Software Design

Prof. A. M. Calvagna

The design activity produces the software architecture (or software design)

The architecture of a software system defines the system in terms of computational components and interactions among those components. (Garlan&Shaw1996)

Design principles

- How to select modules?
- How to define module interfaces?
- How to define USE relations?

How to select modules

- A module is a self contained unit
- USE interconnections with other modules should be minimized
- PRINCIPLE:
  - maximize cohesion and minimize coupling

How to select modules&interfaces

- Distinguish between what a module does for others and how it does that (its secrets)
- Minimize flow information to clients to maximize modifiability
- The interface is a contract with clients and must be stable
- GOLDEN PRINCIPLE: information hiding (Parnas 1974)
  - define what you wish to hide and design a module around it

How to select modules

Something "larger" than a class is needed to help organize large applications: packages (higher level modules)

- Granularità dei moduli è importante

  1. What are the best partitioning criteria?
  2. What are the relationships that exist between packages, and what design principles govern their use?
  3. Should packages be designed before classes (Top down)? Or should classes be designed before packages (Bottom up)?

The Reuse/Release Equivalence Principle (REP)

THE GRANULE OF REUSE IS THE GRANULE OF RELEASE.

- ONLY COMPONENTS THAT ARE RELEASED THROUGH A TRACKING SYSTEM CAN BE EFFECTIVELY REUSED.
- THIS GRANULE IS THE PACKAGE.

The Common Reuse Principle (CRP)

THE CLASSES IN A PACKAGE ARE REUSED TOGETHER.

IF YOU REUSE ONE OF THE CLASSES IN A PACKAGE, YOU REUSE THEM ALL.

The Common Closure Principle (CCP)

THE CLASSES IN A PACKAGE SHOULD BE CLOSED TOGETHER AGAINST THE SAME KINDS OF CHANGES.

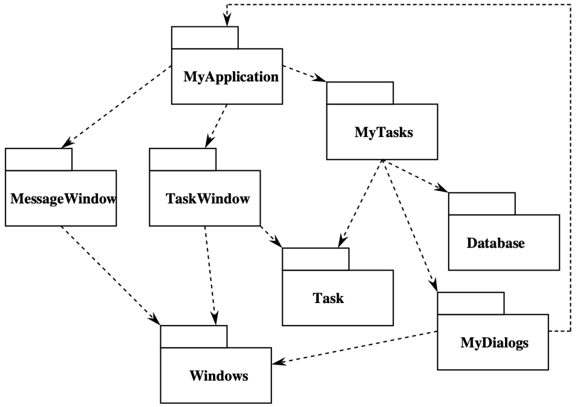
A CHANGE THAT AFFECTS A PACKAGE AFFECTS ALL THE CLASSES IN THAT PACKAGE

The Acyclic Dependencies Principle (ADP)

THE DEPENDENCY STRUCTURE BETWEEN PACKAGES MUST BE A DIRECTED ACYCLIC GRAPH (DAG).

THAT IS, THERE MUST BE NO CYCLES IN THE DEPENDENCY STRUCTURE.

Package diagram with cycles

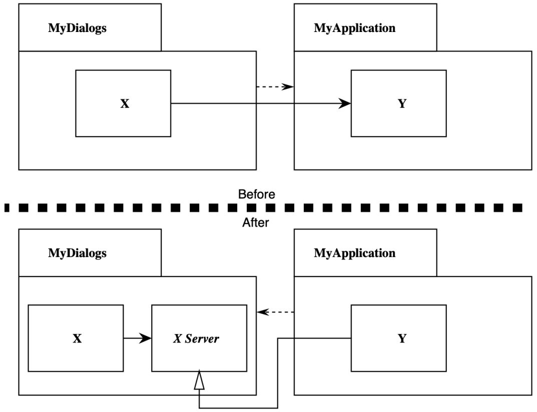


Breaking the cycle

1.Apply the Dependency Inversion Principle (DIP).

2. Create a new package that both MyDialogs and MyApplication depend upon. Move the class(es) that they both depend upon into that new package.

Dependency Inversion Principle (DIP)



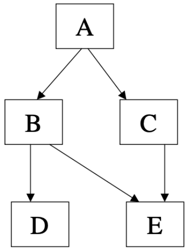
(ADAPTER PATTERN)

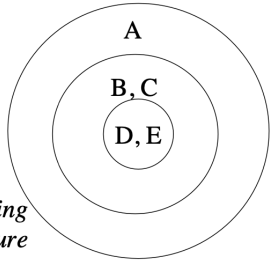
Design styles

- Shared understanding of common design forms is typical of mature engineering fields
- Shared vocabulary of design idioms is codified in engineering handbooks
- Software is going in this direction
  - but there is less maturity

Layered system

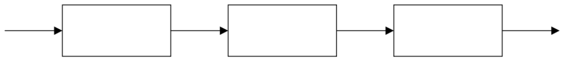
- The system is organized through abstraction levels, as a hierarchy of abstract machines
- Hierarchy is given by the USE relation





the onion-ring structure

Pipes&filters

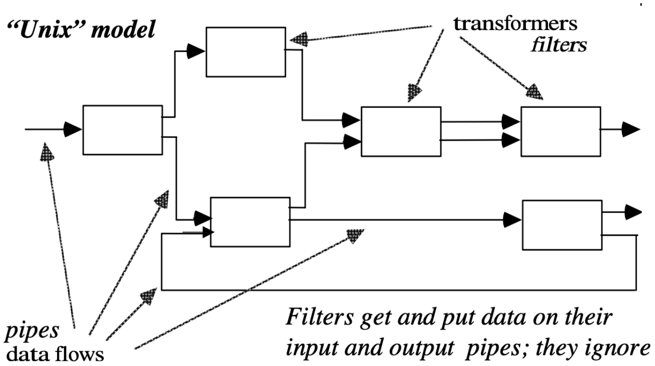


This is a pipeline

- Various control regimes are possible
  - sequential batch
  - concurrent

Pipes&filter style

The "Unix" model



Filters get and put data on their input and output pipes; they ignore the existence (and identity) of other filters.

Pipes&filters

+ compositional
  - overall behavior as composition of individual behaviors

+ reuse oriented
  - any two filters can be put together in principle

+ modifications are easy
  - can add/replace filters

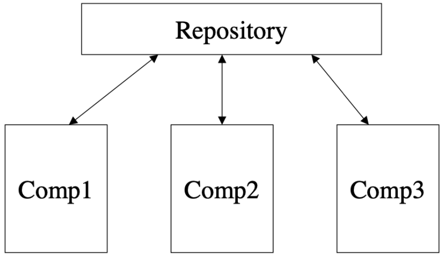
– no persistency

– replications

– tendency to batch organization

Repository-based systems

- Components communicate only through a repository



Repository

Il Repository è di dati: è uno storage, anche remoto

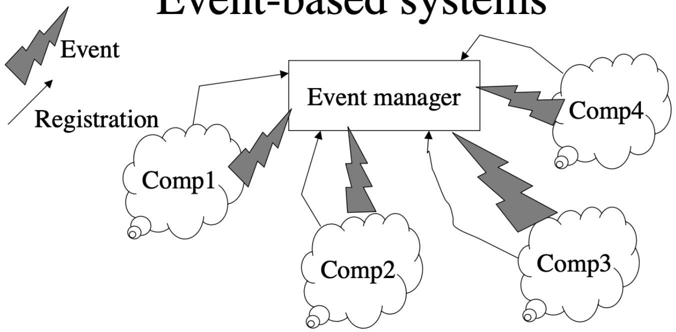
Modello **database**: è un repository passivo. Ha un componente aggiuntivo (dBMS) che sa gestire le transazioni garantendo la coerenza interna

Modello **blackboard**: è un repository attivo. Alle modifiche allo stato del repository (causate dalle scritture) possono corrispondere attivazioni di altri componenti (letture-scritture).

Blackboard:

Known Uses => **No deterministic solution** strategy known. Several subsystems assemble their knowledge to build a possibly partial or approximate solution.

Event-based systems



- Events are broadcasted to all registered components
- No explicit naming of target component

Event-based systems

Facilita le strategie **d'integrazione**

- i componenti sono tra loro privi di dipendenze dirette
- Posso rimuovere o aggiungere componenti facilmente

Può avere problemi di **scalabilità**

i componenti sono **asincroni** tra loro: l'ordine degli eventi non è garantito

- Perché il manager li possa rimbalzare nell'ordine di generazione, gli originatori dovrebbero marcarli e avere gli orologi perfettamente sincronizzati tra loro: non è una cosa banale.