Describing Architectural Design

Communicating the “big picture”
Block diagrams & boxologies
Overall styles

Objectives

• “Orientation” documentation
  - What are the organizing principles for this system
  - What are the major pieces and their interfaces
  - What here are the parts making up those major pieces
Example: Classic Compiler

- Very gross level: many missing details
- Main information is in absent connections
  - e.g., the parser does not interact directly with the code generator

A more realistic diagram of GCC
(not entirely accurate)
Orientation to GCC ...

- Front/back interface is (only)
  - construction of register-transfer-language tree
  - invoking code generator after each procedure
- Code generation for each machine is controlled by table (machdef.h)
- Should say where to look to answer questions:
  - How would I build a native code Java compiler?
  - How would I compile C to Java byte codes?

UNIX layer architecture

from C. Schimmel, UNIX Systems for Modern Architectures (Addison-Wesley 1994)

- What does this diagram tell us about the division of Unix into Kernel & Commands?
Interpreting Block Diagrams

- layers diagram indicates permitted and prohibited interfaces or dependencies (the “uses” relation)
- block diagram shows interfaces
  - but typically not direction of dependence
  - and is often over-simplified (where is symbol table?)

UML Dependencies (of packages)

Figure 31. Dependencies among packages
Boxologies

• The “boxologies” usually have
  - A set of notations for various stages of design and points of view
    (e.g., class hierarchy vs. dynamic architecture vs. static architecture)
  - A corresponding methodology for creating design
• Advantage: Standardization
• Current dominant notation: UML

UML Class Diagram
Describing Interfaces

• Overall style
  - Example: Parser calls lexer to obtain each token
  - Example: Each kernel service is invoked by an SVC, which triggers a context switch

• Precise interfaces
  - int yylex() returns integer code as defined in tokens.h. 0 is always the end-of-input code.
Documenting Interfaces

• javadoc (when used well) is a good example of doing this right
  - with liberal use of header comments

• Well-commented code may be enough
  - but think carefully about navigation
  - comment “extractors” are easy to write

• Diagrams? Maybe
  - but I haven’t yet seen readable detailed interface documentation in diagrammatic form

Navigating from Overview to Code

• “Links” can be hypertext or descriptions
  - but in any case, I should be able to answer: Where do I find the files that make up that module?

• Subdirectories can help
  - although it may be too late if you aren’t already using them
The Bottom Line

• Purpose of internal documentation: Efficiently answer questions
  - First: Where do I need to look?
  - Then: How do I make this change?
• The particular notation or packaging matters less than well-organized content
• It’s a lot easier to document a clean design than a brick