# Design Documentation

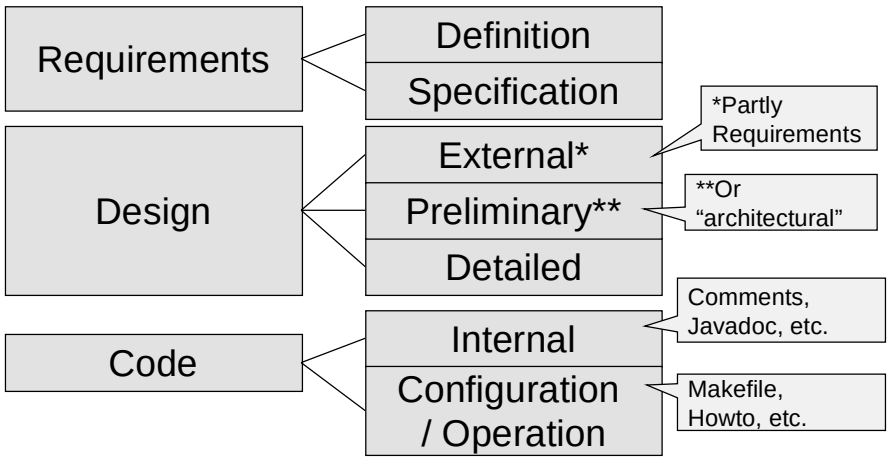
Due next week



# Purposes

- Capture (and demonstrate) the state of an evolving system
  - "Milestone" document
- Freeze decisions
  - "Contracts" among developers, and between developers and clients
- Orient developers to the system
  - Including "maintainers" as developers over the longer term

# Typical Milestone Documents






# Design Doc Requirements

- Essentially an extension of the concept document, with
  - Architectural design, sufficient to
    - Trace from visible features to implementing code
      - and, by extension, ensure completeness
    - Judge feasibility of potential changes
    - Determine potential subsets and build order
  - Module interface specifications
    - Sufficient as "build-to" specifications
    - Optionally augmented with implementation notes

# Architectural Design

- Must include
  - Clear rules for where functionality goes
    - As explicit allocation, AND/OR
    - Implicit but evident in module descriptions
  - Interface specifications
    - Sufficient as "build to" documentation
      - Implies: Sufficient for unit or subsystem tests
    - Sufficient as "build against" documentation
      - Everything I need to build another module



# Architecture: In what detail?

- How far must the system architecture be broken down with precise specifications?
  - At least to main responsibility interfaces
    - Where Mary's part meets Joe's part
  - At least to 1 person-week chunks
    - Sufficient for incremental development
  - At least to good information-hiding modules
    - Sufficient as maintenance firewalls

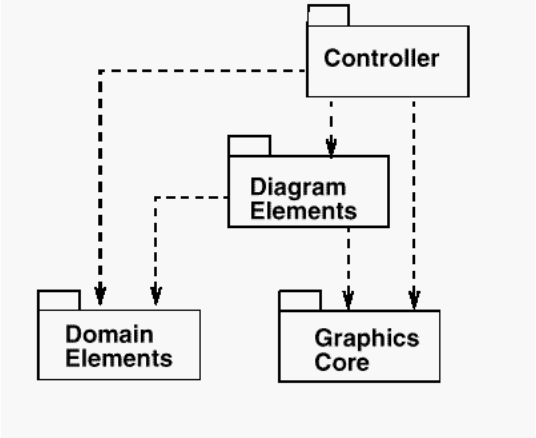
# Architecture: What diagrams?

- I (still) won't impose a single diagram, but I will suggest
  - Dependency diagrams: Layers, uses, etc.
  - Information flow diagrams
  - Information structure diagrams
  - Class hierarchies
- Choose according to appropriateness
  - It is very unlikely that all of these will be useful for any single project



# UML Dependencies (of packages)

Figure 31. Dependencies among packages



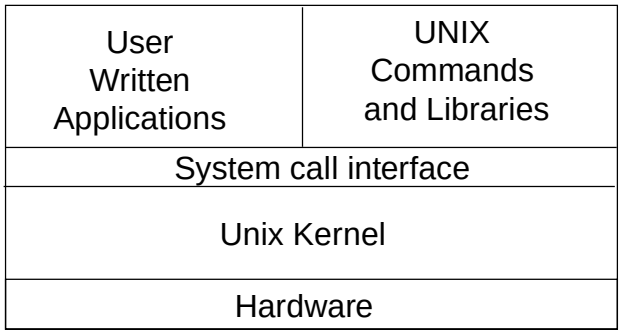
# What is dependence?

- A uses B: requires the presence of
  - Should be acyclic (why?)
  - May differ from visibility and calls relations (how?)
  - May require a breakdown finer than components (why?)



# UNIX layer architecture

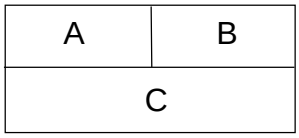
from C. Schimmel, *UNIX Systems for Modern Architectures* (Addison-Wesley 1994)



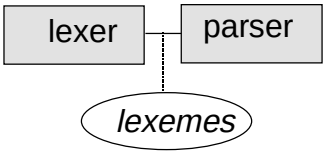

- What does this diagram tell us about the division of Unix into Kernel & Commands?
- How does it differ from “uses”?

# Interpreting Block Diagrams

layers diagram




block diagram




- 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?)

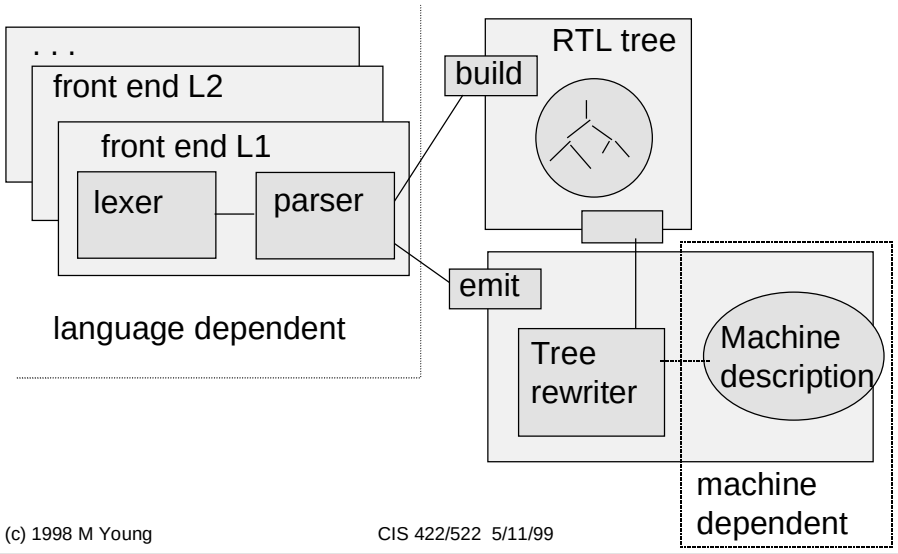


# Information Flow Diagrams

- Most useful when
  - Flow is non-trivial, but not too complex
  - Major functionality is associated with stages in flow
- Examples
  - Information systems
  - Control systems
  - Classic compiler architecture

# Information Flow in GCC

(not entirely accurate)



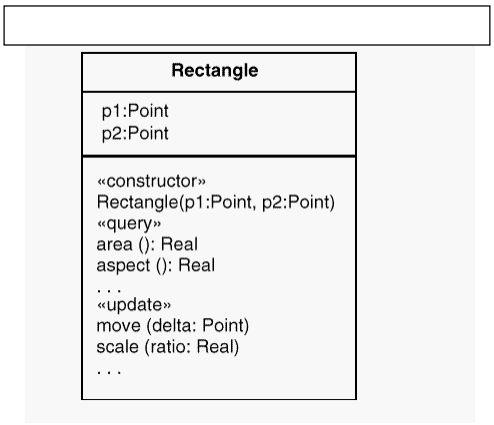

(c) 1998 M Young CIS 422/522 5/11/99 13

# Information Modeling

- For any system with structurally rich data
  - Business systems, but also CAD/CAM, $C^3I$, …
- Part requirements, part design
  - Understanding existing and required information
  - Bringing order and elegance to chaos
- Many alternatives
  - Relational, ER, class/inheritance, …
    - All models emphasize some aspects and discard others

(c) 1998 M Young CIS 422/522 5/11/99 14

# UML Class Diagram

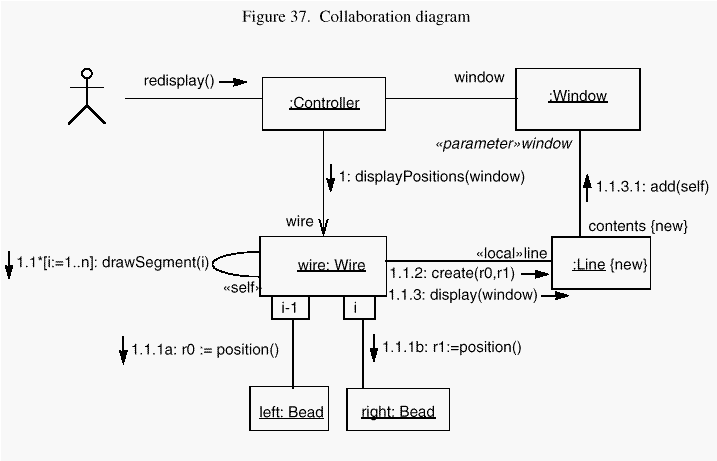





# Scenario-Oriented Diagrams

- Illustrate particular behaviors
  - Scenarios are examples; never complete
- Often used for
  - Cataloguing functionality before module breakdown
  - Validating completeness
  - Augmenting (but not replacing) module breakdown

# UML Collaboration Diagram






# Module Interface Specifications

- Suppose you have
  - .h files for C
  - Public part of Class definitions for C++, Java
- What more is needed?
- Is javadoc enough?

# 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: What do I need to do?
- The particular notation or packaging matters less than well-organized content
- It's a lot easier to document a clean design than a brick