# Evolution of software composition mechanisms

Carlo Ghezzi
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
carlo.ghezzi@polimi.it



# Outline

- Historical evolution of composition mechanisms for software
  - From monolithic to highly decentralized
  - From static to highly dynamic
- Evolution at "product level" in parallel with evolution of "process level"
- Challenges
- Some research directions
- Conclusions



# The concept of binding

- Architecting software requires defining relationships among elements
- Relationships define the logical/physical structure
- Binding is the establishment of a relationship



# More on binding

- Binding occurs at all levels
  - programming level
    - a variable refers to its type, value, scope...
    - a subclass refers to its parent class
  - component level
    - a component refers to other components through a *use* relationship

the focus here is on binding as a the gluing mechanism among components



# Binding time and persistence

- When is the binding established?
  - typical distinction between run-time and "pre" run-time
- How stable is the established binding?
  - can it change?
  - how does it change?
    - explicit
    - automatic



# Evolution thread

- Continuous evolution to accommodate increasing degrees of
  - dynamicity
  - decentralization

  to achieve flexibility
- Concurrent evolution at the process/organizational/business level

## Early days: the “static” scenario (1)

- The **closed**, **static** , **centralized** , **fixed** world assumption
  - requirements are there
    - just elicit them right
  - they are stable
    - if not, we got them wrong
  - changes should be avoided
  - static and centralized system compositions, frozen at design time
  - monolithic, systematic, top-down processes



## Early days: the “static” scenario (2)

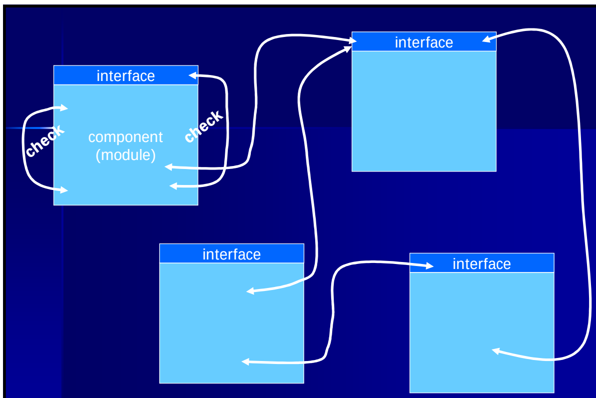
- Response
  - The waterfall process model
    - Refinement, from clearly and fully specified requirements down to code
    - Top-down development → formal deductive approaches
  - Programming languages and methods producing static verifiable architectures
    - static binding → static type checking



## Early days: the “static” scenario (3)

- Software structure
  - From monolithic
    - Changes implied recompilation
  - To separately compiled parts
    - Linked statically and then loaded
    - Changes required partial recompilations
  - Interface separated from implementation
    - From FORTRAN to Ada








## General lessons learned

The source of change is in the world

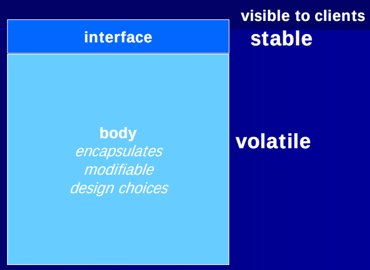
- Requirements cannot be fully gathered upfront
- Requirements cannot be frozen
- Requirements intrinsically decentralized, complete control and pre-plan illusory
- When changed, impact whole product/process



## Initial solutions

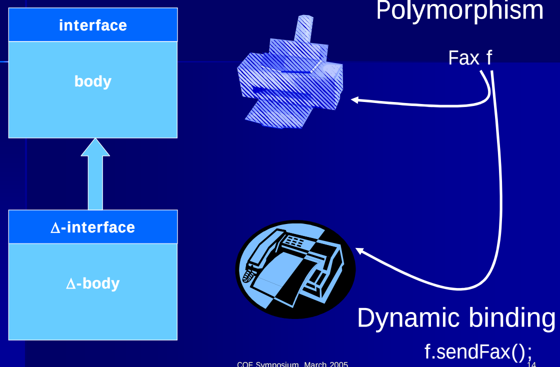
- Evolutionary process models
  - Spiral, prototyping-based
- Design for change
  - Information hiding
  - Careful distinction between
    - specification & implementation
    - interface & body
- Object oriented design and languages
  - Accommodate limited anticipated product changes
  - Towards an open world

# Design for change






# OO design



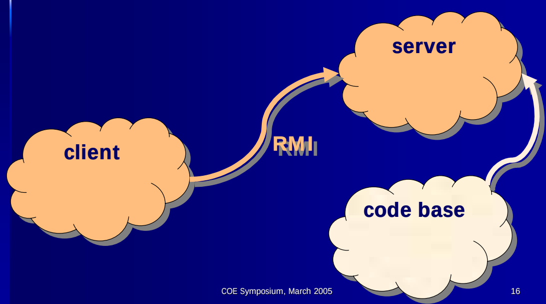



# Open world and type safety

- New subclasses (and new objects) defined as the system is running → methods invoked 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)



# Binding may cross network boundaries






# Conceptual tools

- Distinguish between **logical** structure and **physical** structure
  - modularity vs. allocation
- The goal of a seamless transition from centralized to decentralized deployment



# The "components" scenario

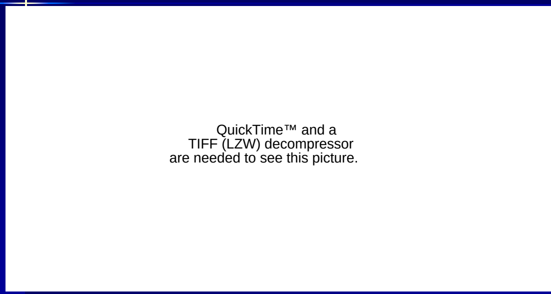
- Systems not developed from scratch, but rather out of existing parts
  - Decentralized developments
- Bottom-up integration vs. top-down decomposition
  - Component-based development

## Gluing software becoming dominant

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



## Middleware



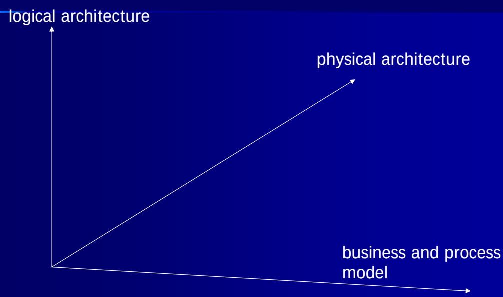



## Mobile scenarios

- With mobility the structure may evolve dynamically
  - physical nodes may appear and disappear
- Logical mobility also possible (i.e., software/agents migrate)
  - physical and logical topology may change dynamically



## Decentralization dimensions






## Dynamicity and decentralization in processes and organizations

- **From** software developed by a single organization or by a group of collaborating organizations
- **To** components developed by independent organizations with different degrees of contractual obligations



## The old world

- Product
  - monolithic
  - centralized
  - static, closed
- Process
  - single authority
  - pre-planned
  - monolithic

## Achievements

- Product
  - monolithic → modular
  - centralized → distributed
  - static, closed → controlled dynamic binding
- Process
  - single authority → static task decomposition
  - pre-planned → pre-planned evolution
  - monolithic → spiral, agile, extreme



## A vision: the "global computing" scenario

- Applications dynamically federated out of distributed components, even at run time
- Motivations
  - the network as a bazaar of components
  - mobility, ubiquitous computing
  - multimodality
- This pushes dynamicity, decentralization and distribution to unprecedented levels
- Problems range from technical to business models



## Problem scale

- From in-the-tiny
  - sensor networks
    - huge numbers of autonomous cooperating devices
- To in-the-large
  - web services
    - different scales possible



## Challenges—1

- How to design components?
- How to federate them?
- How to manage composite systems (without centralized control)?
- How to reason about the "total" quality of provided services?
- What types of business models?



## Challenges—2

- What kind of interface should components provide in such a fluid environment?
  - Interface should support establishment of "contracts
    - Beyond import/export typed lists
- How to ensure a correct "global" behavior?
  - Need for new theories and models?



## Service-oriented architectures

- From now on, I cast my presentation in the context of service-oriented architectures
- in particular, web services

## A definition

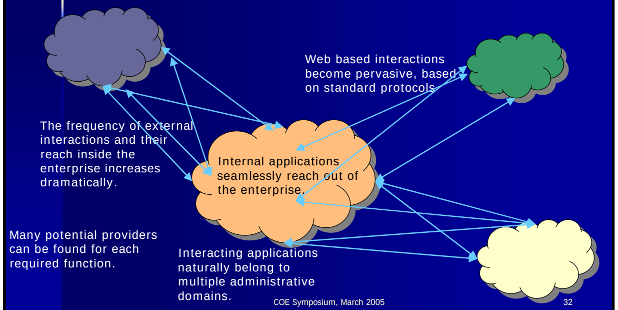
"Web services are a new breed of Web application. They are **self-contained**, **self-describing**, modular applications that can be **published**, **located**, and **invoked** across the Web. Web services perform functions, which can be anything from simple requests to complicated business processes. ...

Once a Web service is deployed, other applications (and other Web services) can discover and invoke the deployed service."

*IBM web service tutorial*



## Motivation: networked enterprises






## More on "service" (1)

- Component encapsulating a business function of possible value for others
  - Different level granularity – coarse grained business services vs. fine grained objects
- Services must support **explicit contracts** to allow independent party access
  - Allow for SLAs that deal not just with functionality
- Services can be the basis for **service compositions**
  - New value is created through integration and composition
  - New components are recursively created



## More on "service" (2)

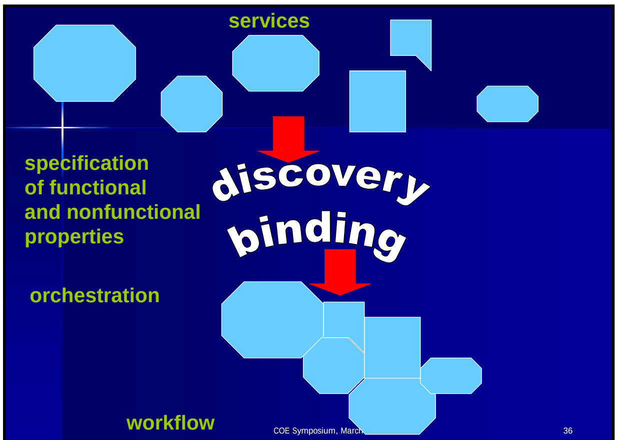
- Services lifecycle phases
  - specified
  - published
  - discovered
  - negotiated
  - delivered
  - composed
  - monitored



## Types of services

- Atomic services: run to completion without interaction with service client
  - a search service
- Service packages: logically related, not interacting, group of atomic services
  - reservation for different theatres
- Workflow services: workflow includes composition of other services
  - state is shared
    - buying a book





## Discovery and binding

- Design time
- Deployment time
- Run time

unstable, evolving environments

ubiquitous, mobile applications

self-organizing behavior



## Dynamic SOAs

- Composite services are specified by workflows
- Workflows contain abstract service invocations
- Concrete services bound dynamically, at run time



## Dynamic SOAs

- Dynamic discovery and dynamic binding
  - the “broker” role
- Self-organizing, self-healing composite services
- Opportunities
  - enjoy use of the “best” available services
  - binding can be “context-aware”
- Threats
  - many things can go wrong



## Service contracts

- Contracts in terms of pre and post conditions
- Exposed services specify what they promise to fulfill
- Workflows specify what they expect from concrete services
- A broker negotiates a contract upon which a binding is established



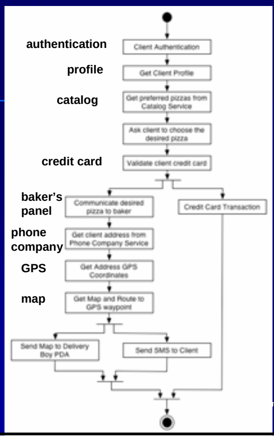
## Threats: contracts can be broken

- We bind to a concrete service that does not satisfy its stated specification
- The bound service evolves autonomously and breaks the contract
- The service is “temporarily down”



## Consequences

- Traditional good software engineering methods stress static reasoning on software architectures
- This has little value in the new world of run time variability
- Improved techniques are needed to monitor and react to unexpected deviations at run-time
  - reaction can lead to self-healing systems




## An example

QuickTime™ and a TIFF (Uncompressed) decompressor are needed to see this picture.



## Monitoring

- In an open environment, reacting to abnormal behaviors is of greater importance than in closed environments
- Recovering from problems has to do with knowing what to do when something goes wrong. But before that we have to:
  - Decide what should not go wrong
  - Detect if and when that happens

This is where monitoring comes in!



## Monitoring

- Defensive programming
  - Workflow handles timeouts and exceptions raised by remote service
- External contract monitoring
  - Collect data
  - Process data
  - Notify workflow



## An assertion-based approach

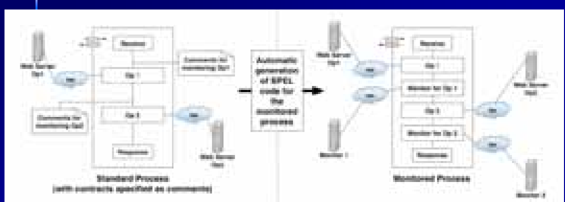
- Contracts expressed in terms of pre- and post-conditions
- These assertions are inserted as comments into our process definition
- External monitors (services) are used to check the assertions

QuickTime™ and a TIFF (LZW) decompressor are needed to see this picture.

We cast our proposal in terms of BPEL processes



## BPEL 2 BPEL Transformation






## What are the advantages?

- Limited design overhead
  - Comments are easy to add and transformation to a monitored BPEL process is automatic
- Business logic remains separate from the monitoring logic
- We stick to BPEL
  - No need for a special workflow engine
- Monitor alternatives
  - Different implementations, possibly co-existing

## Recovery and repair actions

- Retry
  - transient faults
- Rebind
  - find a suitable replacement for previous service
- Restructure (local reconfiguration)
  - find a collection of services that satisfies request, or merge given collection into one



## Restructure

- Workflow process as a graph
- Graph transformation rules express possible local changes
  - sequential composition
  - parallel composition
  - branch composition



## What kind of problems due we monitor?

- Three different kinds of problems:
  - Timeouts
  - External exceptions -> these can be implementation errors in the services or mismatches between how we call the service and how the service expects to be called
  - Functional (and/or non functional) contract enforcement-> this requires an external monitor service



## Monitoring contracts

- An external monitor is needed to monitor a functional or a non-functional contract
- We implemented two different monitors for our assertion-based approach:
  - The first uses C# and .NET framework
  - The second uses CLIX and XlinkIt



## Conclusions (1)

- We are moving towards unprecedented degrees of flexibility, dynamicity, and decentralization *at all levels*
- New challenges to correctness/ reliability, security, performance
- Crucial to understand how we can build on previous approaches and where new ones are needed



## Conclusions (2)

- The global computing scenario requires more intelligence to be moved to run time
- Traditional pre-deployment tools must be moved to run time in a seamless fashion
  - continuous testing
  - run-time verification

## Our work

- We have seen an initial attempt to use defensive programming and an assertion-based approach to monitoring to make system partially self-healing
- The advantage of our approach is that it can coexist with current “standards” developed for SOAs



## Our work

- We developed prototypes for assertion-based monitoring and recovery mechanisms
- We are completing a second wave of prototypes that take into account performance and usability issues
- We will address non-functional properties
- We will try to achieve a better separation between business and monitoring logic to support different monitoring activities for different stakeholders
- Definition of more complex exception handling routines



## Acknowledgments

- This work is mainly funded by the EU IP SeCSE (3 years project; just started)
- Members of the group
  - C. Ghezzi, L. Baresi, E. Di Nitto, S. Guinea and several graduate students
- More on this
  - L. Baresi, C. Ghezzi, S. Guinea, “Smart Monitors for Composed Services”, Int.l Conf. on Service Oriented Computing, New York, Nov. 2004.
  - L. Baresi, C. Ghezzi, and S. Guinea, “Towards Self-healing Compositions of Services” Proceedings of PRISE'04, First Conference on the PRInciples of Software Engineering, November 2004, Buenos Aires, Argentina.