Physical SW design
- some vocabulary -

Benedikt Hegner
(CERN)

Many thanks to Vincenzo Innocente

Level of complexity matters!

Architecting a dog house

- Small problems can be solved with simple techniques

- For large problems you need to use different techniques that are in general more complex and with upfront costs

Architecting a dog house

- Can be build by one person
- Requires
  - Minimal modelling
  - Simple process
  - Simple tools

- Little risk

Architecting a house

- Built most efficiently and timely by a team

- Requires
  - Modeling
  - Well-defined process
  - Powerful tools

Architecting a high rise

- Built by many companies

- Requires
  - Modeling
  - Simple plans, evolving blueprints
  - Scale models
  - Engineering plans
  - Well-defined process
  - Architectural team
  - Political planning
  - Infrastructure planning
  - Time-tabling and scheduling
  - Selling space
  - Heavy equipment

Performance

- “More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity”
(William Wulf (AT&T Professor))

- Overall efficiency is what matters
  - Runtime + Development Time

- Overall design should take performance considerations very much into account, but not down to individual code

- You have to understand and check (!) where you have an individual performance problem

- Reminder:
  - Fast code is nice, incorrect output useless...

But how to get development / design started ?

- Programming does not start at the keyboard but at the whiteboard
- What should the project actually do in the end
- Come up with an initial idea of how the program should be structured
- Start filling the ‘boxes’ in a prototype
- Throw it away and do the real one...

Throw your prototype away!

Don’t be married to your code!

UML

UML

- Unified Modeling Language (UML) is a standardized general-purpose modeling language

- Includes a set of graphical notation techniques to create visual models of software-intensive systems

- Supports the entire software development lifecycle

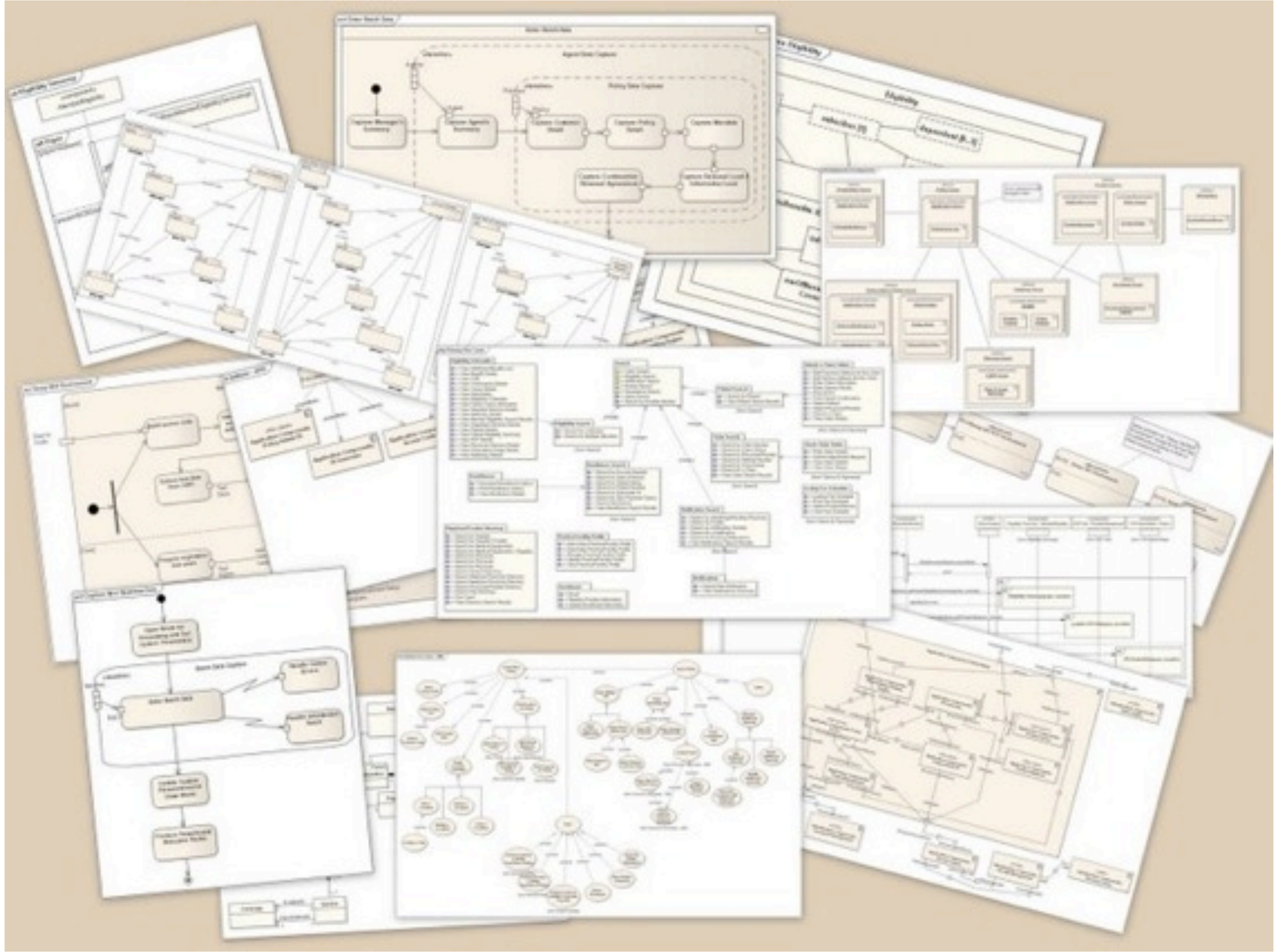
- Supports diverse applications areas

- Is based on experience and needs of the user community

- Supported by many tools

UML

- Structure diagrams
  - Class
  - Component
  - Deployment
  - Object
  - Package

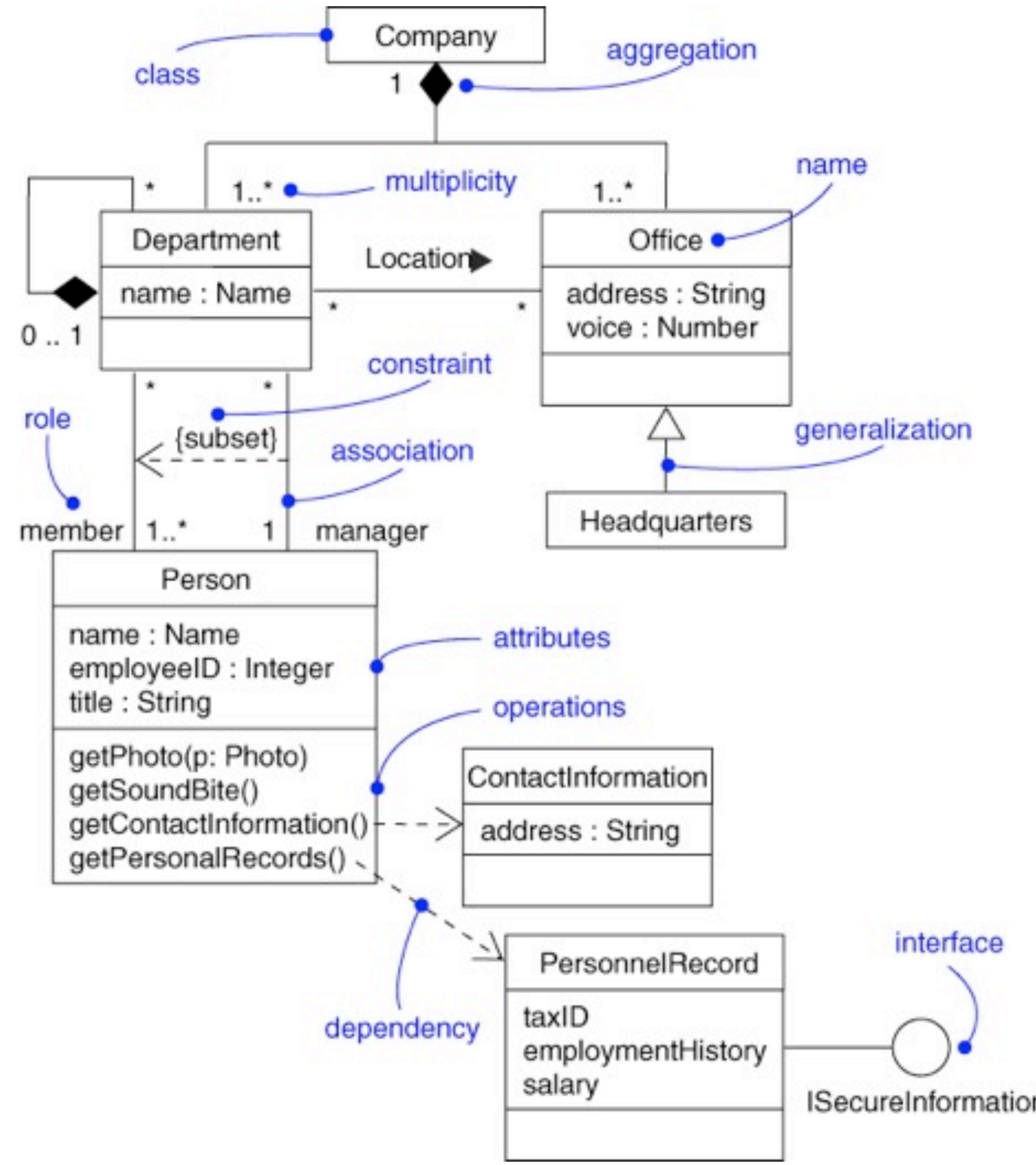


- Behaviour diagrams
  - Activity
  - State machine
  - Use case

- Interaction diagrams
  - Communication
  - Interaction

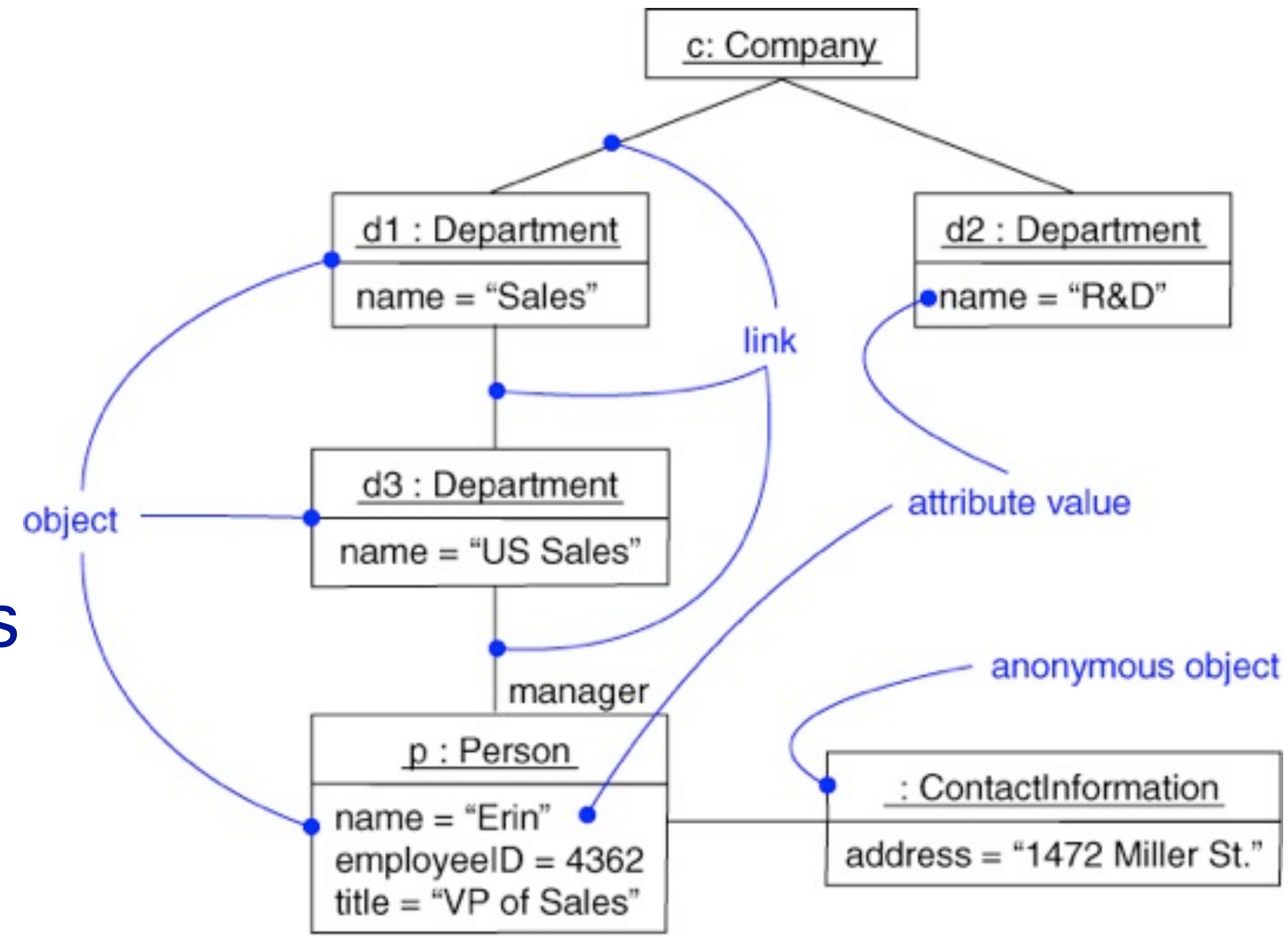
Class diagram

- Captures the vocabulary of a system
- Built and refined throughout development
  - Name models and concepts in the system
  - Specify collaborations
  - Specify DB schemas



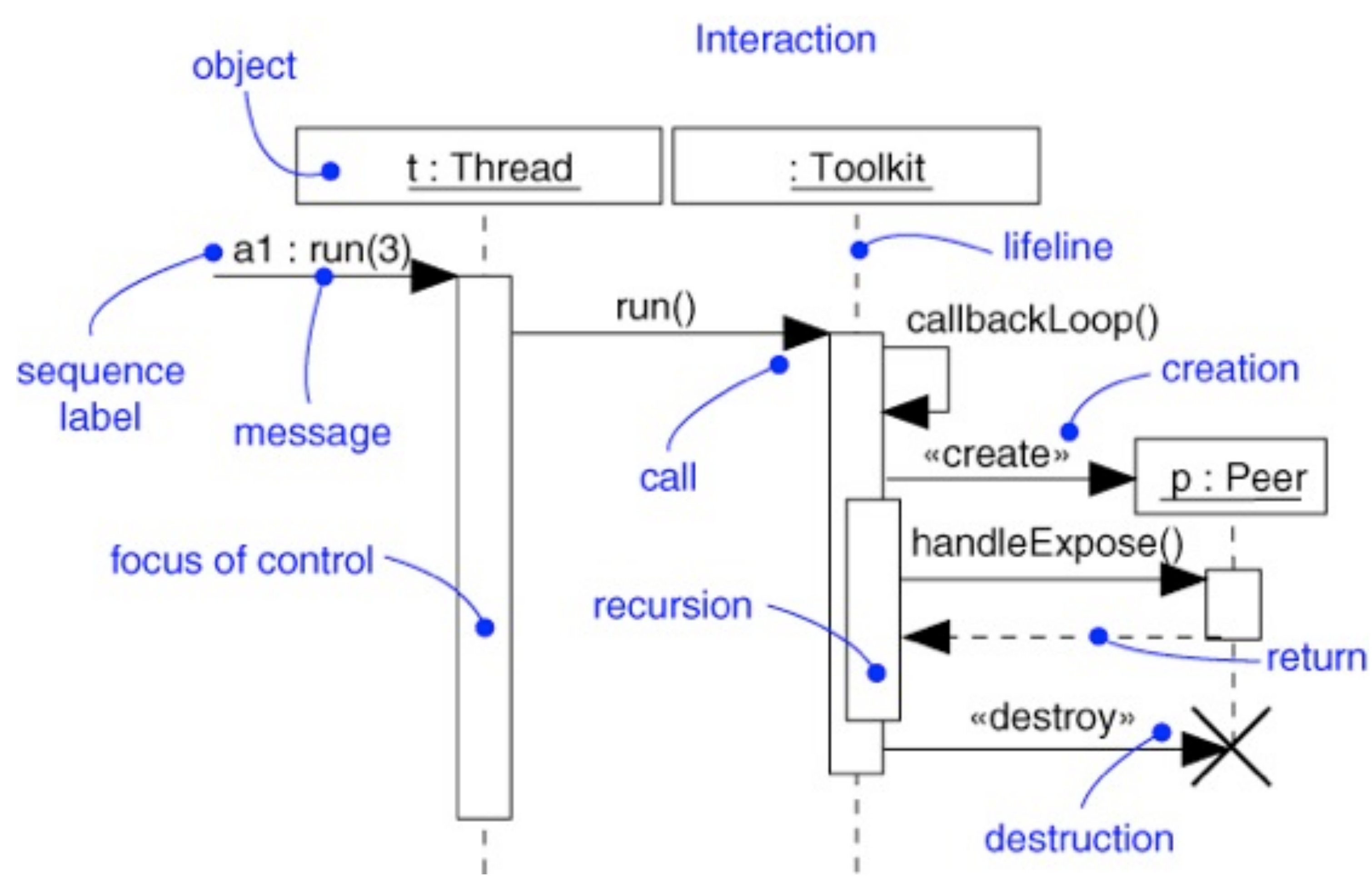
Object diagram

- Shows instances and links
- Built during analysis and design
  - Illustrate data structures
  - Specify snapshots



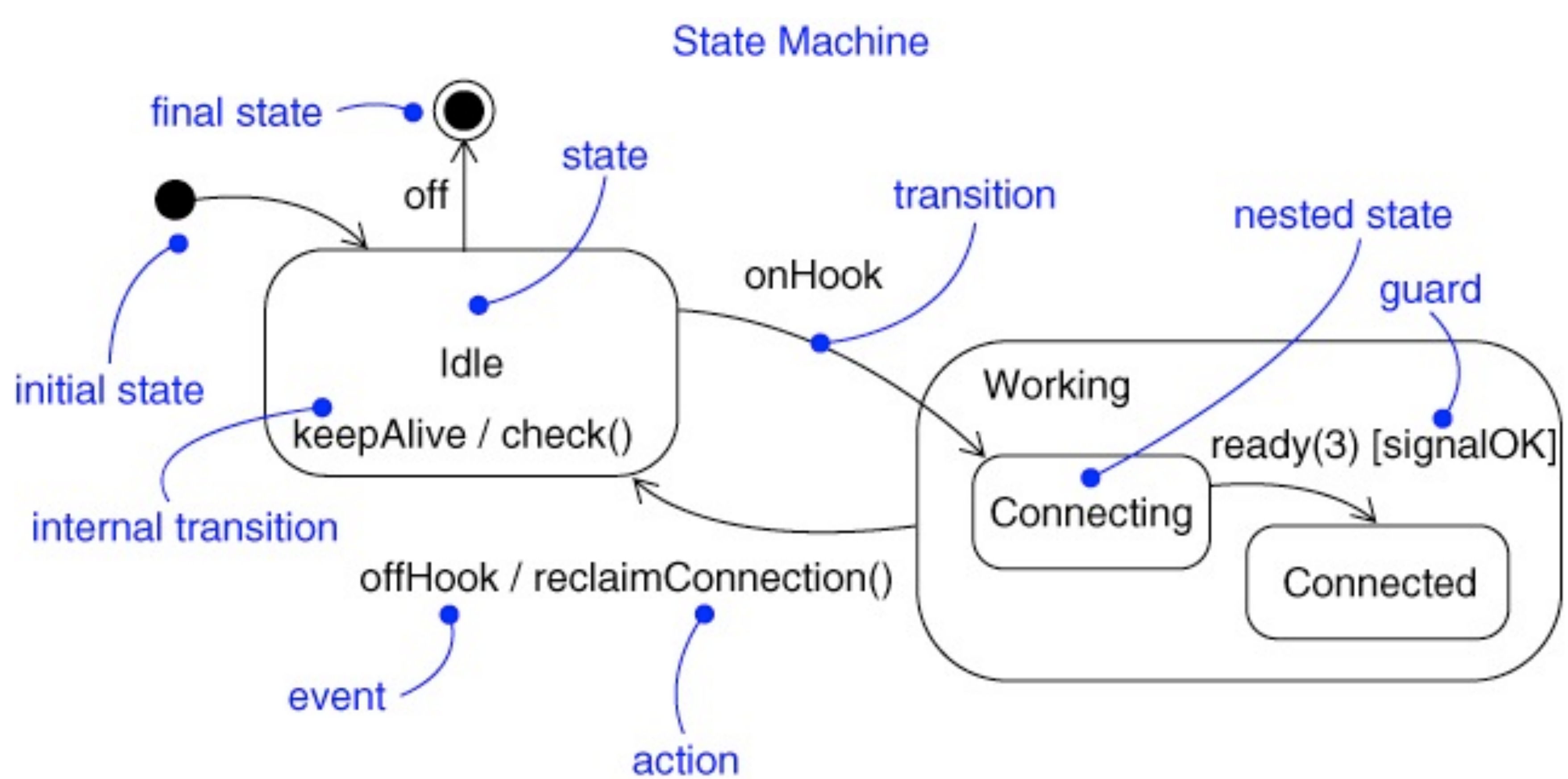
Sequence diagram

- Captures dynamic behaviour (time-oriented)
- Purpose
  - Model flow of control
  - Illustrate typical scenarios



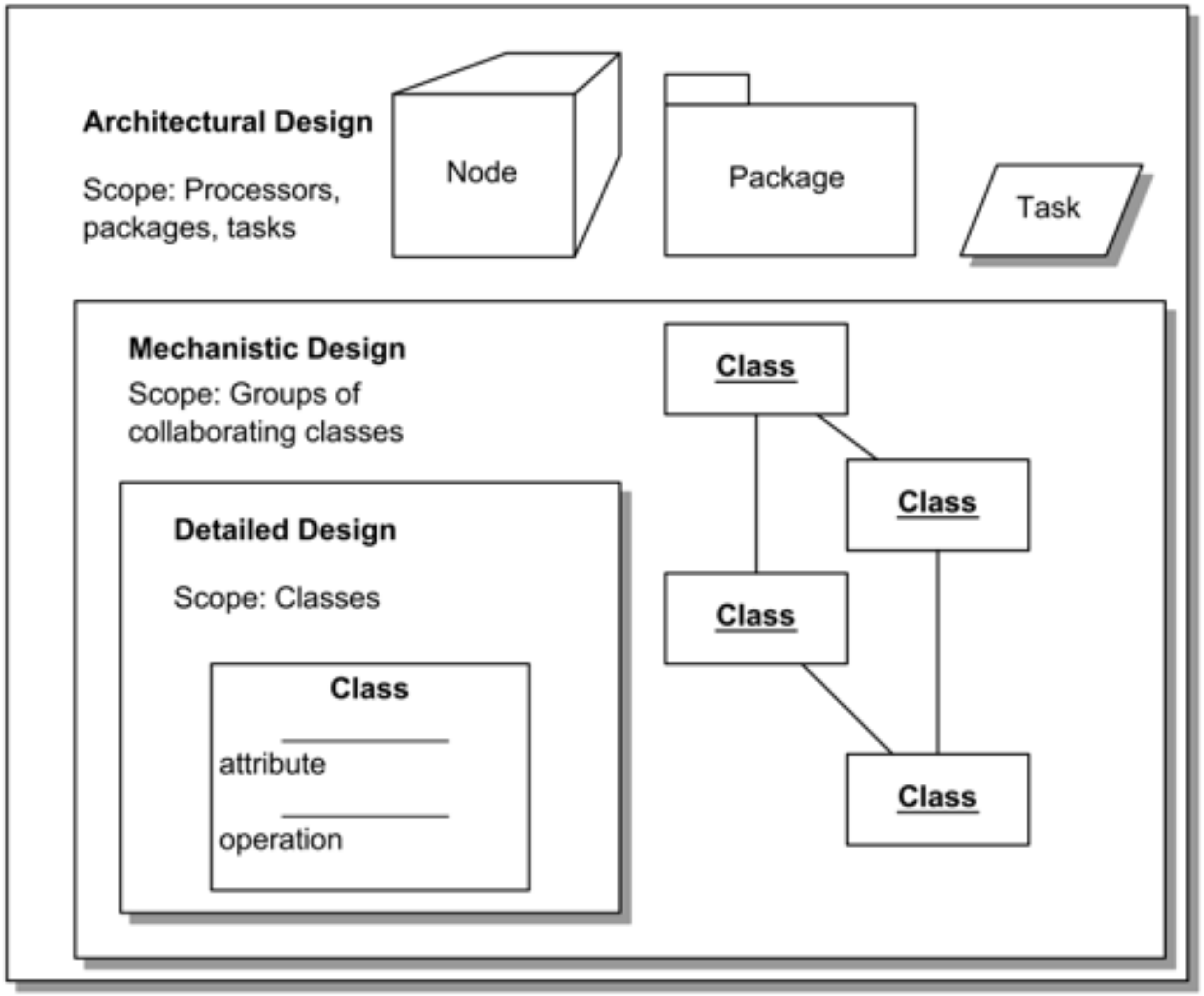
Statechart diagram

- Captures dynamic behaviour (time-oriented)
- Purpose
  - Model object lifecycle
  - Model reactive objects (user interfaces, devices, etc)



Software Design

- System Architecture
- Component Design
- Class Design



Architectural Design

- Capture major interfaces between subsystems and packages early
- Be able to visualize and reason about the design in a common notation
  - Common vocabulary, running scenarios
- Be able to break the work into smaller pieces that can be developed concurrently by different teams
- Acquire an understanding of non-functional constrains
  - Programming languages, concurrency, database, GUI, component re-use

Architecture Defined

- Definition of [software] architecture [1]

  - Set or significant decisions about the organization of the software system
  - Selection of the structural elements and their interfaces which compose the system
  - Their behavior -- collaboration among the structural elements
  - Composition of these structural and behavioral

[1] I. Jacobson, et al. “The Unified Software development Process”, Addison Wesley 1999

Architecture Defined (2)

- Software architecture also involves
  - Usage
  - Functionality
  - Performance
  - Re-use
  - Comprehensibility
  - Economic and technology constraints and tradeoffs

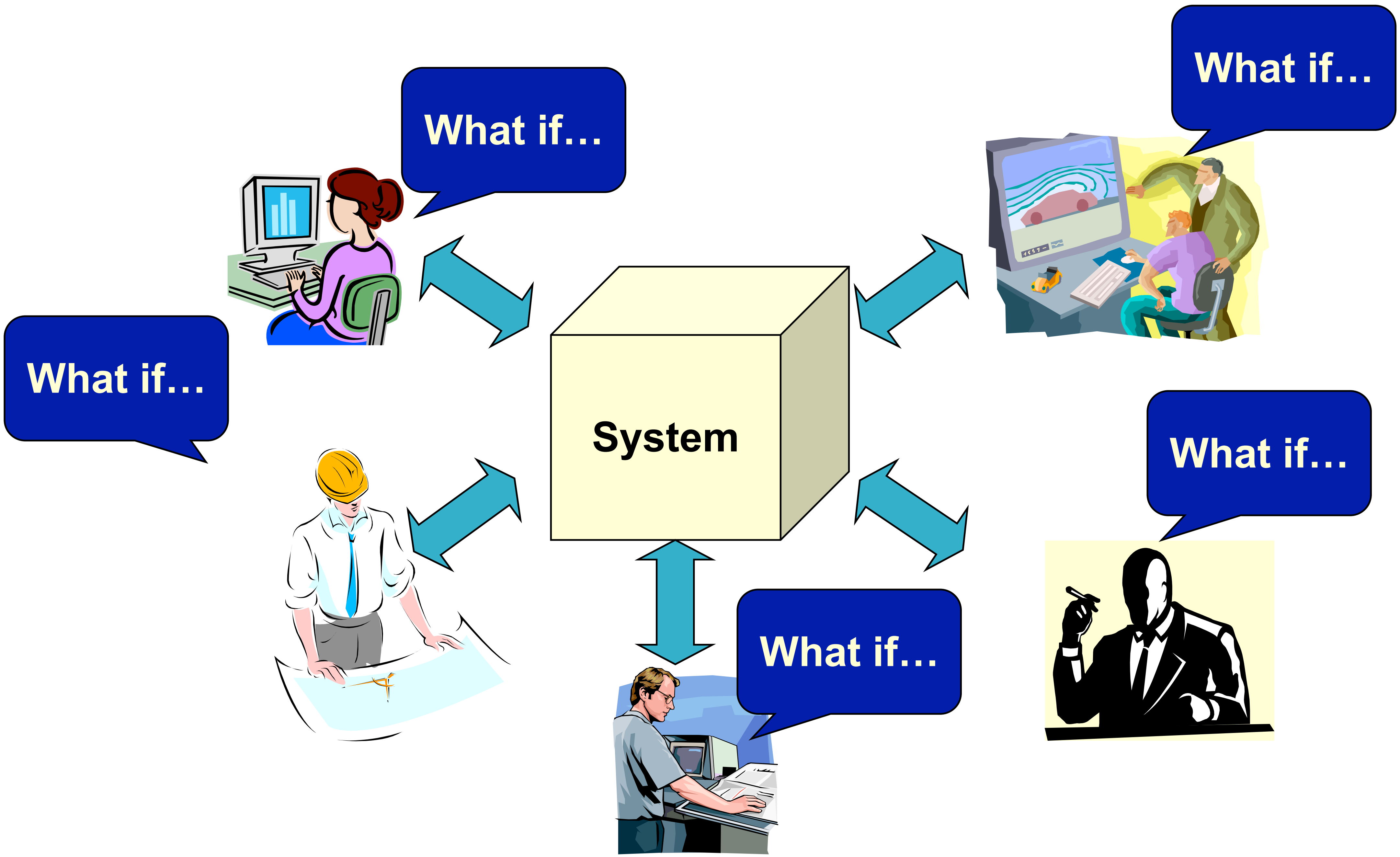
Importance of re-use

- Put extra effort into building high quality components
- Be more efficient by re-using these components
- Many obstacles to overcome
  - too broad functionality / lack of flexibility in components
  - organisational - reuse requires a broad overview to ensure unified approach
    - we tend to split into domains each independently managed
  - cultural
    - don't trust others to deliver what we need
    - fear of dependency on others
    - fail to share information with others
    - developers fear loss of creativity

- Re-use doesn't happen automatically, but needs to be worked for actively

What to consider when designing something?

- Scenario is a brief description of an interaction of a stakeholder with a system



What to consider when designing something? (2)

- User scenarios
  - What if I want to run a new track fit algorithm?
  - What if I need to use the newest calibration?
- Deployment engineer
  - What if we need to port the software to iOS?
  - What if we embed the software in real-time systems?
- Manager
  - What if we need to support some standard data formats
  - What if we integrate a commercial GUI system

Architectural Workflow

- Select scenarios: criticality and risk
- Identify main classes and their responsibility
- Distribute behavior on classes
- Structure in subsystems, layers, define interfaces
- Define distribution and concurrency
- Derive tests from use cases
- Implement architectural prototype
- Evaluate architecture

- Iterate

You'd be amazed how many of these one can map on UML diagrams!

Now moving to Physical Design...

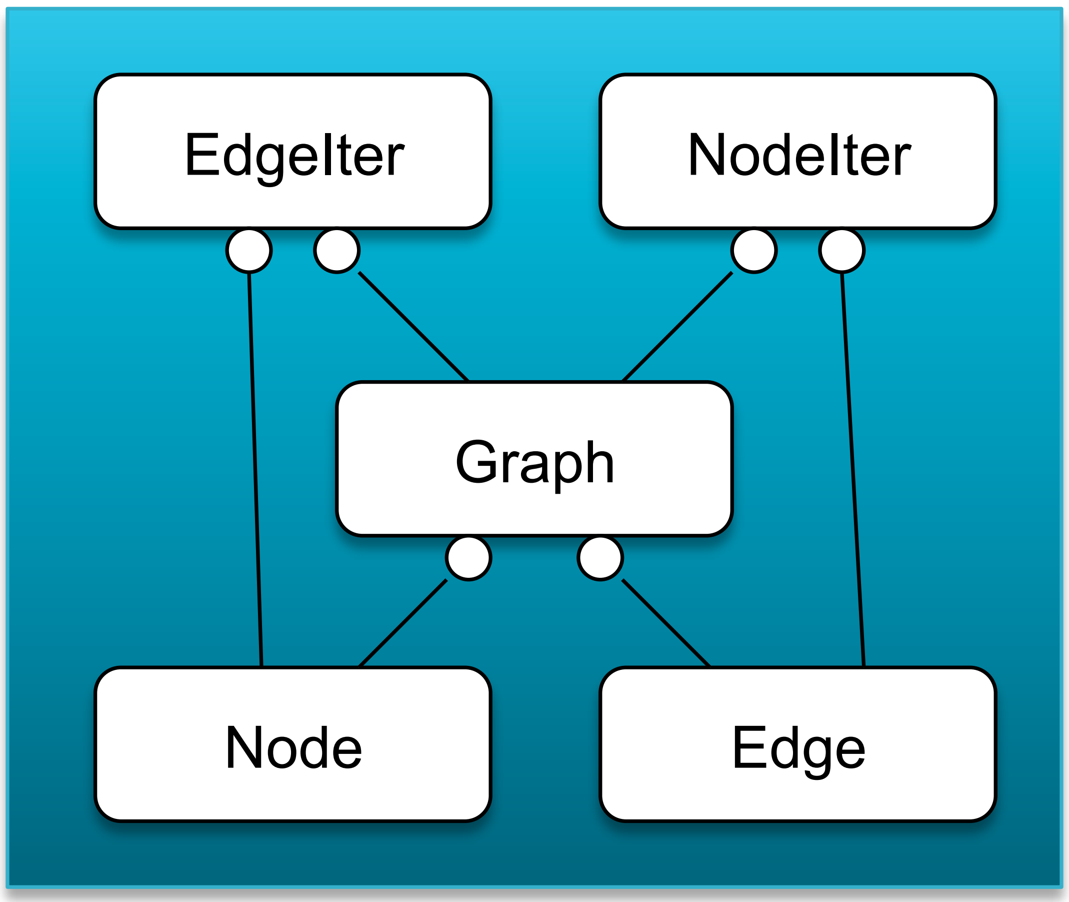
Physical Design Concepts

- Large-scale software development requires more than just logical design issues
  - Distribution of logical entities (classes, functions, etc.) on physical entities (files, directories, etc.)
  - The physical design is the skeleton of the system

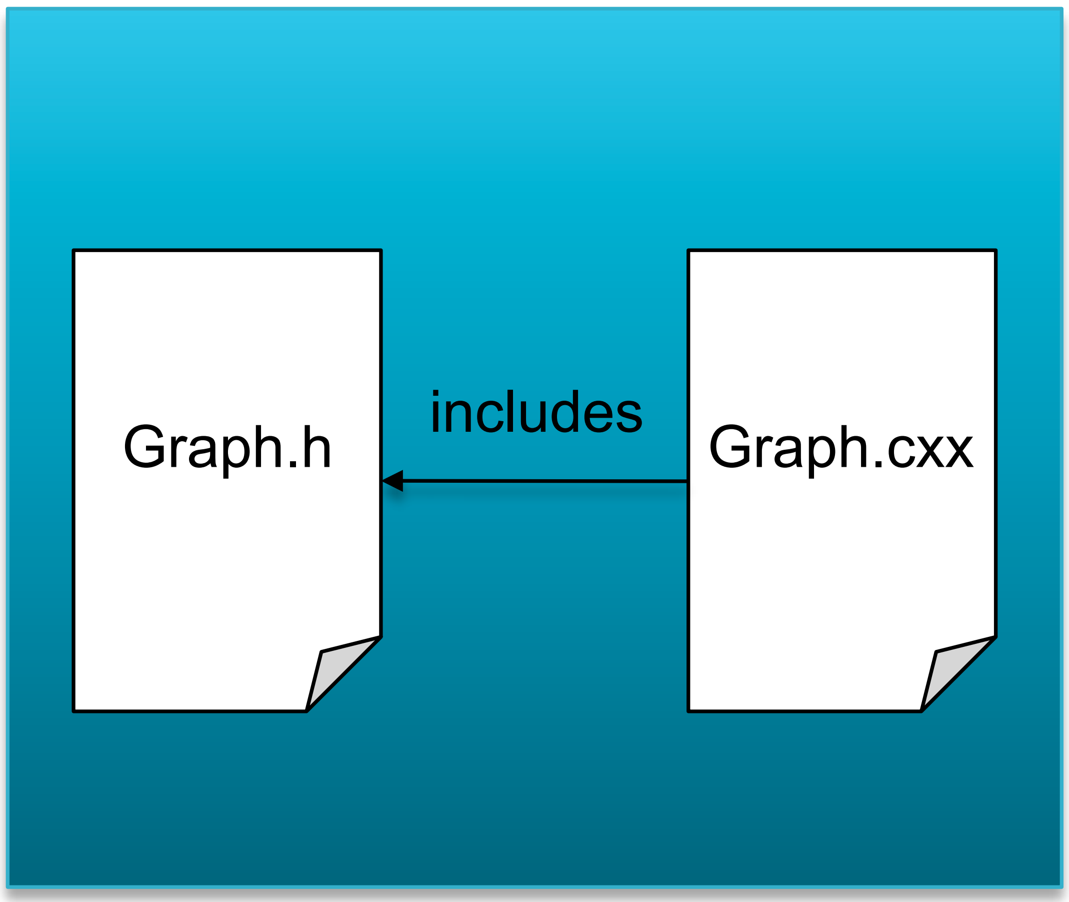
- The quality of physical design dictates from the cost of maintenance to run-time performance
- Additional the potential for re-use
- “Component” is the fundamental unit of design
  - they have a dependency relationship

- Logical design addresses architectural issues; physical design addresses organizational issues

Logical View



Physical View



Components

- Logical design emphasizes interaction of classes and functions in single seamless space
  - It can be viewed as a ‘sea’ of classes and functions
  - It does not take into account physical entities such as files and libraries

- A Component would embody a subset of logical design that makes sense to exists as an independent and cohesive unit

- Typically a Component would consists of a single header file (.h) and implementation files (.cxx)

Packages

- Typically in HEP we put each C++ class in a different file (naming convention & convenience)

- A Package is a collection of components organized as a physically cohesive unit

- A Package is therefore a collection of Classes and functions that implements some functionality
  - Physically a Package is a collection of header files and implementation files organized in some directory structure

- Package is the basic unit in the HEP software development process

- Packages usually depend on other packages

Package as a development unit

- For convenience a Package is developed by one or few developers
    - Concurrent development is essential for large projects
- It is the basic development unit (at least in the HEP communities)
    - It can checked-out and versioned (tagged)
    - It can be tested
    - It can be documented
- Both ATLAS and CMS have a few thousand packages

Package contents

- Public Header Files (.h)
- Private Header files (.h)
- Shareable Libraries (.so)
  - Linker Libraries
  - Component Libraries (plug-ins, i.e. no symbols exported)
  - Other modules (e.g. Python extension modules)
- Programs
- Documentation Files (.html, .doc, ...)

Public Interface of a Package

- Everything declared in its set of public header files
  - Regardless of access privilege (public, protected, private)
  - Any change would cause a re-compilation of clients

- The less information is put on header files the better
  - Favor forward declarations of types used as references and pointers

Package Products

- Linker Libraries
  - Traditional libraries. They export a number of symbols

- Component or plug-in libraries
  - These libraries are loaded at run-time on demand by the application (framework)
  - Typically they do not export any symbol. In some cases a single global one
- Programs, Tests
  - Either direct executables or plug-ins

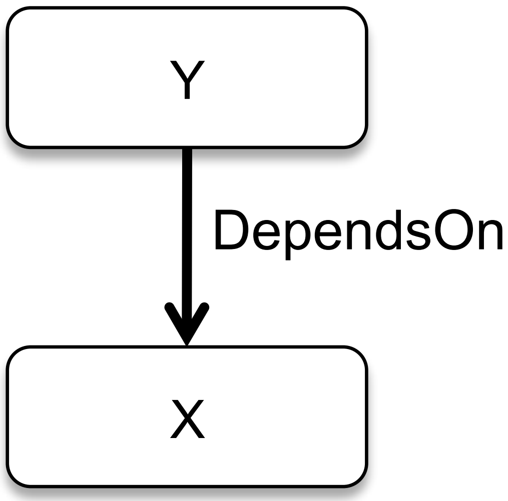
- Documentation
- Additional framework files
- Configuration files, plugin databases, etc.

Package Dependencies

- A package Y DependsOn a package X if X is needed in order to compile or link Y
  - Compile-time dependency if one or more .h files in X are needed for compilation
  - Link-time dependency if one or more libraries in X are needed for linking
  - Run-time dependency if a program/library in package Y requires X for running

- In general compile-time dependency implies link-time dependency and this implies run-time dependency
  - Templates defeat this general rule!

- The DependsOn relation is transitive



Package Dependencies (2)

- A package defining a function will have a physical dependency to any other package defining a type used in the function

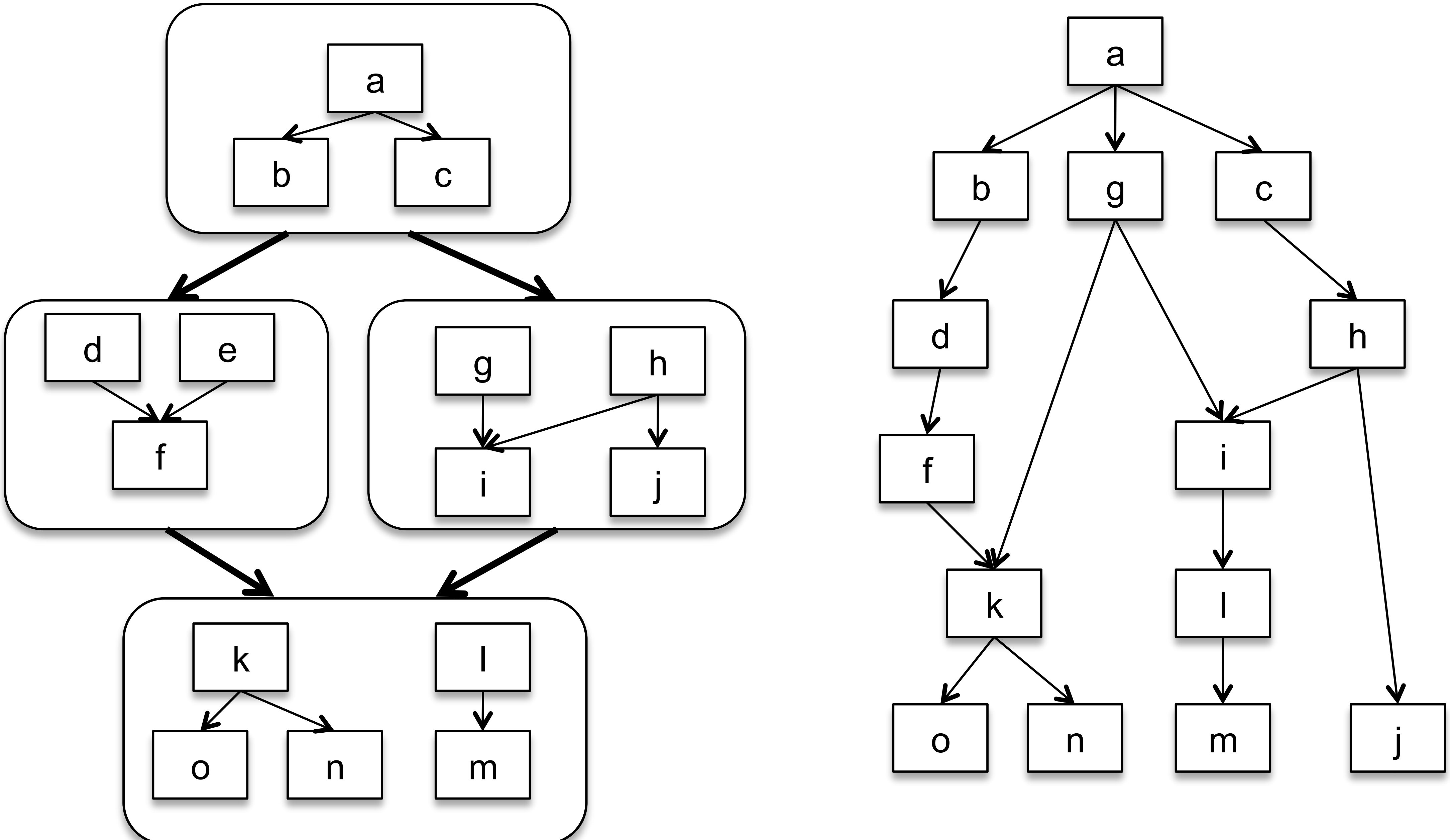
- The logical relationship HasA and IsA translates into a physical dependency

- Dependencies limit
  - flexibility
  - ease of maintenance
  - reuse of components or parts

- Dependency management tries to control dependencies

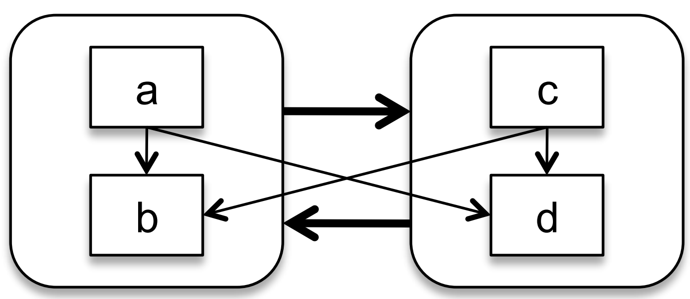
- The more central a package is the more stable it should be
  - Common sense, but frequently violated

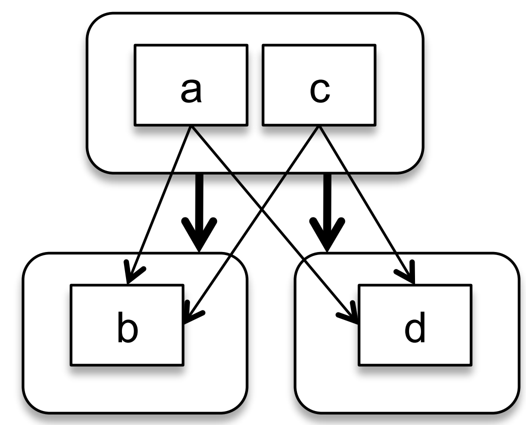
Package Dependencies (3)



sometimes package dependencies don't match logical dependencies

Don't make obvious mistakes...

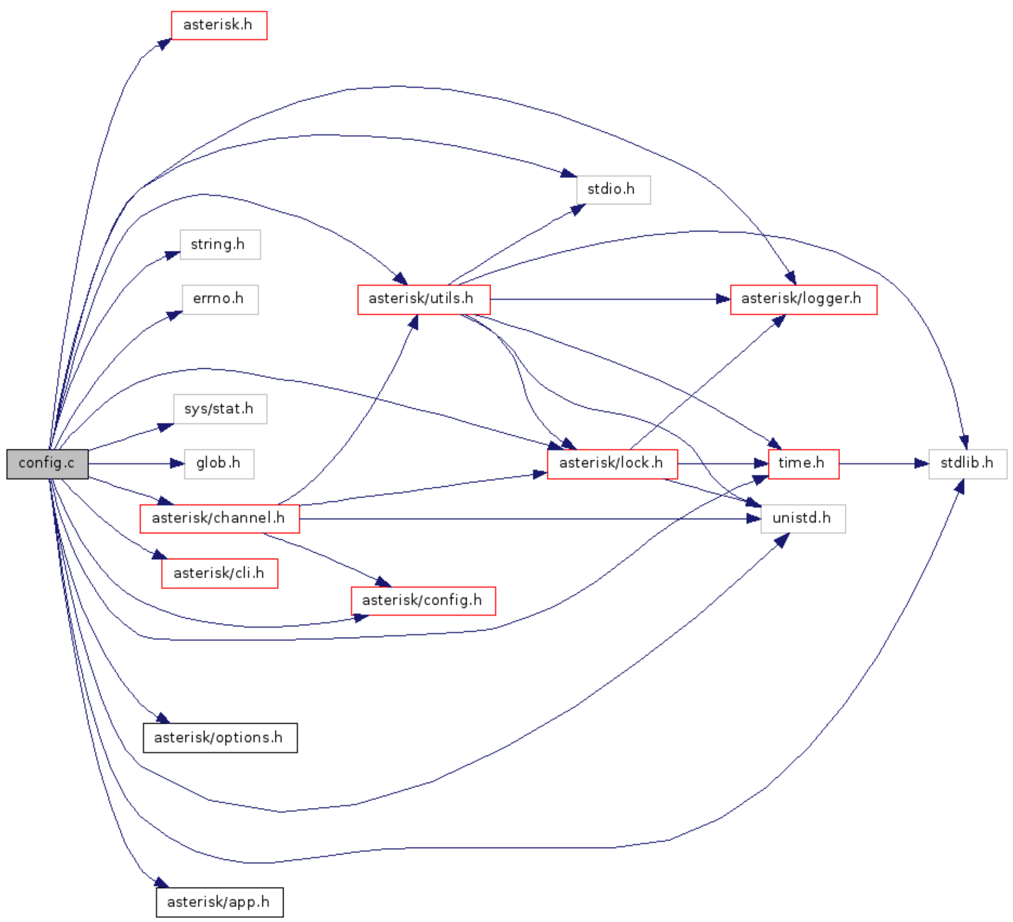




clean up cyclic dependencies

Compile time dependencies

- Cyclic dependencies would prevent building the package. End of story.
- Tools such as Doxygen allows to monitor dependencies
- Thinning header files will speedup building process
- External include guards, or redundant include guards, were suggested by John Lakos




```
#ifndef FILENAME_H_
#include "Filename.h"
#endif // FILENAME_H_
```

Link/Load-Time time dependencies

- The use of dynamic libraries converts link-time dependencies to load-time ones

- Tools such ldd allow to monitor link dependencies
  - Try ldd on one of the examples you played with this morning

- Performance is strongly affected by the number and the size of dependent libraries
  - Interest to keep the them under control

- Reduce the number of needed libraries
  - re-packaging, re-engineering

- Remove unnecessary libraries

- Control package dependencies; use --as-needed flag

Compile and Link Times

- Compile and link times are unproductive

- In a project with N modules compile and link time can grow like N2 (assuming every package is tested) when dependencies are not controlled

- Loss of productivity

- Long turnaround times → slow development

- Dependency management essential in large projects

Run-Time Dependencies

- These dependencies are due typically to the plug-in mechanism, dictionary loading, Python extension modules, etc.

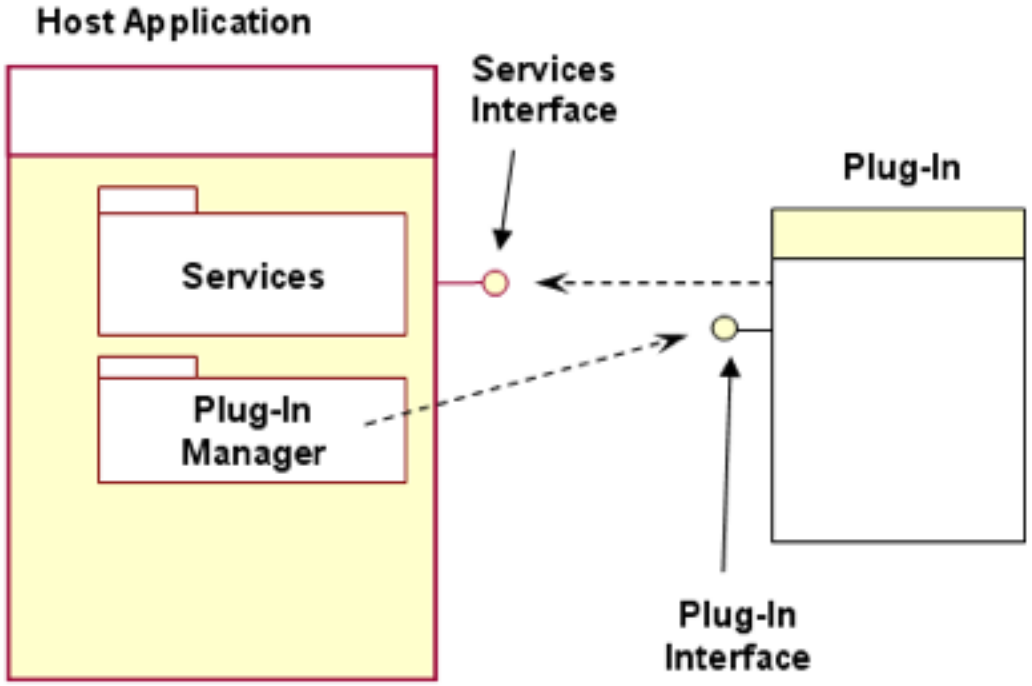
- Frameworks make extensive use of run-time dependencies

- Moving compile and link time dependencies to run-time dependencies is not a bad move
    - Only needed functionality will be loaded

- Packaging and installation of ‘plug-ins’ is non-trivial

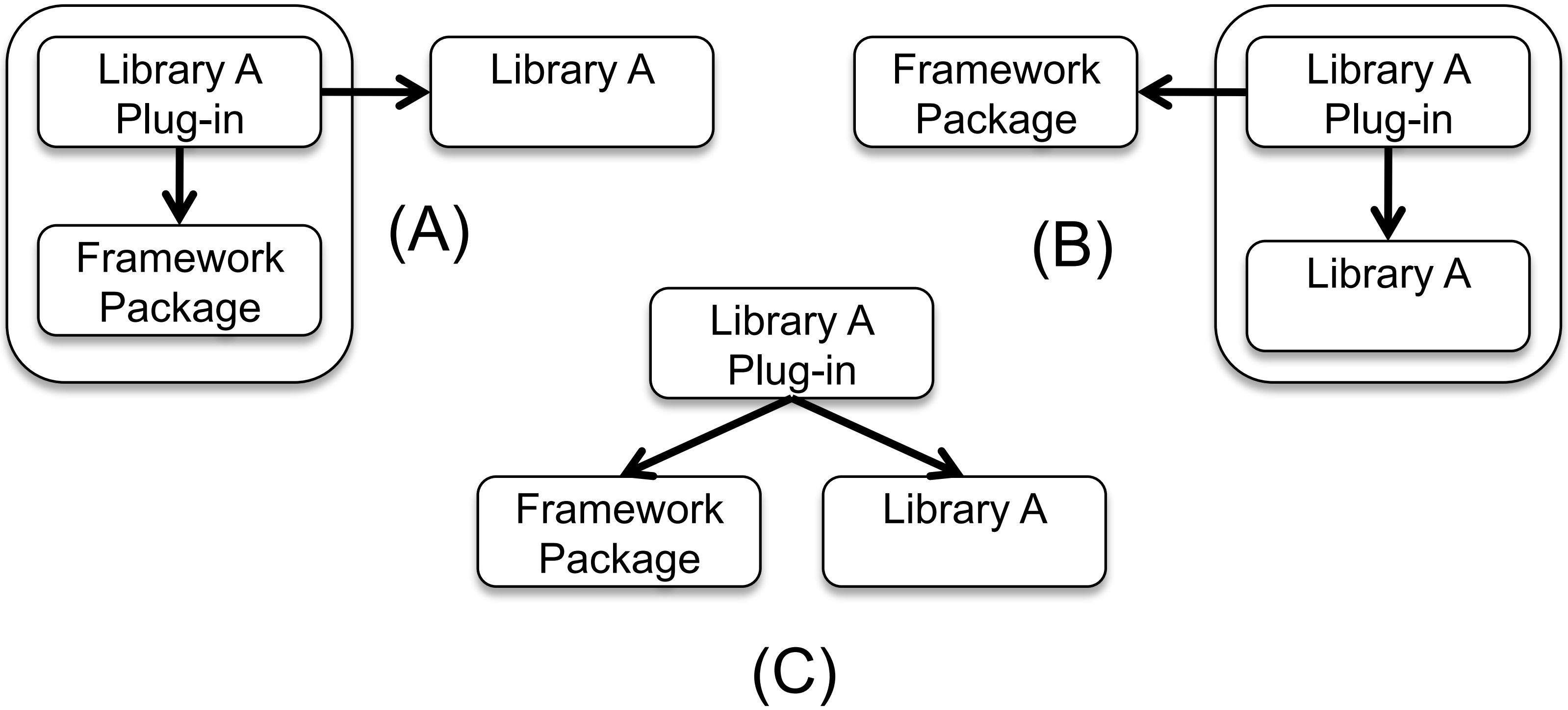
Plugins

- Program extensions to provide a certain, usually very specific function "on demand”
- Applications/frameworks support plug-ins for many reasons (in HEP)
  - to enable third-party developers to create capabilities to extend an application
  - to support features yet unforeseen
  - to reduce the size of the basic application



Plugins (2)

- At least three possibilities for packaging plug-ins
  - (C) is the one that creates less coupling
  - (A) and (B) forces a dependency between the library and the framework



Software Release

- Experiments do not release individual packages
  - Each individual package is ‘tagged’ by developer

- Experiments release complete ‘projects’
  - made of a collection of ‘tags’ for each package
  - “Tag collector” tools helping here

- Again - proper package dependency is essential for ease of release preparation

- You can even measure how good/bad you do by using dependency metrices

And now a little break...