BCT 2308
SOFTWARE DEVELOPMENT TOOLS AND ENVIRONMENTS

CHAPTER 5

Ideal Scenario For the Future

Introduction

- Software engineers have succeeded in automating a wide range of engineering processes but have not been as successful in automating their own processes.

Introduction

- The following are the main features of an ideal scenario for CASE.
  - Phases of development.
  - formality.
  - Incrementality.
  - Infrastructure.
  - Modularity.
  - Methods.
  - User Friendly.

Phases of development

- All phases of the software process should be computer assisted, in an integrated way.
- This means that all the tools for managing, specifying, programming, verifying, etc, software should belong to the same environment, and their integration should be supported through an appropriate infrastructure.

Phases of development

- The software engineer should interact with the facility through an automated assistant, which not only should carry out all clerical activities involved in the process, but also should have enough knowledge of the process and deductive power to provide more intelligent support.

Phases of development

- An example of state of the art integration is Emacs, make and Compilers.

Formality

- This should be supported (not imposed) at every level of the software production process.
- Also the appropriate level of formality should be decided by the user of the CASE environment (the software designer).

Incrementality

- This should be supported in all phases of the software production process (planning, requirement analysis and specification, design, testing, e.t.c).
- Formality should also be achieved incrementally.

Incrementality

- This requirement has a strong impact on the tools of the environment.
- Tools should be able to accept partially specified objects and on-line modifications, incrementally checking their consistency and tolerating the remaining incompleteness.

Infrastructure

- The infrastructure should allow the environment to be open ended i.e. the environment should not be a fixed collection of tools, but rather, it should be easy to enrich the environment progressively by incorporating new tools into it over time.

Infrastructure

- An example, integrating tools that measure program complexity with cost estimation and planning would allow automatic tuning of parameters based on the measures.

Infrastructure

- An example open endedness is the Emacs and UNIX.
- By providing facilities that make it easy to aggregate new tools (e.g. pipes and redirection), UNIX has proved that the programming environment can be progressively enriched by software engineers.

Infrastructure

- However, UNIX also shows the shortcomings of not basing open-endedness on a centralized infrastructure i.e. tools cooperate directly, by accessing their low level output data, with not notion of a central repository that keeps the data that are exchanged among tools.
- Thus tools interact with each other at a very low level.

Modularization

- This should be supported as a major mechanism for structuring complex objects, not only in the design phase, but throughout the production process.
- Emphasis should be that modularity be exhibited in designs, specifications, verification and so on.

Modularization

- Also, the principle of *separation of concerns* should be supported by the environment by allowing the engineer to focus attention on a single issue at a time whenever possible.

Modularity

- E.g. modern word processors provide different views of the same document according to the particular topic of interest.
- Thus, a developer can just see the outline of the document, or a single section, or the first lines of any paragraph, etc.

Modularity

- An example is the capability of exploding a bubble of a DFD into a lower level DFD.
- This capability should be viewed not only as a feature to support stepwise design, but more importantly, also as a structured way of reviewing the specification.

Modularity

- In the review process, one may separate the concerns pertaining to the detailed view of each individual function.

Methods

- The environments should be based on sound methods.
- Tools themselves are of little use if the user does not have a sound understanding of the underlying methods.

Methods

- Thus attention should be paid to associated educational and training material before any tool is introduced.
- Managers should take this seriously.

Methods

- The introduction of a software development environment should be planned and monitored carefully, taking into account the time and money that are needed to educate people on the underlying methods of the environment and to train people on the proper use of the tools.

Methods

- Too often, managers expect a dramatic increase in productivity and other software qualities just from the purchase of a new tool.

User Friendly

- The environment should be user friendly and provide a natural human-computer interaction.
- More precisely, the environment should fit the user's needs and expectations, as well as the characteristics of the application.

User friendly

- Both of these requirements are rather subjective and both apply to the whole life cycle, but are more relevant to the earlier phases.

User Friendly

- The user-interface management systems play a major role in the achievement of these goals, especially user friendliness.
- The issue of naturalness with respect to the application raises the alternative of having several specialized environments for different classes of applications, as against having a single general-purpose environment.
- Both alternatives exist in the present.

End of Chapter 5