Decentralized software development

*Pitfalls and challenges*

*A software engineering viewpoint*

Carlo Ghezzi
DeepSE Group
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
carlo.ghezzi@polimi.it
Plan of the talk

• The software process became increasingly distributed and decentralized
  – Why?
  – How?
• Can we identify basic paradigms?
• Pitfalls and challenges?
• Opportunities?
• How is software engineering education affected?
Premise: terminology

• Decomposition
  – separation into simpler constituents
• Distribution
  – sharing something out among a number of recipients
• Decentralization
  – transfer authority from central to local government
• Decentralization presupposes distribution
• Distribution presupposes decomposition
• Proper decomposition key to decentralization
Premise: process and product

• Process and product are intertwined
• The software process influences the quality of products
  – it can enforce "good practices" and stress certain product qualities
    • e.g., a specification & verification oriented lifecycle favors dependability; prototyping may favor usability
• has its own qualities
  – e.g., cost, timeliness, …
• The quest for improved processes has been pursued since the late 1960s
My goal in this talk

• Analyze decomposition, distribution and decentralization (DDD) in the context of process and highlight their relationships with product, both in terms of
  – expectations
    • goals to achieve, potential benefits
  – pitfalls
A brief history of DDD

• In pre-history, software was developed by its expected users

• A first kind of DDD in the process became soon necessary to support two stakeholder roles
  – user/customer
  – developer

• Motivation
  – specific technical profession to dominate complexity

• Pitfall
  – requirements!
A brief history of DDD

• Initially software was **monolithic**

• Complexity/size of product caused hard to predict/manage projects, and hence failures

• Motivation
  – DDD at product level to achieve DDD at process level which would facilitate prediction/control
    • decomposition of process lifecycle into phases
    • decomposition of product into subsystems/modules

• Pitfalls
  – principles/methods needed to support proper DDD
  – technology (languages, tools)
Key enablers of DDD

• Principle/method level
  – Specification/implementation, modularity, encapsulation, interfaces (Parnas)
  – Abstract data types (Liskov)
  – Design by contract (Meyer)

• Technology level
  – Modular languages
  – Object-oriented languages
    • C++, Eiffel, Java, C#
Traditional software process

• Software developed from scratch
  – limited internal reuse
    • mostly implicit (skills, domain knowledge, …)
• Mono-organization (multiple roles)
  – single "owner" of process and solution
Towards further DDD

• Three main kinds considered
  – Component-based (including libraries)
  – Offshoring
  – Service-based

• Main motivations
  – cost reduction
  – faster lifecycle, timeliness
  – expected benefits from specialization
Pitfalls: preview

- DDD cannot be viewed from a purely economical/organizational aspect
- They require rethinking the whole process and the methods used
  - certain criticalities of software development become even more critical
  - new criticalities arise
Component-based (and libraries)

- "Owns" the application
- Is responsible for the final app wrt the user
Benefits and pitfalls

+ Specialization
+ Standardization
+ Reduced development time
+ Reduced testing, improved correctness
  • components more likely to be correct
  – Solution must match components
  – Dependency on supplier for updates
Offshoring

- One "owner" of the application
- Responsible for the final app wrt the user

Problem

concurrent development
Benefits and pitfalls

• Brings traditional software development into a multi-organization setting
  + Cost reduction
  + Specialization $\rightarrow$ quality
    – Cultural mismatches
    – Coordination
      • distance is still a barrier, technology not a substitute for physical proximity
Component-based development vs. offshoring

- **Offshoring**
  - process dictates/constrains offshored parts
  - mostly top-down
- **Component-based**
  - components dictate/constrain process
  - bottom-up approach
Observations

• Any attempt at *dividing & conquering* only conquers if division follows certain principles—software is no exception
  – decomposition must lead to effective distribution and decentralization (organization and product level)
    • it must follow certain principles
      – information hiding, encapsulation (not just at "module level")
    • it must be specified precisely
      – interface, *explicit* and formal contracts
  • Problems become more critical as the degrees of distribution and decentralization get higher
Service-based DDD

- Service is the unit of composition
- Applications built by relying on autonomous parts
- Service is a component, exposed for use, administered, and run by an independent organization
- Pushes distribution and decentralization to run time
- Vision
  - towards a service-full world
  - applications built by dynamically composition of services, based on changing situation
Service-based DDD
the run time view

service integrator

added-value service

service provider

service provider

service provider

service provider
Service-based vs. component-based systems

- Both components and services are developed by providers (no single authority is in charge of all parts)
  - decentralized developments
- Components deployed and run by the owner of the system
- Services deployed and run by providers
- Components selected/composed (bound) at design/construction time
- Services chosen and bound at run-time
- Services composable to form new services
Service-based DDD

- Supports the goals of dynamic adaptation and evolution that arise in modern systems because
  - requirements change continuously and unpredictably
  - functionalities to offer depend on context
- Software must change accordingly, as it runs and provides service
Example: networked organizations

• Business world: networked enterprises
  – agile federated organizations
    • dynamic, goal-oriented, opportunistic federations
    • reacting with fast responses to rapidly changing requirements and changes in business world
  – changes in business require dynamic adaptation of the software architecture
Networked organizations

- Interacting applications belong to multiple administrative domains
- Web based interactions based on standard protocols
- Many (and changing) potential providers can be found for each required function
- Internal applications exposed for external use
Example: pervasive computing

• The Internet of Things
  – physical objects become part of the system
• Ubiquitous/ pervasive computing settings
  – anywhere, anytime, anything
• The service to provide and the way to provide service change dynamically according to context (situational computing)
• Self* capabilities required
  – self-healing, self-adapting, self-organizing
Pervasive computing

services
Situational computing

• The context in which a service is to be offered changes dynamically
  – goals, requirements
  – type of user (preferences, knowledge, role, …)
  – physical environment (space, time, …, light, temperature, humidity, …)

• Changes occur frequently and must be detected
• Systems must react to detected changes by adapting their behavior dynamically, while service is offered
Claims

- Service-based DDD and situational computing raise difficult challenges in combining extreme *flexibility* and *dependability*:
  - they stress to their extreme what makes software difficult
  - they deeply affect the way we conceive and develop software, invalidating some of our principles and beliefs
Some traditional beliefs vanish

- There is a sharp separation between development time and run time
- Changes require switching from run time back to development time where software is changed and validated
Where are the big challenges?

• Specifications, formal contracts including also extra-functional requirements
  – reliability, response time, performance
• How to deal with partial knowledge and uncertainty at development time
• How to deal with partial ownership of the application
  – new services may be discovered at run time
  – services may evolve independently
  – reconfiguration may occur at run time
Where are the big challenges?

• The development/run-time boundary vanishes
  – verification and validation must extend to run time
  – models continue to live at run time to support
    • early discovery/prediction of violations
    • model-driven adaptation
• Probabilistic approaches are key to to deal with uncertainty
Traditional model-driven view

- Use-case diagram for main functionalities
- Sequence diagram describing logic flow, user interactions
- Deployment diagram describing mapping onto physical resources

Model 1
Goals
Assumptions
Requirements

Model 2
Use Case Diagram with reliability rqmts
Markov model derived from PIM annotated with non-functional aspects
QN model derived from PSM+performance parameters

Model n

Code

2-3 July 2009
Zürich, December 2008
Towards a new paradigm

Goals
Assumptions
Requirements

Model 1
Model n

the world

Code
Impact on education

- Prevalent educational approaches in software engineering are still situated in the traditional *closed-world* scenario
  - in-house, from-scratch development
  - frozen requirements
- However, our students will live in an *open world*
  - from *in-the-small* to *in-the-large* to *in-the-many*
  - at development time and at run time
- Traditional issues, like architecture, specification, verification become harder, inevitable, and pervasive
Thanks for your attention!!!
Questions?