IMPLEMENTATION ARCHITECTURE

Tom McBride
University of Technology, Sydney

UTS:
ENGINEERING AND INFORMATION TECHNOLOGY

Implementation architecture decisions

- Build or buy components
- How will the development be allocated within the organization
- How will modules be packaged for deployment

UNIVERSITY OF TECHNOLOGY SYDNEY

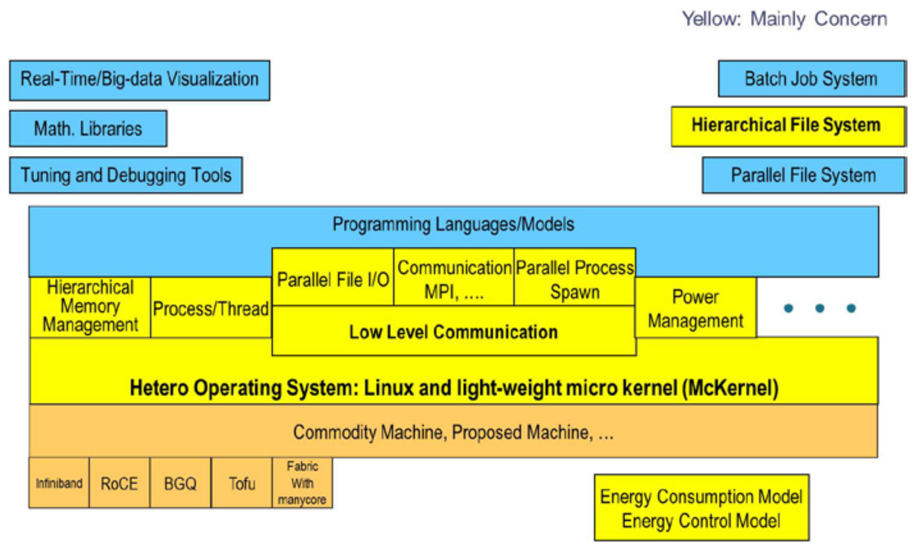
BUILD OR BUY

Build vs Buy is a common engineering problem

- Some components are so well established as COTS components that the decision is easy. E.g. DBMS, specialised library
- Some components are clearly unique and must be developed
- As usual there is a gray border between the two.

Some COTS software is ubiquitous, just buy it.

- DBMS (Oracle, MySQL, Postgres, DB2, etc.)
- Math libraries and other specialised libraries.
- Protocol stacks



Some software modules are part of your organization's value proposition

- Your organization knows something or does something better than others. This is their value proposition.
- Software that realises or supports the value proposition is very likely unique and will need to be developed.

The gray stuff in the middle needs a ‘build or buy’ decision

- If you buy it your architecture might need to accommodate it.
- If you build it you bear all the costs of development and deployment.
- These are significant project decisions, reflected in the architecture

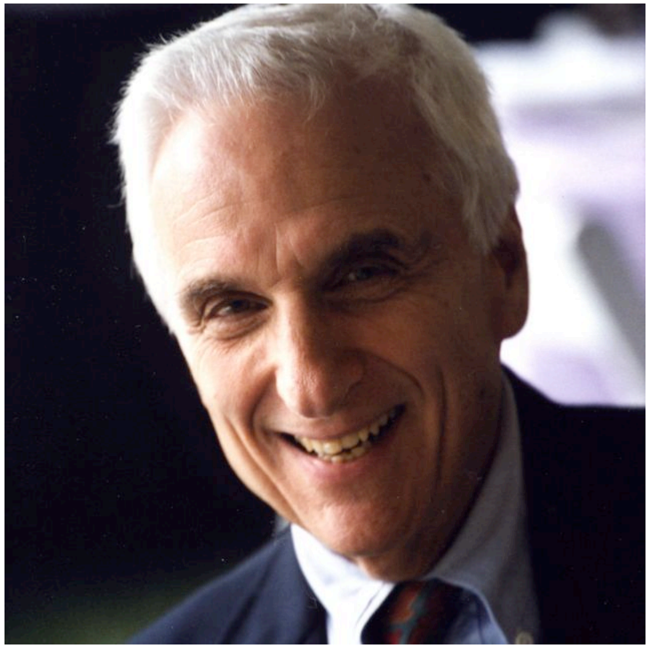
UNIVERSITY OF TECHNOLOGY SYDNEY

Conway’s Law and all that

ALLOCATING THE WORK

Conway’s Law is very powerful

- organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations
  - —M. Conway, 1967



The effect of Conway's Law

- Components that need tighter coupling should be developed by the same or, at least, a co-located team.
- Components that are or can be loosely coupled can be developed by a loosely connected team
- Some companies, e.g. Amazon, Netflix, restructured their company to match the architecture of the system they wanted to build.
- Usually it is the other way around – the software reflects the organization

UNIVERSITY OF TECHNOLOGY SYDNEY

PRINCIPLES OF PACKAGE ARCHITECTURE

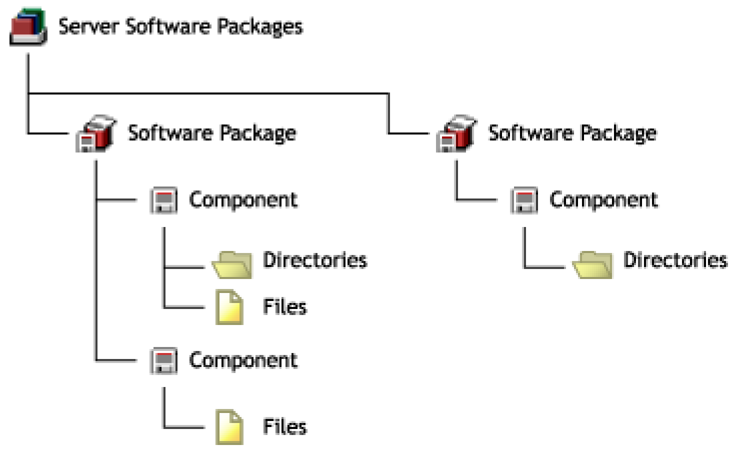
Packaging for deployment

- Consider operational needs.
  - Continuous development
  - Continuous deployment
- Consider maintenance needs.
  - How much of the system is affected by likely changes.
- Packaging the system for deployment and maintenance

OPERATIONS
SUPPORT
MAINTENANCE
OPERATIONS & MAINTENANCE

Designing a deployable package

- Common practice is to collect a set of classes or other components into a package.
- Like classes these should be designed thoughtfully.



Package Principles

- Package Cohesion Principles
  - Release Reuse Equivalency Principle
  - Common Closure Principle
  - Common Reuse Principle
- Package Coupling Principles
  - Acyclic Dependencies Principle
  - Stable Dependencies Principle
  - Stable Abstractions Principle
- Although classes have already been aggregated into components, more may be needed to support deployment and maintenance

UNIVERSITY OF TECHNOLOGY SYDNEY

Deciding what to bring together

PACKAGE COHESION PRINCIPLES

Package Cohesion Principles

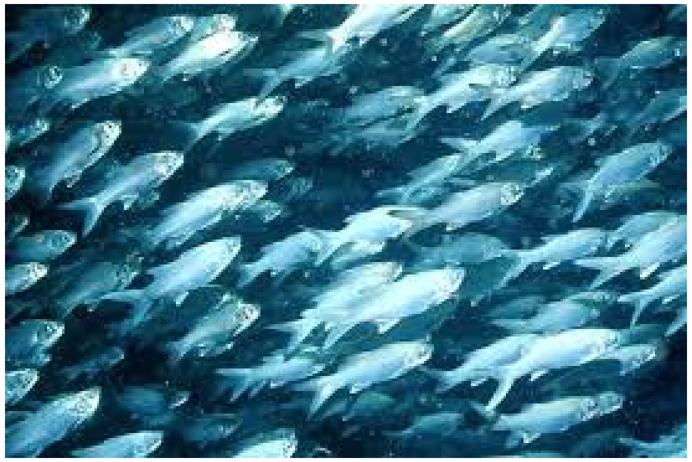
- Classes are a necessary, but insufficient, means of organizing a design.
- The larger granularity of packages are needed to help bring order.
- But how do we choose which classes belong in which packages.

Release Reuse Equivalency Principle

- **The granule of reuse is the granule of release.**
- Users want a ‘sensible’ package
  - A reusable element, be it a component, a class, or a cluster of classes, cannot be reused unless it is managed by a release system of some kind.
  - Users will be unwilling to use the element if they are forced to upgrade every time the author changes it.
  - Thus even though the author has released a new version of his reusable element, he must be willing to support and maintain older versions while his customers go about the slow business of getting ready to upgrade.
- Therefore, one criterion for grouping classes into packages is reuse.
- Since packages are the unit of release, they are also the unit of reuse.
- **Therefore architects would do well to group reusable classes together into packages.**

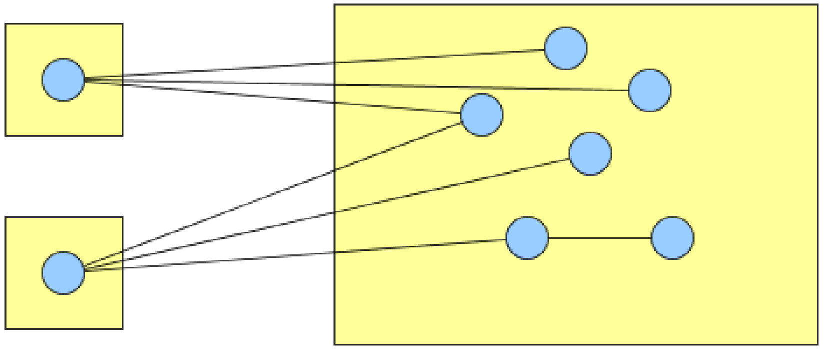
Common Closure Principle

- **Classes that change together, belong together.**
- A large development project is subdivided into a large network of interrelated packages. The work to manage, test, and release those packages is non-trivial. The more packages that change in any given release, the greater the work to rebuild, test, and deploy the release.
- Therefore we would like to minimize the number of packages that are changed in any given release cycle of the product.
- To achieve this, we group together classes that we think will change together. This requires a certain amount of prescience since we must anticipate the kinds of changes that are likely.
- Still, when we group classes that change together into the same packages, then the package impact from release to release will be minimized.



Common Reuse Principle

- **Classes that aren't reused together should not be grouped together.**
- A dependency upon a package is a dependency upon everything within the package.
- When a package changes, and its release number is bumped, all clients of that pack-age must verify that they work with the new package even if nothing they used within the package actually changed.
- We frequently experience this when our OS vendor releases a new operating system. We have to upgrade sooner or later, because the vendor will not support the old version forever. So even though nothing of interest to us changed in the new release, we must go through the effort of upgrading and revalidating.
- The same can happen with packages if classes that are not used together are grouped together. Changes to a class that I don't care about will still force a new release of the package, and still cause me to go through the effort of upgrading and revalidating.



Summary of package Cohesion Principles

- Package Cohesion Principles
  - Release Reuse Equivalency Principle
  - Common Closure Principle
  - Common Reuse Principle

UNIVERSITY OF TECHNOLOGY SYDNEY

Keeping relationships working between packages

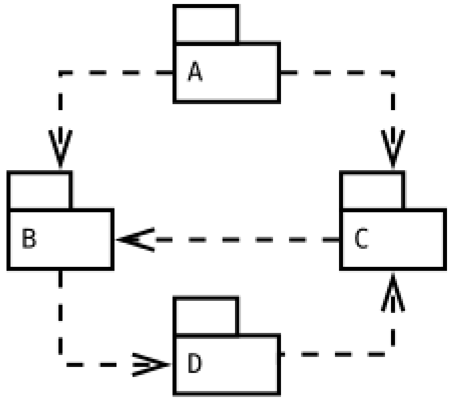
PACKAGE COUPLING PRINCIPLES

Package Coupling Principles

- Applications tend to be large networks of interlaced packages.
- The rules that govern these interrelationship are some of the most important rules in object oriented architecture.
- Package Coupling Principles
  - Acyclic Dependencies Principle
  - Stable Dependencies Principle
  - Stable Abstractions Principle

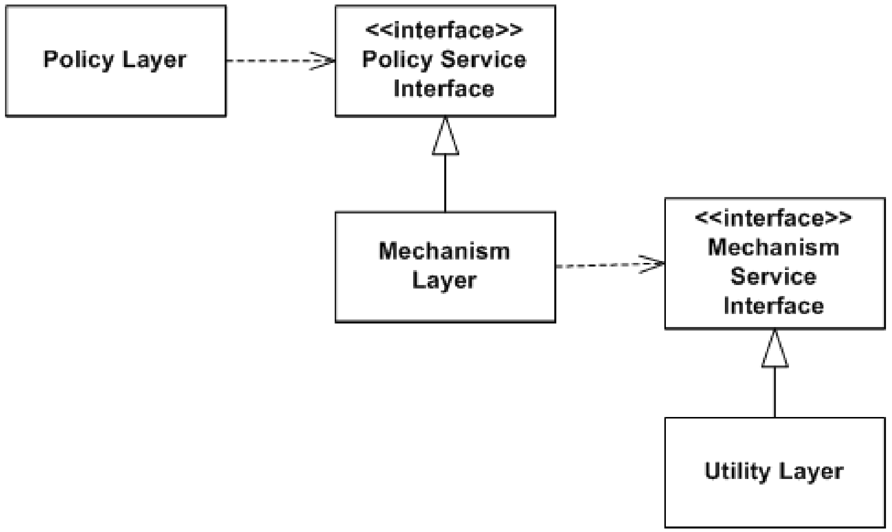
Acyclic Dependencies principle

- The dependency graph of packages or components should have no cycles
- Example;
  - package *A* depends on packages *B* and *C*. Package *B* in turn depends on package *D*, which depends on package *C*, which in turn depends on package *B*. The latter three dependencies create a cycle, which must be broken in order to adhere to the acyclic dependencies principle



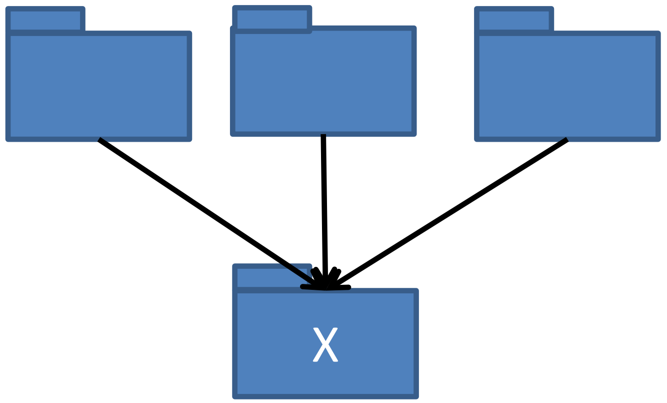
Breaking cyclic dependencies

- Create a new package and move the common dependencies there.
- Invert the dependencies by inserting an abstraction between them

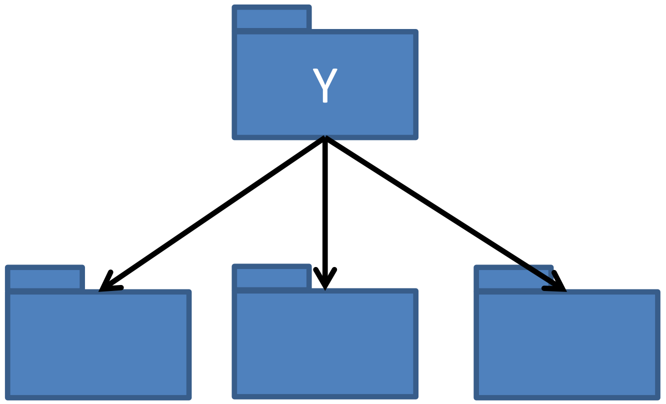

Policy Layer
<<interface>> Policy Service Interface
Mechanism Layer
<<interface>> Mechanism Service Interface
Utility Layer

Stable Dependency Principle

- Stability is related to the amount of work required to make a change.
- A software package with lots of other packages dependent on it is very stable.
- A package that depends on a lot of other packages in unstable



X is stable



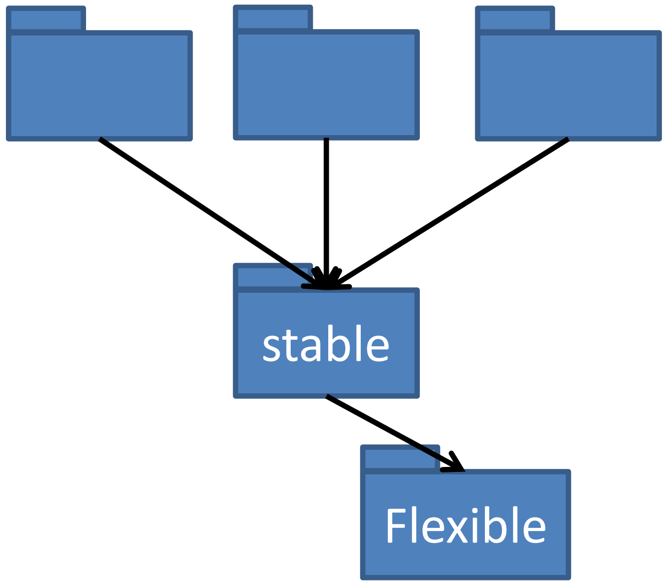
Y is unstable

Should all software be stable

- One of the most important attributes of well designed software is ease of change.
- Software that is flexible in the presence of changing requirements is thought well of. Yet that software is unstable by our definition.
- Indeed, we greatly desire that portions of our software be unstable. We want certain modules to be easy to change so that when requirements drift, the design can respond with ease.

Stable Abstractions Principle

- Flexibility requires that stable packages are easy to extend.
- But some parts may need to change
- Solution – insert an abstract class to act as an interface.
- Requires that the abstraction is well designed.



Summary of Package Coupling Principles

- Package Coupling Principles
  - Acyclic Dependencies Principle
  - Stable Dependencies Principle
  - Stable Abstractions Principle