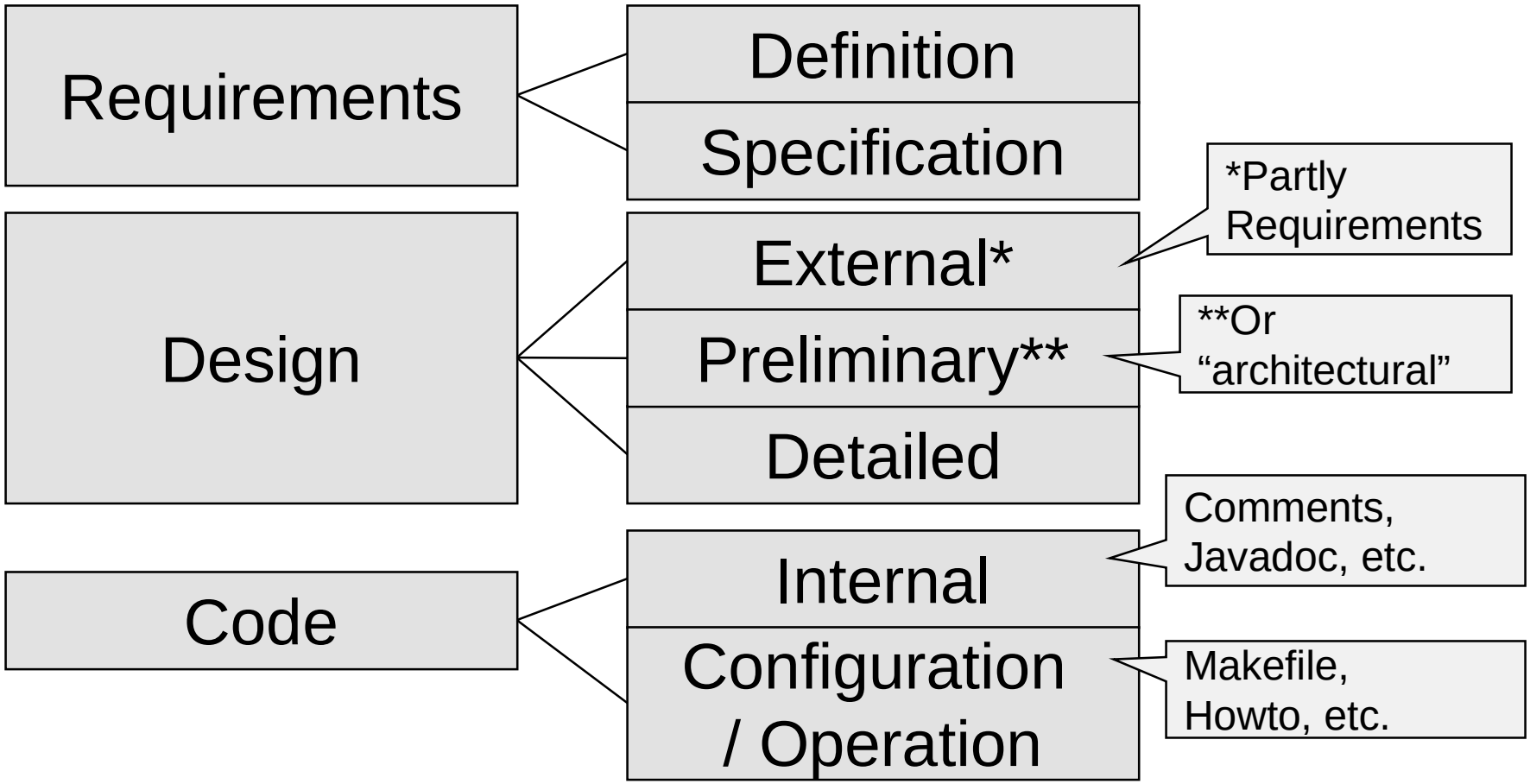
# Documentation for Developers

(As distinguished from
documentation for end-users)

# Purposes

- Capture (and demonstrate) the state of an evolving system
  - "Milestone" documents for each stage
- 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

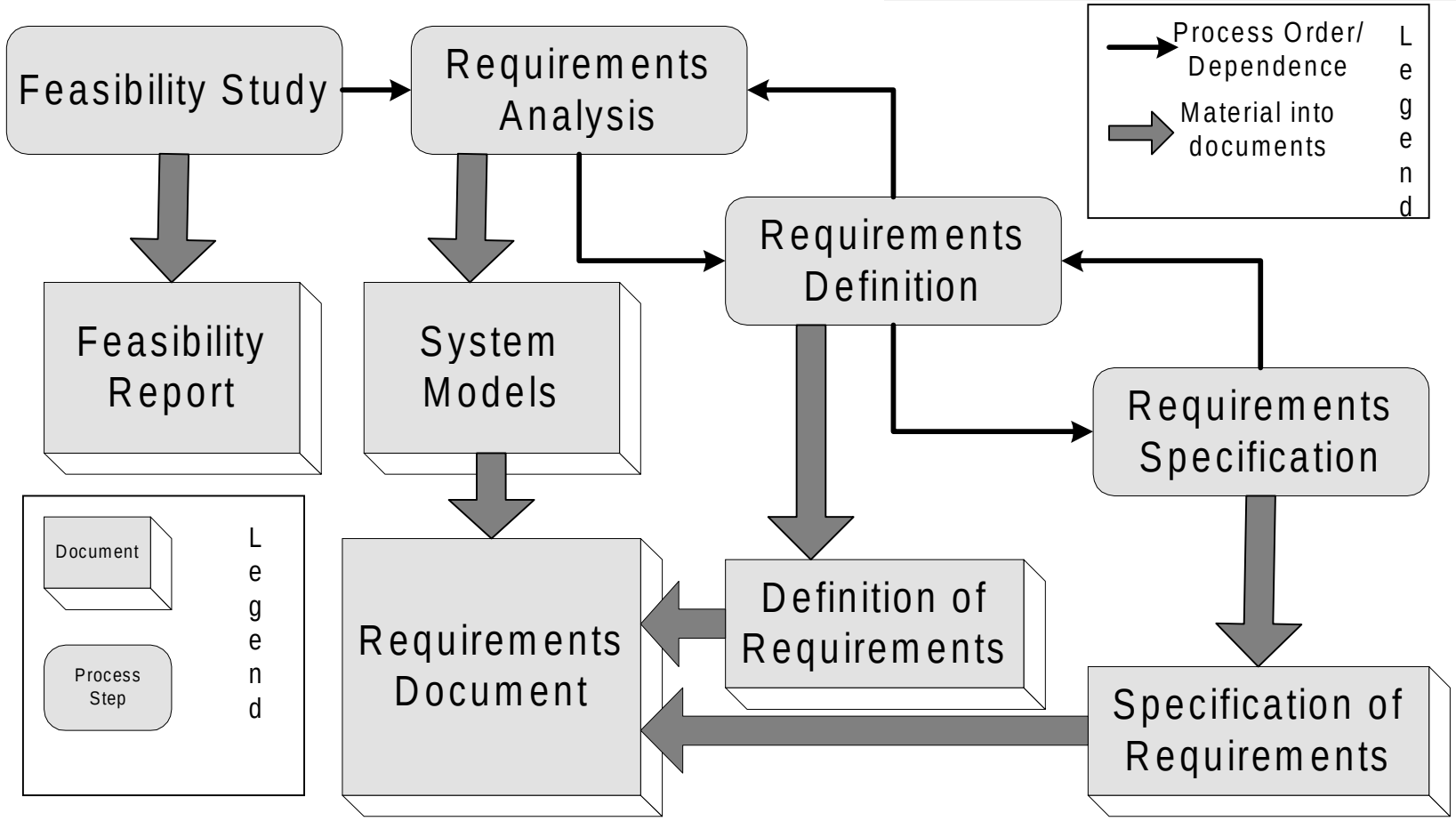


# Requirements

- Definition vs. Specification
  - Definition: What the client wants or needs
  - Specification: What the developer promises
- Requirements definition (elicitation)
  - In the problem space, in the users language
  - Should avoid design
- Requirements specification
  - In the solution space, necessarily involves design

# Requirements Process

Diagram from Sommerville, pg. 67



# System Models: Context

- Context Model
  - Explores question: What is the environment of this system
  - OR: What system is this system a component of
- Particularly useful for information processing
  - ex., how does class registration fit with all the other human and automated information systems?

# 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

# The SRS Document

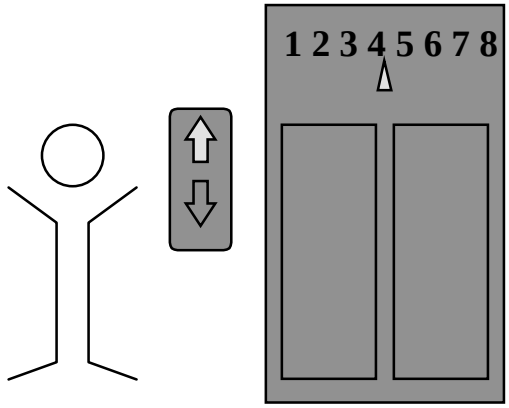
- Describes both requirements and specification
  - Maybe together, maybe separately
- Includes
  - Problem statement: Why this system?
  - Rationale (for specification choices)
  - Likelihood of change
  - Precise Specifications (next slide)

# Specifications

- Discriminate between acceptable and unacceptable systems
  - Should say *just enough*; should not over-specify
- Should include (among other things)
  - Negative specifications: What must not happen
    - a.k.a. safety specifications
  - Desirable responses to undesirable events
    - robustness
  - Glossary of terms

# Narrowing for Checkability

•**Example:** Elevator response




*Uncheckable requirements can often be **narrowed** to checkable properties (often sufficient but not necessary conditions)*

- **Objective:** Passengers are not frustrated by waiting
- **Specified property:** Elevator responds within 60 seconds, 99% of the time
- **Excluded solution:** Install a mirror in the waiting area

# External Design

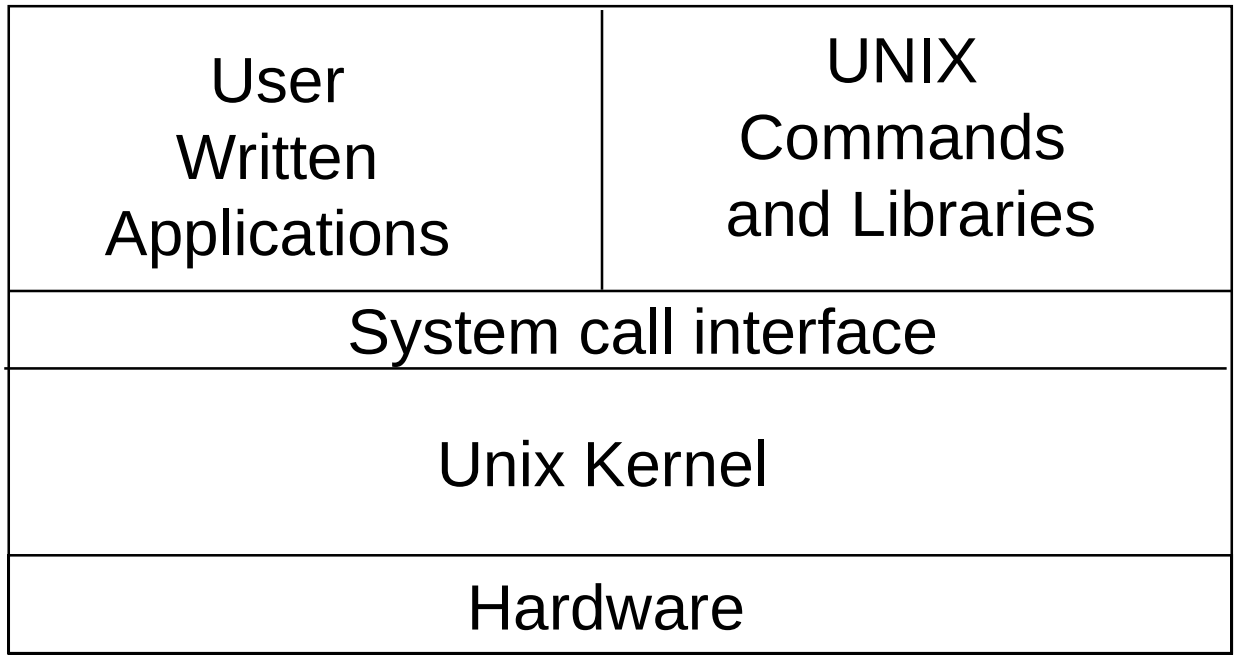
- Partly requirements, partly design
  - May bind some parts early and some late
- Can be specified in user documentation
  - User manuals can be written before code
    - Avoids redundant external design
    - Accelerates document development
    - Moves responsibility for detailed external design decisions
      - Probably must be shared among writer, usability expert (if available), and designers/coders

# Design Documentation

- Objectives
  - Orientation
  - Specification (serving as contract and record)
  - Prompting (thought tool)
- Dozens of notations and methods to choose from
  - Like any other model: Emphasize and discard
  - Important to be well-defined, whether standard or ad hoc

# UNIX layer architecture

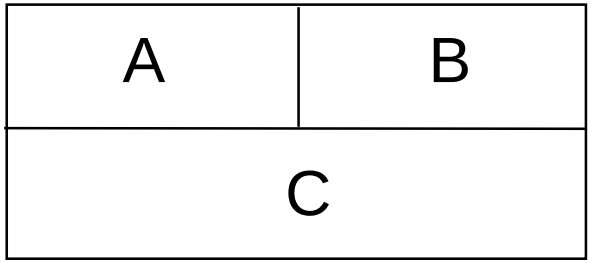
from C. Schimmel, *UNIX Systems for Modern Architectures* *(Addison-Wesley 1994)*



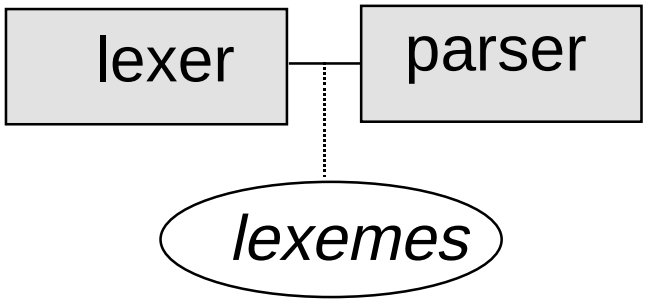

- What does this diagram tell us about the division of Unix into Kernel & Commands?

# Interpreting Block Diagrams

layers diagram

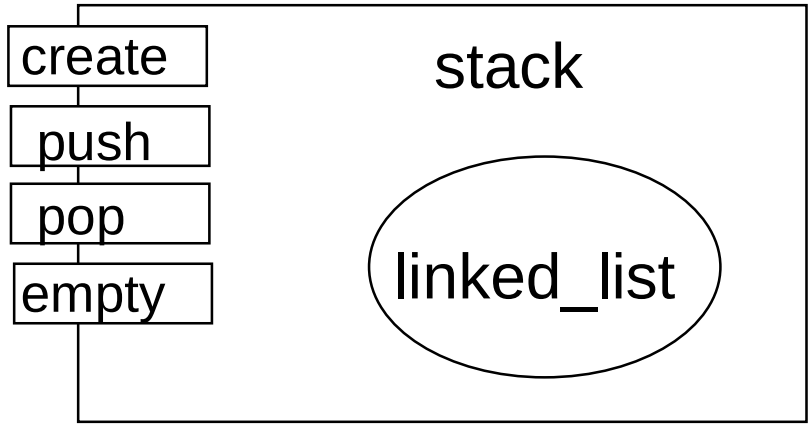



block diagram




- 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?)

# Boxologies

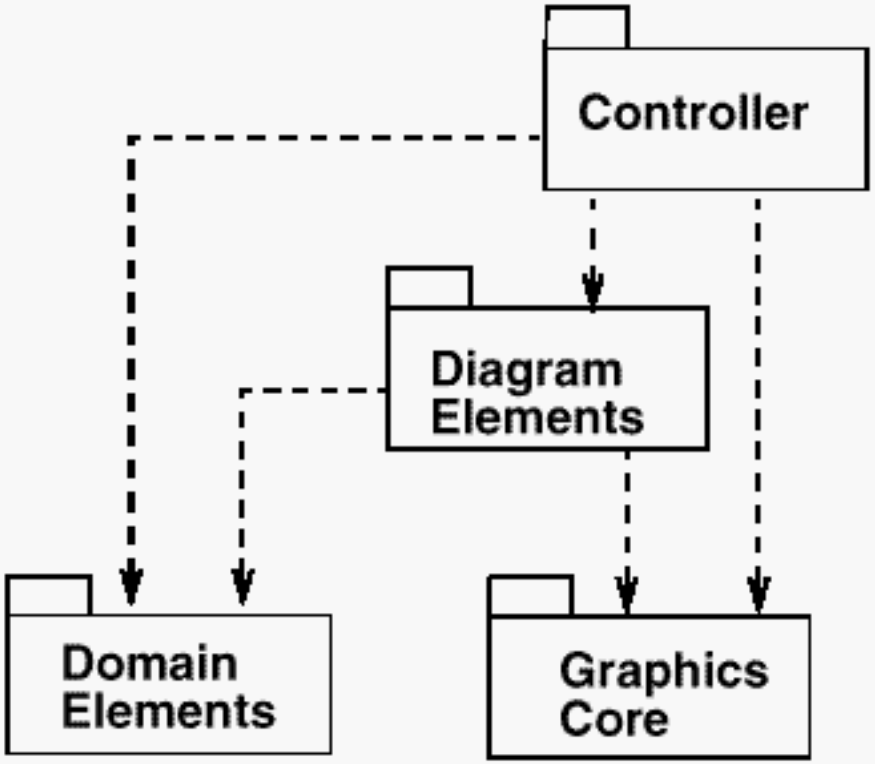



*A design notation*
*for object-based design*
*circa 1985*

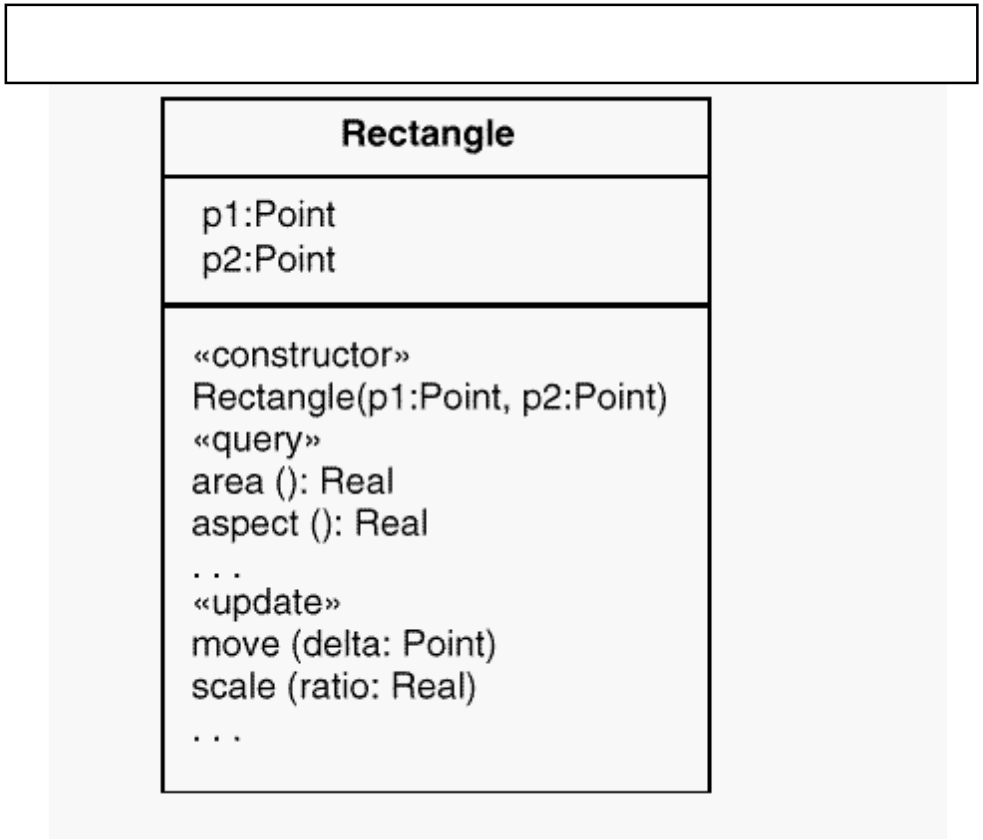
- The "boxologies" usually have
  - A set of notations for various stages of design and points of view (e.g., class hierarchy vs. dynamic architecture vs. static architecture)
  - A corresponding methodology for creating design
- Advantage: Standardization
- Current dominant notation: UML

# UML Dependencies (of packages)

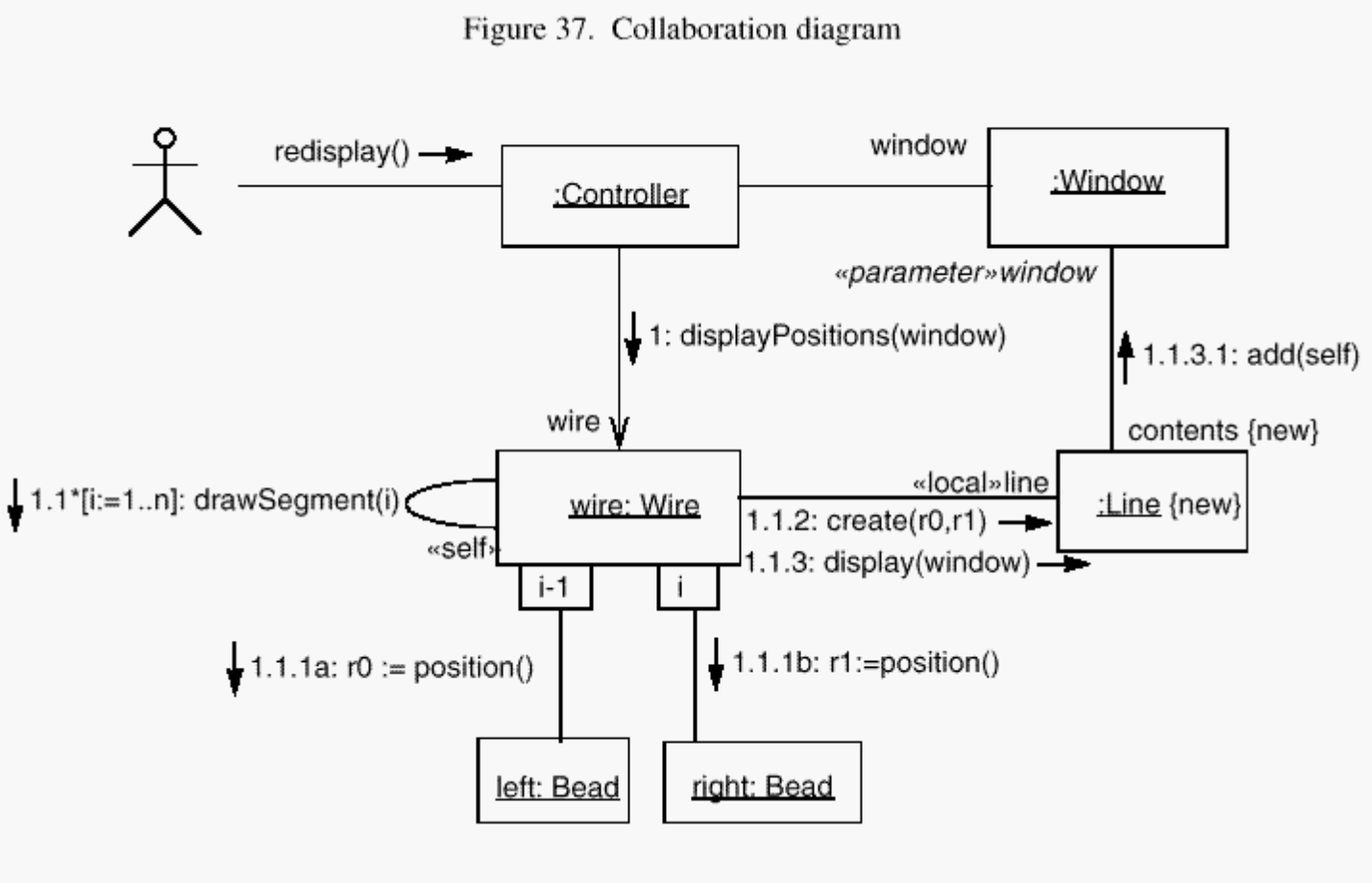
Figure 31. Dependencies among packages



# UML Class Diagram



# UML Collaboration Diagram



# Boxology Assessment

- Boxologies have been good for
  - Standardizing communication
  - Making syntactic distinctions (e.g., arrow types)
- Some problems and limitations
  - Precision only at low (language) level
  - Incomplete semantic definition
    - Lots of room for ambiguity
- An alternative (sometimes): Domain-specific notations / languages