O
O S E
E

Program Families

Programmer's Practice

> We were behind schedule and wanted to deliver an early release with only a proper **subset of intended capabilities**, but found that that subset would not work until everything worked.[5]

> We wanted to add **simple capability**, but to do so would have meant rewriting all or most of the current code.[5]

> We wanted to simplify and speed up the system by removing the **unneeded capability**, but to take advantage of this simplification we should have had to rewrite major sections of the code.[5]

Software as Product Family

- understanding a program as a single product . . .

$$\text{is} \left\{ \begin{array}{c} \text{good} \\ \text{bad} \end{array} \right\} \text{practice from the perspective of} \left\{ \begin{array}{c} \text{marketing} \\ \text{development} \end{array} \right\}$$

- product-line development helps to let do even marketing a better job
  - aim at looking for already existing solutions
  - provide an infrastructure of reusable (software) components
  - don't re-invent wheels, waste man power, and extent the time to market
- single-product thinking goes back to the "Stone Age" of programming

Product-Line Development

- what was (is) true for "hardware" over a long period is also true for software
  - i.e., hardware such as e.g. car, watch, house, camera, handy and so on
  - i.e., hardware in the sense of an industrial good
- software-product development is an evolutionary process
  - a software product may exist for years, or even decades, at the market
  - typical examples are operating systems: UNIX, e.g., exists since 1970 [1]
- *software manufacturing* (e.g. operating systems) is an engineering discipline

[1] But this doesn't mean that UNIX has been a product-line development from the very beginning. Refer to [6].

Product Line $\Longrightarrow$ Program Family

- a product line is made of a number of individual products:
  - the Volvo 40, 60, 70, and 80, with its V and S models and variations

    ⋮
  - the Leica-M system with its M3, M2, M1, M4, M5, and M6 cameras
- all products of the same line share the same (sub-) set of properties:
  - SIPS and WHIPS is available for the V/S 40, 60, 70, and 80 models, resp.

    ⋮
  - M-lenses are upward compatible from the M3 to the M6 (i.e. 1954 – today)
- a program family is a **software-product line** with programs being the products

Program Family $\Longrightarrow$ Function Reuse

> We consider a set of programs to be a program family if they have so much in common that it pays to study their **common aspects** before looking at the aspects that differentiate them. [5]

> We want to **exploit** the **commonalities**, **share code**, and **reduce maintenance costs**. [5]

Function Reuse $\Longrightarrow$ Separation of Concerns [2]

- for a function to be reusable, it needs to be "componentized"
  - concentrate on the function's essentials
  - hide the function's implementation details as far as possible
  - design a complete and, yet, easy to employ function interface
  - provide a full specification and extensive documentation of the design
- differentiate functional requirements from non-functional requirements
  - free a function's implementation from any non-functional property
  - particularly, don't hard-code assumptions about the function's use pattern
- remind the experiences made over time and reflect the lessons learned

What you demand is what you get — WYDIWYG

Some users may require only a subset of the services of features that other users need. These **"less demanding" users** may demand that they **are not be forced to pay for the resources consumed by the unneeded features**. [5]

Family-Oriented Design

minimal subset of system functions

- captures common functions that are useful to build specialized systems
- provides mechanisms in the form of fundamental building blocks
- is made of a distinct (i.e., small) number of reusable assets

minimal system extensions

- reuse and specialize or customize the fundamental system functions
- encapsulate the strategic and application-specific design decisions
- are stepwise developed bottom-up, directed in a top-down manner

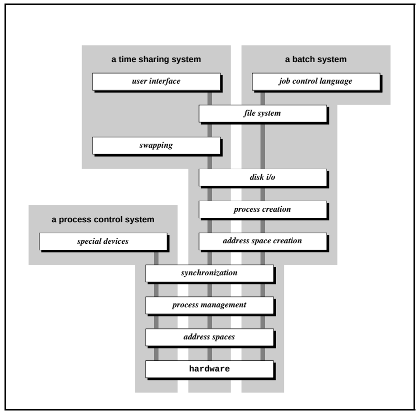
Incremental System Design

- program families tend to exhibit a distinguished, deep hierarchical structure
  - the design process is somewhat challenging and proceeds in “tiny” steps

  $\Rightarrow$ “*decisions which restrict the family are postponed as far as possible.*” [3]

  - the result is a multi-level hierarchy of a large number of thin abstractions
- proceeding iterative and bottom-up, from generalization to specialization

Example of a Family of Operating Systems — FAMOS [3]



- the sample shows three family members:
  - a process-control system
  - a time-sharing system
  - a batch system
- . . . exploiting a set of commonalities:
  - synchronization
  - process management
  - address space
- . . . sharing the same design decisions

Software Framework

Rather than write programs that perform the transformation from input to output data, we design **software machine extensions** that will be useful in writing many such programs. [5]

Summary

- a major concern is what ideas to exclude from the design
  [Liskov, 1981]

- keep things as simple as possible
  [Lampson, 1983]

- the challenge lies in doing it 'right', and 'right' often means staying simple
  [Svobodova, 1985]

- simple systems work surprisingly well
  [Birrell, 1986]

- simple things nearly always work, and simple thinks are extensible
  [Needham, 1986]

Bibliography

[1] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999. ISBN 0-201-82467-1.

[2] E. W. Dijkstra. *A Principle of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[3] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[4] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-5(2):1–9, 1976.

[5] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.

[6] W. Schröder-Preikschat. UNIX System Programming, 1999. Lecture Notes.