Packages

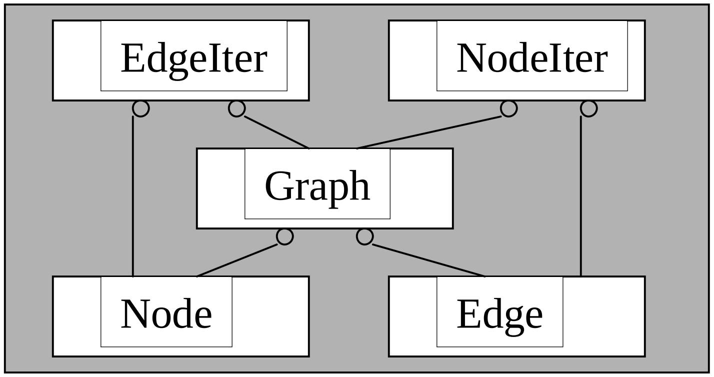
(extracted from “Large-scale C++ software design” by John Lakos)

J.Harvey

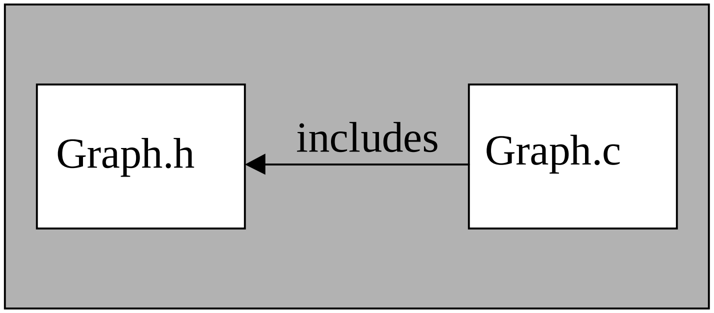
26 October 1998

Logical vs Physical Design

- Logical design addresses architectural issues; physical design addresses organisational issues
- Physical design takes account of physical things such as compile-time coupling, link-time dependency, executable size



Logical View



Physical view

What are components?

- A component is smallest unit of physical design. It allows for consideration of physical issues not addressed by class level design.
- It is an indivisible physical unit, none of whose parts can be used independently of the others.
- It consists of exactly one header file (.h) and one implementation file (.c).
- It defines one or more closely related classes and free operators deemed appropriate for abstraction it supports.
- The logical interface of a component is the set of types and functionality defined in the header file that are programmatically accessible by clients of that component.
- The physical interface of a component is everything in its header file.
- A component *y* DependsOn a component *x* if *x* is needed in order to compile or link *y*

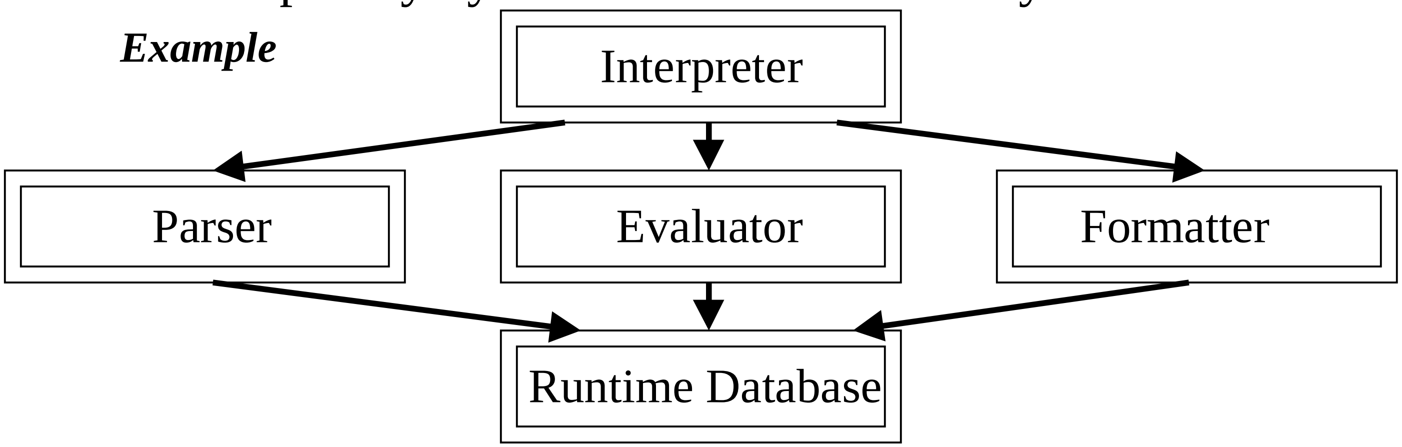
Why packages?

- Focuses on physical structure of system
- Reflects on :
  - logical structure of application
  - organisational structure of development team
- Large systems require hierarchical physical organisation beyond hierarchy of individual components
- Need a macro unit of *physical design* referred to as a *package*
- A package is a collection of related components in a logically cohesive physical unit.
- It has an associated registered prefix that identifies both files and file-scope logical constructs as belonging to package.

From Components to Packages

- A *component* is smallest unit of *physical design* containing :
  - 1,2, or even several *classes*
  - several hundred lines of C++ source code and .h files
- Address complexity by abstraction and hierarchy.

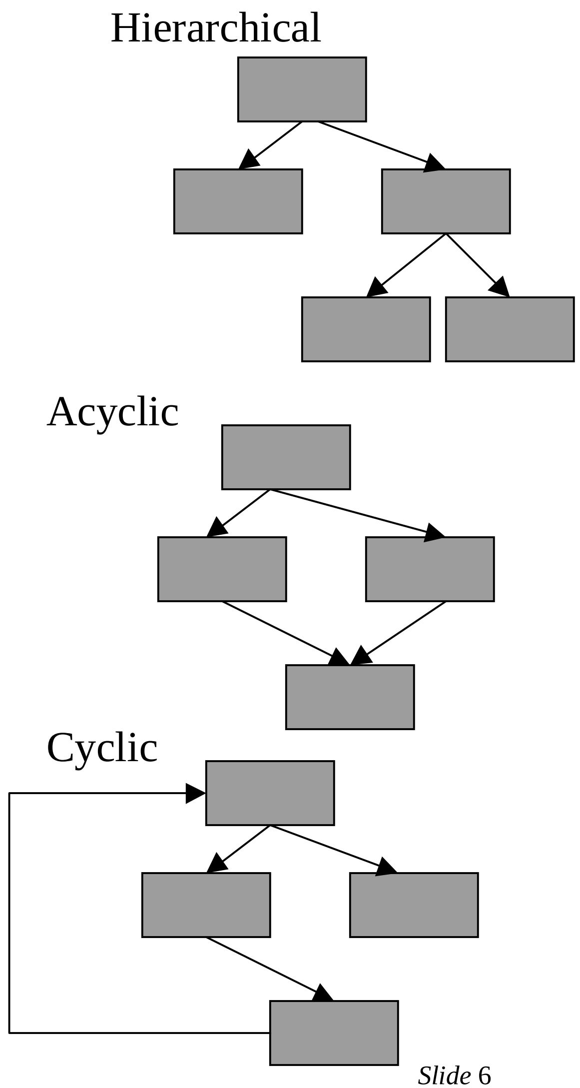
Example



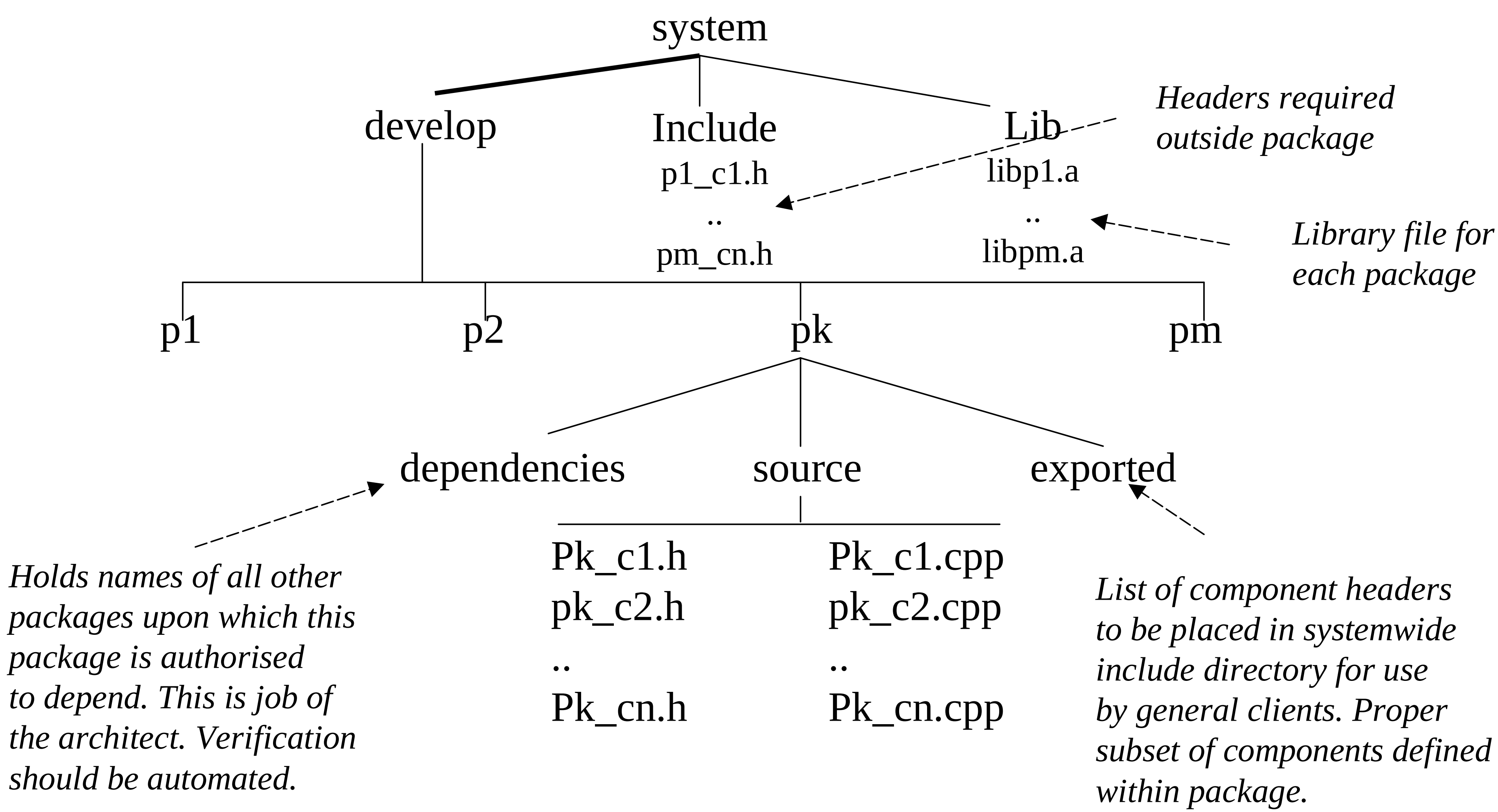
- Dependencies between larger units represent an envelope for aggregate dependencies among the components comprising each subsystem
- Once database is designed, can launch 3 concurrent efforts on Parsing, Evaluating and Formatting and finally top level Interpreter

Definitions

- A package is a collection of components organised as a physically cohesive unit
- It refers to a generally acyclic, often hierarchical collection of components that have a cohesive semantic purpose.
- Physically it consists of a collection of header files along with a single library file
- It might consist of a loosely-coupled collection of low-level re-usable components, such as STL



Possible Organisation

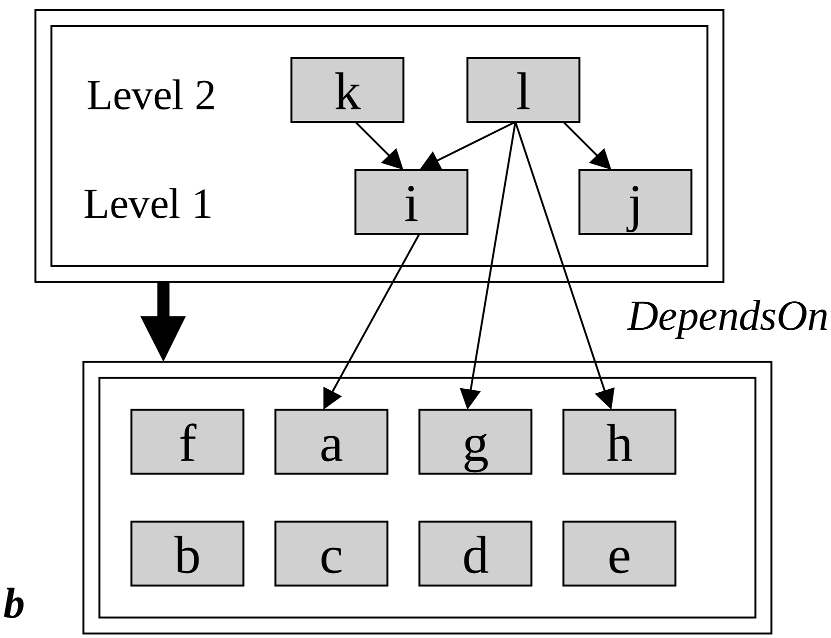


DependsOn, Levilisation

- **A package *x* DependsOn another package *y* if 1 or more components in *x* DependsOn one or more components in *y***

Package Level 2

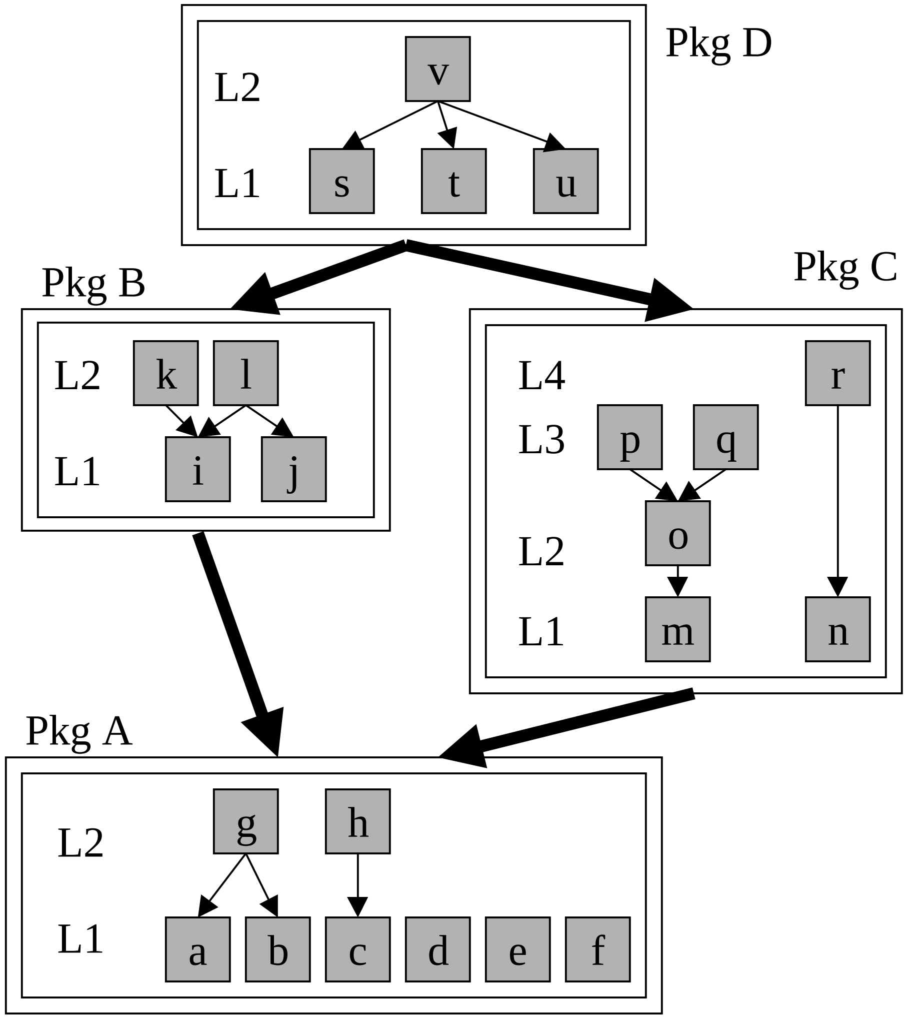
Package a

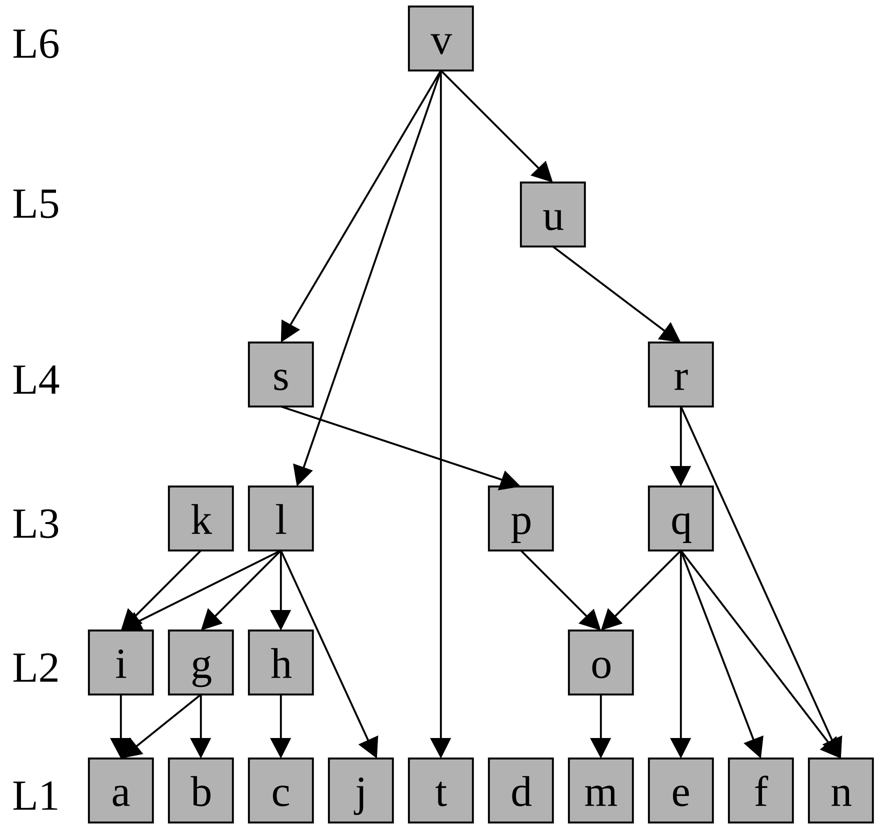


Package Level 1

Package b

Decomposition of System into Packages of Components





Advantages of Packages

- Develop architecture at higher level of abstraction
- Delineate responsibility for a package - each package can be owned/authored by single developer
- Specify acceptable dependencies as part of overall system design without addressing individual components
- Putting at same level in directory structure makes them easily accessible to developers
- Physical dependencies can be extracted by tool and compared to architect's specification
- Highly coupled parts of system can be assigned to single package with single developer - change management easier

Package Prefixes

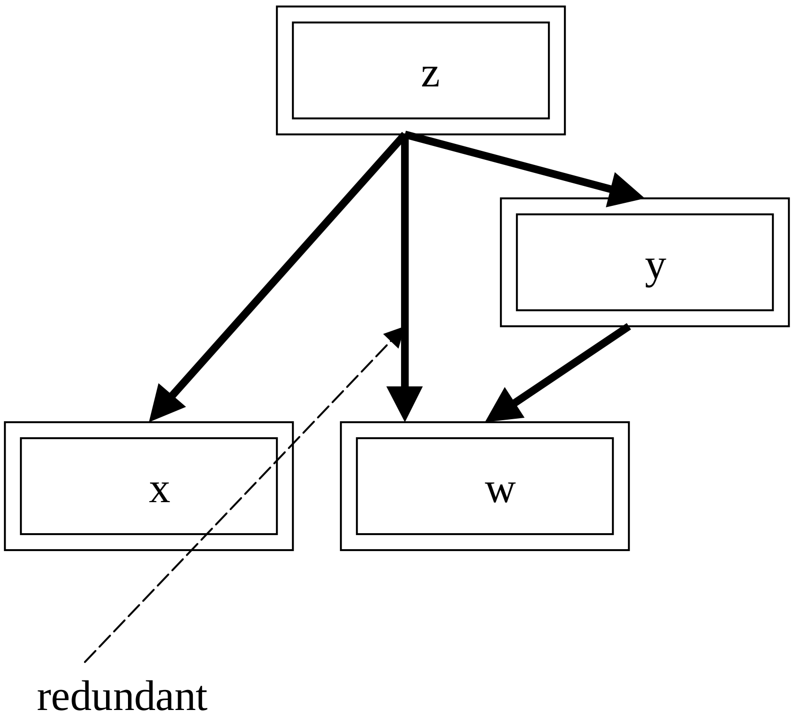
- Structured approach required to avoid name collisions
- Each package must be associated with *unique registered* prefix consisting of 2-5 characters
- Each construct in header file is prepended with package prefix as are .cpp and .h files implementing component.
- Major design rules
  - Prepend every global identifier with its package prefix
  - Prepend every source file name with its package prefix
- Principles : Purpose of prefix is to :
  - identify uniquely physical package in which component resides
  - indicate logical and organisational characteristics

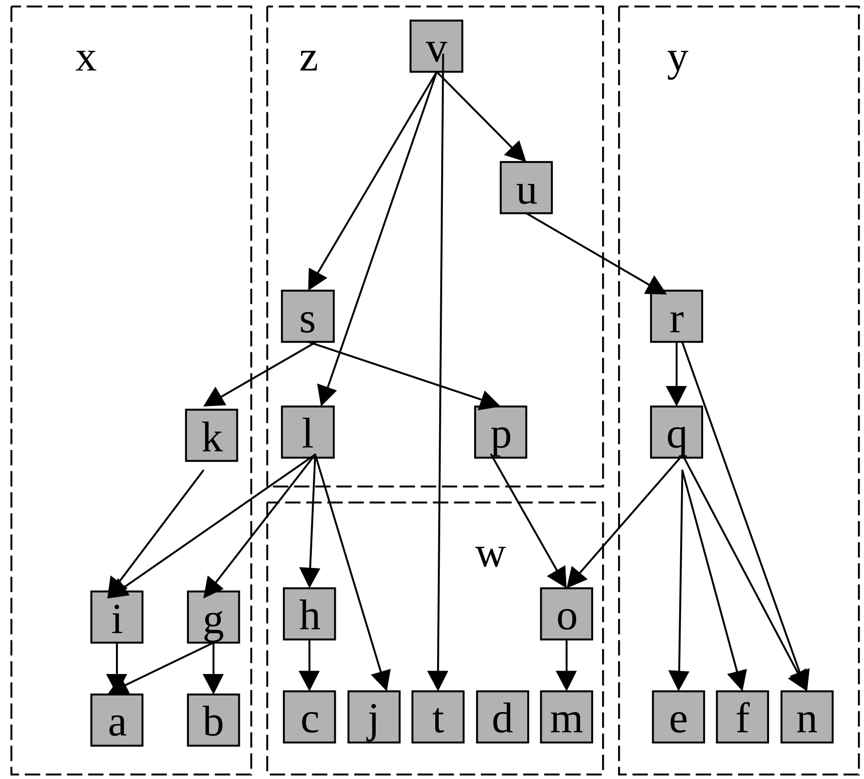
Avoid Cyclic dependencies

- Important design goal - aids incremental comprehension, testing and reuse.
- Avoid among packages too! In general minimise package interdependencies
  - optimises linking
  - usability - don’t link huge libraries just to use simple functions
  - reduces number of libraries that must be linked
  - minimises size of executable image
- Need to test large system incrementally and hierarchically
- Techniques to avid - escalate component to higher level package, repackage

Partitioning

- A package should consist of components that make sense to be packaged together and treated abstractly at higher level.

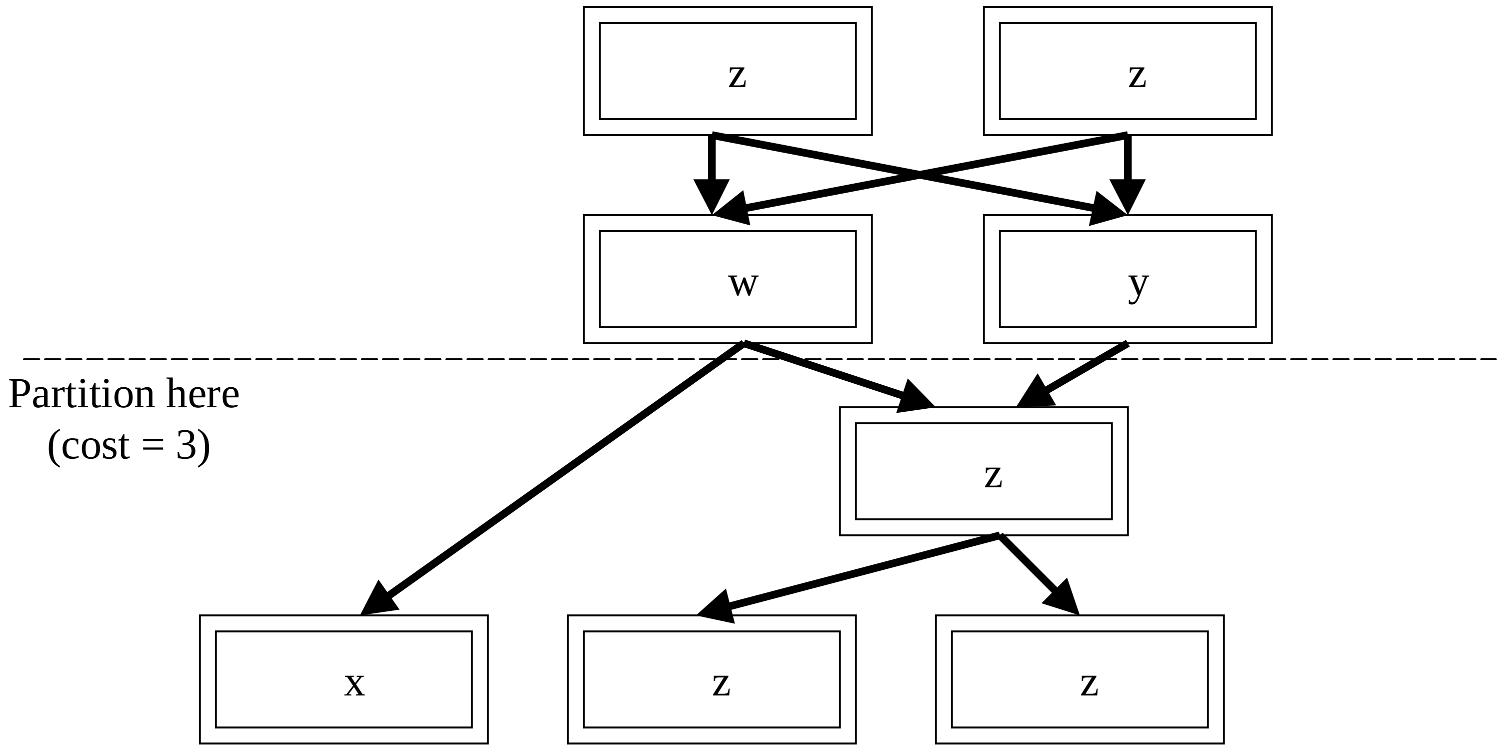




- When adding a component to a package both logical and physical characteristics of component should be considered

Multi-site development

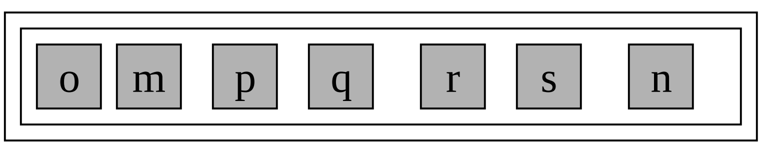
- Geographical distribution influences how package ownership is distributed among developers



Package Insulation

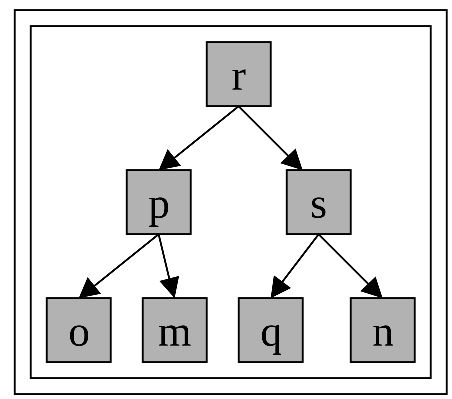
- Minimising number of exported header files enhances usability

Exported headers
o.h
m.h
p.h
q.h
..
n.h



Logical abstraction only

Exported headers
r.h



Logical and physical abstraction only

Must header for particular component be exported?

- Do clients of package need access to component to use functionality provided by package?
- Does any other exported component fail to insulate its clients from this components definition?
- Do other packages need access to this component e.g. to reuse its functionality?

Other package issues discussed

- Groups of packages (very large systems - us?)
- Release structure
  - directory hierarchy
  - cost of compiling - function of #.h files, but also #directories
    - **Put header files in just a few directories**
- A patch is a local change to previously released software to repair faulty functionality within a component. It must not affect internal layout of any existing object.
- Start-up time is time between when a program is first invoked and when thread of control enters main. Time when non-local static objects are created.
- Clean-up. Provide mechanism for freeing dynamic memory allocated to static constructs within a component.

Other Topics discussed

- Architecting a component
  - component interface design
  - degrees of encapsulation
- Designing a function
  - interface specification
  - types used in the interface
- Implementing an Object
  - member data
  - function definitions
  - memory management
  - using templates in large projects