CHAPTER 10: EMERGING TRENDS IN OOP

The major goals of object-orientation are to produce *well-structured software* and to build more *extensible and reusable systems*. It seems that the second goal is more important because it is more difficult to achieve and there remains more work to be done. We believe that the following areas of OO will most prominently contribute to this goal in the future:

Frameworks

Frameworks have been recognized as one of the most promising OO technologies because they provide *large scale* reuse. The reuse of individual classes, while helpful, will not bring significant productivity leaps. Only the reuse of taylorable systems *as a whole* will lead to noticeable results. Although frameworks are widely used today, there remain many open problems to be tackled, for example:

- **How can a framework be designed systematically?**
  Current frameworks are developed in a rather ad-hoc manner by comparing existing systems and extracting their common features. This process is iterative and good frameworks are only obtained after many iterations.
  Why is framework design not more systematic? One reason is that current design methods hardly apply to frameworks. Their aim is to model *one specific* problem but not a set of *various similar* problems. Traditionally, the main question is "what are the objects in a given situation?" wheras for frameworks the main question should be "what is to be kept flexible so that it fits several situations?". We need special methods and guidelines for designing frameworks. Such methods could be based on design patterns that help putting together a framework from templates of proven OO knowledge.

- **How can a framework be documented so that it is easy to use?**
  The documentation of a framework has to be different from the documentation of traditional software because it must not only describe what the framework does but also what the user-provided parts are expected to do. It must show the "hot spots", i.e., the places where custom functionality can be hooked in. Providing the source code of the framework is a common solution but not a good one. Besides finding better graphical and/or formal notations, a promising approach seems to be the employment of more sophisticated development environments that use hypertext, visualization and other online assistance to understand and extend a framework.

- **What makes a framework good or bad?**
  Is it the number of hot spots, the reuse factor (i.e., the ratio between the framework code and the user code), or the simplicity with which a framework can be adapted? There are hardly any guidelines, let alone metrics, to make an objective judgement and thus to drive the framework development process. The work on frameworks has spawned off the idea of *componentware*. Instead of delivering a system as a prepackaged monolith containing any conceivable feature, modern systems consist of a light-weight kernel to which new features can be added (often dynamically) in the form of black box components. This helps keeping systems small and frees

users from carrying along unnecessary functionality. Pluggable components will also become an interesting issue for small software vendors and for creative programmers who want to distribute free software via the Net.

Design Patterns

OO software development is still in a state comparable to traditional programming in the sixties. There are sufficiently powerful languages but programs have to be assembled from low-level building blocks: in traditional languages these building blocks are integers, arrays, pointers; in OO languages they are objects.

In traditional programming, the invention of data structures such as lists, trees or queues solved common problems and abstracted from the building blocks involved. Similarly, design patterns can be viewed as *"object structures"* that capture common OO solutions and abstract from the underlying building blocks, namely the objects. In the same way as the traditional pattern of a "binary tree" is assembled from nodes and pointers, the object-oriented pattern of a "decorator", say, is assembled from objects of certain classes. Design patterns are data structures (and sometimes also the algorithms) in object-oriented software development. Research challenges of the future will be:

- Finding *new patterns* and unifying existing ones.
- Finding *better notations* to describe patterns, their structure, their dynamic behavior, and their integration into larger systems. Beside graphical notations, formal pattern languages might be a promising solution.
- *Integrating patterns with programming languages*. If a programming language could express and check the correct use of design patterns, programming could be raised to a higher level and source code could become better understandable.

Distributed Objects

Distributed objects that operate in a concurrent and active way have been a research topic in the OO community for a long time because the metaphor of communicating objects lends itself to distribution well. With the success of the Internet and appropriate programming languages such as Java this topic has gained additional relevance. How can intelligent and active objects improve the use of the Net? How can security problems be solved that result from such a distribution? How can objects in heterogeneous environments (different computers, languages, and operating systems) cooperate effectively?

Languages and Environments

If one looks through the proceedings of the recent OOPSLA and ECOOP conferences one will notice that languages and their implementations are still one of the most intensive research areas. Although I don't expect really spectacular innovations here in the near future I believe that there is still room for improvement in most widely used OO languages of today. These improvements do not have so much to do with new language features but with the more fundamental concepts of safety and dynamic extensibility.

- **Safety.**
  Object-orientation is intended for building large and complex systems. The larger a system the more important is safety and memory integrity. Any violation of semantic contracts must be detected, the earlier the better. This requires that contracts can be expressed in a language and that the compiler checks them. Memory integrity means that programming errors must not have disastrous effects on the data. For example, a garbage collector eliminates the frequent error of bad memory deallocation. Yet many languages do not rely on such mechanisms.

- **Dynamic extensibility.**
  Extensibility is one of the virtues of object-orientation. Yet many systems are restricted by static extensibility. They can only be extended by interrupting their use, relinking and reloading them. Relinking requires that the individual object files are available, which is usually not the case in commercial systems. In contrast, dynamic languages such as Smalltalk and Self allow extending a system at run time without disrupting its use. This should also become possible in compiled languages and systems (It is already the case in Java or Oberon, for example.)

Finally, an important challenge will be to bring all these techniques to the mainstream software industry. While many software products claim to be object-oriented, they often just use an OO language or at best classes as a structuring aid. The idea of (custom-) extensible systems is not yet common in practice.