Software Architectures and Their Design

Maura Cerioli

Software System Design and Modeling 2022/23

Dibris

Architecture vs Design

- In the first lesson we have seen the relevance of design
  - it leads from problems to solutions
  - design = collect possible solutions + evaluate them + select the best *for the goals*
    - this also applies to the micro-design of code, but usually it is left implicit
  - each decision restricts the possibilities in further design steps
    - the earlier, the greater impact on the final system

All architecture is design but not all design is architecture.
Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.

Grady Booch (2006) Post “On design” on IBM developer forum

Architecture: the Metaphor

- Architecture is both the process and the product of planning, designing, and constructing buildings or any other structures.
  Esp. the art or practice of designing and building edifices for human use, taking both aesthetic and practical factors into account.
  Latin *architectura*, from the Greek ἀρχιτέκτων (*chief* + *creator* = architect).
- Commonalities
  - Both process and its result
  - Focus on the main structure
  - Design for specific users and their goals
    - stakeholders
  - Environmental constraints
  - Building = assembly of smaller parts
    - components + their relationships
  - Reuse of known concepts
    - 99% of buildings do not present breakthrough novelties
    - selecting and combining know elements create different buildings

Form ever follows function
Louis Sullivan

Group these pictures in

- Schools
- Factories
- Houses

What is a software architecture?

- The software architecture represents the *significant decisions*, where significance is measured by cost of change

Grady Booch

- Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled

Eoin Woods

- The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both

(Architectural) Design is hard

- Design decisions are not correct or wrong *per se*
  - usually, they are good or bad in a specific context and w.r.t. an expected result
  - solving one problem may very well result in an entirely different problem elsewhere in the system
- Architectural design involves trade-offs
  - e.g., between speed and robustness
  - $\Rightarrow$ not one best solution, but several acceptable solutions
    - Thus, it is important to *record* choices and discarded alternatives not to waste work when we need to modify our *best* choice
- Architectural design has no stopping rule
  - no criterion that tells when the architecture is finished
  - in many cases design is completed when we run out of time ☹

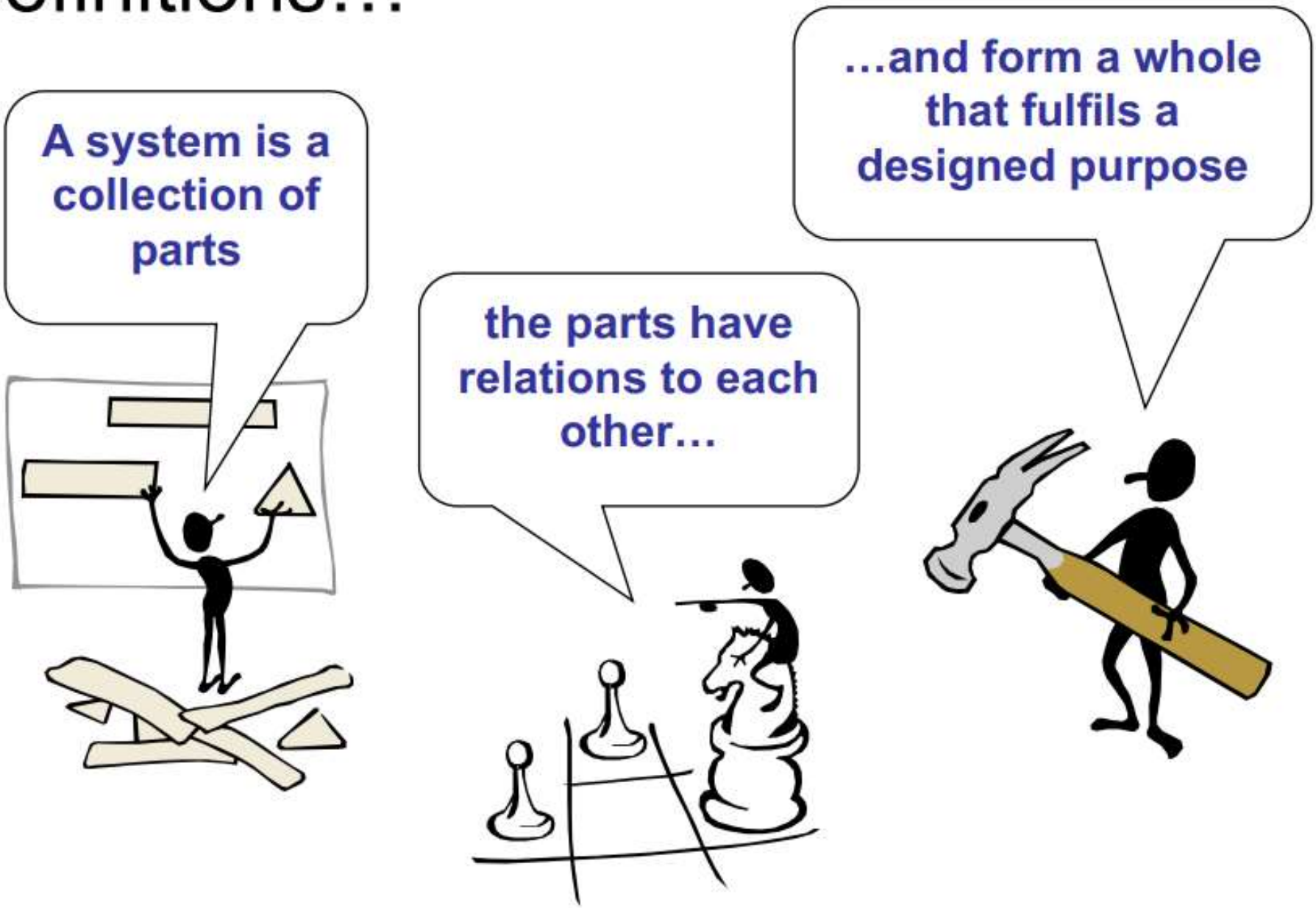
Architectural Design…of what?

- We want to design the architecture of a *system*
- But what is a system?

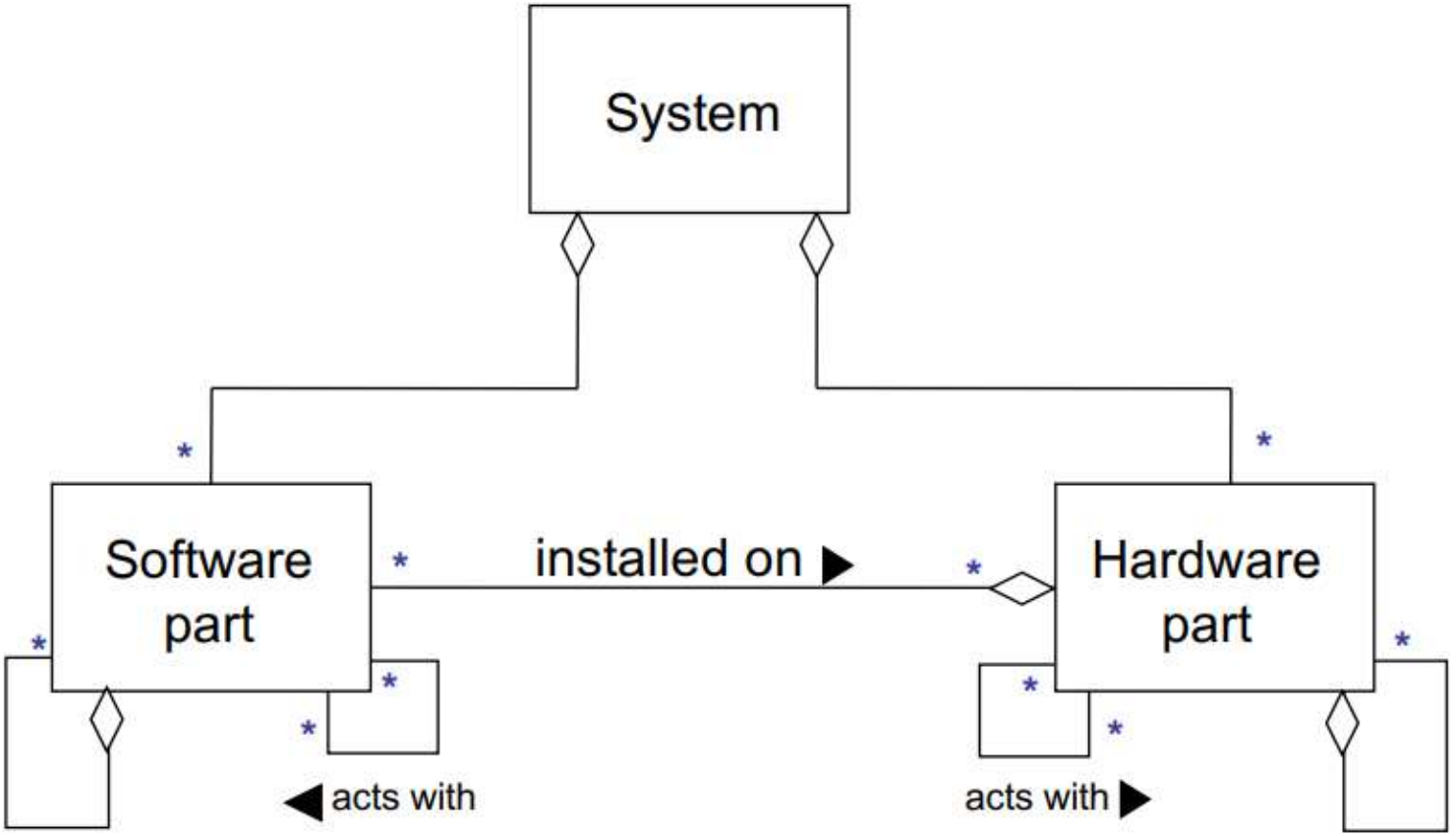
Definition: system [standard IEEE]

- [ISO/IEC 15288] systems that are man-made and may be configured with one or more of the following: hardware, software, data, humans, processes (e.g., processes for providing service to users), procedures (e.g. operator instructions), facilities, materials and naturally occurring entities
- software-intensive systems as described in [IEEE Std 1471:2000]: “any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole” to encompass “individual applications, systems in the traditional sense, subsystems, systems of systems, product lines, product families, whole enterprises, and other aggregations of interest”.
- A system is a collection of components organized to accomplish a specific function or set of functions
- A system exists to fulfill one or more missions in its environment

Definitions…



(Software intensive) System



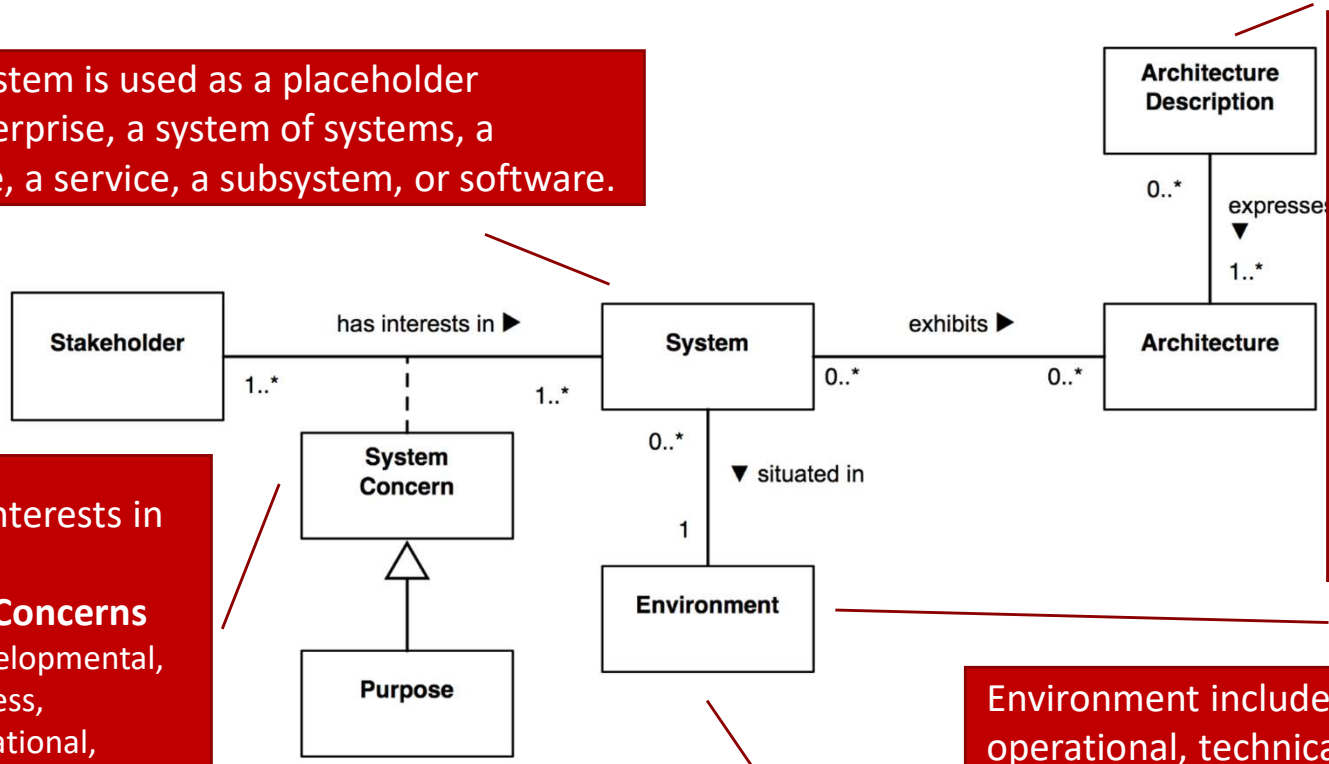
Most systems include also:

- Natural elements
- Humans

SW Architectures – Standard ISO/IEC/IEEE 42010

Architecture: 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 term system is used as a placeholder
e.g., an enterprise, a system of systems, a product line, a service, a subsystem, or software.



Architects and stakeholders use Architecture Descriptions
- to
  - understand
  - analyse
  - compare
- as "blueprints" for planning and construction

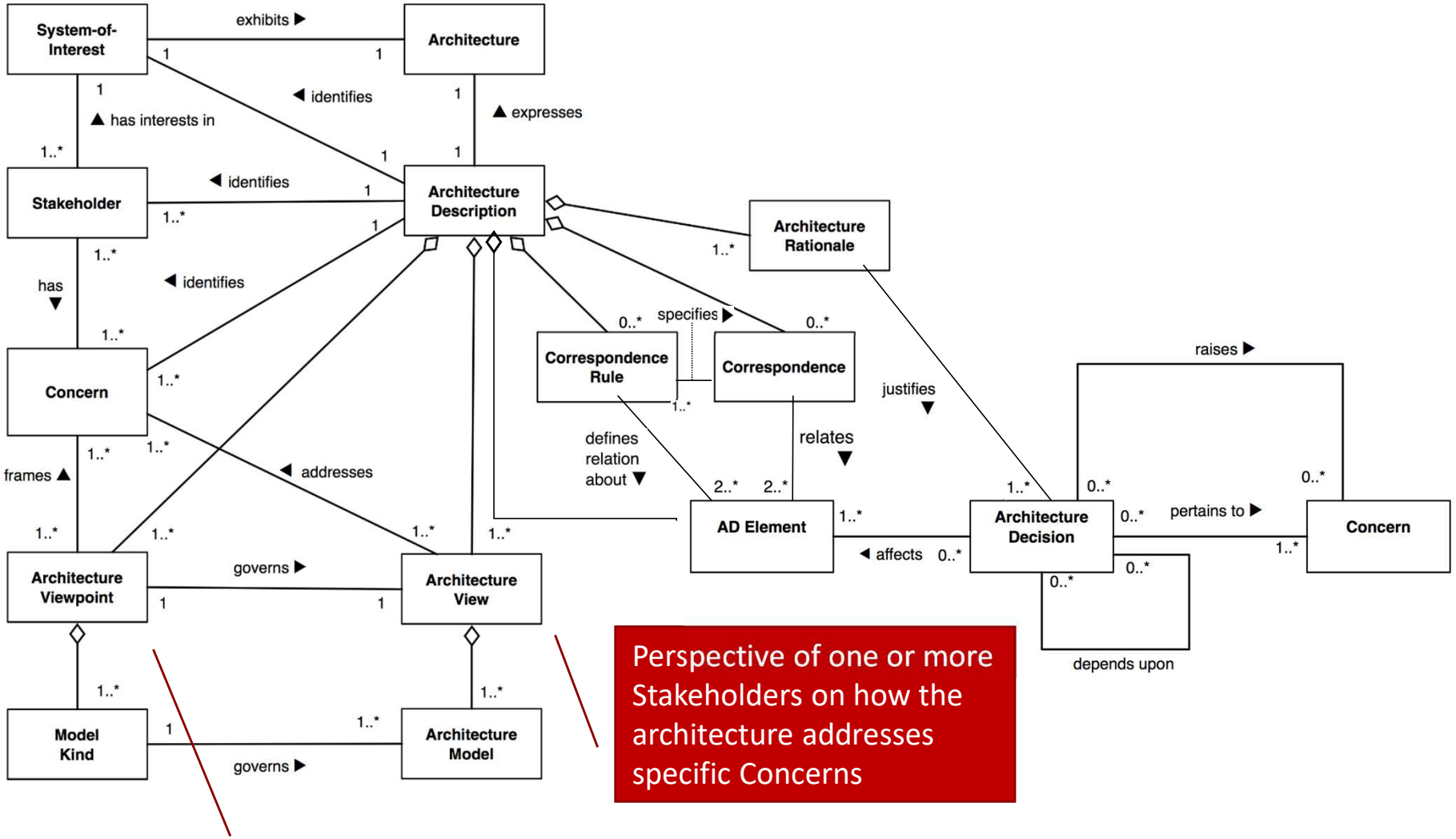
the architecture

Stakeholders have interests in a System
interests are called **Concerns**
- could pertain to developmental, technological, business, operational, organizational, political, regulatory, social, or other influences
- Special case: system's **Purpose** (or *mission*)

Environment includes developmental, operational, technical, political, regulatory, and all other influences which can affect the architecture

A System is situated in its Environment.
That environment could include other Systems

SW Architectures – Standard ISO/IEC/IEEE 42010



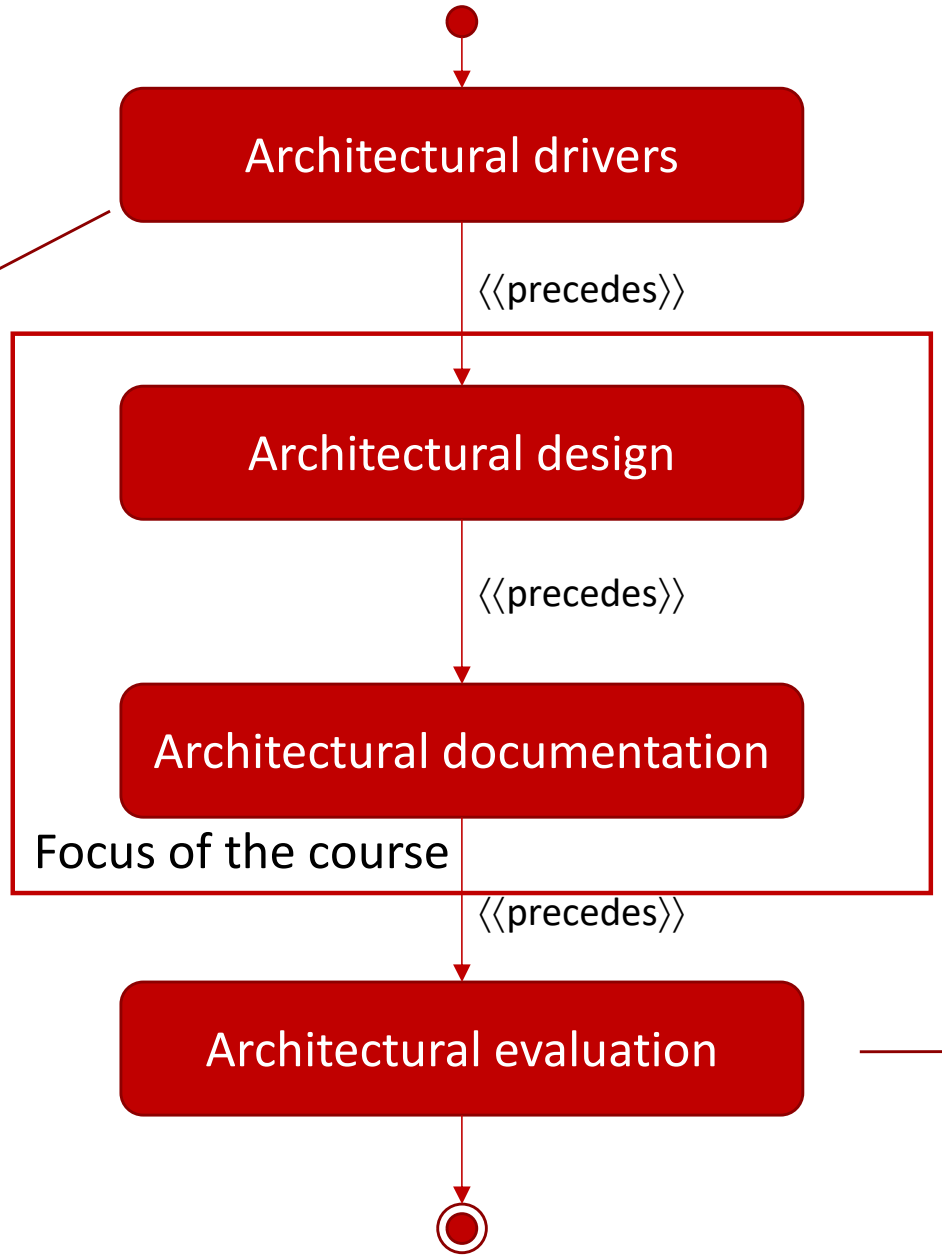
Perspective of one or more Stakeholders on how the architecture addresses specific Concerns

A set of conventions for constructing, interpreting, using and analyzing one type of Architecture View
Examples : operational, systems, technical, logical, deployment, process, information.

Dibris

Software Architecture Lifecycle

Ideally



We recap what they are, but do not present methods to identify them

In reality the activities may overlap and have feedback

Processes must norm and make provisions for that

We limit ourselves to informal and simple evaluations

Software Architecture Design

- Many different approaches
  - processes
  - choice of architecture description techniques
- Focus on the method we will follow
  - many concepts recur also in other methods
    ...possibly with different names
  - something more about it at the end

Architecture Design

Problem domain

Design

Solution domain

Design **drivers** (inputs)

Architecture Description (output)

Objectives

Quality Attributes

Primary Functional Requirements

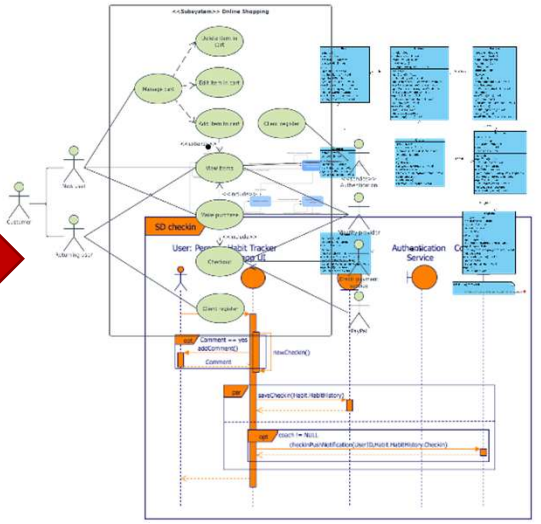
Constraints

Concerns

drive

Software Architect

produces



help

Design patterns

Deployment patterns

Tactics

Architectural styles/patterns

Design Concepts

Design Drivers - Requirements

- The engineer's first problem in any design situation is to discover what the problem really is
  -- Anonymous

software

- Requirements
  - Primary functional requirements
    - not all. Only the most critical ones
  - Quality attributes
  - Constraints

  } Non-functional requirements

- Discussed in a different part of the course
- Must be prioritized w.r.t. both
  - stakeholder needs
  - technology risks/difficulties
  - we will see some techniques to prioritize the requirements

Design Drivers – Design Purpose

- The expected use of the design is a force shaping the design
- Design goal
  - Pre-sale: just enough structure to guess the price
    - in mature fields may be very little (similar systems exist, the architect has experience)
    - in novel application types, or innovative technology the risk is high ⇒ more detailed design is in order, a prototype could be the best choice
  - Throw away prototype: no need to design for maintenability, scalability...
  - Development: enough structure and details to start the process
- Greenfield/Brownfield
  - Greenfield: requirements could be less stable
  - Brownfield: legacy system could be difficult to master
- Family of products: designing for future extensions/subsetting, high reuse
- Optimization of development process: best team allocation and skill reuse
- Timely delivery
- ...

Design concerns

All other concerns besides requirements, constraints and purpose

- General concerns
  - overall architecture, allocation of functionalities to modules and modules to team, development process management...
- Specific concerns
  - specific system-internal issues. E.g. exception management, dependency management, configuration, logging, authorization...
- Internal requirements
  - also *derived requirements*
  - implicit technical requirements addressing aspects to facilitate development, deployment, maintenance or operation
- Issues
  - result of the analisys of the current architecture

Design concepts

- Design is about finding solutions

Unfortunately, designers often reinvent

- Effective and flexible designs difficult to get “right” the first time
  - expert designers reuse solutions that have worked for them in the past
  - when facing new problems, let’s avoid pitfalls where others have already fallen
- Avoid reinventing the wheel
  - expensive and time consuming, even if you know how
- Predefined solutions to common problems free time to dive deeper on other aspects
  - save your energy and creativity for new challenges

Looking at someone else's vices, the wise fixes his
(Ex vitio alius sapiens emendat suum)
Publilio Siro

A good scientist is a person with original ideas.
A good engineer is a person who makes a design that works with as few original ideas as possible.
Freeman Dyson

Design Concepts

- Problem: experience reuse
  - capture
  - reuse
- Hard to know
  - how things were done before
  - why things were done a certain way
  - how to reuse solutions
- Many proposals
  - different names in literature for similar concepts
  - The ADD considers
    - Design patters
    - Architectural patterns/styles
    - Deployment patterns
    - Tactics
  - We present only those
    - main characteristics
    - a few examples

Dicebat Bernardus Carnotensis nos esse quasi nanos gigantium humeris insidentes, ut possim plura eis et remotiora videre, non utique proprii visus acumine aut eminentia corporis, sed quia in altum subvehimur et extollimur magnitudine gigantean

We are like dwarfs sitting on the shoulders of giants. We see more, and things that are more distant, than they did, not because our sight is superior or because we are taller than they, but because they raise us up, and by their great stature add to ours.

— Bernardus Carnotensis

quoted by John of Salisbury

Patterns

- Patterns
  - roughly speaking **solution schemas**
    - to be personalized and instantiated for the individual problem/context
  - many different kinds depending on
    - granularity
    - purpose
- First to be proposed: design patterns
  - best established
  - more standardized
- More complex and higher level patterns emerged
  - architectural

What are Design Patterns…

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

(from A Pattern Language, Alexander et al. 1977)

A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts.

(from "*Understanding and Using Patterns in Software Development*", Dirk Riehle and Heinz Zullighoven)

"Patterns communicate insights into design problems, capturing the essence of the problems and their solutions in a compact form. They describe the problem in depth, the rationale for the solution, and some of the trade-offs in applying the solution.“

(from The Design of Sites, Van Duyne et al. 2003)

Christopher Alexander

115 COURTYARDS WHICH LIVE**

❖

The courtyards built in modern buildings are very often dead. They are intended to be private open spaces for people to use – but they end up unused, full of gravel and abstract sculptures

There seem to be three distinct ways in which theses courtyards fail.

1. There is too little ambiguity between indoors and outdoors…
2. There are not enough doors into the courtyard…
3. They are too enclosed…

Therefore:

Place every courtyard in such a way that there is a view out of it to some larger open space; place it so that at least two or three doors open from the building into it and so that the natural paths which connect theses doors pass across the courtyard. And at one edge, beside a door, make a roofed veranda or a porch, which is continuous with both the inside and the courtyard.

Build the porch according to the patterns for ARCADE (119), GALLERY SURROUND (166), and SIX-FOOT BALCONY (167)…

Dibris

'A Pattern Language' Alexander et al., 1977

Image: www.saraphina.com

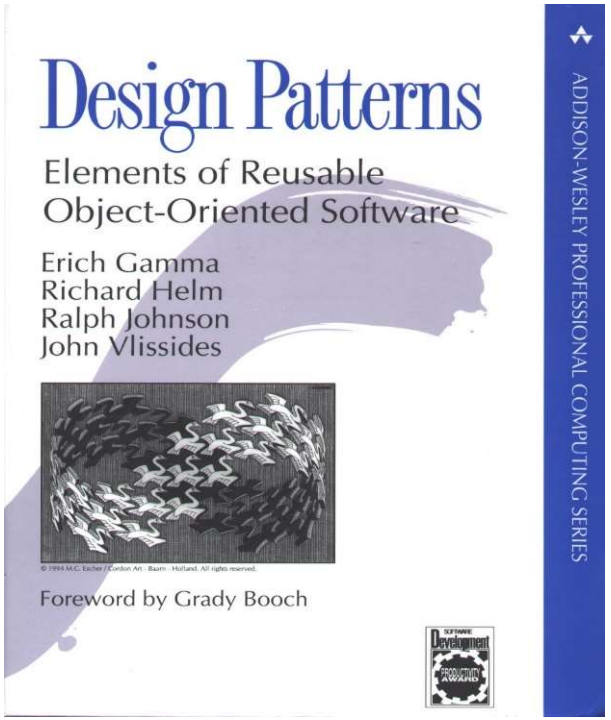
Design Patterns in Software Development

- Started in 1987 by Cunningham and Beck
  - Smalltalk + GUIs.
- Popularized by Gamma, Helm, Johnson and Vlissides
  - The gang of four

Creational
Abstract factory
Builder
Factory method
Prototype
Singleton

Structural
Adapter
Bridge
Composite
Decorator
Facade
Flyweigh
Proxy

Behavioral
Chain of responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template
Visitor



Design Patterns in Software Development

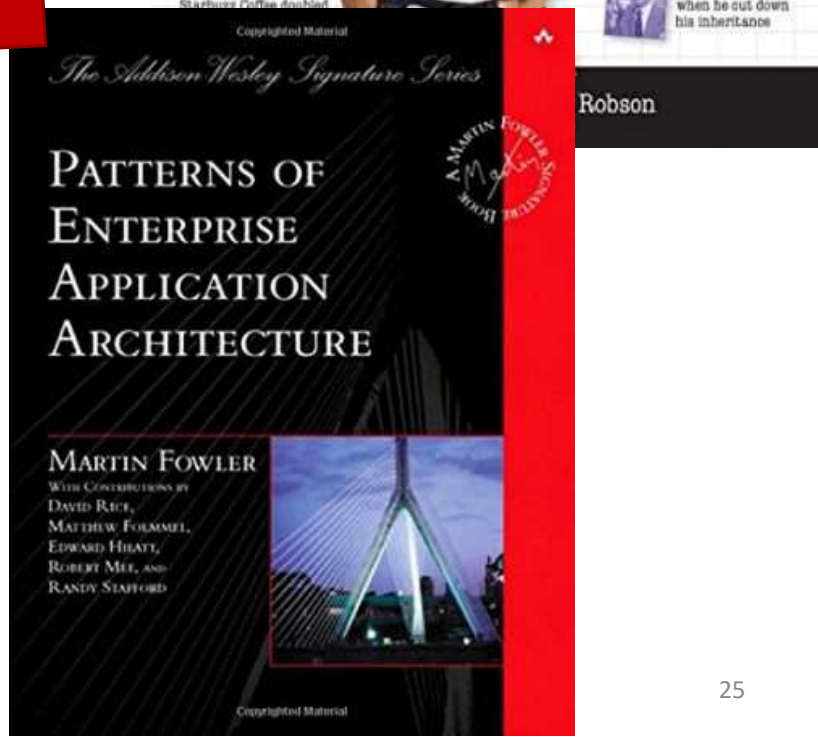
Now thousands of more specific patterns

Great book

A few cover with design pattern in it

Not a suggested reading program

Great author



Design Pattern =?

- Common solution to a recurring problem in design
  - Distils design experience
  - Abstracts a recurring design structure
- Comprises classes and/or objects
  - Dependencies
  - Structures
  - Interactions
  - Conventions
- Names & specifies the design structure explicitly
- Basic parts:
  - Name
  - Problem
  - Solution
  - Consequences and trade-offs of application
- Language- and implementation-independent
- No mechanical application
  - The solution needs to be translated into concrete terms in the application context by the designer

Design Pattern Advantages

- Distil and disseminate experience
  - Aid to novices and experts alike
- Capturing and preserving design information
  - Articulate design decisions succinctly
  - Improve documentation
  - Facilitate restructuring/refactoring
- Explicit names for design structures
  - Common vocabulary
  - Reduced complexity
- Codify good design
  - E.g. gang of four design patterns promote
    - design against interfaces (not classes)
    - object composition over inheritance
- Management of types without counterpart in the domain
  - array become composite pattern
  - behaviour encapsulating object (e.g., command)
  - bridge between domain model and programming needs

How to use Design Patterns

1. Read description (intent and motivation/forces)
   - Focus on Consequences and trade-offs
2. Understand structure, participants, and collaborations
3. Check-out coding examples
4. Chose participant names significant for the application (domain)
5. Define
   - classes/interfaces (identify those already existing in the application!)
   - operations
6. Coding

Observer Design Pattern

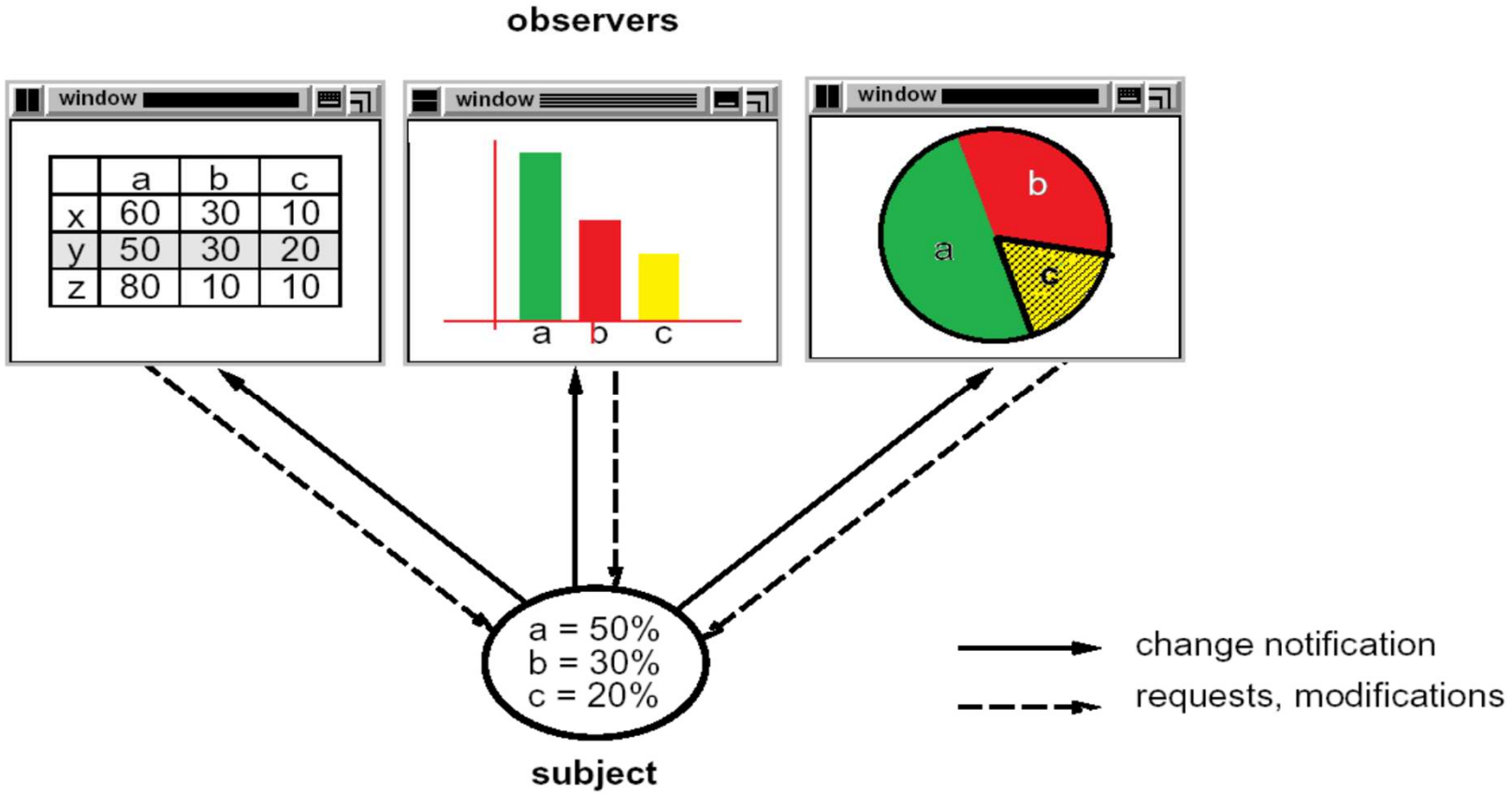
A Design Pattern everybody should know

Dibris

Observer (Behavioral)

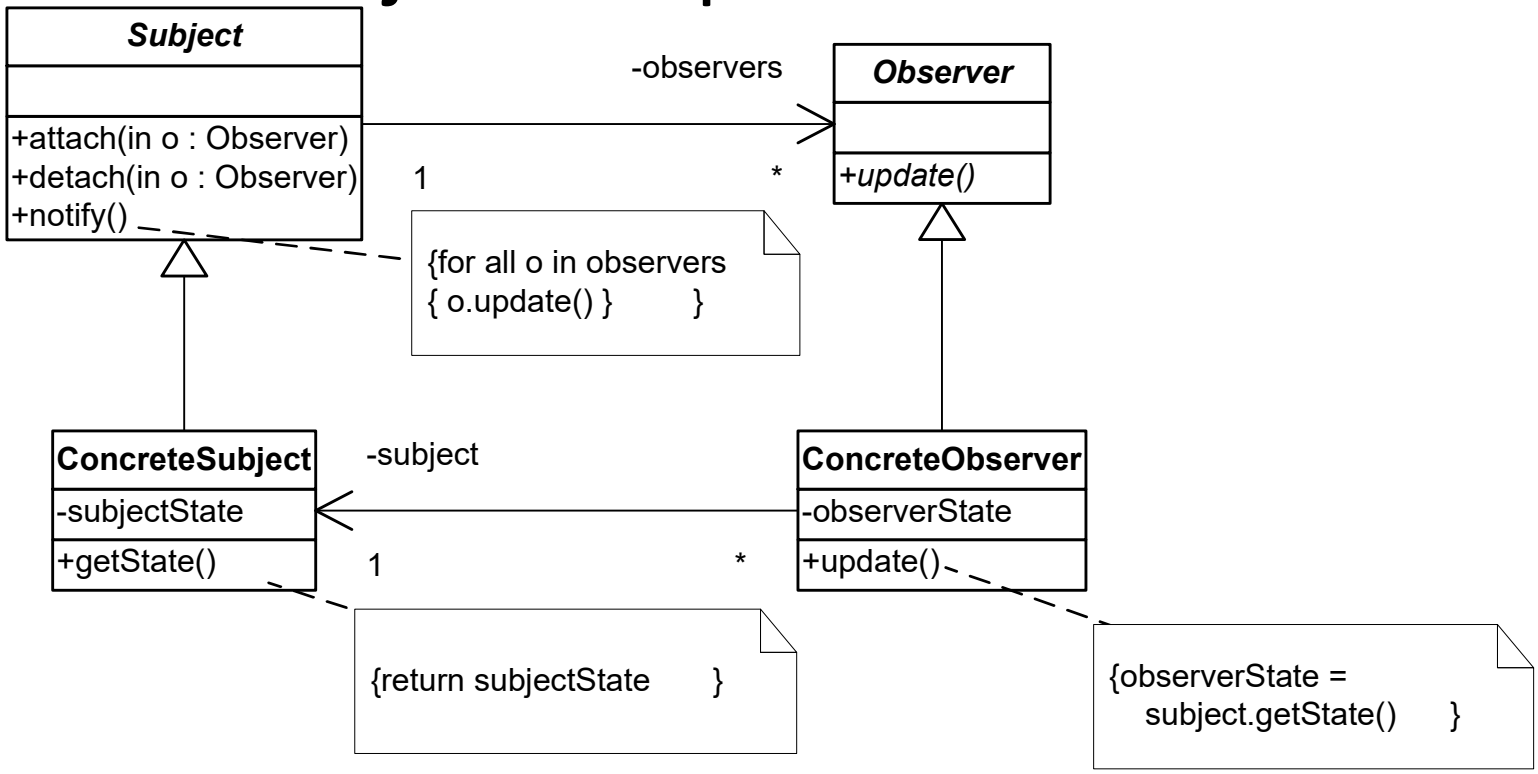
- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Applicability
  - When a change to one object requires changing others, and you don't know how many objects need to be changed
  - When an object should notify other objects without making assumptions about who these objects are
- Can you think of some example in everyday life?

Schematic Observer Example

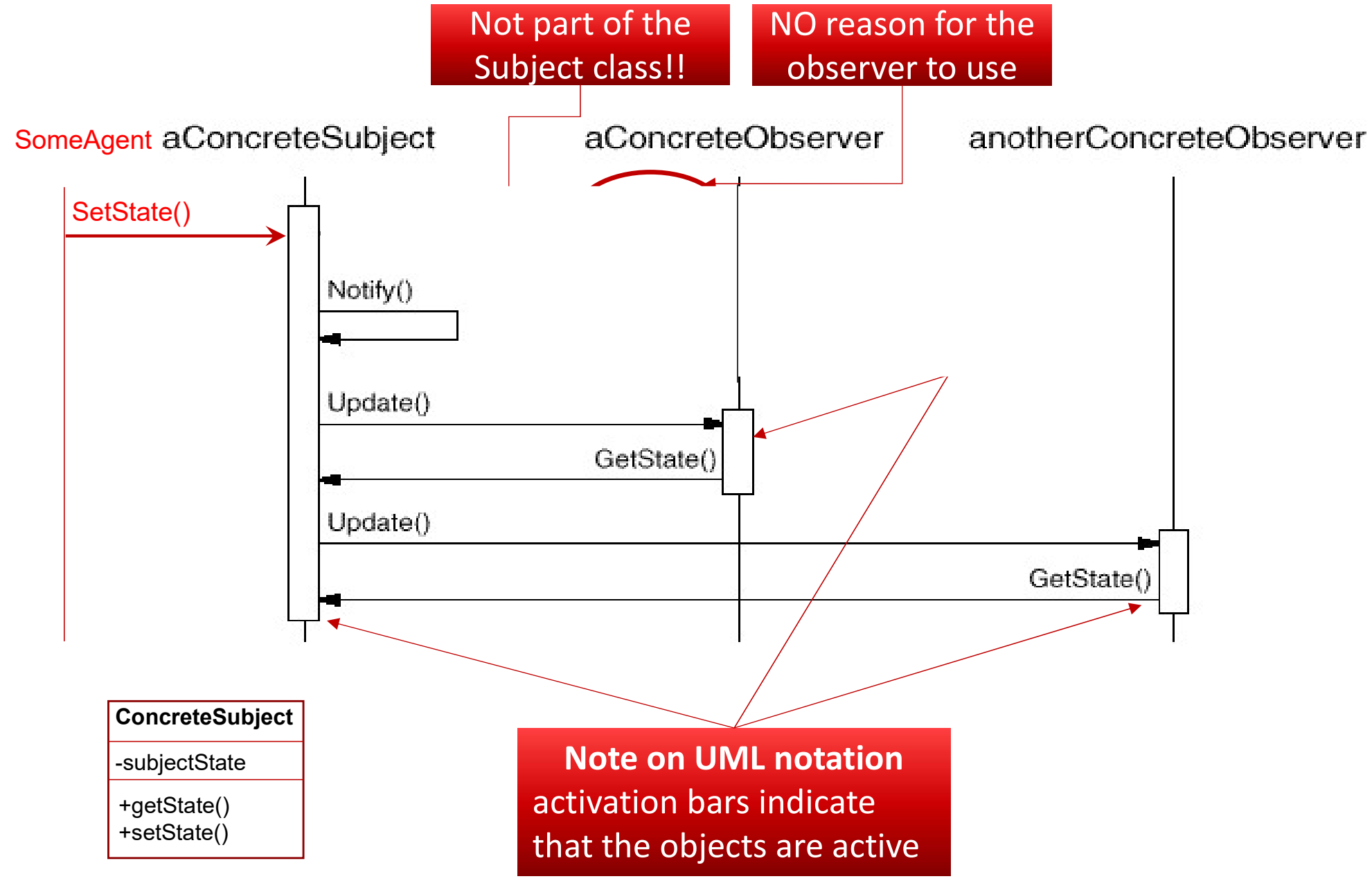


Observer- Structure

- When (Concrete) Subject changes its state, all the observers are notified
- Any (Concrete)Observer, when notified, may contact the subject to update its information



Observer- Collaborations



Observer Consequences

- Modularity
  - subject and observers may vary independently
  - subject depends only on the interface Observer
- Extensibility
  - can define and add any number of observers
  - subject broadcast to all the observers without knowing them
- Customizability
  - different observers provide different views of subject
- Unexpected updates
  - observers don't know about each other
- Update overhead
  - any operation may cause thousands of useless updates
  - no information on the changes carried by notifications

Observer Implementation

- Subject-observer mapping
- Dangling references
- Avoiding observer-specific update protocols: the push and push/pull models
- Registering modifications of interest explicitly

Architectural patterns

- Same principles as design patterns, but coarser grained
  - not necessarily the overall architecture of the system
  - architecture of some subsystem/module/component
- Several different formats in the literature
  - as for design patterns
  - the required info are the *same*

Elements of an architectural pattern

(from *Pattern-Oriented Software Architecture: A System of Patterns*)

- Name
- Problem
- Context
- Forces
- Solution
- Resulting Context
- Examples
- Rationale
- Related Patterns
- Known Uses

A description of the relevant forces and constraints, and how they interact/conflict with each other and with the intended goals and objectives. The description should clarify the intricacies of the problem and make explicit the kinds of trade-offs that must be considered. The notion of "forces" equates in many ways to the "qualities" that architects seek to optimize, and the concerns they seek to address, in designing architectures. For example: Security, robustness, reliability, fault-tolerance, ...

The post-conditions after the pattern has been applied. It describes which forces have been resolved and how, and which remain unresolved. It may also indicate other patterns that may be applicable in the new context.

An explanation/justification of the pattern.
The Solution element of a pattern describes the external structure and behavior of the solution.
The Rationale provides insight into its internal workings.

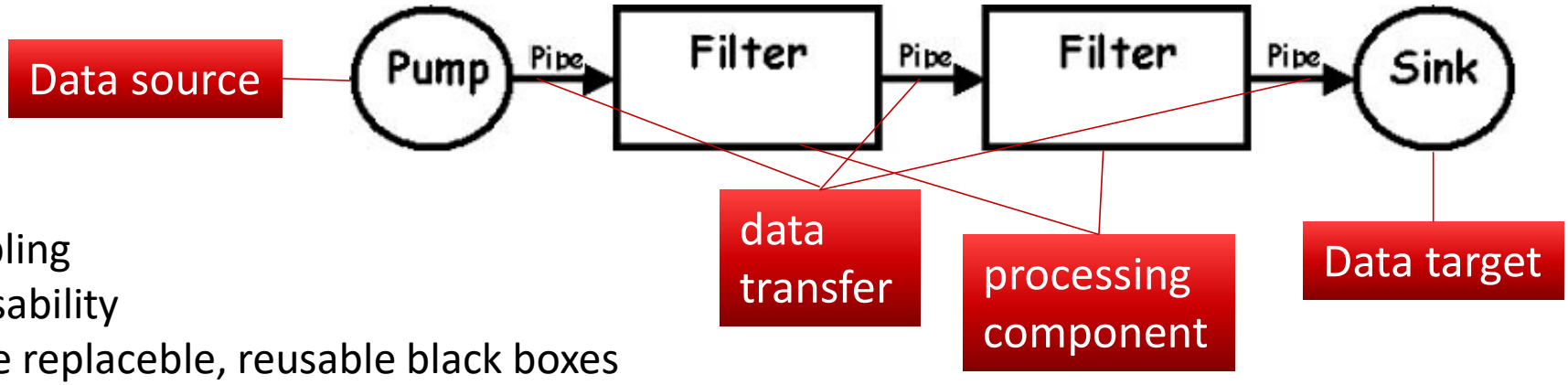
The relationships between this pattern and others. Predecessor, successor, alternative , or co-dependent patterns

Architectural patterns vs. architectural styles

- Elements of architectural patterns are not classes/objects
  - ...because they are coarser grained
  - nor packages/modules/components...
  - more *abstract* elements of the architecture
- To express an architectural pattern (in most cases) we need a vocabulary
  - architectural *styles*
  - in many sources in literature no difference between arch. styles and arch. patterns

Architectural pattern/style example: Pipe and filters

- Intent: stream/asynchronous processing of data
- Applicability: large processes that can be broken down into multiple steps
- As a style, it defines the *terms*



- Pro
  - low coupling
  - high reusability
  - filters are replaceble, reusable black boxes
- Cons
  - starvation/flooding when filters cannot keep up the pace
  - overhead
    - to keep filters independent checks, parsing etc must be repeated, data unpacked and re-packed...
- As a *pattern,* it provides possible solutions, for instance the *linear* pipe and filter

A few patterns for graphical interfaces

Examples of pattern evolution, composition, comparison

Dibris

Web Development = MVC pattern (??)

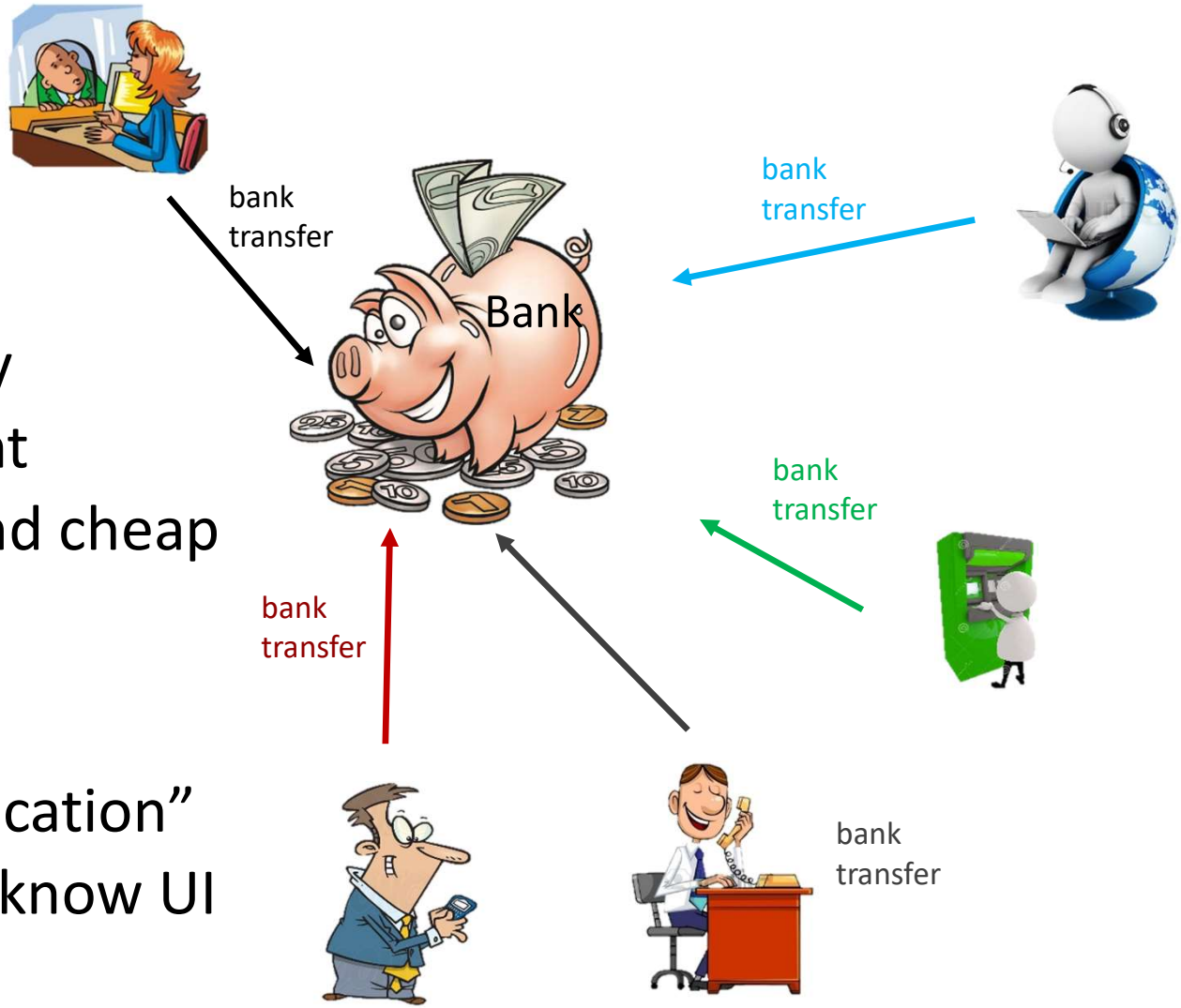
- Most modern framework claim to be MVC-based
- Different descriptions of MVC

> *Different people reading about MVC in different places take different ideas from it and describe these as 'MVC'. If this doesn't cause enough confusion you then get the effect of misunderstandings of MVC that develop through a system of Chinese whispers.*
>
> *-- Martin Fowler*

- Variants of MVC: MVP, MVVM…
- What to choose?
  - start with needs
  - solution from available patterns
    - historical prospective

Problem 1

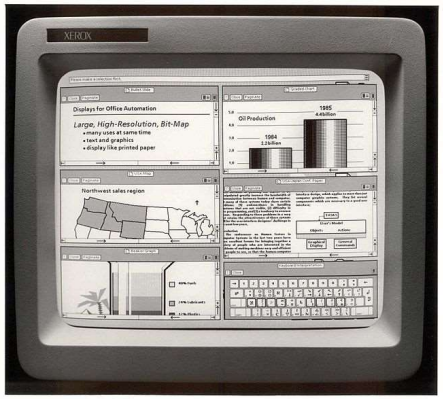
- Problem context: many ways to work on the same data
  - same application
  - different interfaces
- How to
  - guarantee consistency
  - minimize development
  - make changes easy and cheap
  - allow easy testing
- Solution
  - UI distinct from “application”
  - “application” doesn’t know UI



MVC History

- 1979: 2 pages note by Trygve Reenskaug introduced MVC into Smalltalk-76
- 1980s
  - MVC implemented in the Smalltalk-80 class library
  - MVC expressed as a general concept in a 1988 article in The Journal of Object Technology

way ahead of patterns!



Xerox Alto GUI
1972

A pioneering attempt at design reuse

MODELS - VIEWS - CONTROLLERS

Trygve Reenskaug - 10 December 1979

- MODELS represent knowledge. (…) There should be a one-to-one correspondence between the model and its parts on the one hand, and the represented world as perceived by the owner of the model on the other hand.

Objects not raw data

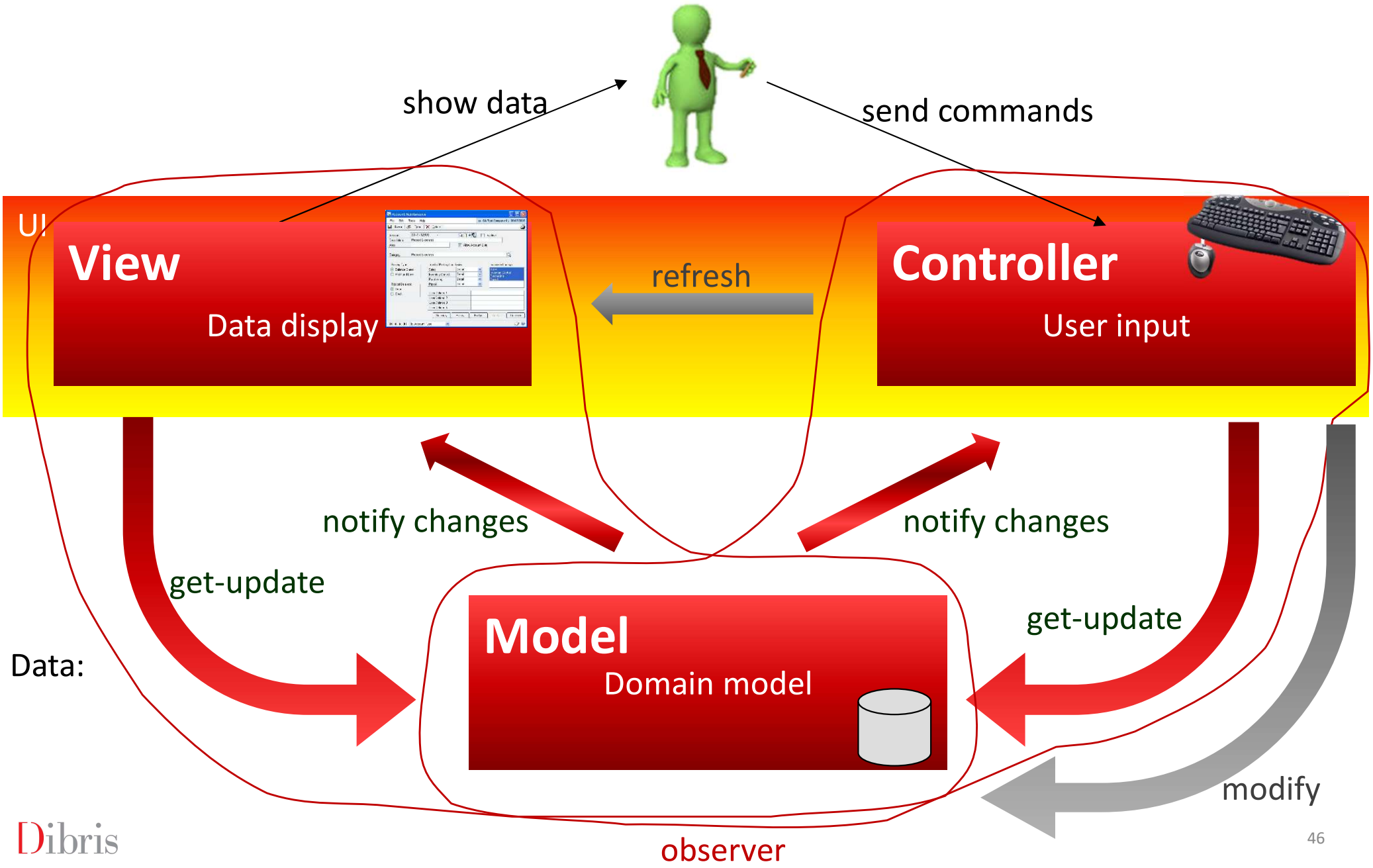
- VIEWS are a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a presentation filter. A view is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages.
- CONTROLLERS are the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. The controller receives such user output, translates it into the appropriate messages and pass these messages on to one or more of the views..(…)
  Conversely, a view should never know about user input, such as mouse operations and keystrokes. It should always be possible to write a method in a controller that sends messages to views which exactly reproduce any sequence of user commands.

Controllers did jobs now performed by os/platform

Solution Key Idea

- Decoupling data access and business logic from user interface
- Data access + business logic = (Domain) model
  - one model for each (group of) application(s)
    - guarantees consistency
    - minimizes effort
  - presentation agnostic
- One model + several UI
  - UIs change more often
  - UIs present data and operations from model
- Can we use the observer pattern?

Model-View-Controller (MVC)



Details of MVC Design Pattern

- Context
  - used when developing interactive applications
- Forces
  - the same information is presented differently in different windows or devices
  - the display and behavior of the application must reflect data manipulations immediately
  - changes to the user interface should be easy, and even possible at run-time
  - support different look and feel standards or porting the user interface should not affect code in the core of the application

In memory not on DB

MVC Solution

- Divides the application into three areas (separation of concerns)
- **Model** encapsulates the core data and functionality
  - structure of the data in the application
  - application-specific operations on those data
- **View** encapsulates the presentation of the data
  - there can be many views of the common data
- **Controller** accepts input from the user and makes requests from the model for the data to produce a new view
  - translates user actions (mouse motions, keystrokes, words spoken, etc.) and user input into application function calls on the model
  - selects the appropriate View based on user preferences and Model state

MVC Consequences

- The view is easily replaced or expanded
- Model data changes are reflected in all interfaces because all views are Observers In memory
- Better scalability since UI and application logic are separated
- Distribution over a network is greatly simplified
- Business logic bleeds into the Controller
- Excessive coupling between the Model and View and the Model and Controller

MVC Granularity

- As ***design*** pattern MVC applies to elementary items in forms (and forms)
  - model ⇔individual data
  - view ⇔item visualization
  - control ⇔ event-handlers
- As ***architectural*** pattern MVC applies to application design
  - model ⇔domain model
  - view ⇔output part of UI
  - control ⇔ input part of UI

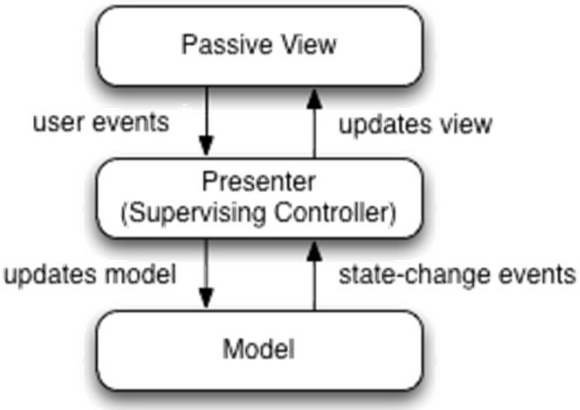
Problem 2

- Problem context: developing UI is expensive
  - different interfaces similar building blocks
  - each building block difficult to design and implement
  - need for a framework/platform to assemble them
- How to
  - reuse interface elements/assembly line
  - allow people with different skills to work independently
- Solution
  - UI built in layers
    - lower layer knows the application/domain
    - higher layer built from standard blocks

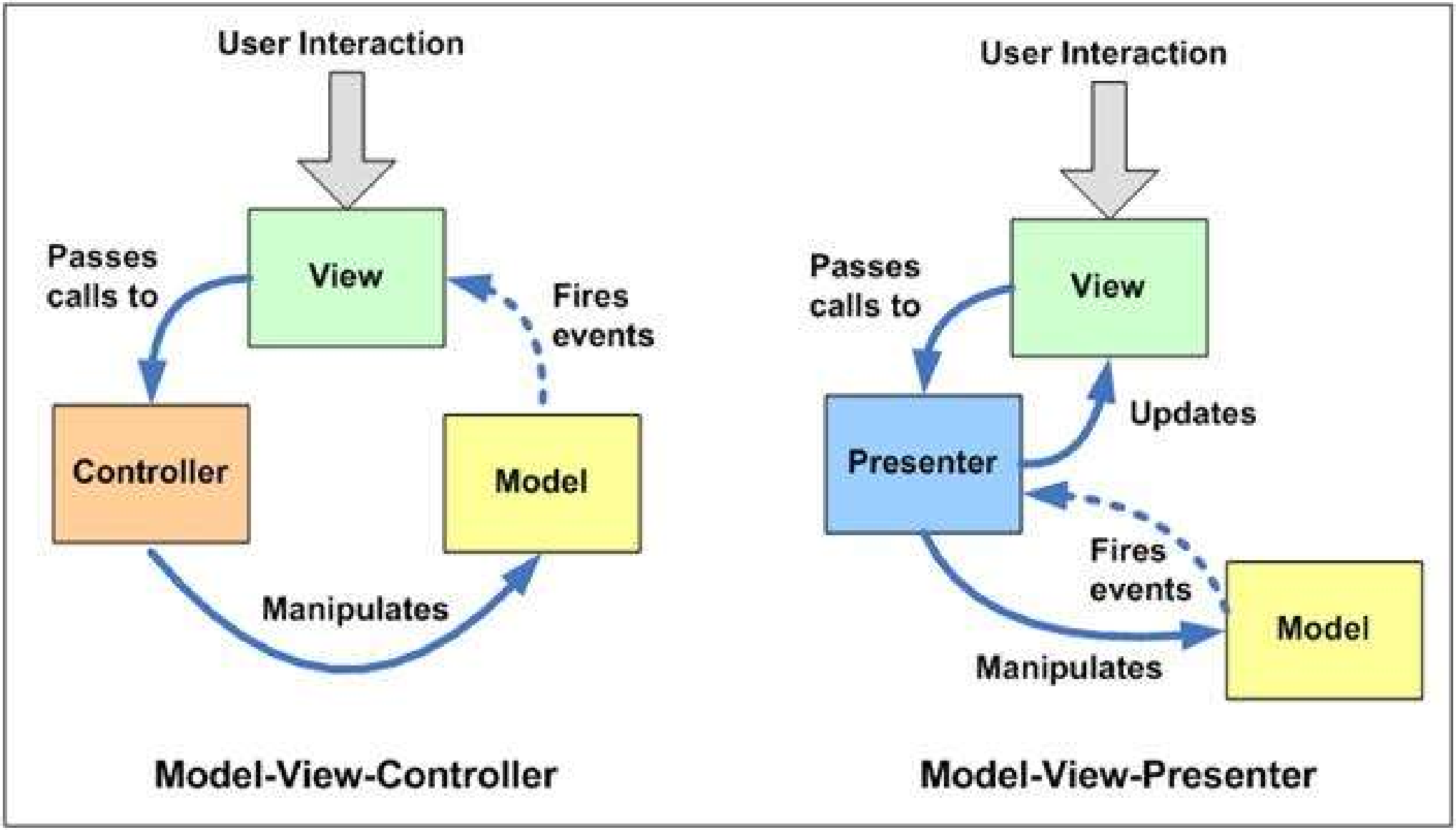
views in MVC mix the two aspects

MVP - Model View Presenter

- Variant of MVC
  - View interacts with Presenter and Presenter with Model
  - View built from standard elements
  - State of view + specificity to support model moved in Presenter
  - Standard data binding between View and Presenter
- Can use dependency injection (View interface) to improve testability
- More apt for distributed systems
  - View on client
  - Model and Presenter on server(s)
- Hybrid variant
  - View interacts **trivially** with Model (data binding)
  - Specific interaction mediated by Presenter



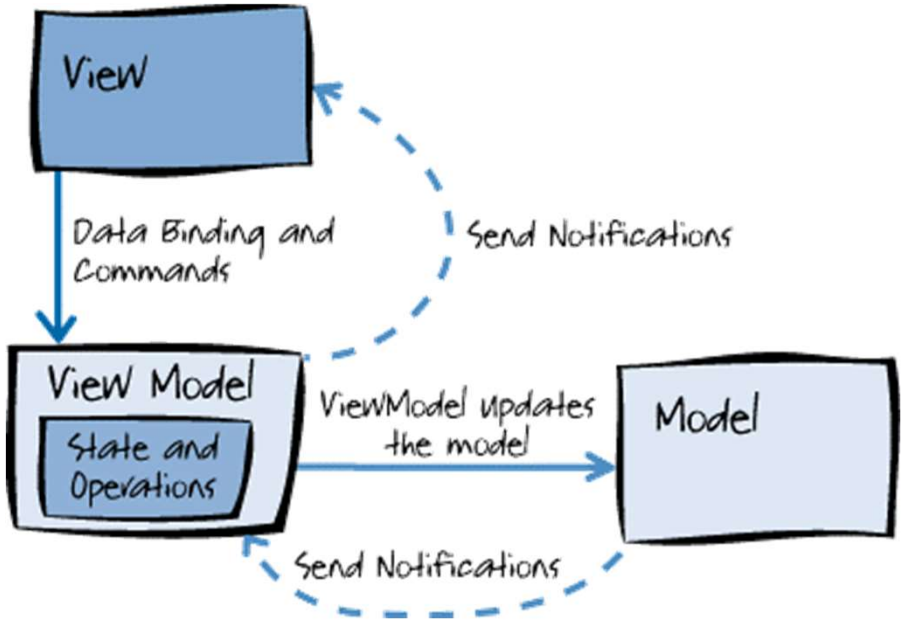
MVC ⇔ MVP


User Interaction
Passes calls to
View
Fires events
Controller
Model
Manipulates
Model-View-Controller
User Interaction
Passes calls to
View
Presenter
Updates
Fires events
Model
Manipulates
Model-View-Presenter

Problem 3

- Problem context: where data about user/app interaction live?
  - e.g., editing/reading mode, buttons enabled only after some input is received…
  - no part of the domain logic, no part of visualization
- Solution
  - have a model of the UI
  - hook the view model to the actual UI on one side and to the application core on the other

Model View ViewModel



- Not a thong twister ;-)
  - introduced by Microsoft (WPF – Windows Presentation Foundation and Silverlight)
  - also called Model View Binder (mostly outside of Microsoft stack)
- Variant of Model View Presenter
- Model and View mostly as before
  - Model = domain model, including business logic
  - View = GUI with presentation concern
- View Model abstract representation of interactions with users and data needed for that
  - interact with model
  - translate data format/organization to/from view and model
  - View delegates to View Model all specific operations (through event usually)

Deployment patterns

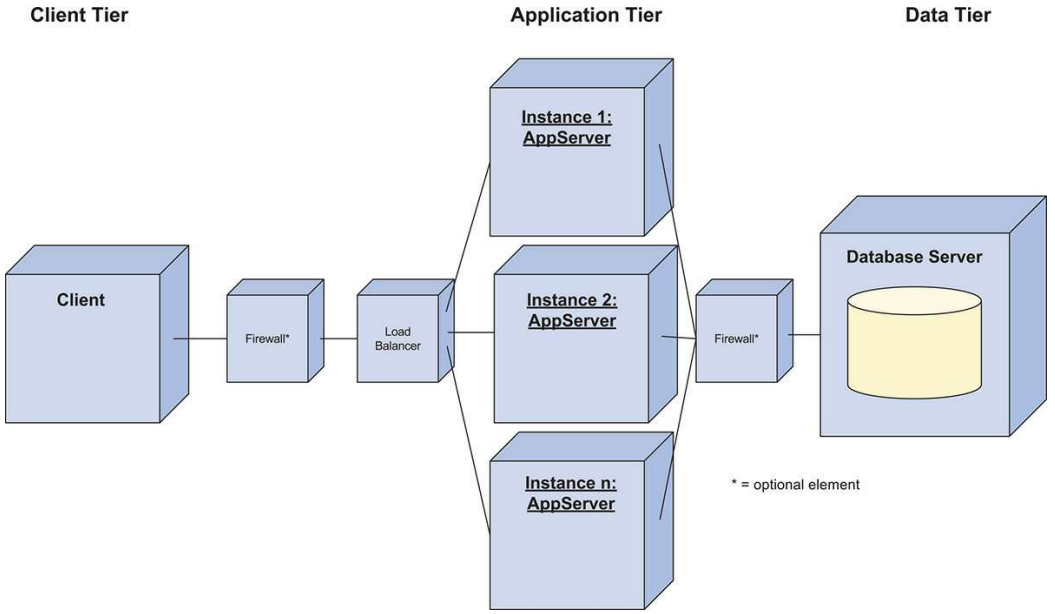
- Subtype of architectural patterns: those concerning *physical* distribution of software
- Usually required by/selected for improving quality attributes
  - security (separation and protection through firewalls)
  - performance (e.g., co-location for synchronous operations)
  - scalability (e.g., replication)
- Sometime needed to satisfy policies or legal constraints
  - e.g., company policies about data management, laws about geographical data distribution
- Constrained by
  - hardware availability
  - operative costs

Deployment patterns: an example
Load balanced cluster

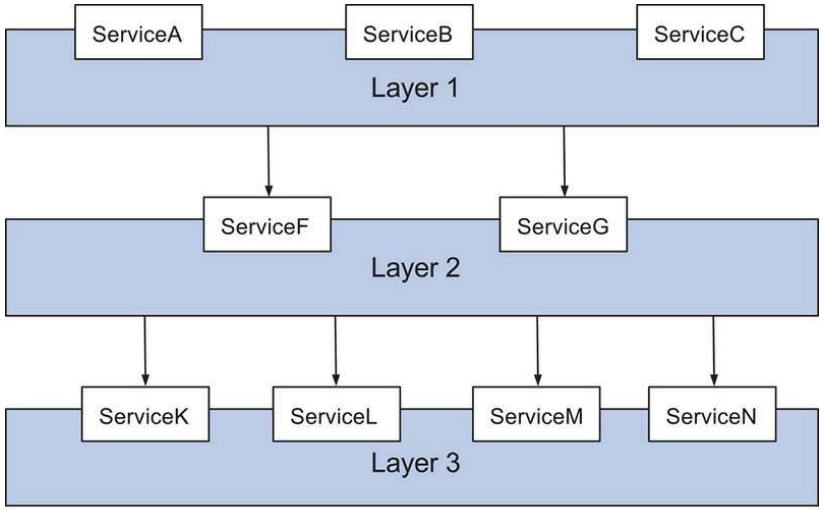
- Context: in a many-tier architecture, the server in one layer needs to be duplicated
  - for security or performance reasons
- Problem: how to distribute incoming requests
  - optimizing performance
  - improving availability
- Forces
  - server maximum load capability
  - server maximum phyisical performance
  - single point(s) of failure vs complexity of managing and monitoring server replications
- Solution: *cluster* of server managed by a *load balancer*

Load balanced cluster

- Implementation issues
  - choice of algorithm for load balancing
  - for state based interactions, session management (client side, server side)
- Unsolved issues
  - load balancer is a single point of failure
    - possible mitigation: replications

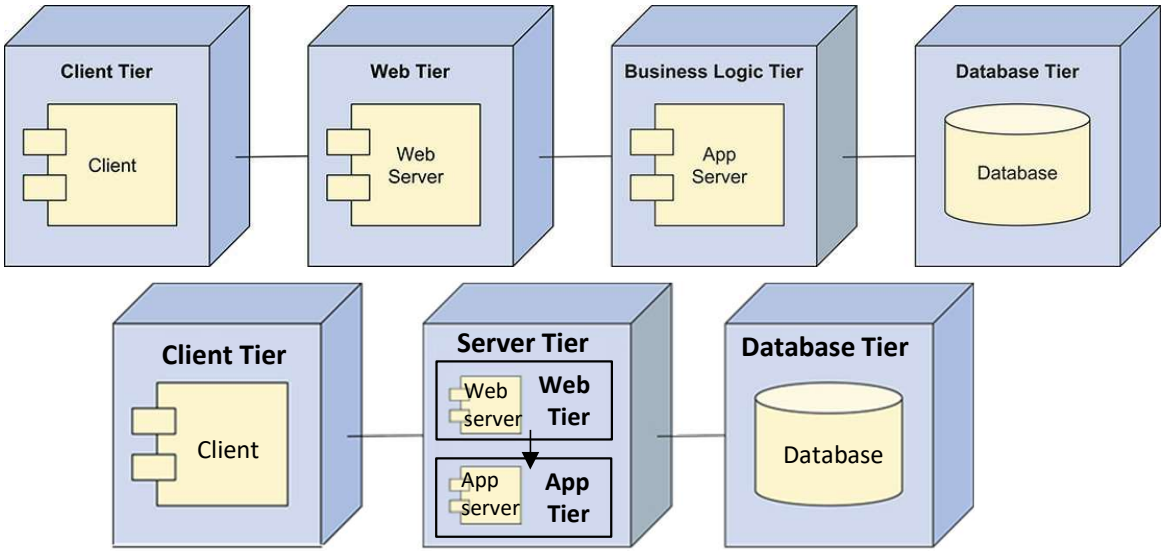


The two faces of many tiers patterns (and of many other distribution patterns)



Layered ***logical*** structure
Architectural pattern
Key idea: modules are clustered in logical layers, those in a layer provide services for those in the layer(s) above, use services from those in the layer(s) below
Main goal: decoupling ⇒scalability+maintenability

Layered ***physical*** structure
Deployment pattern
Key idea: deploy different layers on different hardware
Main goal: physical separation ⇒ security+performance (with some drawbacks)



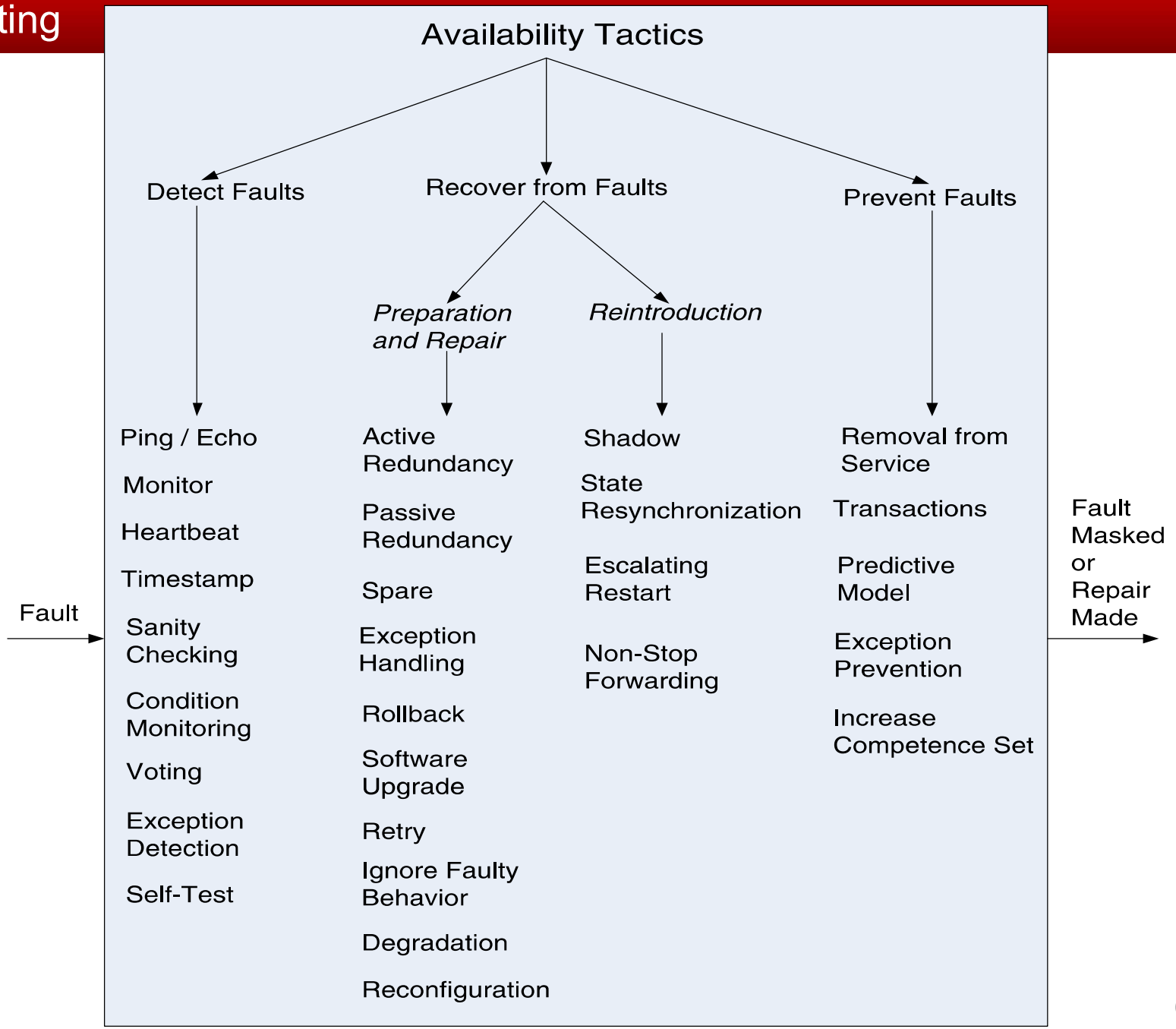
Often, a logical distribution enables, but does not enforce, a similar physical distribution
= Architectural patterns enable deployment patterns by the same name

Tactics

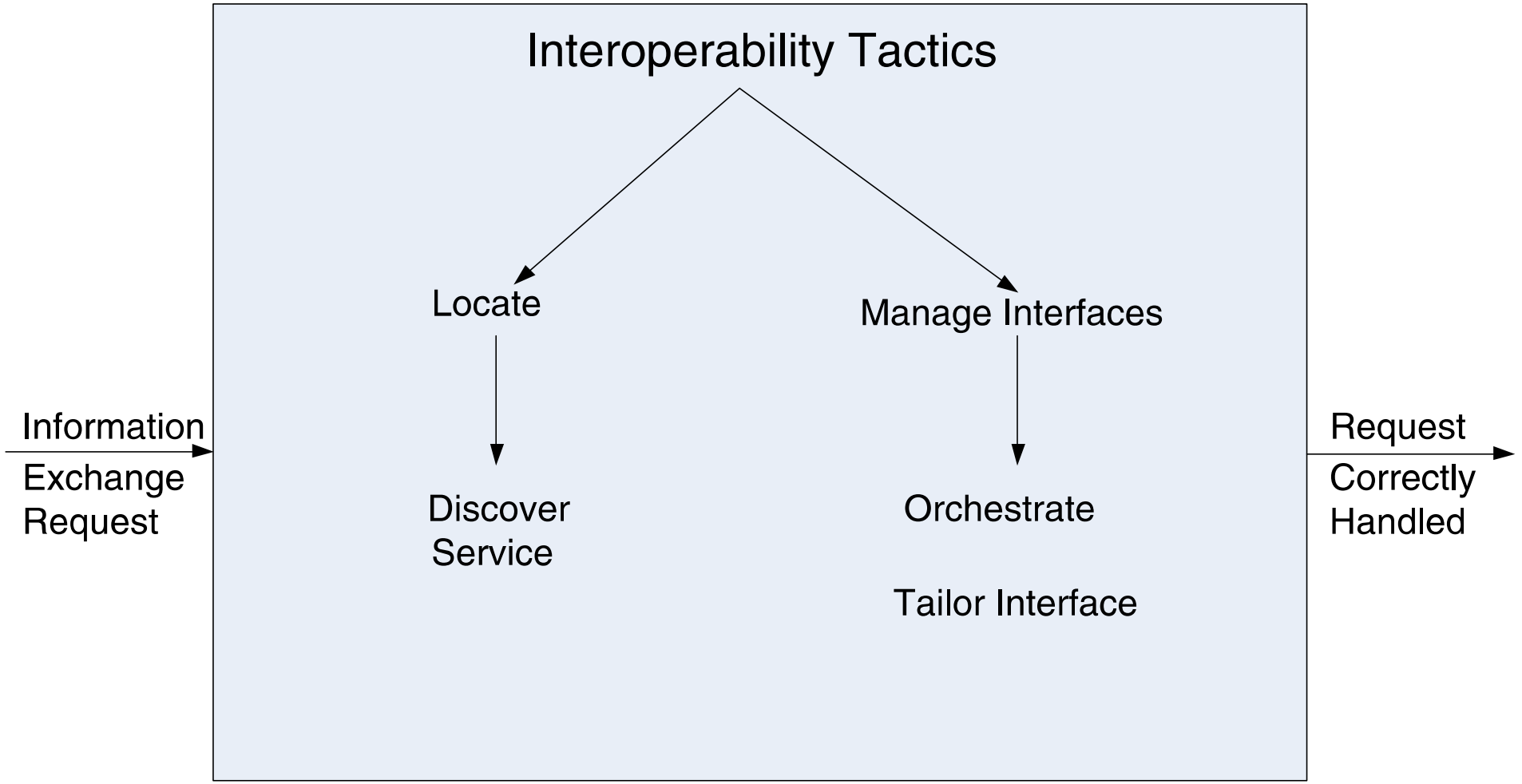
Tactic: an action or strategy carefully planned to achieve a specific end.

- Embody *experience as decisions*
  - as opposite to patterns, embodying schematic solutions
- For each quality attribute/non-functional requirement category a list of possible decisions to improve that aspect
- Less structured than patterns
  - no trade-offs
  - no context/problem/other design concepts...
- Patterns may capture some tactic
  - some tactic cannot be expressed by patterns

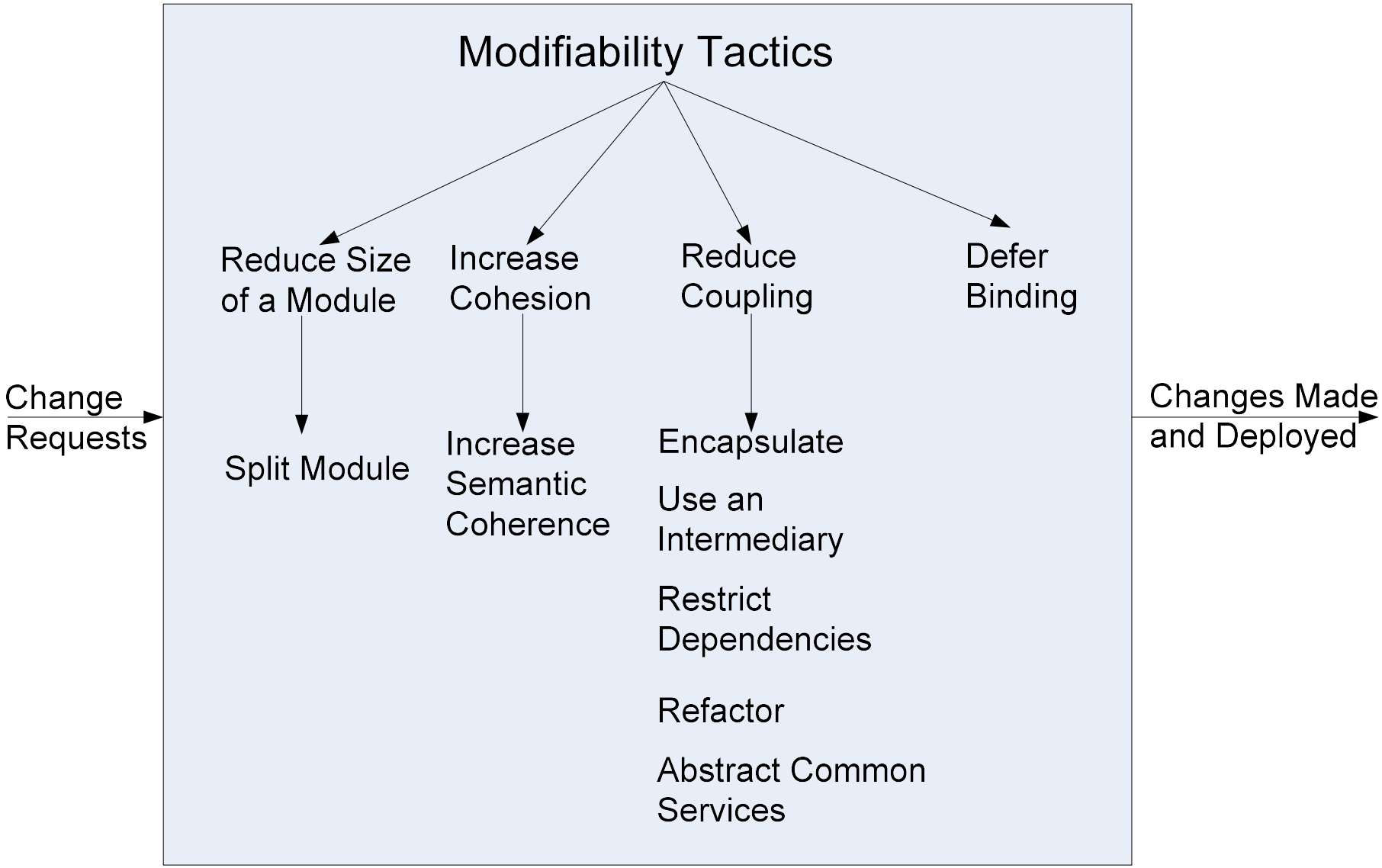
Availability - the probability that a system is operational at a given time, i.e. the amount of time a device is actually operating as the percentage of total time it should be operating



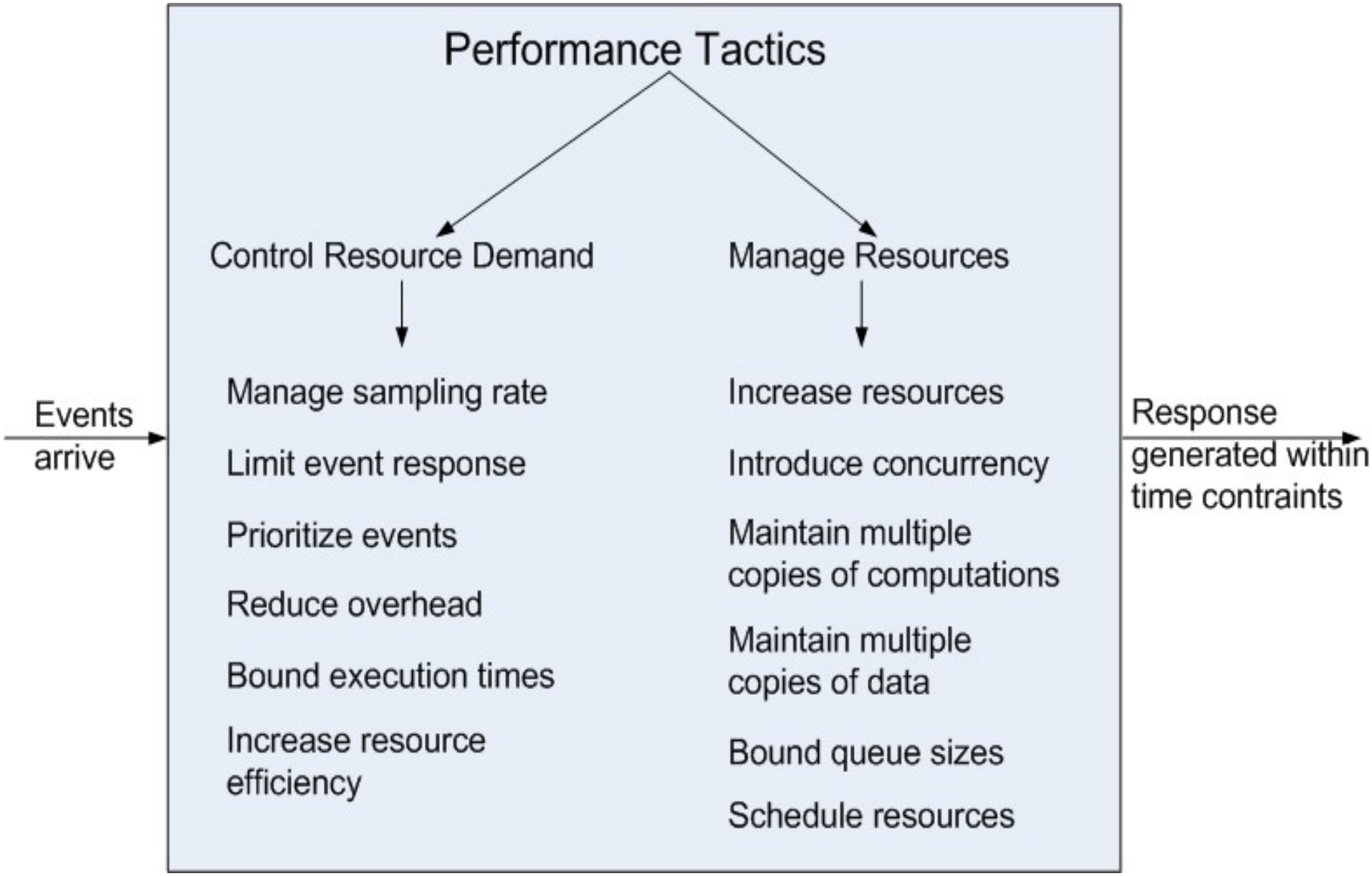
Interoperability - the ability of different systems, devices, applications or products to connect and communicate in a coordinated way, without effort from the end user.



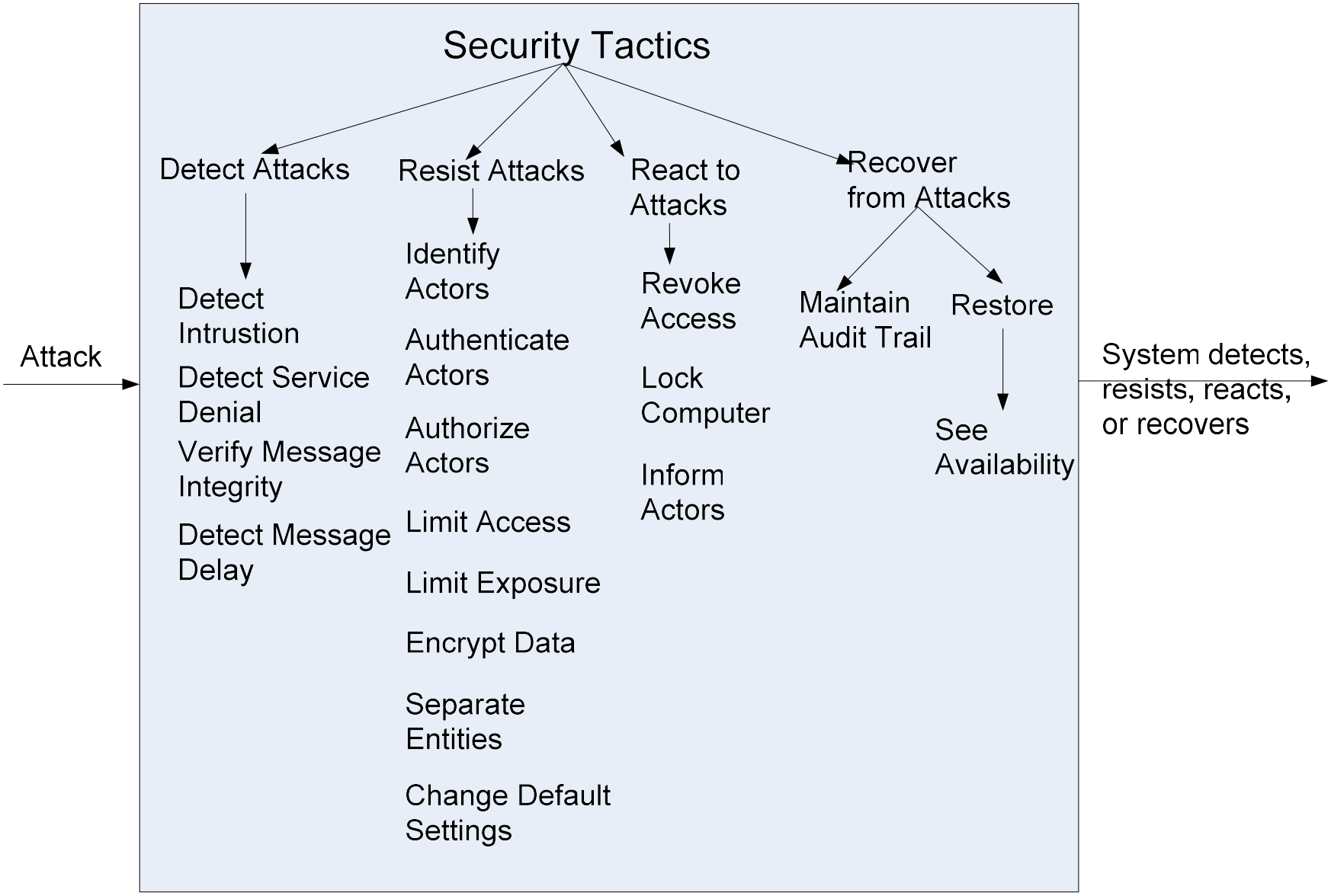
modifiability - the ease with which a **software** system can be modified to changes in the environment, requirements or functional specification



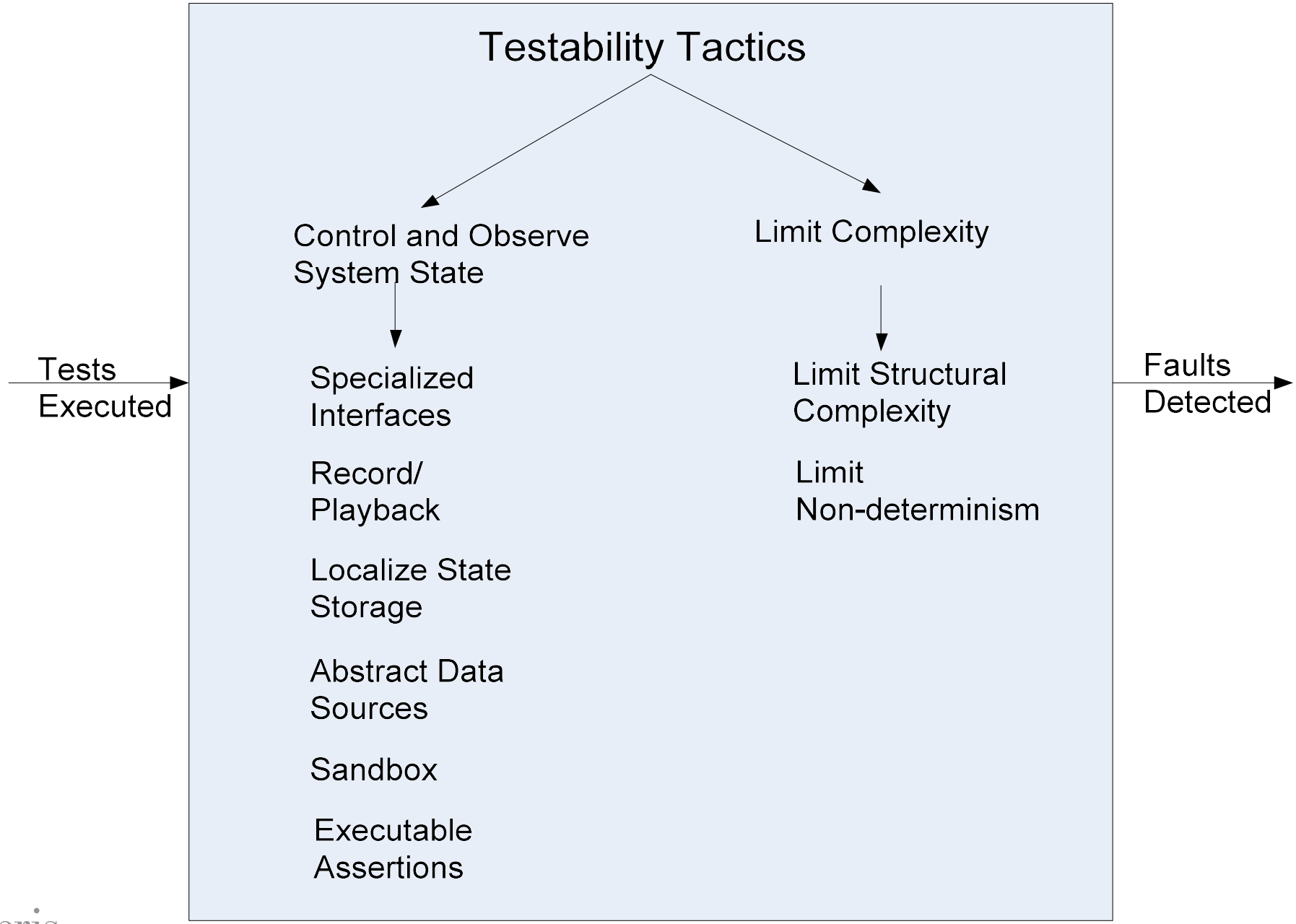
Performance - the response time or latency between a request to the system and its answer



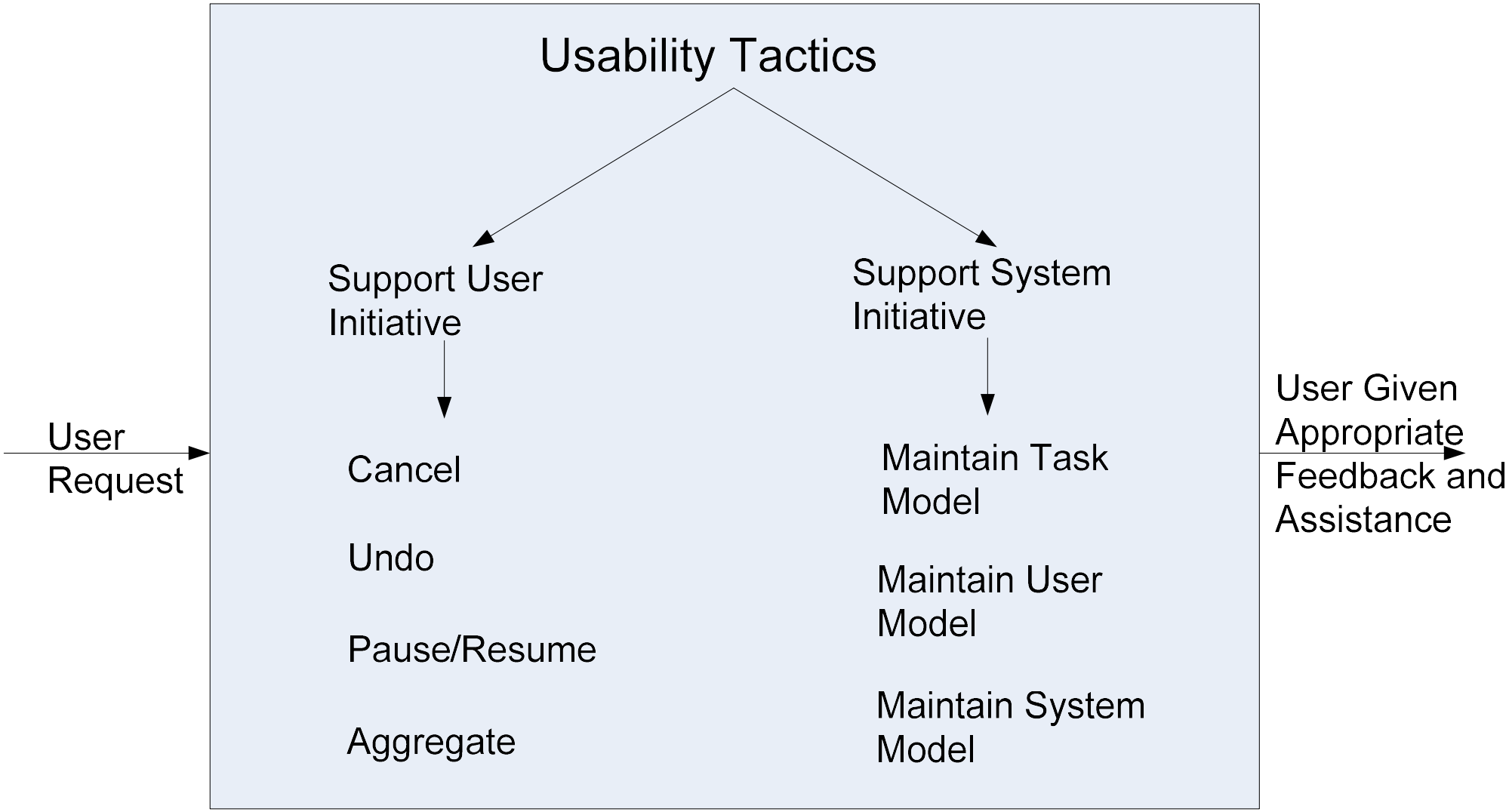
Security - the extent to which the system can deliver services whilst under hostile attack



Testability - the degree to which a software system supports testing in a given test context



Usability - the degree to which a software can be used by specified consumers to achieve quantified objectives with effectiveness, efficiency, and satisfaction in a quantified context of use.



- https://web.archive.org/web/20120623081009/http://aahninfotech.com/arct_pattern.html
  Definition of architectural pattern and catalogue
- https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658117(v=pandp.10)?redirectedfrom=MSDN
  Same but queer (examples of architectural styles are object-oriented or domain-driven)

Further readings

- Microsoft Application Architecture Guide, 2nd Edition https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff650706(v=pandp.10)?redirectedfrom=MSDN online reading http://ce.sharif.edu/courses/91-92/1/ce474-2/resources/root/App%20Arch%20Guide%202.0.pdf pdf download
- Peter Eeles, What is a software architecture? Published on February 15, 2006 https://www.ibm.com/developerworks/rational/library/feb06/eeles/index.html
- ADD 3.0: Rethinking Drivers and Decisions in the Design Process https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=436536
- Interesting discussion about graphical interface design by Martin Fowler http://martinfowler.com/eaaDev/uiArchs.html
- On tactics http://etutorials.org/Programming/Software+architecture+in+practice,+second+edition/Part+Two+Creating+an+Architecture/Chapter+5.+Achieving+Qualities/5.1+Introducing+Tactics/