The challenges of open-world software

Carlo Ghezzi
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
carlo.ghezzi@polimi.it
What is "open world"?

- Software methodologies and technologies have been evolving from
  - closed world
  - to controlled evolution
  - to open world
- Progress in the field may be analyzed through this perspective

The external world with which the system interacts is fixed and known. System re-design as system is running. Architecture evolves as fixed, monolithic, and centralized system is running.
Plan of the talk

• Briefly revisit software progress through the closed to open world evolution perspective
• How object orientation fits into the picture
• Identify challenges and opportunities
  – In particular, dependability issues
Postulates

• Open world asks for dynamic software compositions
• Software structure may change at run-time, even in unanticipated ways
• We need to rethink the way we validate systems
  – from pre run-time validation to perpetual validation
Viewpoint

• Focus on composition/structuring
  – how is an application made out of parts?
    • how/when is binding among parts established?
  – what is its structure/architecture?
also at process level
  – how is an application's lifecycle structured?
  – how are phases and activities organized?
Archeology

• Software engineering "officially" born in 1968, as a recognition of
  – increasing importance of software in society
  – failures, poor quality, out-of-control costs
and a quest for methods to support industrial-strength quality
  – because software production did not follow any precisely formulated process
    • continuous iteration of coding and error fixing
Early day assumptions

- Monolithic organizations
  - centralized solutions
- The **closed world** assumption
  - the boundary between the real and the virtual world is clearly identified and fixed
    - owner of problem and owner of solutions clearly identified
  - requirements are there, they are stable
    - *just elicit them right!*
  - changes considered harmful
    - *avoid them! they disrupt the "normal" flow! they are the culprit of schedule and cost problems!*
Early day 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, centralized, verifiable architectures
    • static and centralized system compositions, frozen at design time
Early lessons learned

• Requirements cannot be fully gathered upfront
• Requirements cannot be frozen
• Requirements intrinsically decentralized, corporate and pre-plan illusory
• When changed, they may impact whole product/process
• Evolution increasingly intrinsic to software
• It is NOT a “post-delivery” nuisance
Controlled evolution

• Evolution should be anticipated and controlled
  – progressive departure from the closed (and fixed, static, centralized) world assumption

• Process level
  – evolutionary models
    • incremental, prototyping-based
Product level

- Methods
  - likely changes anticipated at requirements level
  - design for change (Parnas)
    - information hiding
    - specification/interface vs. implementation/body
- Technology (languages)
  - from monolithic structures
    - changes imply complete re-compilation + re-deployment
    - to incremental construction mechanisms
    - partial re-compilations + re-deployment
    - interface separated from implementation
  - to distributed solutions
    - client-server architectures
OO design and languages: further controlled evolution

• Accommodate limited anticipated dynamic product changes
• New subclasses (and new objects) defined even as the system is running → methods invoked may become known at run time
• Partially open world + type safety
  – if changes are anticipated and they can be cast into the subclass mechanism, dynamic evolution and dynamic binding can co-exist with static checking (and type safety)
OO design

Fax

Polymorphism

Fax with phone

Dynamic binding

f.sendFax();
Binding may cross network boundaries

Diagram:
- Client
- RMI
- Server
- Code base
Components: opening the process world

- Systems not developed from scratch, but rather out of existing parts
- Open world at process level
  - decentralized developments
  - multiple owners/stakeholders
    - improved flexibility/reduced control
  - components cooperate through middleware
Summing up

• The need for adapting to external changes
  – of the business world
  – of the physical world
has been the dominating force

• We have been *reasonably* successful at supporting it without compromising the dependability of our solutions through *controlled evolution*
What is happening today?

• Open world
  – Unprecedented degrees of change
  – "Perpetual beta"
Where are the sources of change? (1)

• Changes originate in the business world
  – agile networked organizations
    • dynamic, goal-oriented, opportunistic federations
    • reacting with fast responses to rapidly changing requirements
    • intra and extra organization changes require continuous changes in their information system
Where are the sources of change? (2)

- Physical mobility generates environment changes in ubiquitous/pervasive computing settings
  - request for context-aware dynamic bindings
    - invocation of a print service binds to a printer based on proximity
    - request to light a room binds to actuator that opens shades or to switch turning electric light on depending on external conditions
Open world

• In an open world requirements change continuously and unpredictably

• The boundary between the systems we build and the real world cannot be frozen
  – In the extreme case, it changes as the systems we build runs and must continue to provide service
    • the system structure cannot be frozen
    • mechanisms needed for the structure to evolve dynamically

• Compositions change because both components and the structure change

• No single authority is in charge of all parts
Closed world

- In a closed world there is a sharp separation between **pre**-runtime and runtime
  - structuring
  - verifying
  - deploying
  - executing

- Changes require switching from runtime back to development time where software is changed and validated
From components to **services**

- Both are developed by others
- Components are run in the application's domain, they become part of the application
- Services are run in their own domains
- Normally, components chosen and bound at design/construction time
- Services chosen and bound at run-time
- Services must support explicit contracts to allow independent party access
  - allow for SLAs that deal not just with functionality
- Services can be composed to form new services
Ownership

• Traditional applications are largely owned by the organization that runs them
• Components are run but not really owned
  – organization depends on owner for enhancements
• Services are used, not run
Binding mechanisms

- **Two-phase explicit**
  - discovery-based
    - publication states all functional and nonfunctional properties
    - discovery/selection may involve optimization/negotiation
- **Implicit**
  - via some "logical" coordination space
    - E.g., publish/subscribe
    - *by contract?*
Discovery-based binding

**Service Specification**

**Discovery Agency**

**Query**

**Publish**

**Interact**

**Service Requestor**

**Request**

**Response**

**Service Provider**

Requirements
Implicit binding: Publish-Subscribe

- Asynchronous communication mediated by a *dispatcher*
  - anonymous
  - multipoint
  - implicit addressing
    - subject vs. contents-based
- Application components
  - *subscribe* to relevant message patterns
  - *publish* messages
- The dispatcher matches published messages against previously issued subscriptions
- Support dynamic addition, removal and replacement of application components
Binding by contract

required interface \[\text{COMPONENT}\] provided interface

interfaces described by a SPECIFICATION

**Specification** describes
- signature
- functional behavior
- nonfunctional behavior

We must manage contract violation
Potential benefits

• Clear separation between the "what" and the "how" and "when"
  – different policies possible
    • early/late
    • context-aware
    • optimizing certain figures of merit
      – bind to the "current best"
  – "self-organization" possible
  – "rich" interface descriptions needed
    • "contracts" specify QoS
The challenges of open world

• How can flexibility and dynamism coexist with the required dependability?
• All phases of software development are affected and must be re-thought
  – requirements
  – specification
  – design&implementation
  – verification&validation
Verification & validation

- **Good news** is flexibility and dynamism
- **Bad news** is continuous V&V
  - Late binding + dynamic compositions add flexibility at the expense of reduced safety
  - Full range of binding regimes imply full range of V&V activities