

Software Architecture

Carlos Noguera,
Kennedy Kambona, Janwillem Swalens
2013-2014



This course

- Material on-line in pointcarré (Software Architecture Course)
 - Slides
 - Planning
 - Assignments
- Additional material
 - Articles in Pointcarré
 - Book: L. Baas, P. Clements, R. Kazman. *Software Architecture in Practice*. Addison-Wesley

This course

- Theory and Practice sessions
- Five exercises
- Final (written) exam

Towards Software Architecture

Programming

In the Small

- Single developer
- Understanding of all aspects
- Effort is in solving the problem

In the Large

- Large team
- No single person knows all
- Effort is in communication of information about the project

Programming vs Software Engineering

- Simple applications
 - Short life span
 - Few (one) stakeholders
 - One-off development
 - Build from scratch
 - Minimal maintenance
- Complex systems
 - Long life span
 - Multiple stakeholders
 - System families
 - Reuse
 - 60% of cost is maintenance

Programming vs Software Engineering

- | | |
|--------------------------|------------------------------|
| ● Simple applications | ● Complex systems |
| ● Short life span | ● Long life span |
| ● Few (one) stakeholders | ● Multiple stakeholders |
| ● One-off development | ● System families |
| ● Build from scratch | ● Reuse |
| ● Minimal maintenance | ● 60% of cost is maintenance |

Complexity and Cost

Coping with Complexity

Abstraction

- Languages
- Middleware
- Models

Tools

- IDEs
- SCM
- Bug trackers

Process

- Life Cycle Models
- Software Project Management

Coping with cost (Reuse)

- Less development time
- Less cost
- Improve reliability

Requirements

Design fragments

Code fragments /
modules

Beyond artifact reuse

Capture a general solution to a re-occurring problem

- Program idioms
 - Design Patterns
 - Architectural Patterns
- Iterate a collection
- Model-View Controller
- Client/server

Why Architecture

- ✓ Consider the wishes of the landlord
- ✓ Make buildings solid so that they last long, survive rain, earthquakes, etc.
- ✓ Modify buildings while keeping stability and other properties
- ✓ Plan buildings in a way that they are extendable in the future
- ✓ Keep infrastructure in mind: streets, power lines, ...
- ✓ Make buildings easy to operate: make repairs easy by making parts accessible and interchangeable
- ✓ Make building comfortable, cool in summer, easy to heat in winter
- ✓ Make building with as little cost as possible (to build and to maintain)
- ✓ Guide workers who will actually build it
- **Architecture: collect knowledge on best practices, materials, architectural styles, patterns, idioms, ...**



Why Software Architecture?

- ✓ Consider the wishes of customer
- ✓ Make software stable so that it can cope with errors and failures
- ✓ Modify software while keeping stability and other properties
- ✓ Design software in a way that it can evolve in the future
- ✓ Keep infrastructure in mind: network, storage, cooperating systems, ...
- ✓ Make software easy to operate: make repairs easy by making components exchangeable without affecting other components
- ✓ Make software easy to install, administer and use
- ✓ Make software with as little cost as possible (to build and to maintain)
- ✓ Guide programmers who will implement it
- **Software Architecture: collect knowledge on best practices, technology, architectural styles, patterns, idioms, ...**

Is it engineering yet?

In the beginning: Implicit, anonymous,
accidental software architectures

Goal

- Communication
- Productivity
- Quality

Obstacles

- Legacy systems, people
- Changes in technology
- Organization

Now: Software architecture as an emerging
discipline

Software Architecture

where does it come from?

Requirements and influences in SA

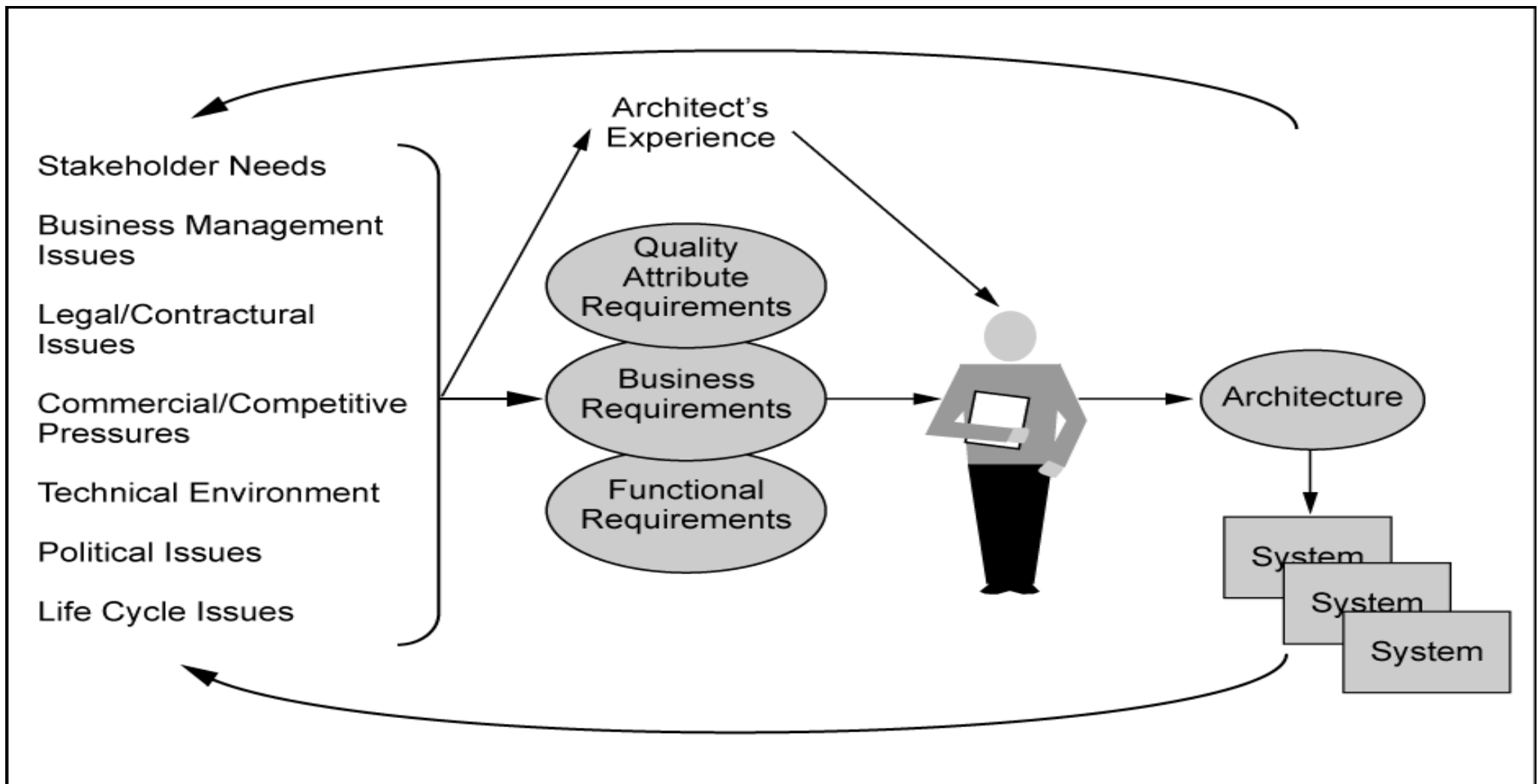
Stakeholders

- End Users
- Costumer

Organization

Technical
Environment

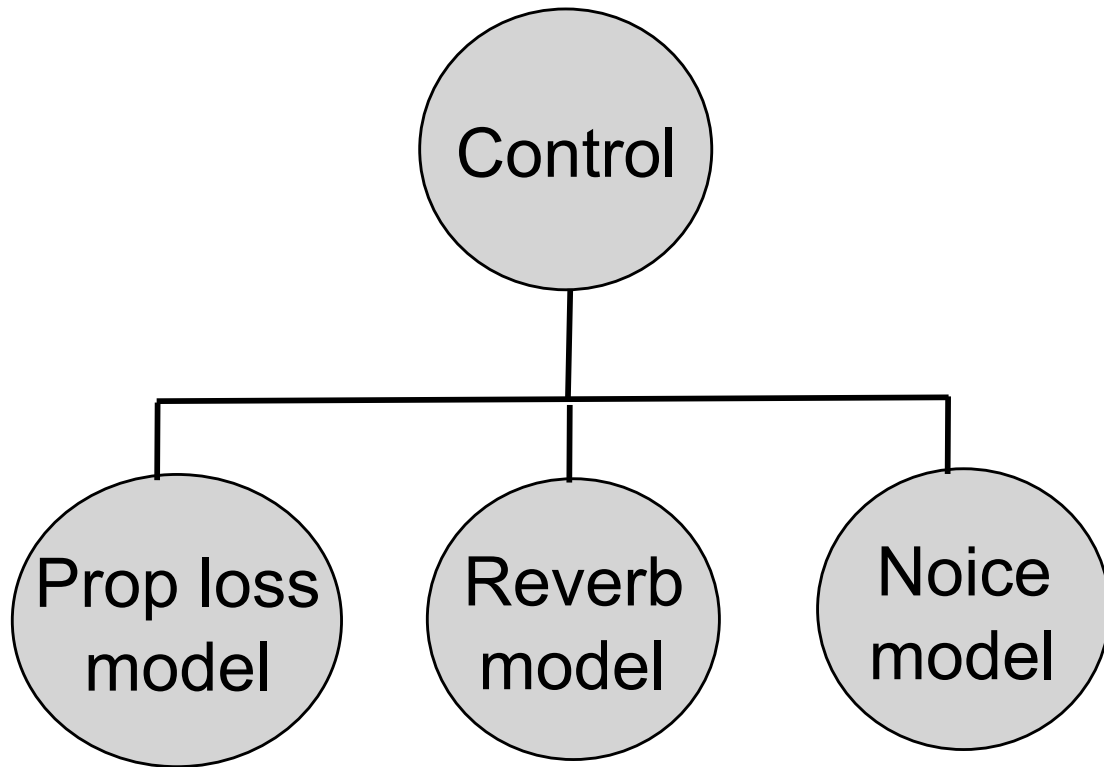
Architect's
knowledge



Software Architecture

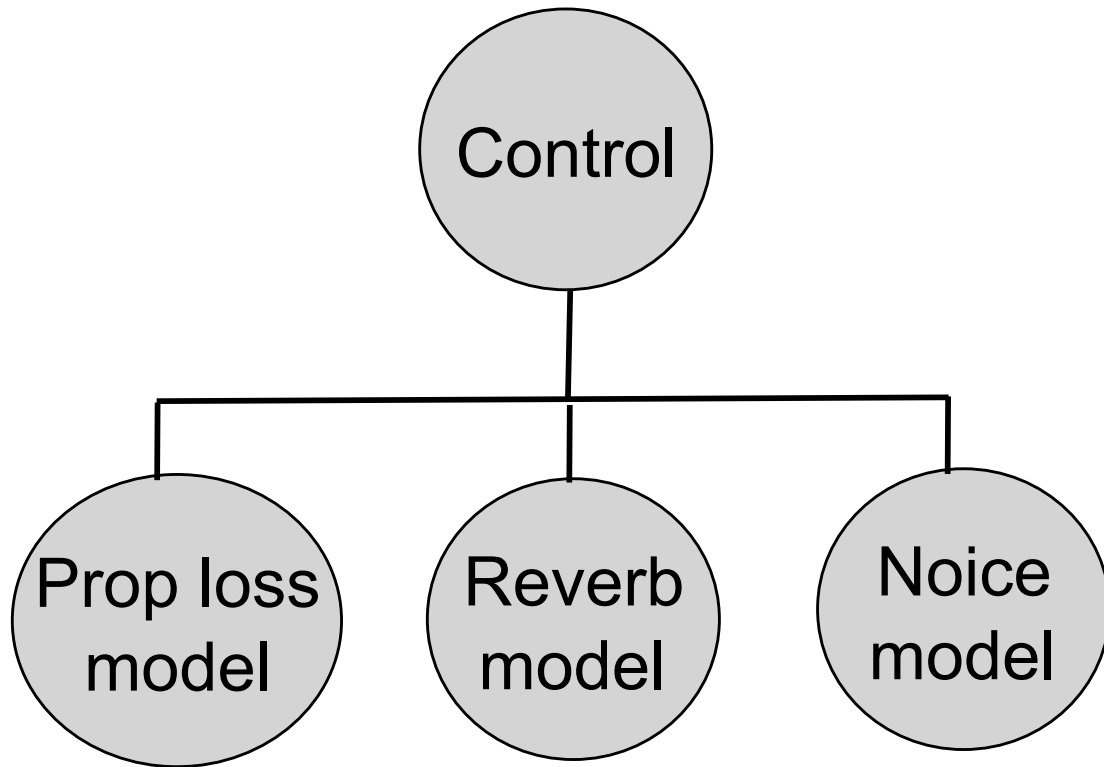
what is it, actually?

What does this mean?

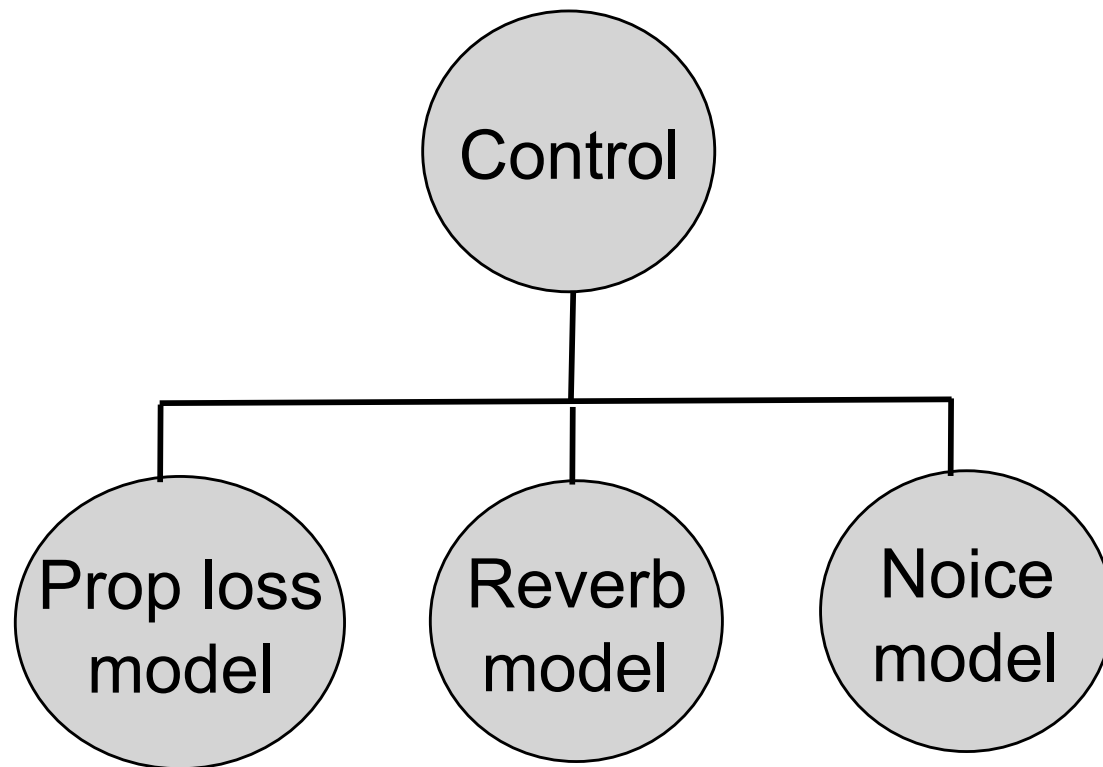


What does this mean?

- 4 elements

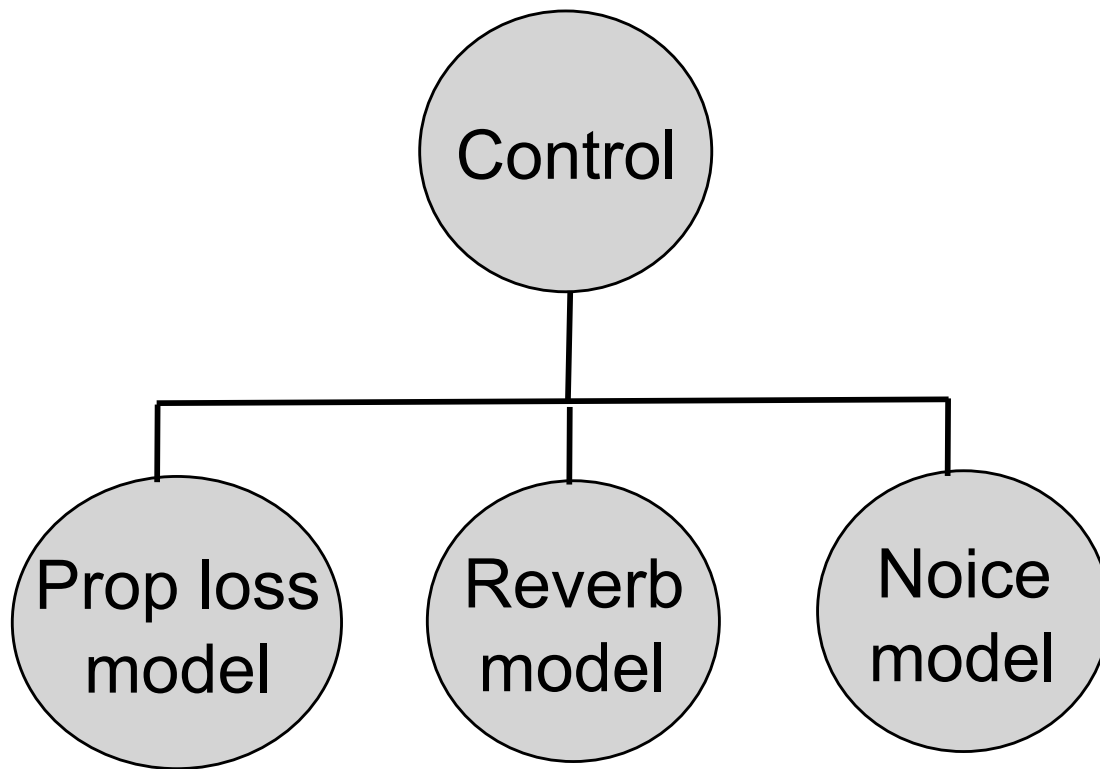


What does this mean?



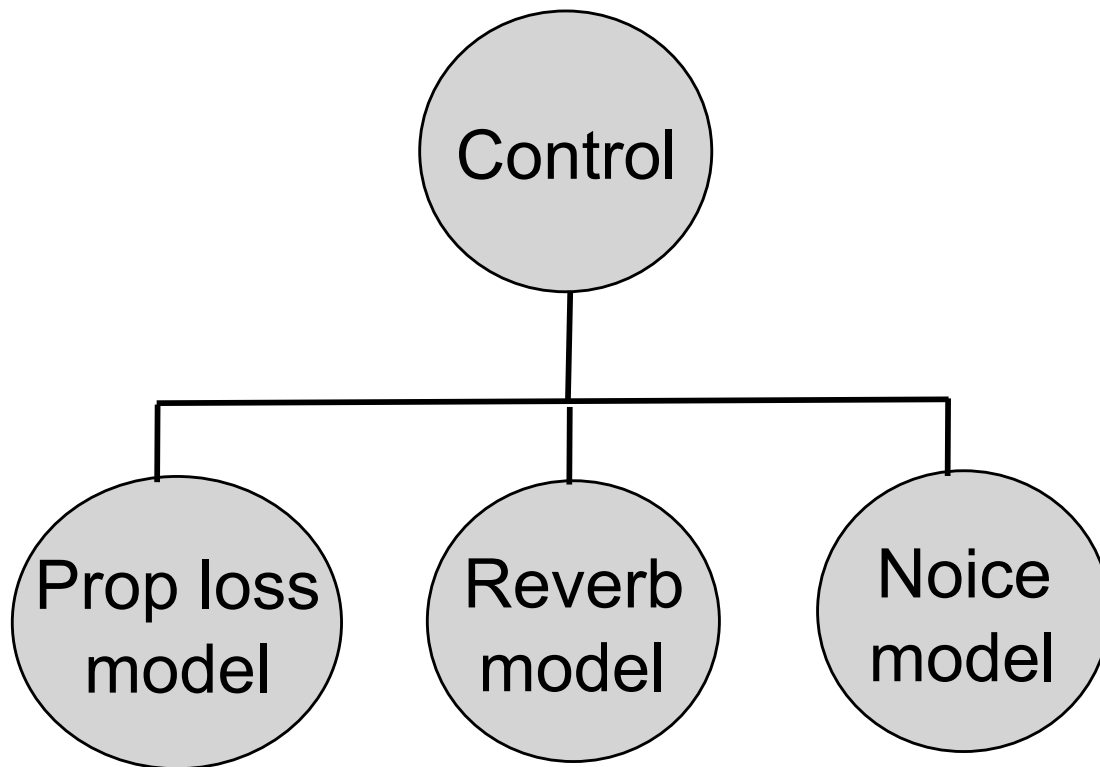
- 4 elements
- All related

What does this mean?



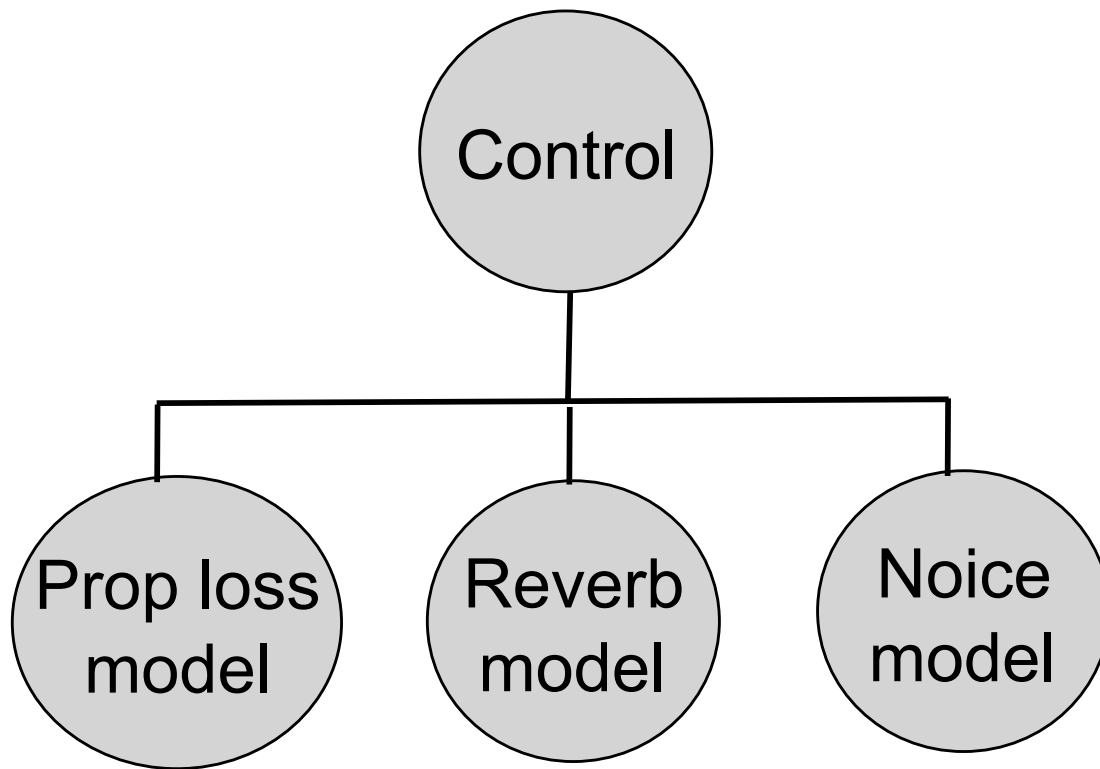
- 4 elements
- All related
- 3 elements have more in common

What does this mean?



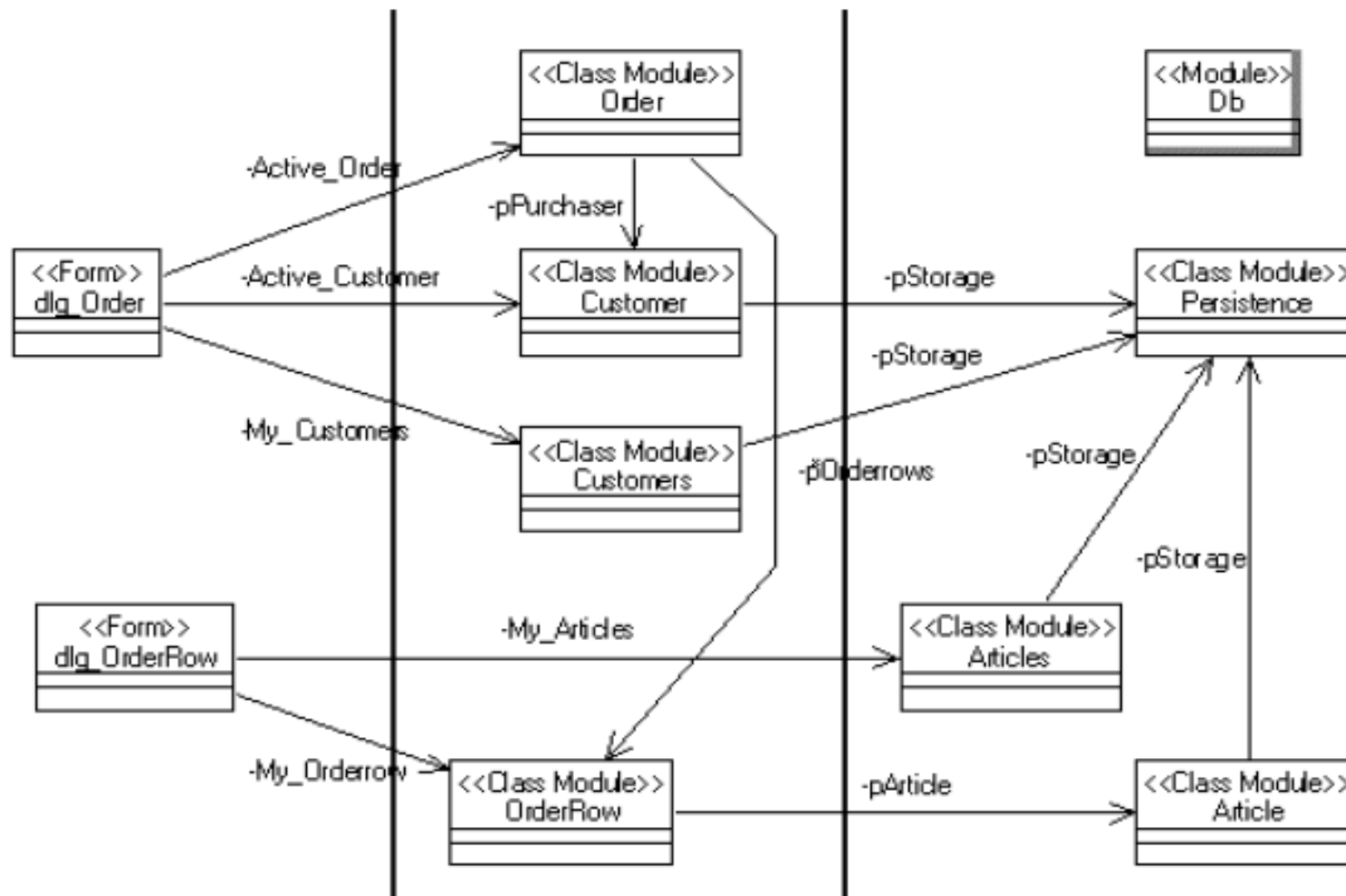
- 4 elements
- All related
- 3 elements have more in common
- Place

What does this mean?



- 4 elements
- All related
- 3 elements have more in common
 - Place
 - Name

Drawings everywhere



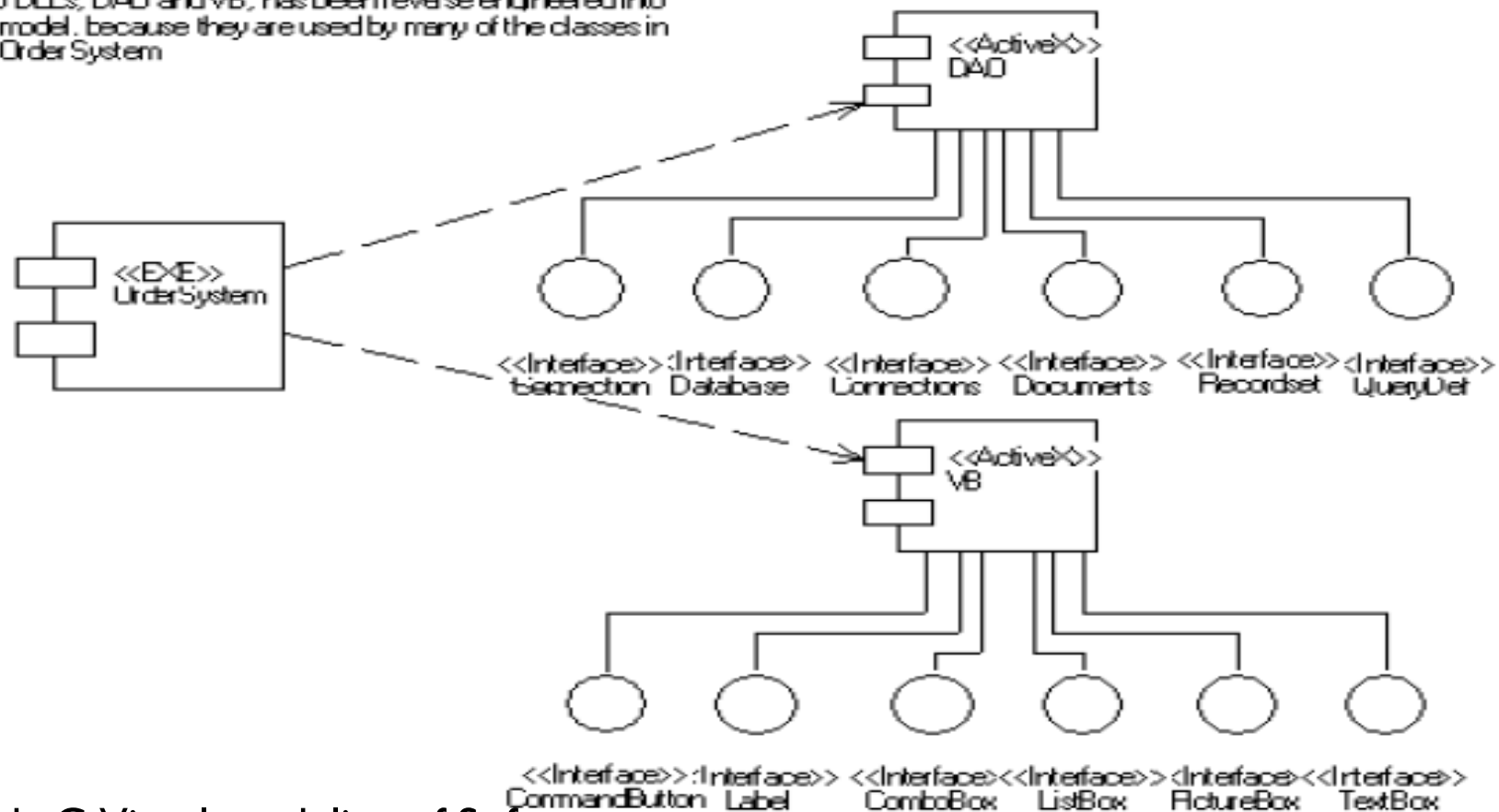
From Booch, G. Visual modeling of Software
Architecture for the enterprise. MSDN
Home, 1998

Drawings everywhere

Order System Components

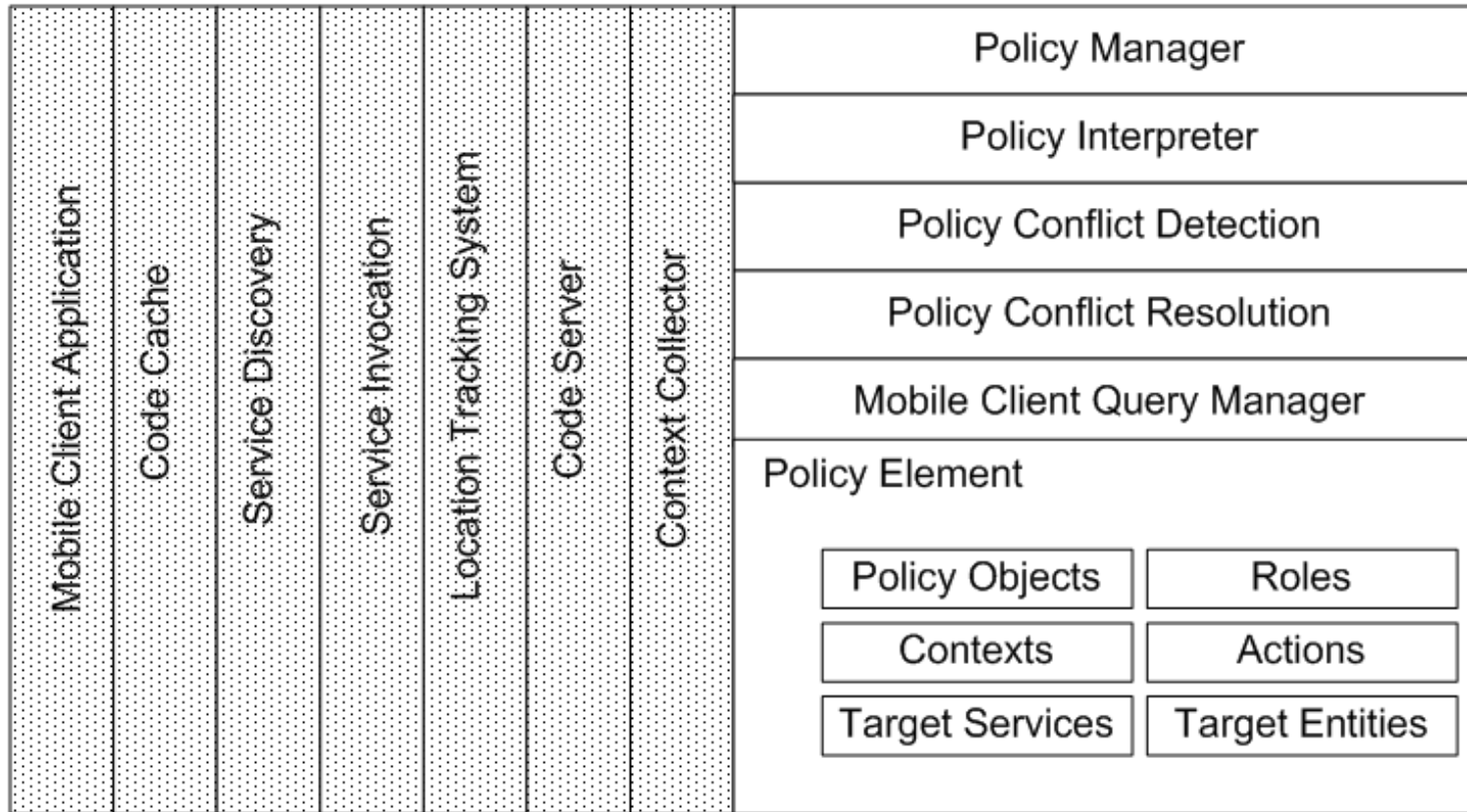
The Order System is implemented as one single component an executable called OrderSystem, which corresponds to the ORDERSYS.VBP project.


Two DLLs, DAD and VB, has been reverse engineered into the model, because they are used by many of the classes in the Order System



From Booch, G. Visual modeling of Software
Architecture for the enterprise. MSDN Home,
1998

Drawings everywhere

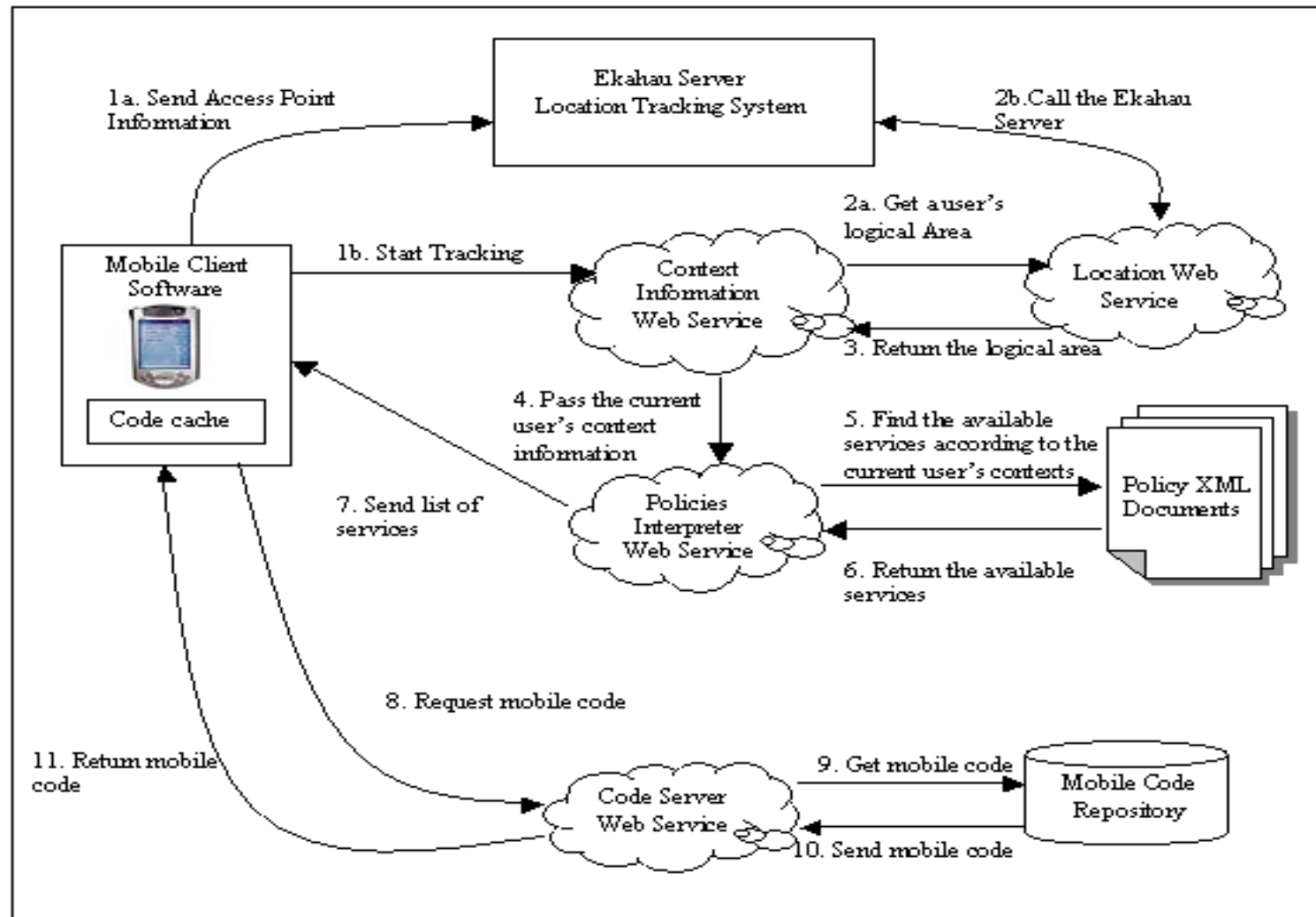


 = Context awareness software components

 = Policy software components

From Syukur and Loke, Policy Based Control of Context-Aware Pervasive Services. Journal of Ubiquitous computing and intelligence

Drawings everywhere



From Syukur and Loke, Policy Based Control of Context-Aware Pervasive Services. Journal of Ubiquitous computing and intelligence

from drawing to architecture

Nature of elements

- Separation
- Semantics
- Representation

Responsibilities of elements

- What do they do?

Nature of Connections

- Communication
- Mechanisms
- Information flow

Nature of Layout

- Levels
- Containment
- Esthetic

Definitions



Software architecture is a level of design that involves:

- The description of elements from which the system is build
- Interactions amongst these elements
- Patterns that guide their compositions
- And constraints on these patterns

Software architecture involves:

- The structure and organization by which modern system components and subsystems interact to form systems, and
- the properties of systems that can best be designed and analyzed at the system level

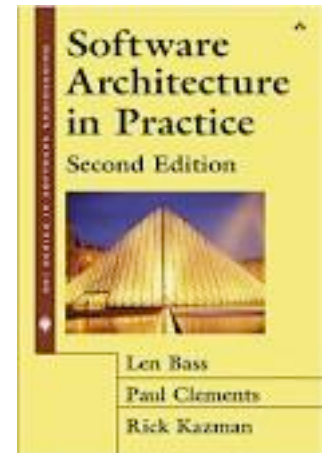


Definitions



"fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution"

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."



What does this mean?

- ✓ Architecture defines software elements and embodies information on how they relate. So architecture is an abstraction, details that are not relevant for element interaction are omitted.
- ✓ Systems can comprise more than one structure and non of these can claim to be "the" architecture. E.g. one structure can focus on how functionality is divided up and eventually assigned to an implementation team while another structure can focus on the way elements interact with each other at run time. In a distributed or parallel setting a third structure can specify order of execution and synchronization.
- ✓ Every computing system has an architecture although that architecture may remain unknown or undocumented
- ✓ The behavior of each element is part of the architecture in so far that it can be observed or discerned from the point of view of another element

Minority report



- ✓ **Architecture is high level design.** But they are not interchangeable. For example deciding on important data structures that will be encapsulated is not an architectural concern
- ✓ **Architecture is the overall structure of the system.** Ignores the fact that multiple structures co-exist
- ✓ **Architecture is the structure of the components, their interrelationships and the principles and guidelines governing their design and evolution.** Although it may be good professional practice to document the rationale for architectural decisions, they are not part of the architecture.
- ✓ **Architecture is components and connectors.** Connectors imply a run time mechanism for transferring control and data around in the system. Relationships are more general and can capture run-time but also non run-time relationships

Importance of an architecture

- ✓ Serves as a common abstraction in the communication amongst stakeholders: understanding, negotiation, consensus, ...
- ✓ Manifests the earliest set of design decision:
 - Constrains the implementation
 - Dictates organizational structure (of the development project)
 - Inhibits or enables System Quality Attributes
 - Allows predicting system qualities
 - Helps to manage change
 - Enables more accurate cost and schedule estimates
- ✓ Is a transferable re-usable model
 - Software product lines share an architecture
 - Systems can be built using large, externally developed components
 - An architecture can be the basis for training

Components of an architecture

Structures and Views

- ✓ Complex systems are difficult to grasp at once. At any one moment in time attention should go to a limited number of the system's structures
- ✓ To communicate meaningfully it is necessary to make explicit which structures are being discussed - i.e. which view is taken
- ✓ In Bass, Clements and Kazman 3 groups of architectural structures are distinguished: Module structures - Component and connector structures - Allocation structures
- ✓ The ontology established in ANSI/IEEE 1471-2000 mentions 7 viewpoints: Functional/logic view - Code/module view - Development/structural view - Concurrency/process/thread view - Physical/deployment view - User action/feedback view - Data view"
- ✓ Kruchten from Rational advocates the 4+1 view: Logical view - Process view - Development view - Physical view + Scenarios

Architectural Structures and views (bass et al.)

✓ **Module structures**

- Elements are units of implementation: a code based way of considering a system
- Modules have functionality assigned to them and are often hierarchically decomposed in module/submodules relations
- Modules are further related by uses and used-by relations or by generalisation and specialisation relations

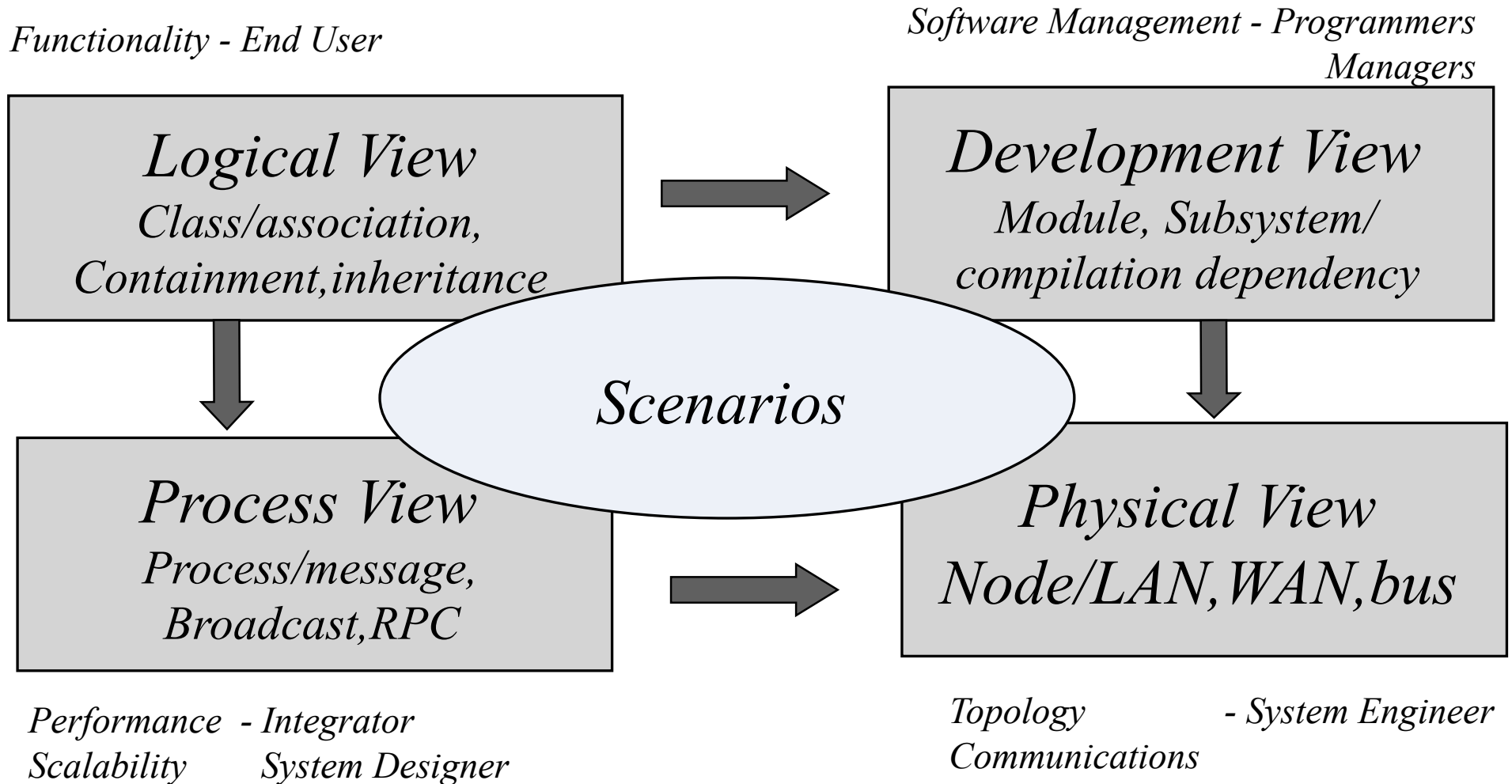
✓ **Component and connector structures**

- Elements are run-time components (units of computation): a computation based way of considering a system
- Connectors are communication vehicles between the elements
- Shows shared data stores, replicated parts, data flow, parallelism

✓ **Allocation structures**

- Shows relationships between software elements and the external environment in which the software is created and executed, i.e. on what processor does each component execute, in what file is each module stored during development, the assignment of modules to development teams, etc.

4+1 Views



Software Architecture

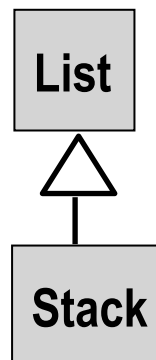
OO Design Principles

Revisiting...

OO Programming

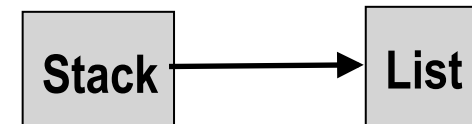
- Encapsulation
- Message passing
- Polymorphism
- Inheritance

Inheritance

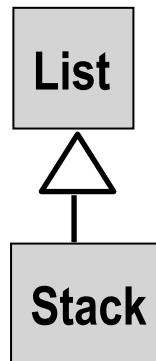


vs

Composition

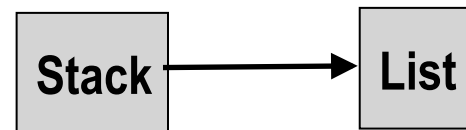


Class Inheritance



- White box
- Compile-time

Object Composition



- Black-box
- Runtime

Quality of a Design

✓ A good design strives for low coupling and high cohesion

- Coupling and cohesion were introduced in the context of modular programming

and revised later on in the context of object oriented programming.

- Coupling describes the relationships between modules or classes; cohesion describes the relationships within them.

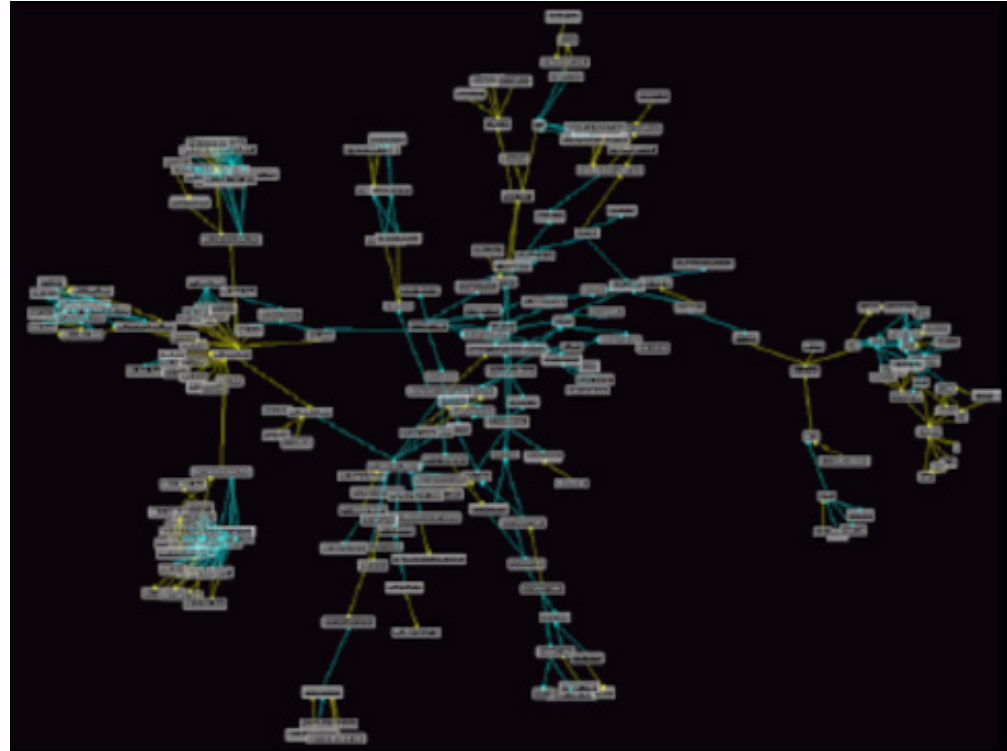
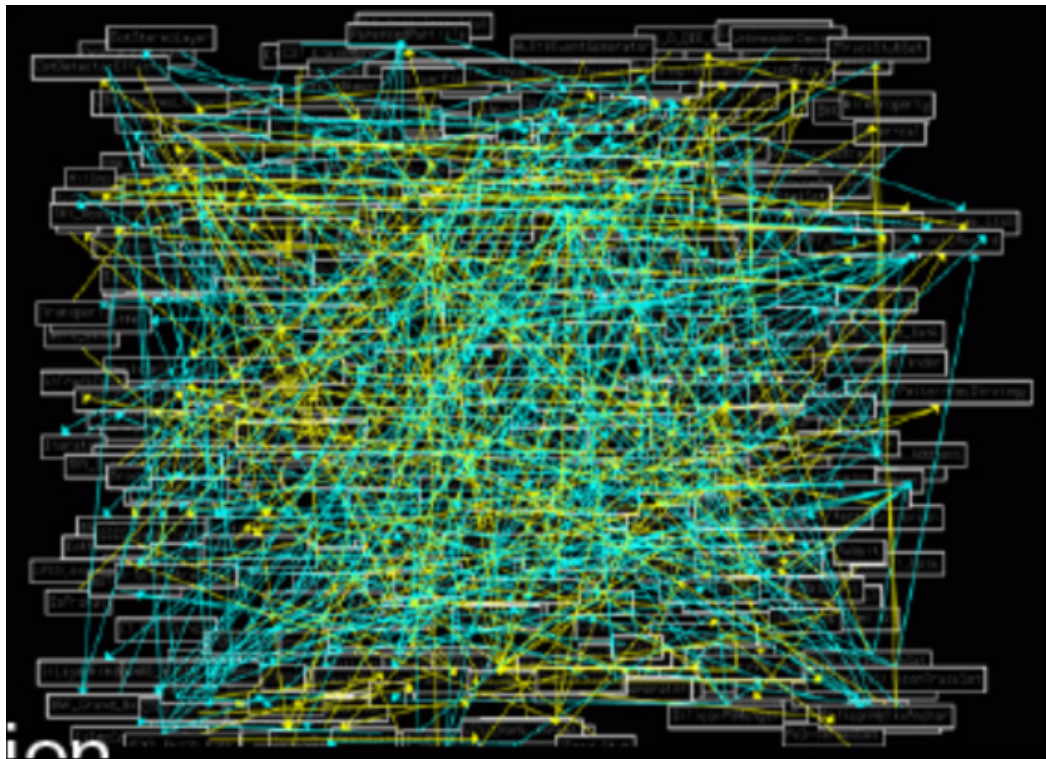
- High coupling is bad because:

- A change in one module will force a ripple effect of changes in other modules
- Assembly of modules might require more effort and/or time due to the increased inter-module dependencies
- A particular module may be harder to reuse and/or test because the dependent modules must be included

- High cohesion is good because:

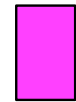
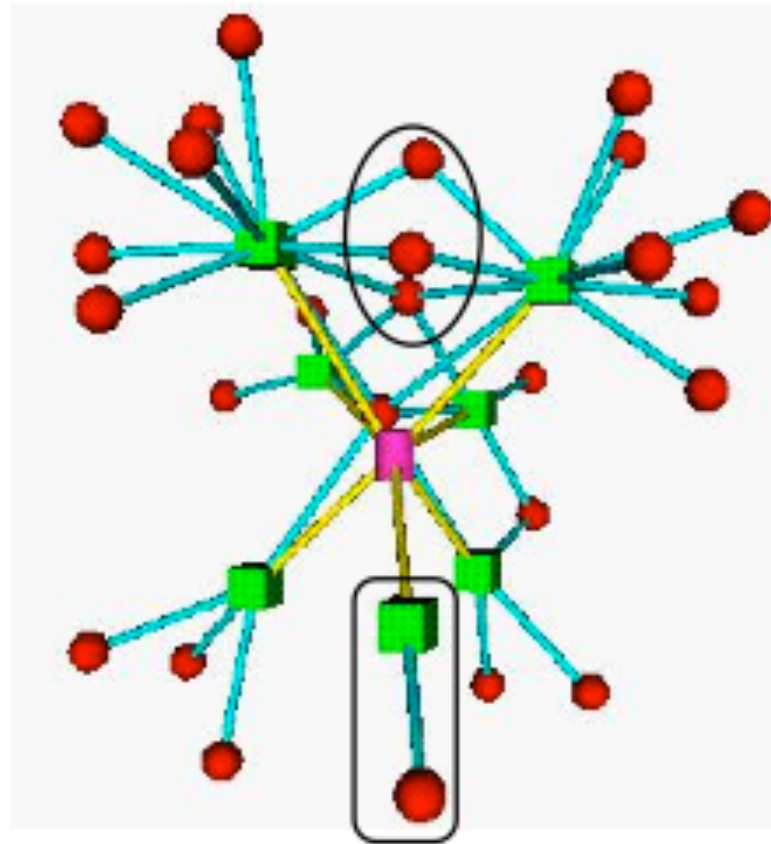
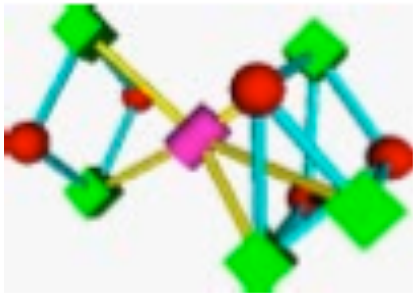
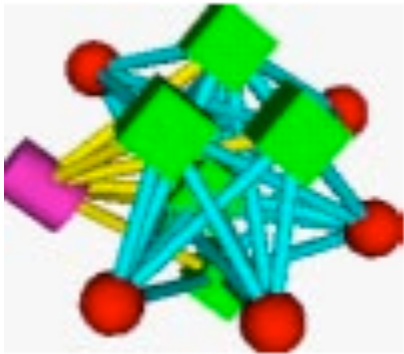
- A particular module has a clear purpose and can be understood in isolation

Coupling



Cohesion

High



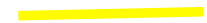
class



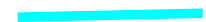
attribute



method



has



uses

Refactor to 2 classes?

Coupling

(from bad to good)

- Content
- Common
- External
- Control
- Stamp
- Data
- Message
- No coupling

One module modifies or relies on the internal workings of another module (e.g. accessing local data of another module). Therefore changing the way the second module produces data (location, type, timing) will lead to changing the dependent module.

Coupling

(from bad to good)

- Content
- Common
- External
- Control
- Stamp
- Data
- Message
- No coupling

Two modules share the same global data (e.g. a global variable). Changing the shared resource implies changing all the modules using it.

Coupling

(from bad to good)

- Content
- Common
- External
- Control
- Stamp
- Data
- Message
- No coupling

Occurs when two modules share an externally imposed data format, communication protocol, or device interface.

Coupling

(from bad to good)

- Content
- Common
- External
- Control
- Stamp
- Data
- Message
- No coupling

One module controlling the flow of another, by passing it information on what to do (e.g. passing a what-to-do flag).

Coupling

(from bad to good)

- Content
- Common
- External
- Control
- Stamp
- Data
- Message
- No coupling

When modules share a composite data structure and use only a part of it, possibly a different part (e.g. passing a whole record to a function which only needs one field of it). This may lead to changing the way a module reads a record because a field, which the module doesn't need, has been modified.

Coupling

(from bad to good)

- Content
- Common
- External
- Control
- Stamp
- Data
- Message
- No coupling

When modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data which are shared (e.g. passing an integer to a function which

Coupling

(from bad to good)

- Content
- Common
- External
- Control
- Stamp
- Data
- Message
- No coupling

This is the loosest type of coupling. Modules are not dependent on each other, instead they use a public interface to exchange parameter-less messages or events.

Coupling

(from bad to good)

- Content
- Common
- External
- Control
- Stamp
- Data
- Message
- No coupling

Modules do not communicate
at all with one another.

Cohesion

(from worst to best)

- **Coincidental**
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- Functional
- ADT

Parts of a module are grouped arbitrarily (at random); the parts have no significant relationship .

Cohesion

(from worst to best)

- Coincidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- Functional
- ADT

Parts of a module are grouped because they logically are categorized to do the same thing, even if they are different by nature (e.g. grouping all I/O handling routines, grouping all mathematical functions).

Cohesion

(from worst to best)

- Coincidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- Functional
- ADT

Parts of a module are grouped by when they are processed - the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).

Cohesion

(from worst to best)

- Coincidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- Functional
- ADT

Parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file).

Cohesion

(from worst to best)

- Coincidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- Functional
- ADT

Parts of a module are grouped because they operate on the same data (e.g. a module which operates on the same record of information).

Cohesion

(from worst to best)

- Coincidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- Functional
- ADT

Parts of a module are grouped because the output from one part is the input to another part (e.g. a function which reads data from a file and processes the data).

Cohesion

(from worst to best)

- Coincidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- Functional

Parts of a module are grouped because they all contribute to a single well-defined task of the module (e.g. parsing XML).

OO Design Patterns refresher



Design Patterns



- Collect and Characterize recurring architectures
 - Provide solution to a problem
 - Common language
 - Tend to be small (large ones exist)
- No immediate implementation
 - Partial implementation
 - Smaller than a Framework

A Pattern Language: Towns, Buildings, Construction (1977) Christopher Alexander

Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, Helm, Johnson, Vlissides)

Elements of a Pattern

- Name
- Problem
 - Conditions of applicability
- Solution
 - Elements (classes, objects), Roles, Responsibilities
 - No concrete design, implementation
- Consequences
 - Tradeoffs
 - Implementation issues

Kinds of Patterns

Creational Patterns:

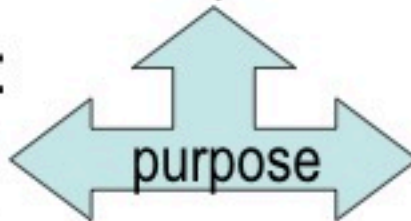
are concerned with the process of object creation

Structural Patterns:

are concerned with how classes and objects are composed to form larger structures

Behavioural Patterns:

are concerned with algorithms and the assignment of responsibilities between objects



Class Patterns deal with static relationships between classes and subclasses



Object Patterns deal with object relationships which can be changed at run time

Overview

Creational Patterns

- Singleton
- Abstract factory
- Factory Method
- Prototype
- Builder

Structural Patterns

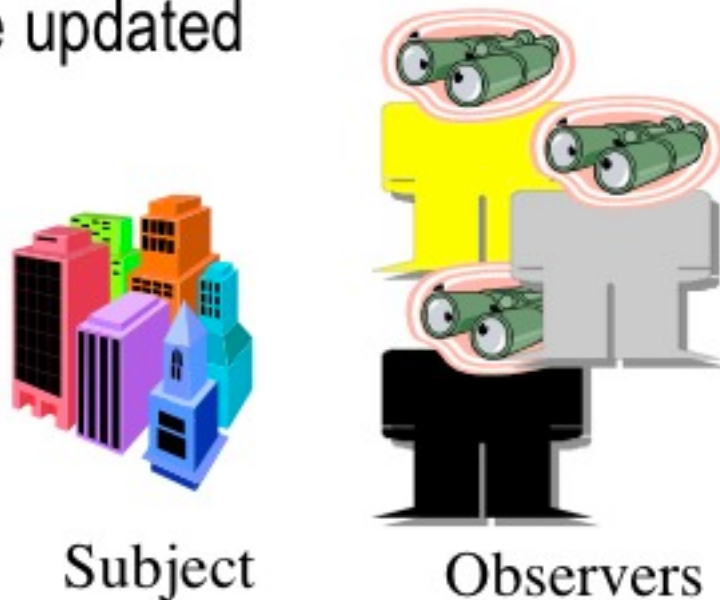
- Composite
- Façade
- Proxy
- Flyweight
- Adapter
- Bridge
- Decorator

Behavioral Patterns

- Chain of Respons.
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Observer problem

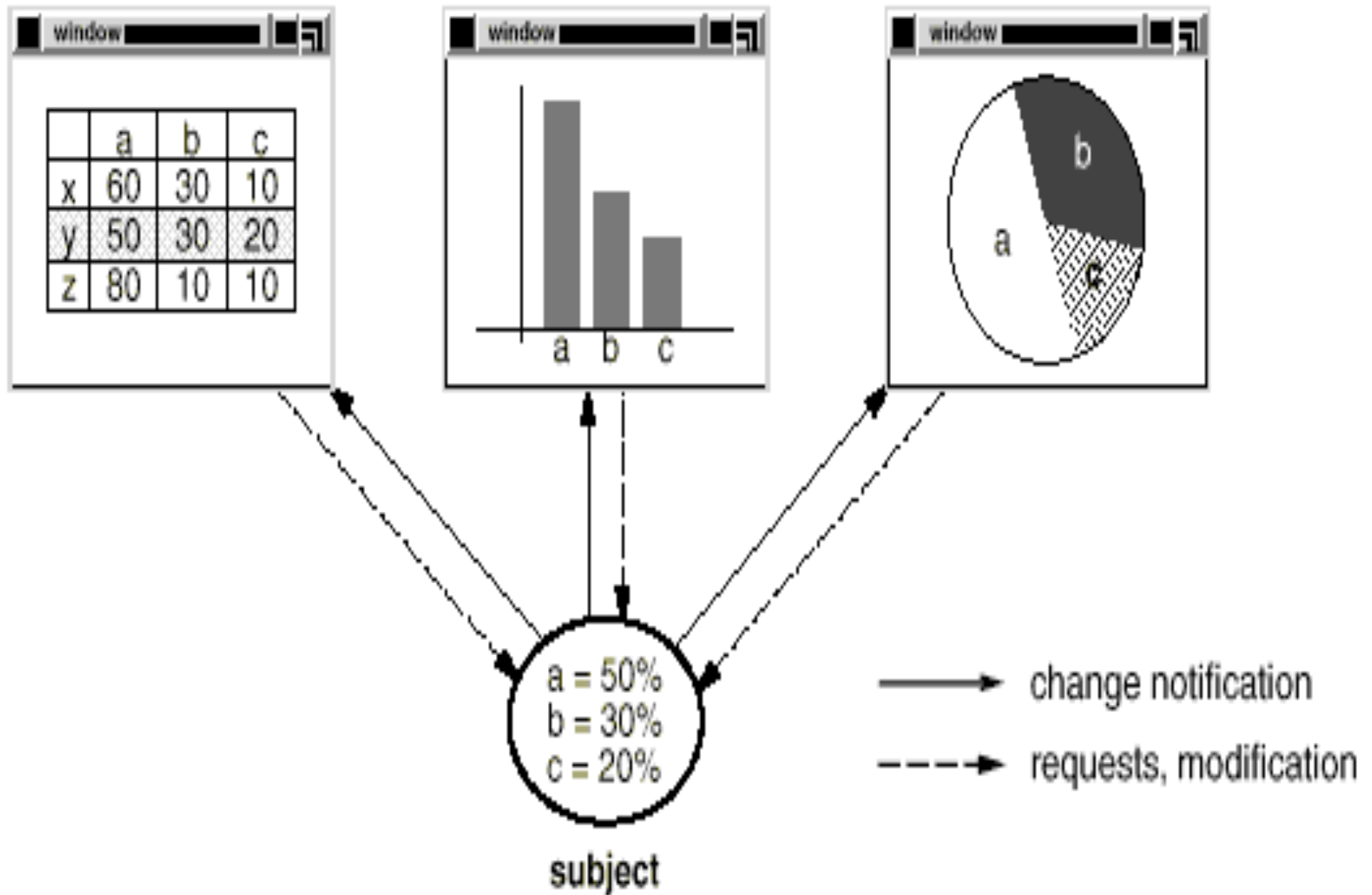
Assume a one to many relationship between objects, when one changes the dependents must be updated



- different types of GUI elements depicting the same application data
- different windows showing different views on the same application model

Also known as : Dependants, Publish-Subscribe

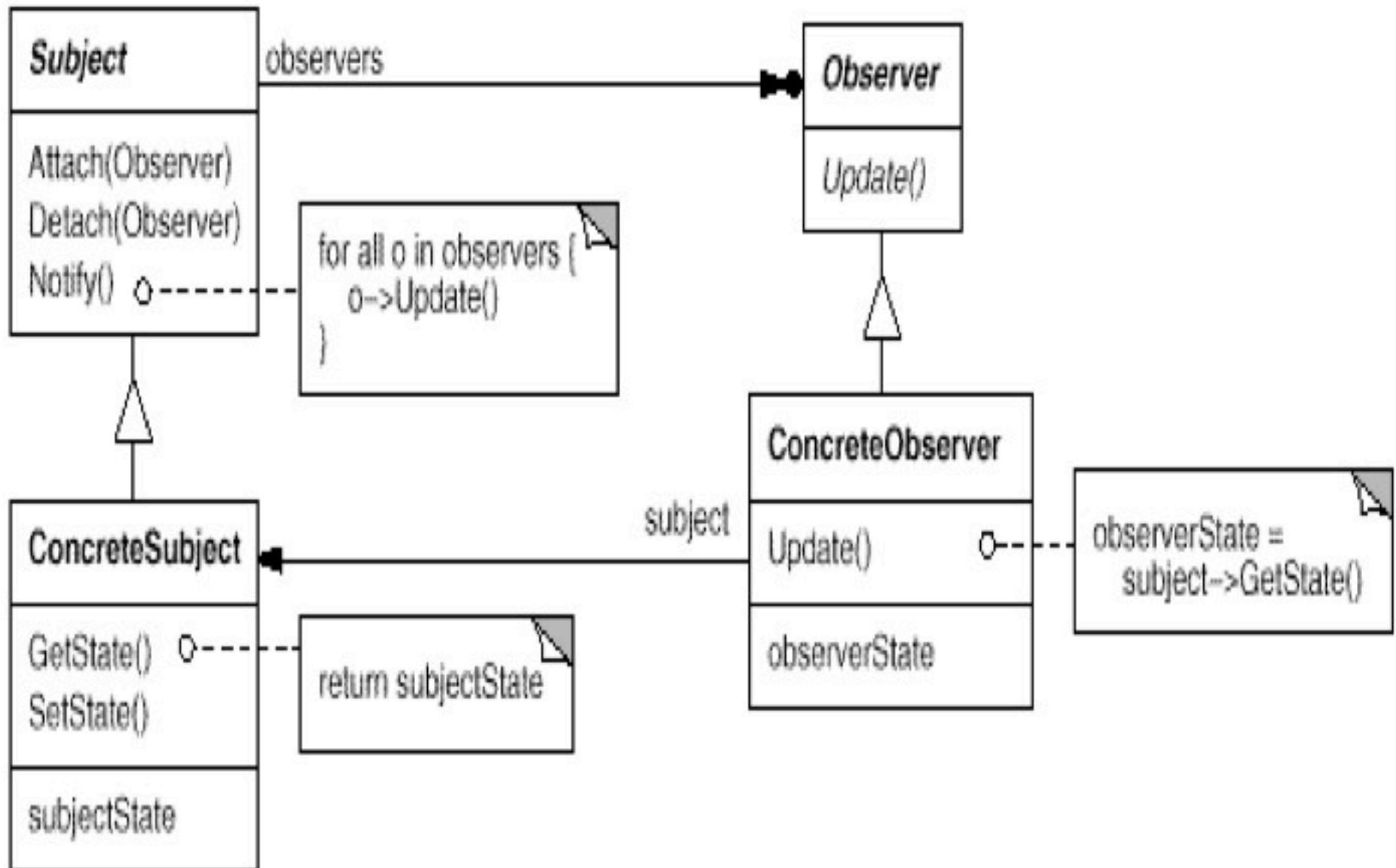
observers



Observer participants

- **Subject:** knows its observers, provides an interface for attaching (subscribe) and detaching (unsubscribe) observers and provides a *notify* method that calls *update* on all its observers
- **Observer:** provides an *update* interface
- **ConcreteSubject:** maintains a state relevant for the application at hand, provides methods for getting and setting that state, calls *notify* when its state is changed
- **ConcreteObserver:** maintains a reference to a concrete subject, stores a state that is kept consistent with the subject's state and implements the observer's *update* interface

Observer



Software Quality and Architecture

Functional Requirements

- ✓ Functional requirements specify what functions a system must provide to meet stated and implied stakeholder needs when the software is used under specific conditions.
- ✓ Examples:
 - The system shall allow users to buy and sell securities.
 - The system shall allow users to review account activity.
 - The system shall monitor and record inputs from meteorological sensors.
 - The system shall notify operators of reactor core temperature changes.
 - The system shall compute and display the orbit and trajectory for all satellites.

Design Constraints

✓ Design constraints are decisions about a system's design that must be incorporated into any final design of the system. They represent a design decision with a predetermined outcome.

✓ Examples:

- Oracle 8.0 shall be used for persistent storage.
- System services must be accessible through the World Wide Web.
- The system shall be implemented using Visual Basic.
- The system shall only interact with other systems via Publish/Subscribe.
- The system shall run on both Windows and Unix platforms.
- The system shall integrate with legacy applications.

Quality Attributes

✓ Quality attribute requirements are requirements that indicate the degrees to which a system must exhibit various properties.

✓ Examples:

- buildability: The system shall be buildable within six months.
- availability: The system shall recover from a processor crash within one second.
- portability: The system shall allow the user interface (UI) to be ported to a new platform within six months.
- performance: The system shall process sensor input within one second.
- security: The system shall deny access to unauthorized users 100% of the time.
- testability: The system shall allow unit tests to be performed within three hours with 85% path coverage.
- usability: The system shall allow users to cancel an operation within one second.
- capacity: The system shall have a maximum of 50% CPU utilization.

Quality properties



A word cloud of quality properties. The words are arranged in a roughly circular pattern, with some words being larger and more prominent than others. The colors of the words include shades of blue, green, yellow, orange, and brown. The words are: Scalability, Evolvability, Modifiability, Reliability, Robustness, Portability, Reusability, Interoperability, Testability, Heterogeneity, Availability, Understandability, Performance, Correctness, Verifiability, Usability, and Security.

Scalability
Evolvability
Modifiability
Reliability
Robustness
Portability
Reusability
Interoperability
Testability
Heterogeneity
Availability
Understandability
Performance
Correctness
Verifiability
Usability
Security

System Quality Attributes

- ✓ Performance
- ✓ Availability
- ✓ Usability
- ✓ Security

End User's
view

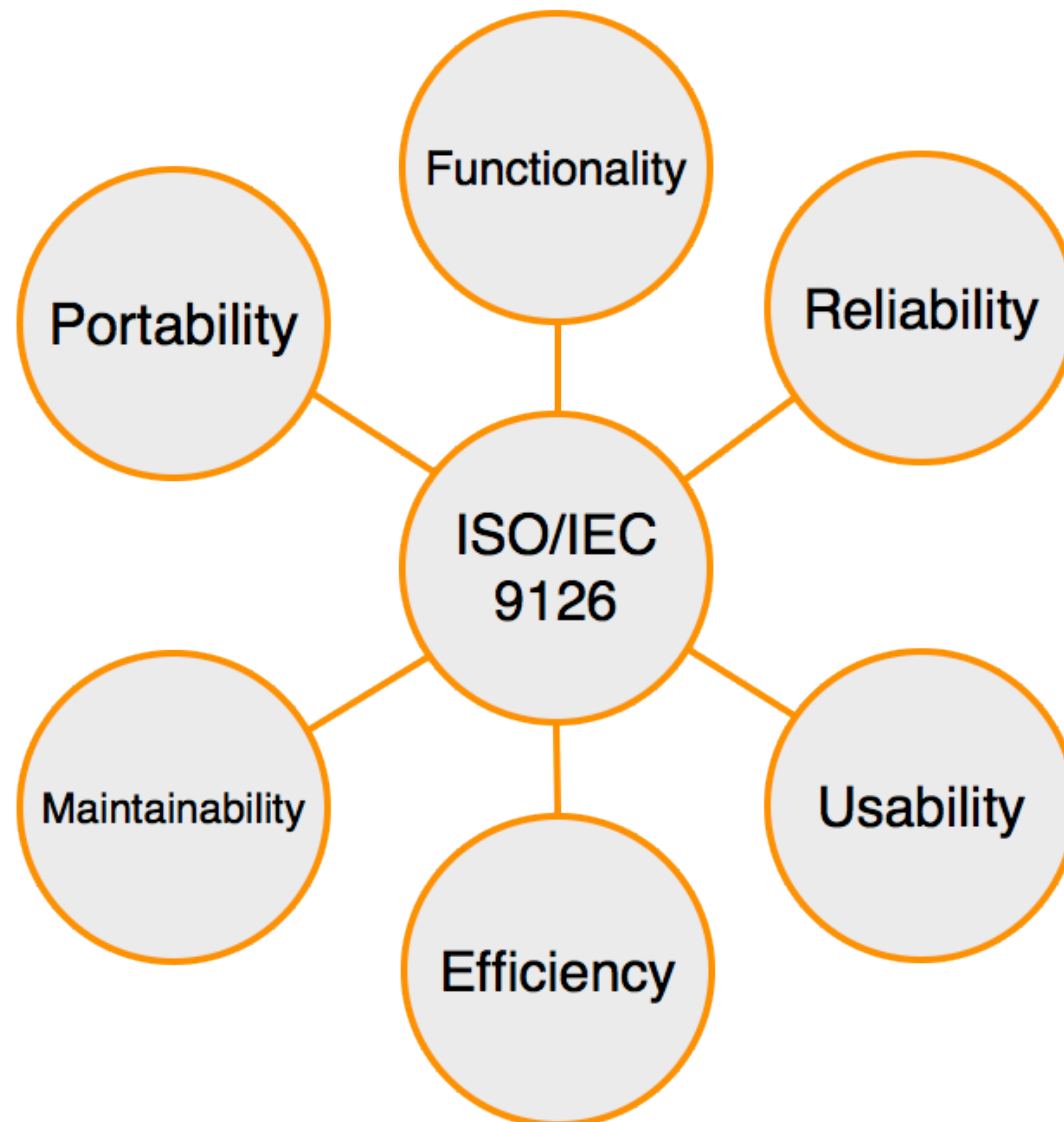
- ✓ Maintainability
- ✓ Portability
- ✓ Reusability
- ✓ Testability

Developer's
view

- ✓ Time to market
- ✓ Cost and Benefits
- ✓ Targeted Market
- ✓ Integration with Legacy System

Business
Community
view

ISO 9126 Software Quality Model



Ambiguous and Conflicting Quality Attributes

- Ambiguous categorization
 - User authentication is a functional or a quality requirement?
 - Interoperation with legacy system: design constraint or functional requirement?
- Conflicting
 - E.g., Portability->overhead->bad performance

Achieving Quality

- Development process (quality process -> quality product)
- Tactics
 - Design decisions that influence/control a quality attribute
 - Tactics are reflected in architecture

Availability Tactics

Fault detection

- Ping/Echo
- Heartbeat
- Exception

Recovery preparation and Repair

- Voting
- Active Redundancy
- Passive Redundancy
- Spare

Recovery and Reintroduction

- Shadow
- State Resynchronisation
- Checkpoint/Rollback

Prevention

- Removal from Service
- Transactions
- Process monitor

Fault



Tactics to control availability



Fault masked or repair made

Modifiability Tactics

Localise Changes

- Semantic Coherence
- Anticipate Changes
- Generalise Modules
- Limit Possible Options
- Abstract Common Services

Prevention of Ripple Effect

- Hide Information
- Maintain Existing Interface
- Restrict communication Paths
- Use an Intermediary - façade, proxy, broker, location manager, factory, ...

Defer Binding Time

- Runtime registration
- Configuration Files
- Polymorphism
- Component Replacement
- Adherence to Defined Protocols

Change Request



Tactics to control
modifiability



Change implemented,
tested and deployed
within time and budget

Performance Tactics

Resource Demands

- Increase Computational Efficiency
- Reduce Computational Overhead
- Manage Event Rate
- Control Frequency of Sampling

Resource Management

- Introduce Concurrency
- Maintain Multiple Copies - caching, replication
- Increase Available Resources

Resource Arbitration

- Scheduling Policy

Events arrive



Tactics to control performance



Response generated within time constraints