Issues of adaptable software for open-world requirements

Carlo Ghezzi
Politecnico di Milano
Deep-SE Group @ DEI
carlo.ghezzi@polimi.it

Lectures in the context of the other lectures

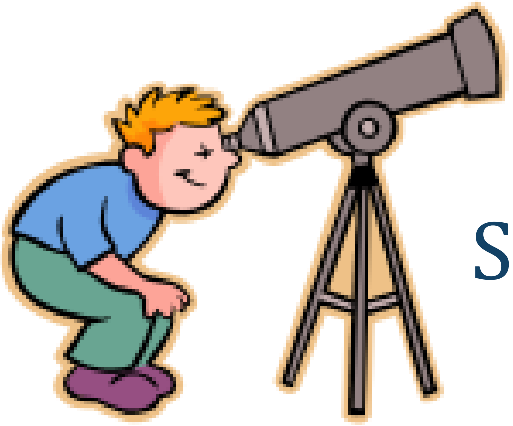
- I will focus on **software evolution**
  - how to support evolution in a sound way using formal methods
  - in particular, on **adaptation** (self-managed evolution)
- Evolution&change traditionally viewed as antagonistic to **formal methods**
- To support evolution, formal methods are instead necessary; they **need to extend to run-time**
- I will not discuss new formal approaches, but rather discuss how formal methods can be **used/adapted/packaged** to support software engineers

Outline

- **Lecture 1**
  - Introduction&motivations; historical perspective of software evolution
    - from the closed world to the open world
    - the new main challenges
- **Lecture 2**
  - Software architectures and languages for adaptation and evolution
    - architectural styles and middleware support
    - architectures supporting self-organization
    - language support for dynamic software evolution
- **Lectures 3, 4**
  - Lifelong quality management for adaptive evolvable systems
    - functional (*behavioral*) and nonfunctional (*quality*) properties
    - development-time vs run-time
    - specification and verification
    - run-time adaptation

Self Managing Situated Computing

Introduction and motivations



Historical perspective

Software and software evolution

"Pre-history" of software engineering

- Software production did not follow any precisely formulated process
  - continuous changes
    - iteration of coding and error fixing
- Code&fix not compatible with the desired industrial standards

Early history: facts and assumptions

- **Monolithic**, **stable** organizations
- Slow change
  - the **closed world** assumption
    - requirements are there, they are stable
    - just elicit them right
  - software changes should be avoided
    - they disrupt a rational development
      - causing schedule and cost problems

Early history: solutions

- **Process level**
  - The sequential (waterfall) process model
  - Refinement, from clearly and fully specified requirements down to code
  - Top-down development -> formal deductive approaches
- **Product level**
  - Programming languages and methods producing static verifiable architectures
    - Static and centralized system compositions, frozen at design time

A waterfall model

Maintenance
can be up
to 80% of total costs

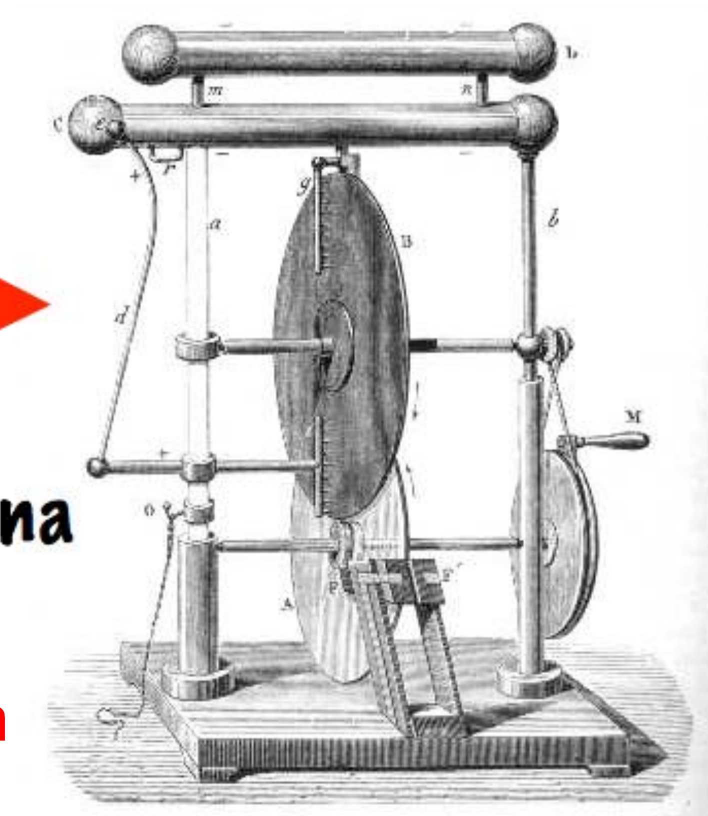
The *machine* and the *world*

World (the environment)

Machine

Domain properties (assumptions)

Shared phenomena



Goals

Requirements

Specification

What changes in the environment?

- The **requirements** we wish to achieve
  - e.g., because business goals change
- **Domain assumptions**
  - e.g., because the context/situation changes
    - users, user profiles
    - external resources/services/libraries/devices

Maintenance

- Traditionally, any change in the software is handled as maintenance, and managed **offline**
  - **corrective** maintenance
    - corrects the machine
  - **adaptive** maintenance
    - achieves compliance with domain changes
  - **perfective** maintenance
    - achieves compliance with requirements changes

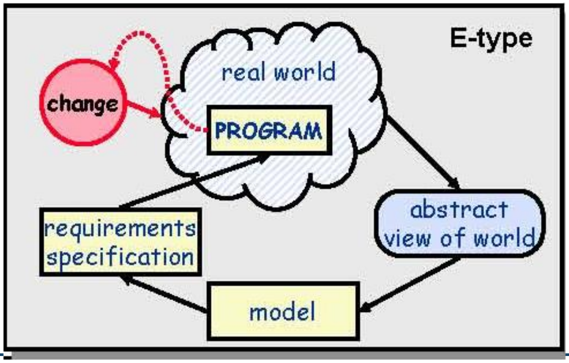
Software evolution

- Early work in the 1970s by M. Lehman and L. Belady, then continued until the 2000s by M. Lehman
- Empirical observations lead to the "laws" of evolution
- Much empirical research active today, especially mining data from open-source software repositories

Lehman, M.M., 'Software's Future: Managing Evolution', IEEE Software, Vol. 15, No. 1, Jan-Feb 1998.

Lehman's original classification

- **S-type software** (rarely observed in practice)
  - software has the sole criterion of being mathematically correct with respect to a fixed and constant specification
- **E-type software**
  - solution to real-world problem, used and embedded in a real-world domain



Lehman's "laws" of evolution (1)

I 1974	Continuing Change	Software must evolve continuously otherwise it becomes progressively less satisfactory in use
II 1974	Increasing Complexity	As a system evolves its complexity increases unless work is done to maintain or reduce it
III 1974	Self Regulation	Global system evolution processes are self-regulating [System attributes such as size, time between releases and the number of reported errors are approximately invariant for each system release]
IV 1978	Conservation of Organizational Stability	Average effective global activity rate in an evolving system tends to remain constant over product lifetime

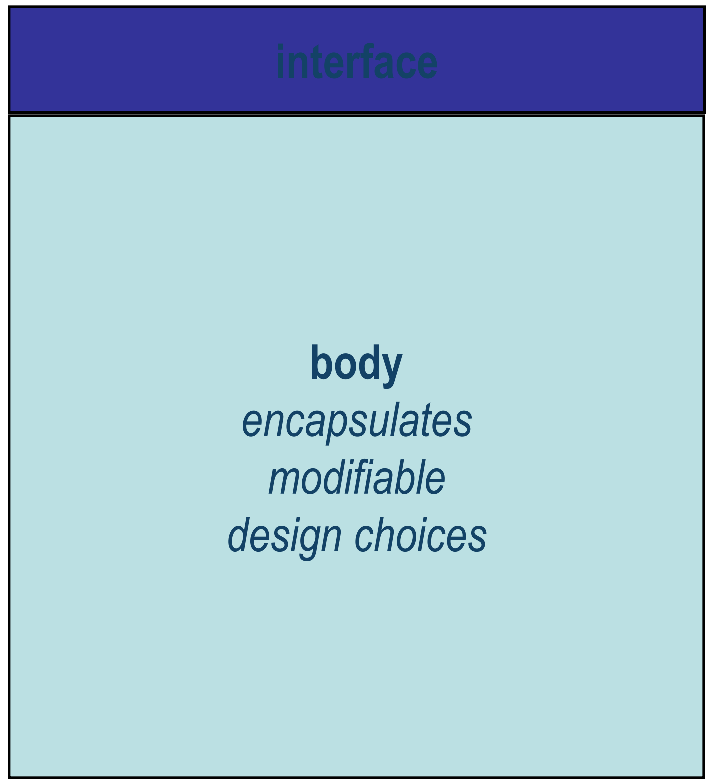
Lehman's "laws" of evolution (2)

V 1978	Conservation of Familiarity	During the active life of a program the amount of change in successive releases is roughly constant to allow people to maintain mastery of the system (excessive growth diminishes that mastery)
VI 1991	Continuing Growth	The functional capability of a systems must be continually increased to maintain user satisfaction over the system lifetime
VII 1996	Declining Quality	Unless special care is taken, the quality of evolving systems will appear to be declining
VIII 1996	Feedback System (Recognised 1971, formulated 1996)	Evolution processes are multi-level, multi-loop, multi-agent feedback systems

How to deal with evolution

- More flexible **processes** have been invented
  - from iterative to agile
- To support evolution of the software **products**, different approaches to modularity were used, leading to current mainstream OO languages and OO design approaches

Design for change (Parnas)

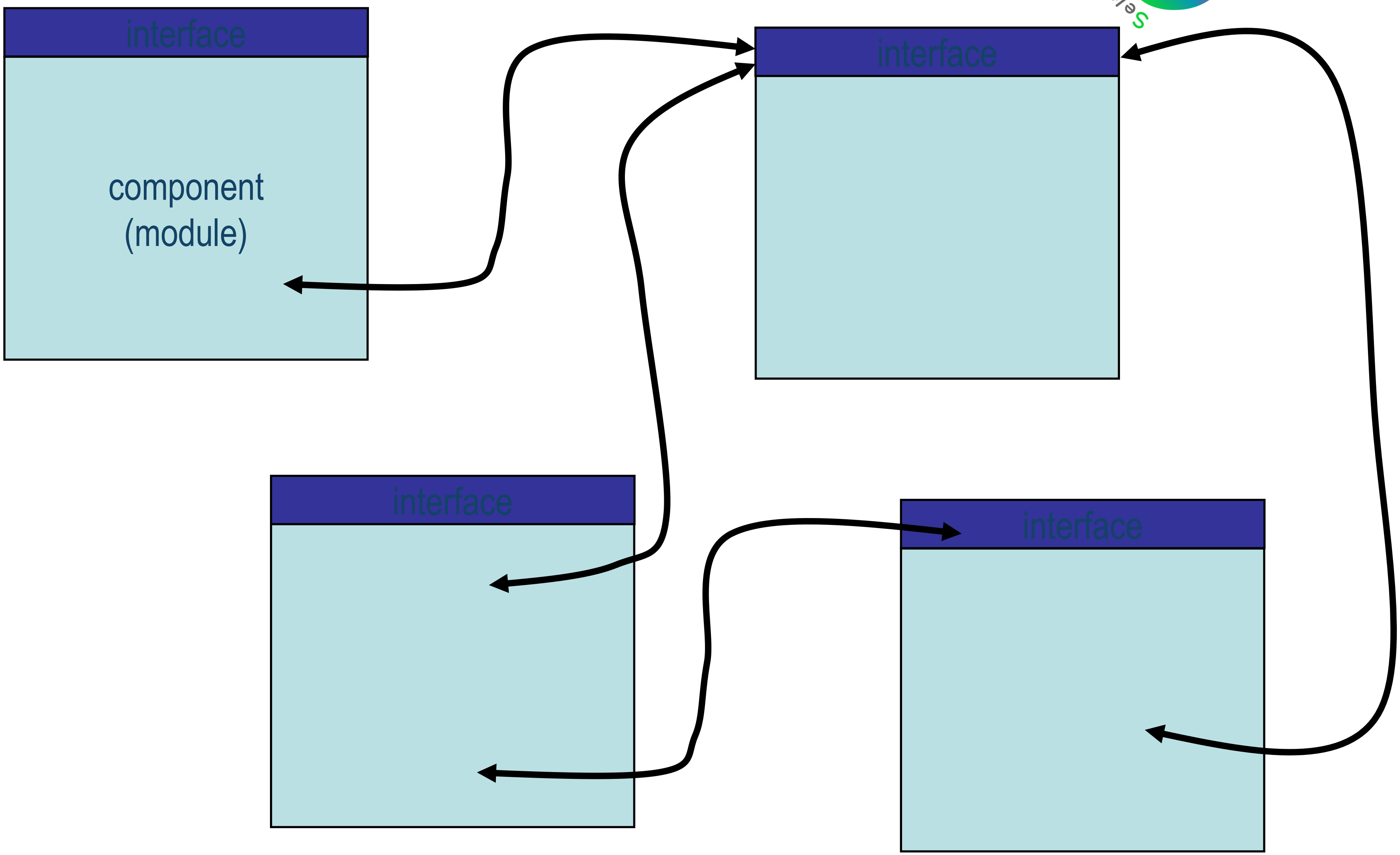


visible to clients

stable

hidden to clients

volatile


interface
component
(module)
interface
interface
interface

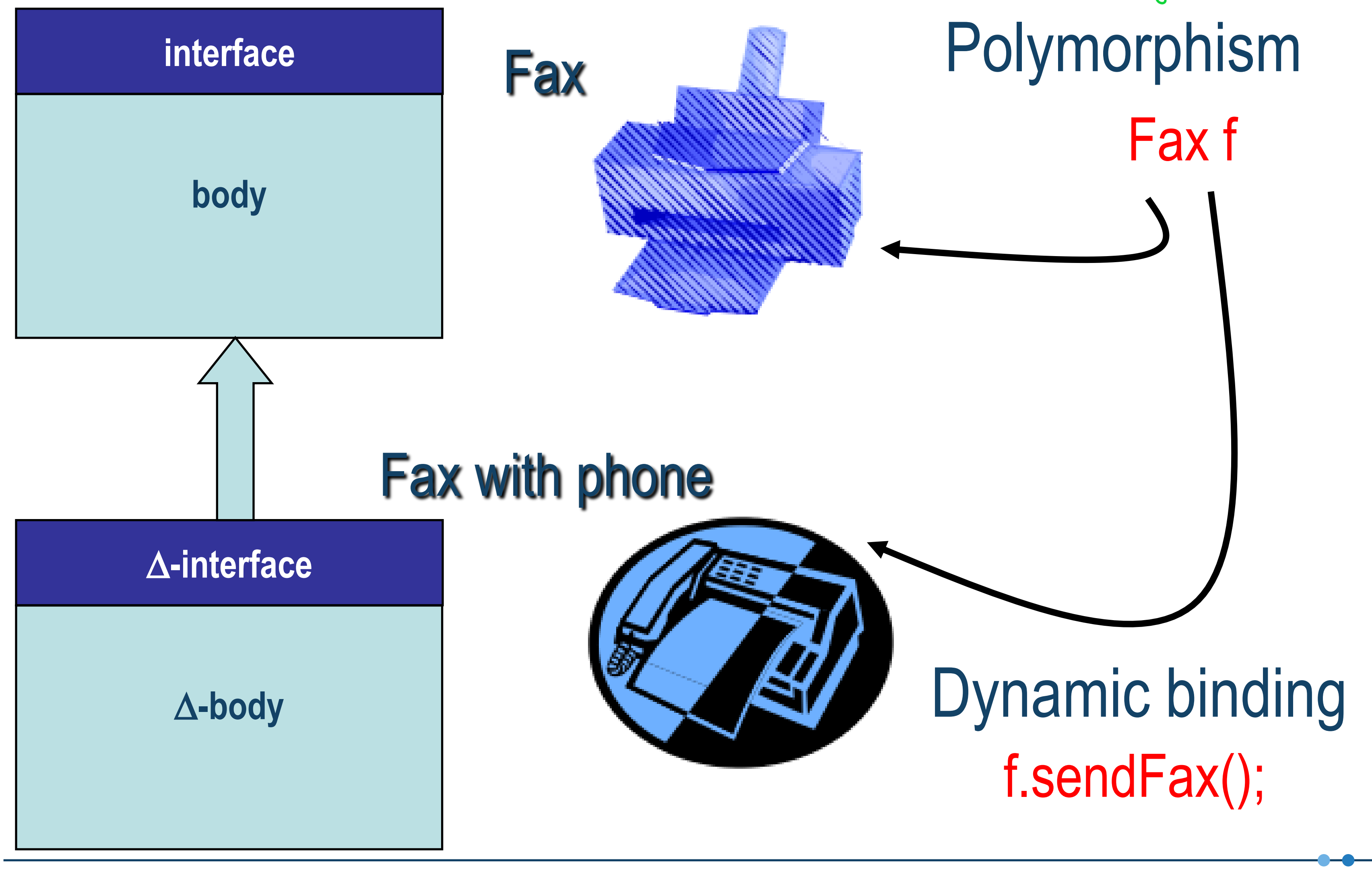
OO methods

- Support to **design for change** through **encapsulation**
  - data abstractions
- Support to **dynamic binding** to add flexibility to modularity
  - dynamic binding constrained to achieve statically checkable strong typing

Dynamism and type safety

- New **subclasses** as **change units**
- Changes are **not disruptive** (just added to old software, also at run-time)
  - methods to invoke on objects may become known at run time
- If changes are **anticipated** and changes can be cast in the subclass mechanism, dynamic evolution and dynamic binding can co-exist with static checking (and **type safety**)

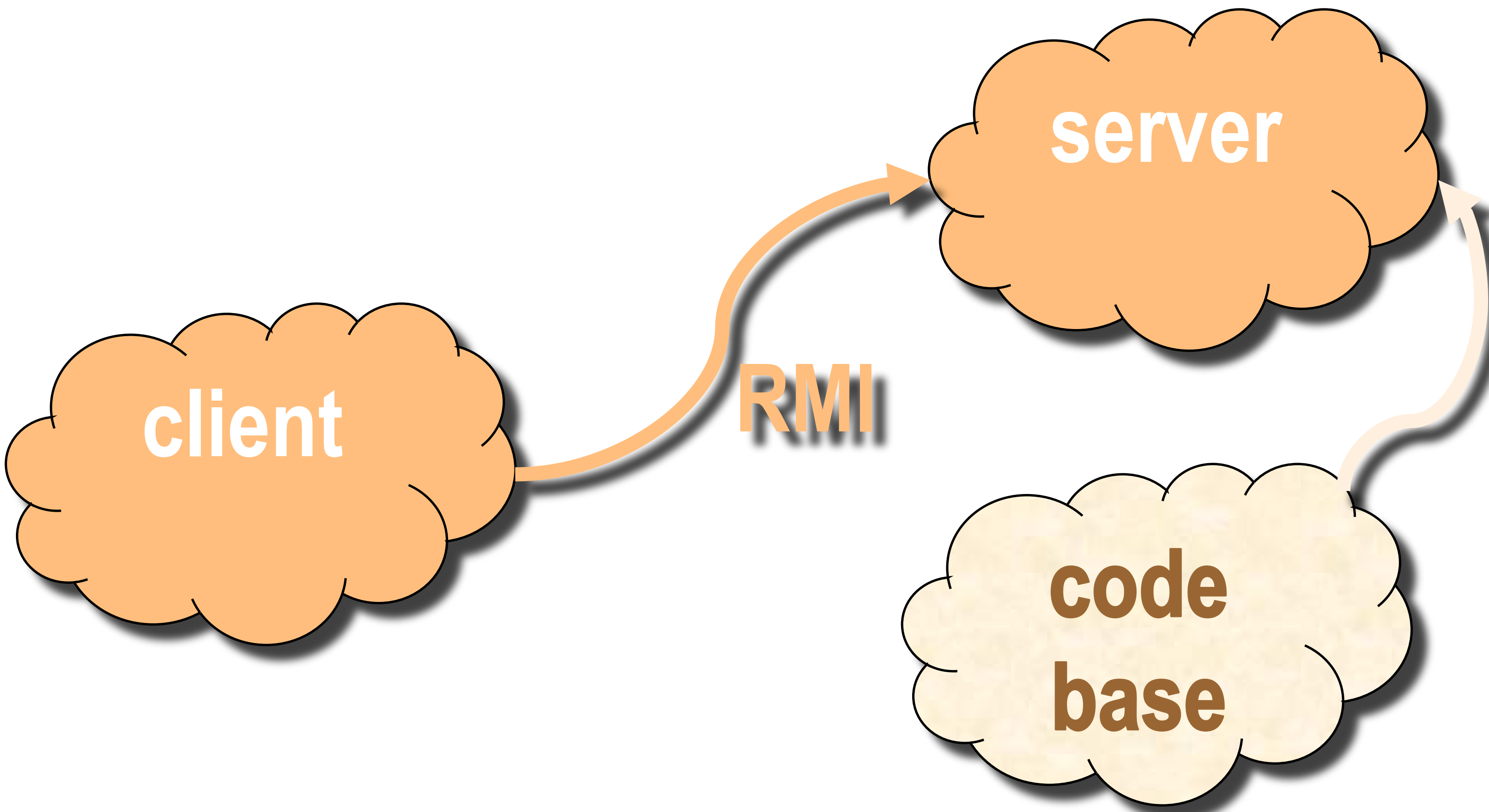
OO design


interface
body
Δ-interface
Δ-body
Fax
Fax with phone
Polymorphism
Fax f
Dynamic binding
f.sendFax();

A further step: distribution

- Components may be deployed in different address spaces
- Distinction between **logical** structure and **physical** structure
  - modularity vs. allocation
  - goal of a seamless transition from centralized conception to decentralized deployment

Binding crosses network boundaries



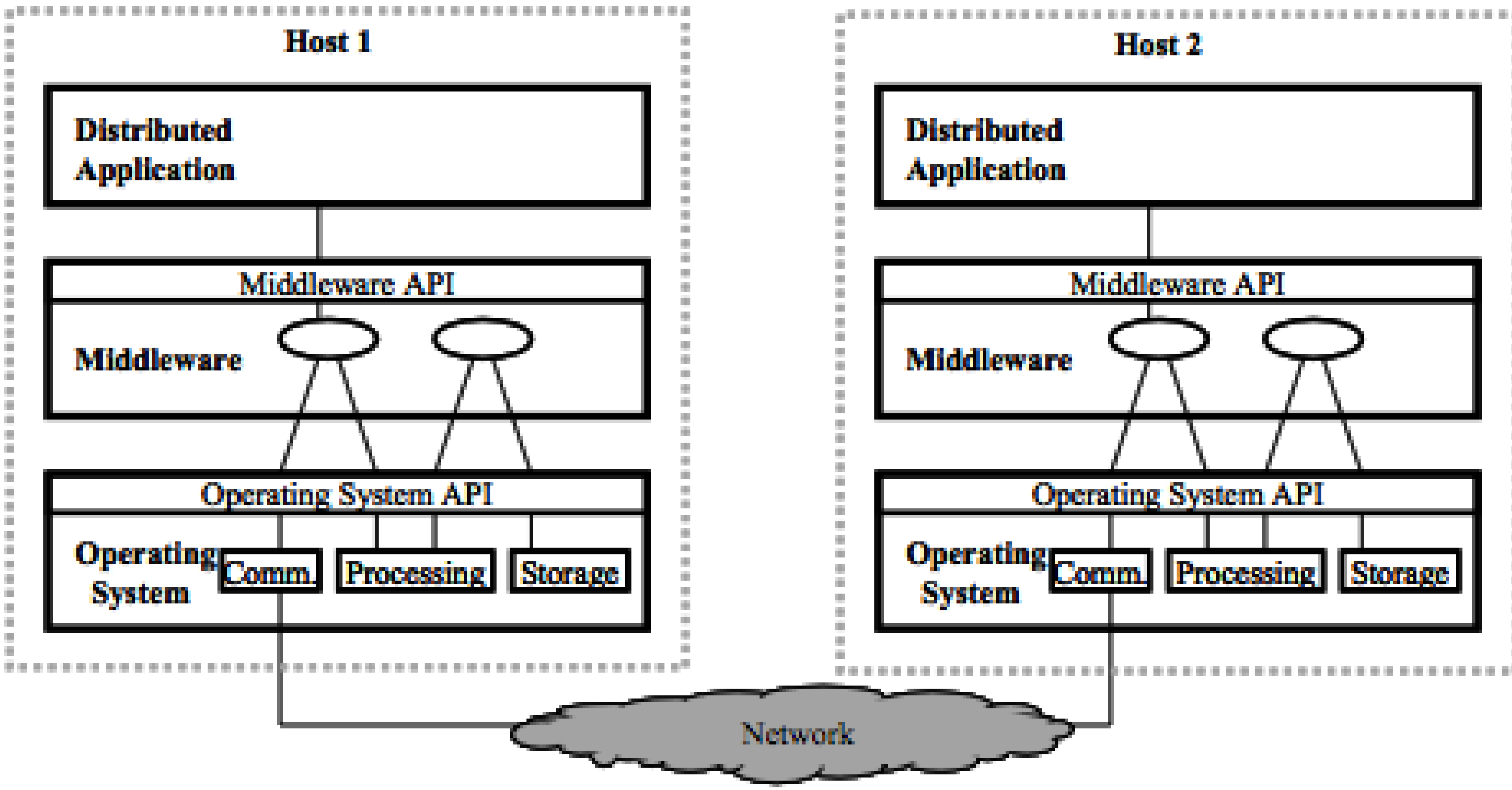
The "components" scenario

- Systems not developed from scratch, but rather out of existing parts
  - Decentralized developments
  - Bottom-up integration vs. top-down decomposition
    - ***Component-based development***
- **From** software developed by a single organization
- **To** components developed by independent organizations with different degrees of contractual obligations
    - **No control over evolution of components**

Gluing software becoming dominant

- Distinction between **components** and **connectors**
- **Middleware** provides binding mechanisms
  - Middleware as a decoupling layer
    - separation of concerns
      - separate component logic from intricacies of communication/cooperation

Middleware


Host 1
Distributed Application
Middleware API
Middleware
Operating System API
Operating System
Comm.
Processing
Storage
Host 2
Distributed Application
Middleware API
Middleware
Operating System API
Operating System
Comm.
Processing
Storage
Network

Software product lines

- A software product line (SPL) is a set of software systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way
  - e.g. a product line for TV sets, or for the software of different cars
- **Variants** and **variation points**

Summing up

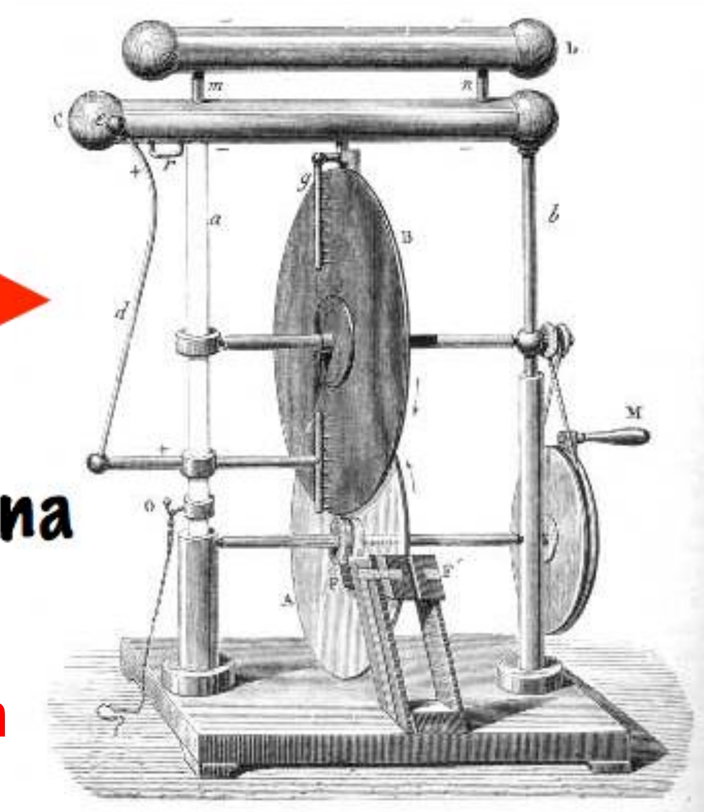
- **Product**
  - monolithic → **modular**
  - centralized → **distributed**
  - hard to modify → **controllable changes**
  - static, closed → **constrained dynamic compositions**
- **Process**
  - single authority → **multiple (components)**
  - pre-planned, 1 end → **incremental, iterative, spiral**

The *machine* and the *world*

World (the environment)

Machine

Domain properties (assumptions)



Shared phenomena

Goals

Requirements

Specification

“Open world”

- In an open world **requirements** and **domain** change continuously
- Domain includes “parts” (components, services) → **multiple ownership**
  - No single stakeholder oversees and controls all parts
  - Parts may change over time in an unannounced manner
- Increasingly, reactions to changes must be **self-managed**

Adaptation and evolution

- **Adaptation** is the ability of software to detect changes and react to them in a self-managed manner
- **Evolution** requires the designer in the loop
- To cope with open-world requirements we need to empower the run-time behavior of software to
  - improve its self-managing capabilities
  - provide designer support for evolution

The challenge

- Can we support continuous **adaptation and evolution** without compromising **dependability**?
  - *the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers*
- We need to understand which are the **invariant** properties that should be preserved by changes and ensure that they hold

Self Managing Situated Computing

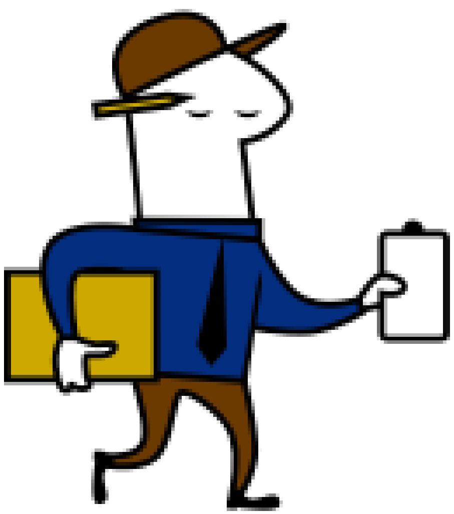
Sources of change? (1)

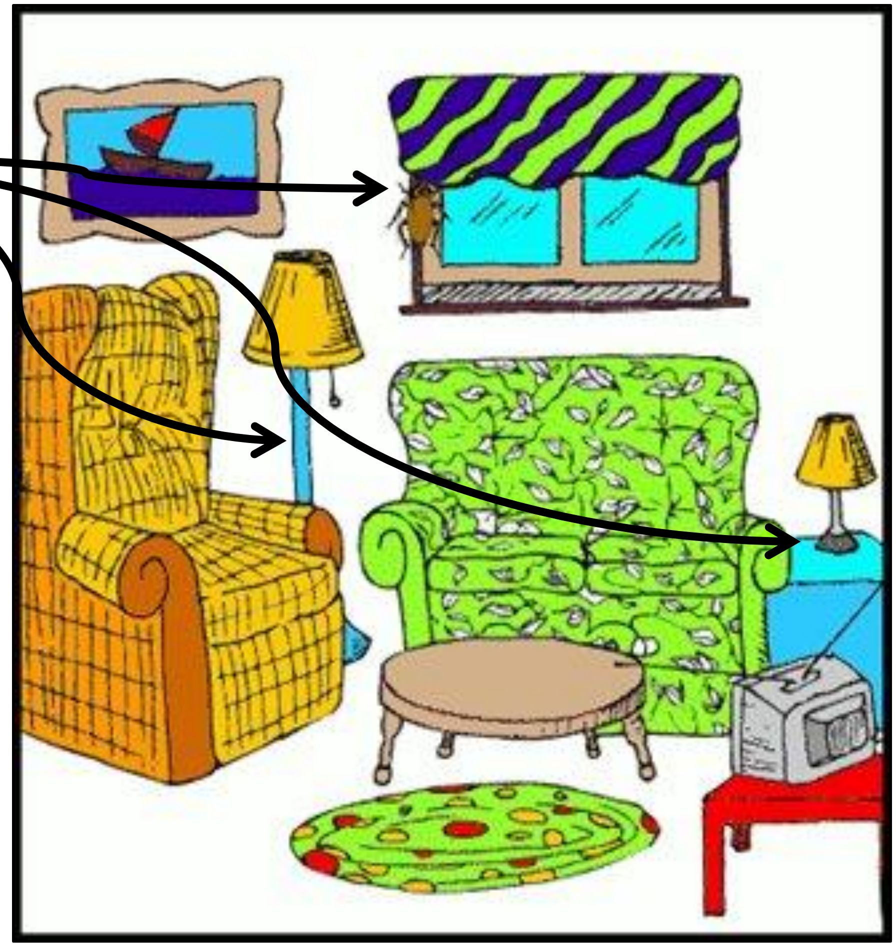
- Changes originate in the **interaction** with the **physical environment**
- Implied by **pervasive/ubiquitous computing** requirements
  - mobility and context awareness
  - ambient intelligence and disappearing computer
    - external world changes unpredictably
      - because context changes
      - because new computational objects are encountered + old disappear

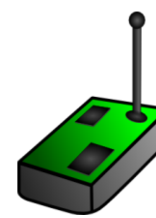
Pervasive computing

active devices

offering services







Context awareness

- Dynamic context-aware bindings established to deal with dynamic **context changes**
  - *invocation of a print service binds to a printer based on proximity*
- Context is not just *location*, nor just *physical*
  - light, temperature, ... emotional
  - e.g., light the room bound to
    - open window shades
    - switch electric light on

    depending on weather condition

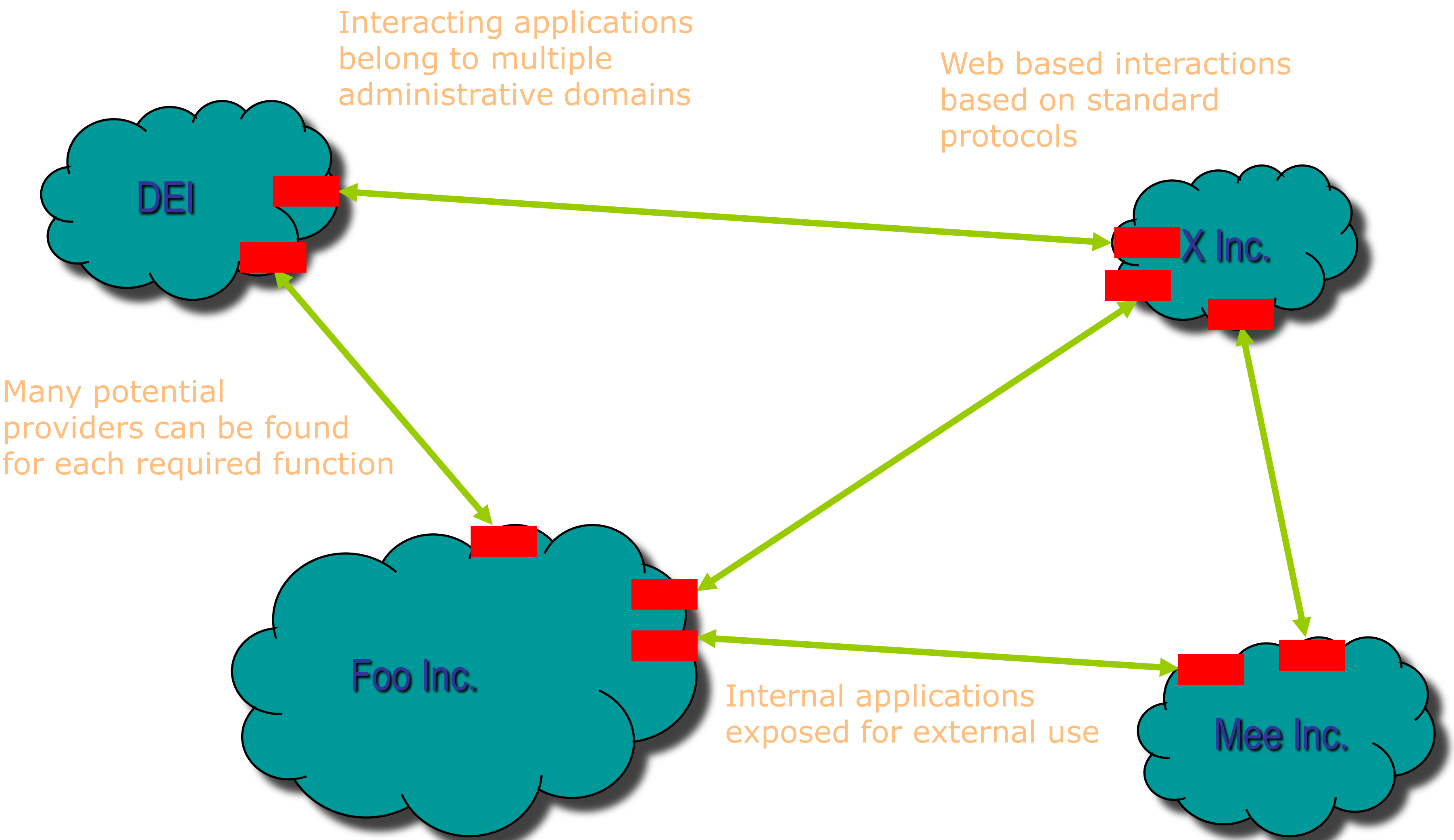
Sources of change? (2)

- Changes originate in the **business** world
  - agile networked organizations
  - fast organizational responses to rapidly changing requirements
    - intra and extra organization changes require continuous adaptation of the information system

Service orientation

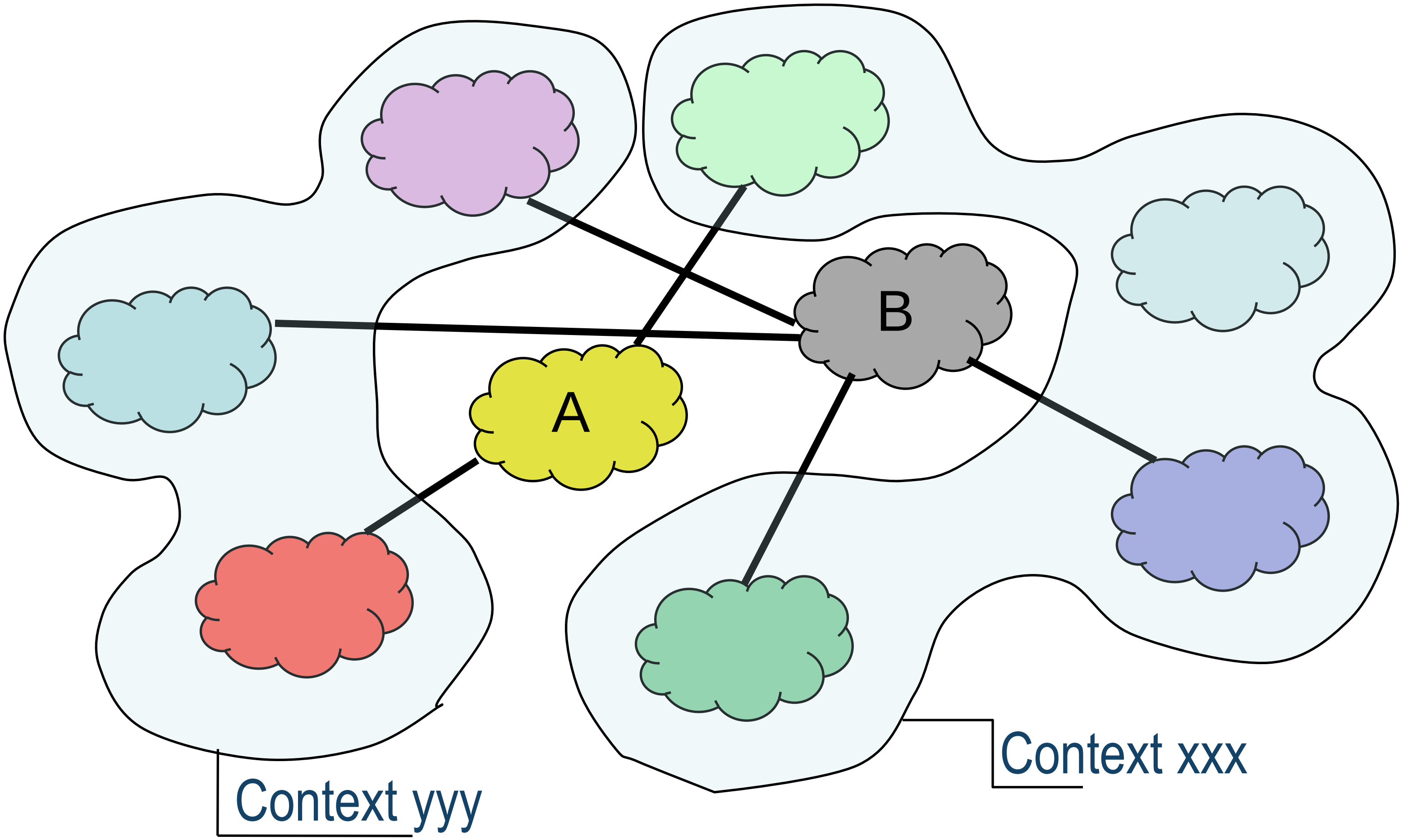
- The central role of service, as **unit of value**
- Service-oriented **business**, **process** and **product** architecture to support
  - dynamic, goal-oriented, opportunistic federations of organizations
  - rapidly adapting to changing requirements

Networked organizations



What do we need?

Flexible and dependable composition schemes



Components/services

- Both are parts developed by others
- **Components**
  - are normally selected at design-time
  - cannot change after they have been deployed
  - are run by the application owner
- **Services**
  - run autonomously
  - can be discovered and selected dynamically
  - can be invoked remotely

Self Managing Situated Computing

What do we need? Ability to *detect* change

- We need to get real data from the world through (abstract) sensors; e.g., by activating suitable probes
  - ***MONITOR***
- We need to transform data into information
  - ***LEARN***

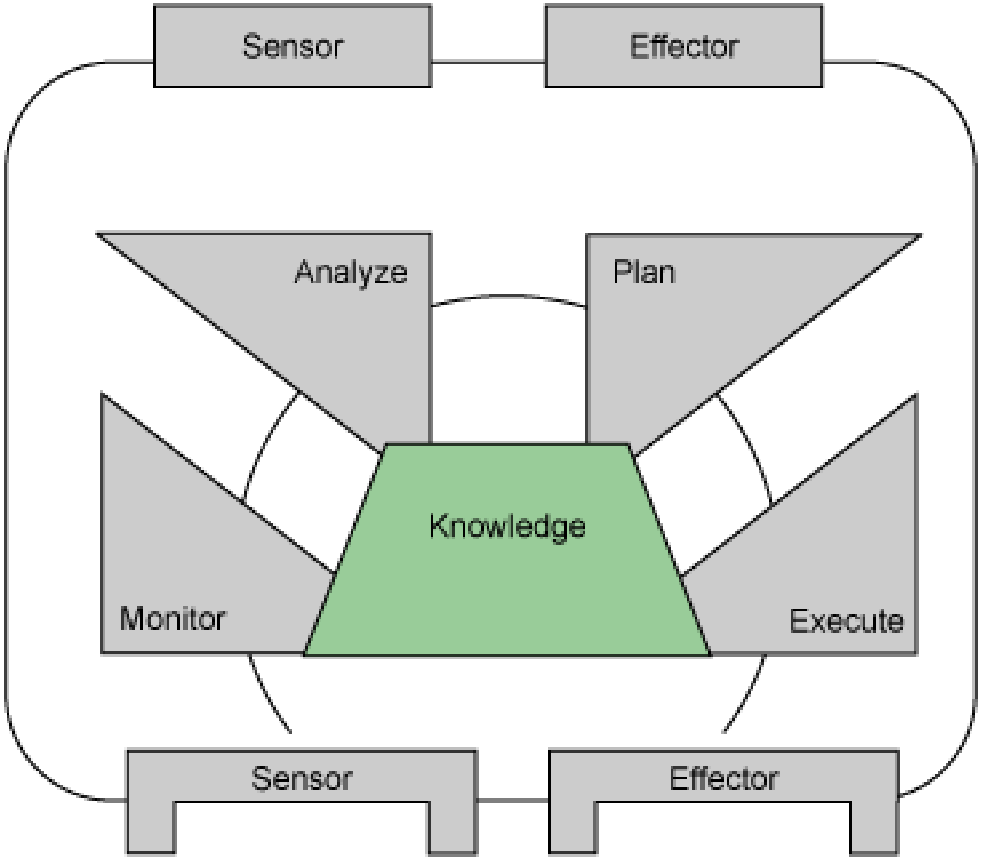
What do we need?
Ability to *react* to change

- How can detected changes be used to react by generating a feedback loop to “development” activities?
- Different timescales require different strategies
  - Off-line, with human intervention
    - Re-design/re-deploy/re-run
  - **On-line, self-managed**
    - **A must for perpetual applications**

evolution

adaptation

The MAPE autonomic manager



MC Huebscher, JA McCann, A survey of autonomic computing—degrees, models, and applications - ACM Computing Surveys, 2008

Where do we focus next?

- **Architecture** (and **languages**)
  - how can an architecture support/facilitate adaptation and evolution?
  - can languages help? why?
- **Specification** and **verification**
  - how can specification and validation be performed for continuously evolving/adapting systems?