

work independently on the same software. Repositories and configuration management tools help *control* the software process. However, they don't provide *active support* for the processes of cooperation which are fundamental in software engineering.

Indeed, it can be argued that the current pre-occupation with process modelling actually *inhibits* cooperation as it ossifies the cooperation patterns in a system. Studies by sociologists have shown that cooperative processes are very subtle and complex. Many IT systems have failed simply because they didn't understand the social context in which they were installed. I predict that few organisations will devote sufficient resources to understanding their process before writing a process model and I am confident the resulting environment will be quite ineffective.

Of course, teamwork is part of many different activities and the potential for computer support for these explicitly cooperative activities has been recognised since the mid-1980's. A new interdisciplinary research area called 'Computer Support for Cooperative Working' (CSCW) has developed rapidly since the first conference on this topic in 1986. CSCW brings together workers from computer science, cognitive science and social science who are interested in how groups of workers cooperating can be provided with computer support. The first products in this area (sometimes called 'groupware') are now available.

Software engineering has always been a cooperative process so a reasonable question which might be asked is 'Why haven't tool vendors included support for cooperation in their products'. There are two obvious reasons for this:

1. The development of CASE tools has been technology-driven. The requirements for current CASE tools were, simplistically, to support existing manual methods. We don't have a good understanding of how software engineers cooperate in the specification and design processes.
2. Cooperative tools need powerful hardware and networks. It wasn't practical to develop these applications on the personal computers and workstations of the 1980s.

I also think there is a third reason. Software engineers feel uneasy about many developments in CSCW. Because of the problems of configuration management, they think that cooperation is a necessary evil and that the environment should control cooperation. For example, most configuration management systems ensure that a document can only be changed by one person at any one time. CSCW systems turn this round and allow, for example, many people to be working on the same document at the same time.

The costs of collaboration, in one form or another, are probably the most significant software project costs. Therefore, if environments are to reduce software development costs, they should explicitly support cooperative activities.

Software composition

The most common general model of system development is based on creating a set of program components and putting these together to form a system. The created components are often like existing components and, in many organisations, an informal reuse strategy is adopted where existing components are modified to create the new components. However, this isn't usually called software reuse!

An objective of many research projects has been to explore systematic software reuse where a development strategy is based on systematically discovering reusable components in some kind of component repository and then using these in the creation of new systems. Created components are then added to the library for further reuse.

For a variety of reasons, these efforts have not yet come to fruition and systematic reuse is not widely practised. I believe that reuse is the most effective way to reduce the amount of software which has to be written. However, we will not achieve this through imposing systematic methods on software developers; rather we must take these existing informal reuse practices and look at ways of supporting them.

Informal reuse has a number of characteristics:

1. It is domain specific. Development teams use domain knowledge to discover reusable components.

2. The entities which are reused are large-grain entities. Engineers don't waste time looking for individual procedures or objects. Rather, they start with whole sub-systems which are commonly used in an application domain.
3. It doesn't need complex cataloguing or information retrieval systems. There are only a small number of sub-systems which are widely used.
4. The sub-systems are invariably modified before being reused.

System composition is a method of software development which is based on supporting the informal reuse of large-grain components. In order to support these practices, we need:

1. For a given application domain, the appropriate sub-systems which are commonly used in that domain.
2. An architectural framework for applications so that they are composed of loosely rather than tightly coupled sub-systems.
3. Active support for system modification and change. This means specialist tools to help the engineer discover the effects of system change and tools to make these changes.

I believe that moving to a development strategy based on composition rather than creation is currently practical and likely to be cost-effective. This therefore suggests that SDE's should provide composition support facilities to reduce development costs.

Cooperative systems engineering

Work at Lancaster University is currently underway to explore the facilities required in a distributed, domain-specific SDE which can support cooperation and system composition. The application domain which we are currently concerned with is CSCW systems i.e. we want to support the construction of systems which explicitly support cooperation. In the usual traditions of computer science therefore, we ought to be able to bootstrap this environment using itself!

We are in the early stages of developing a Cooperative Systems Integration Environment (COSIE) which is predicated on the notion that developers will build cooperative systems by selecting components which are themselves used in COSIE. The system is bootstrappable because, as COSIE evolves, more and more sub-systems become available so the range of systems and environment tools which can be built using COSIE increases.

The model of use which we envisage is as follows:

1. From a set of system requirements, engineers identify the sub-systems which are required in the final application.
2. Appropriate sub-systems are abstracted from COSIE and modified to meet the application requirements.
3. Application specific sub-systems which may be required are written.
4. The sub-systems are assembled into an application system.

Obviously this is not a sequential process. Rather, we would hope that the mode of development is based on evolutionary prototyping where a version of the application is produced quickly using un-modified sub-systems and the modifications required identified after user experiments. Indeed, we envisage that some modifications may actually be carried out by end-users after the system has been delivered.

Currently, COSIE exists but has only user interface components (X-window libraries and a generic user interface model) and associated modification support (a library browser and a user interface generator). However, it is already useful and, over the next year, we plan to add significantly to the available sub-systems.

COSIE components

The sub-systems which will be incorporated in COSIE fall into a number of different classes:

1. *Object classes* These are definitions of entities with state and associated operations. Examples of object classes are DOCUMENT (plus many associated sub-classes such as FORM, etc.), MENU, DIRECTORY, RESPONSIBILITY-MODEL, etc.

2. *Libraries* These are collections of procedures which are used in a particular part of an application or in an application domain. Examples are MOTIF library for constructing user interface applications, or X-window sharing libraries for cooperative systems which support window management and interaction across several distinct workstations.
3. *Servers* These are stand-alone sub-systems which provide a range of services to other sub-systems. Examples are a configuration manager or a hypertext engine.
4. *Tools* These are the obvious facilities such as word processors, spreadsheets, compilers, etc.

A characteristic of COSIE is that ALL reusable sub-systems have an associated change manager which is used to modify that system for use in other applications. The different classes of component, in general, need different types of support for change.

1. *Objects* Change is mostly by specialization. Change management requires tools to browse and manipulate the inheritance hierarchies.
2. *Libraries* Changes are accomplished by setting parameter values or by adding new procedures to the library. Replacement of existing procedures is possible but unwise. Change management requires support for impact analysis as, typically, libraries have a significant number of inter-dependencies. Program understanding tools help the user predict the effect of parameter changes. System generators may relieve users of detailed library knowledge.
3. *Servers* Servers are principally modified by source code changes. The principal requirement is for good browsing and editing tools.
4. *Tools* Tools are only changed by parameterisation. We assume that source code is not available. Change management support essentially means online documentation.

Component composition

We have only just started exploring appropriate composition mechanisms for cooperative systems. Possible approaches which can be adopted include:

1. *Configuration specification* A specification language may be used to define the configuration of a system. This has the advantage that hardware and software may be specified at the same time.
2. *User interface driven composition* Using this approach, the systems engineer starts by defining the user interface then incrementally adds to the system by defining how user interface events should be handled.
3. *Composition around a standard kernel* Using this approach, we assume that cooperative systems are built around a kernel whose principal responsibility is event-handling and event distribution. Composition might involve defining kernel events and their associated handling and distribution mechanisms.