What SOA can do for Software Dependability

Karl M. Göschka
Karl.Goeschka@tuwien.ac.at
Vienna University of Technology

Overview

- Dependability challenges
- Control loop: Adaptivity and evolution
- The SOA potential

Challenges of today‘s applications

- heterogeneity (SOA, GRID)
- large-scale (pervasive, GRID, ultra-large-scale)
- dynamic (MANET, SOA)
- run continously (24*7)
- time to market
- cost pressure
- → dependability degradation

The dependability gap

- (short-/long-term) changes of ...
  - the system itself (e.g., resource variability)
  - the context (environment, failure scenarios)
  - users' *needs and expectations*
- Complexity and emerging behaviour
  - Interactions and interdependencies prevail properties of a systems' constituents
- → Human maintenance and repetitive software development processes
  - error-prone and costly
  - slow, sometimes prohibitively
  - BUT: self-learning and highly adaptive ;-)

Software development

- Defects in software products and services ...
  - may lead to failure
  - may provide typical access for malicious attacks
- Problematic requirements
  - incomplete
  - most users are inarticulate about precise criteria
  - competing or contradictory (due to inconsistent needs)
  - will certainly change over time

Requirements

Requirements are the things that you should discover before starting to build your product. Discovering the requirements during construction, or worse, when your client starts using your product, is so expensive and so inefficient, that we will assume that no right-thinking person would do it, and will not mention it again.

Robertson and Robertson
Mastering the Requirements Process

Needs, expectations, and requirements

Walking on water and
developing software from a specification
are easy

– if both are frozen

Edward V. Berard
Life Cycle Approaches

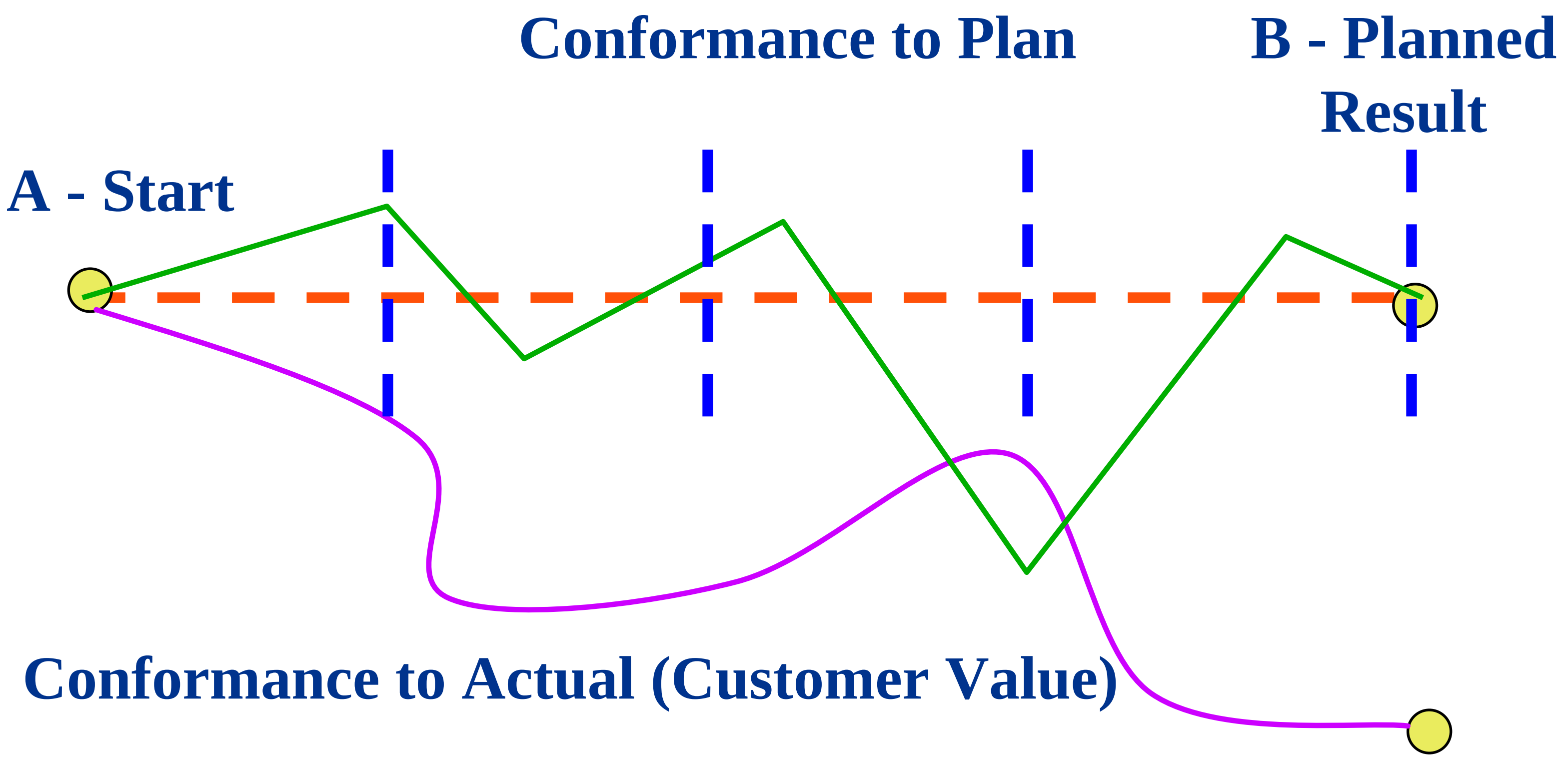
Requirements do change ...

- ... continously!
- Trade-offs change as well
- Domain know-how changes
- Technical know-how changes
- Retrofit originally omitted requirements
- Impossible to predict all changes

Answer on the process level

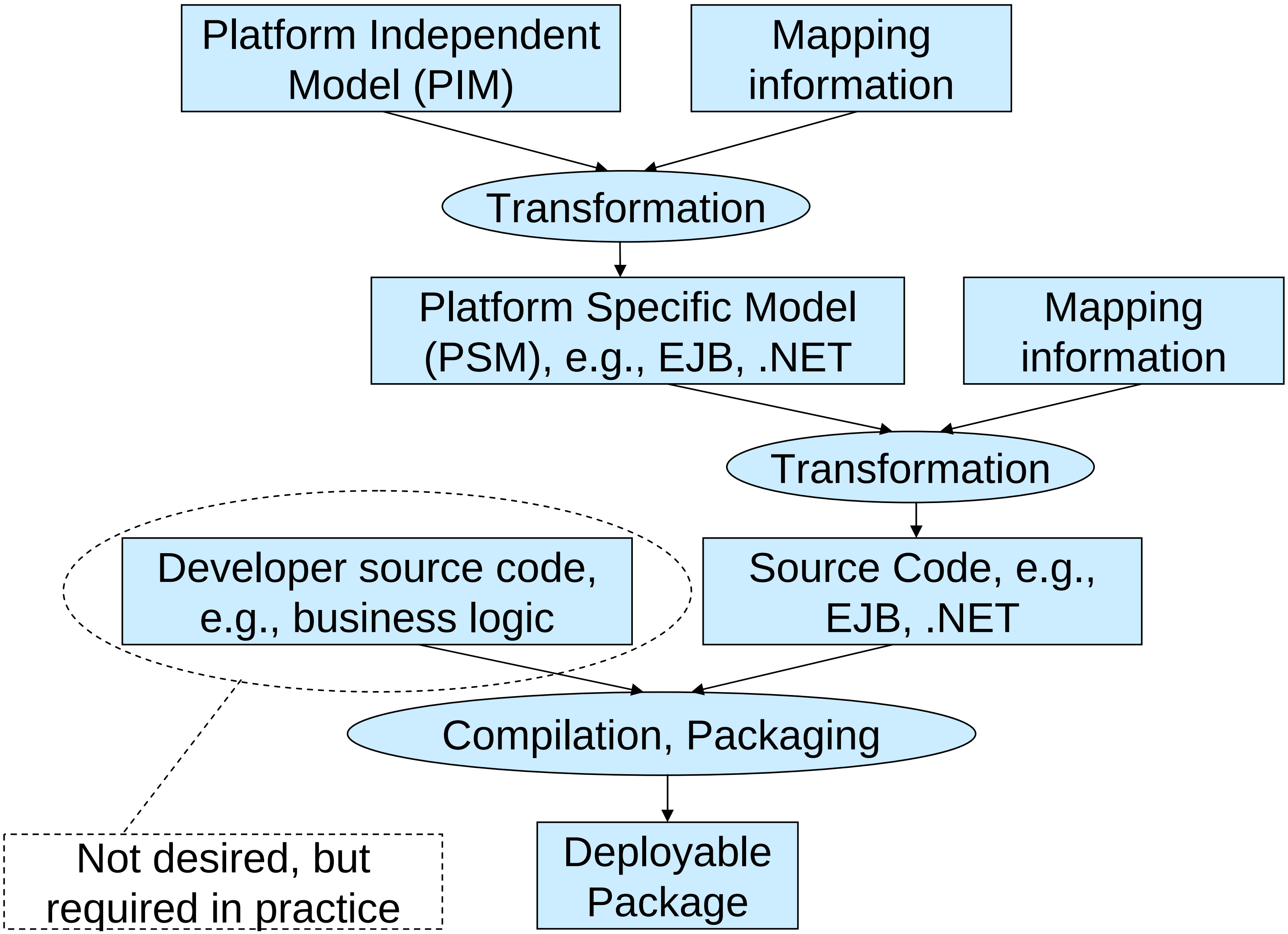
- Design for change in highly volatile areas!
- Heavy weight (CMM) → light weight (ASD) processes
- Differentiation:
  - development in-the-small: Component, service,...
    → agile development (ASD, XP), MDA, AOP, ...
  - development in-the-large: Procurement/discovery, generation, composition, deployment, ...
    → EAI, CBSE, (MDA), SOA, ...

Agile Development (ASD)



"In an extreme environment, following a plan produces the product you *intended*, just not the product you *need*."

Model-Driven Architecture (MDA)

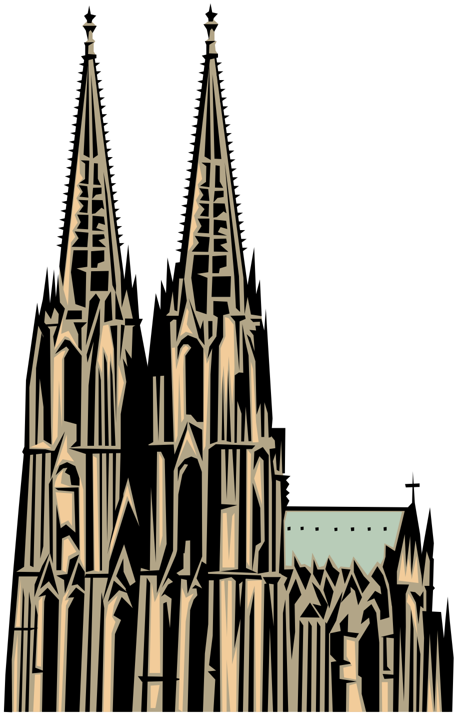


Dependability arguments for MDA

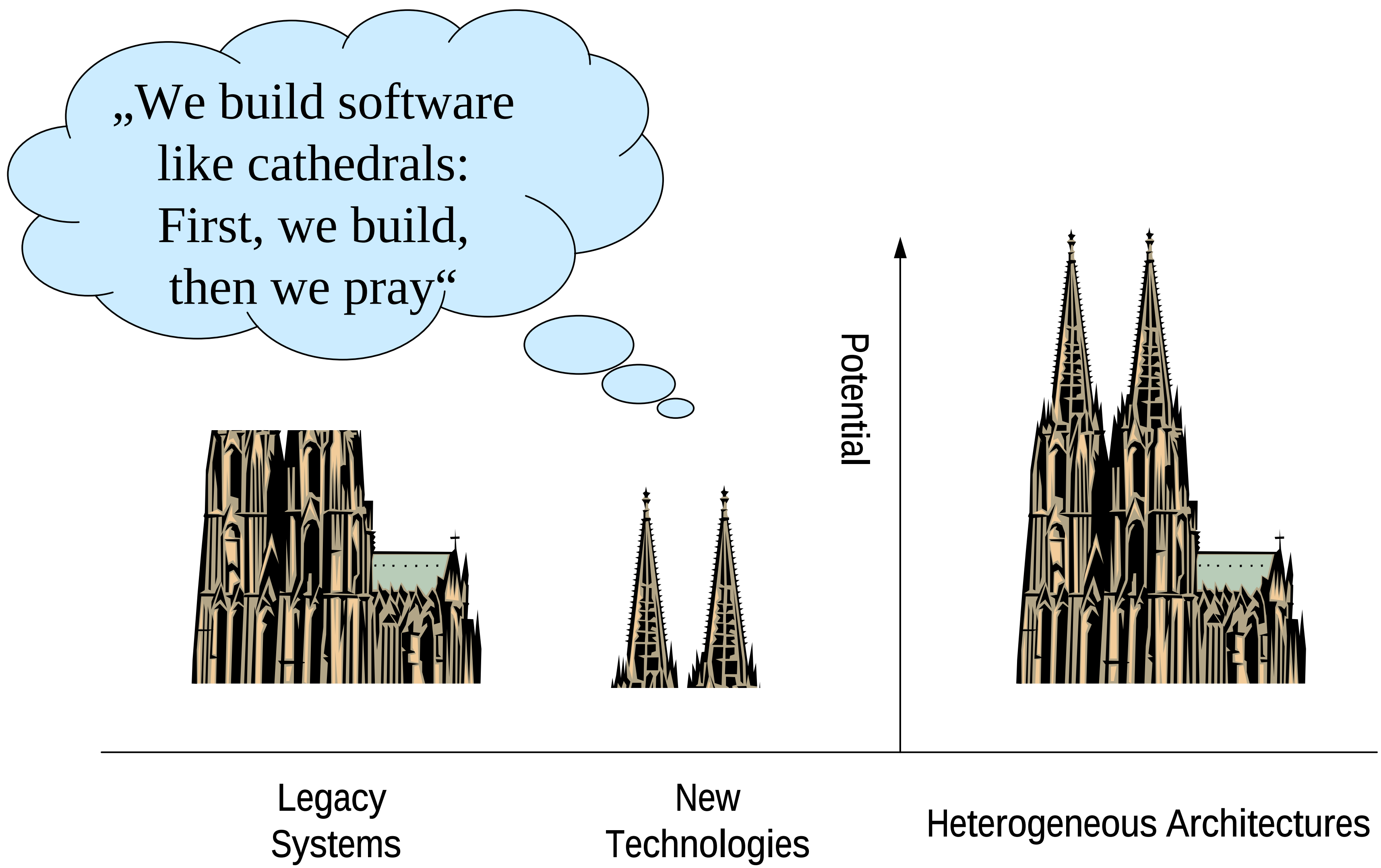
- Verification of system properties at PIM level
  - Formal verification
  - Testing (?)
- Verification of system properties at PSM level
  - Formal verification
  - Testing
  - Required platform specific properties
- In theory no component testing at code level necessary
  - Only System Test
- Documentation always up-to-date

EAI: Software Cathedral

- Robust, long Lifecycle
- Co-Existent of diverse different Technologies
- dynamic, extensible
- Re-usable Designs
- Based on a common Framework-Architecture



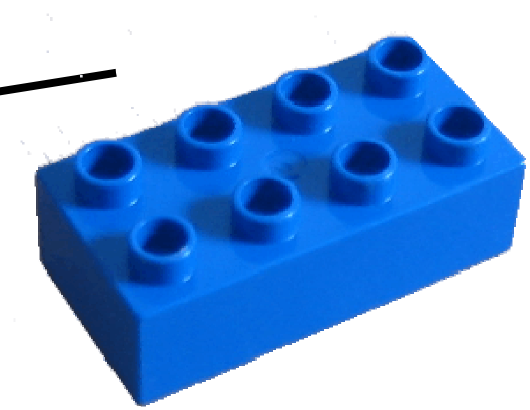
Heterogeneous Architectures



Component-based Software Engineering

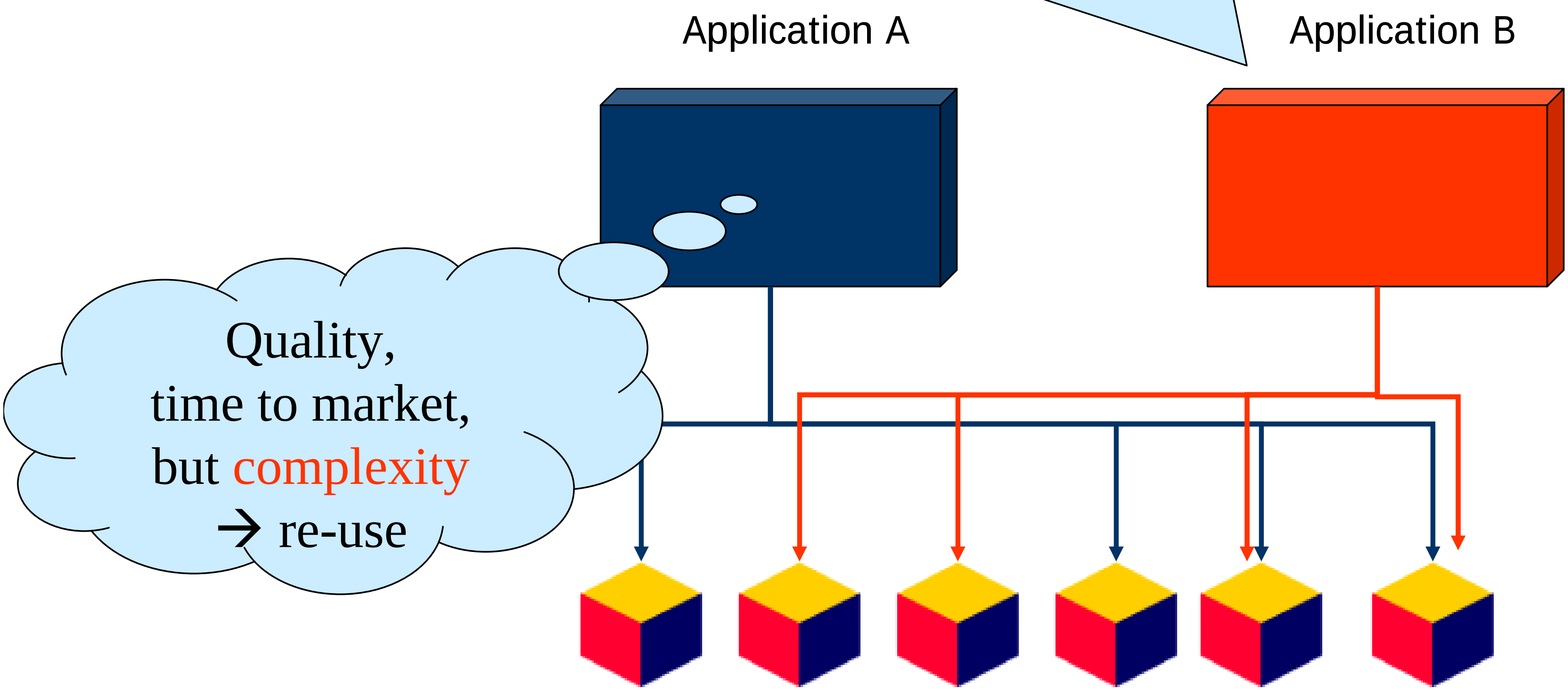
„Buy before build.
Reuse before buy“
Fred Brooks 1975(!)

Components: CBSE
and Product Lines



Product Line

Components of Mercedes E class cars are 70% equal.
Components of Boeing 757 and 767 are 60% equal.
→ most effort is integration istead of development!



Fault tolerance techniques

- persistence (databases)
- transaction monitors
- replication
- group membership and atomic broadcast
- reliable middleware with explicit control of quality of service properties
- also addressing scale and dynamics: e.g., gossipping protocols

Overview

- Dependability challenges
- Control loop: Adaptivity and evolution
- The SOA potential

Control loop approach

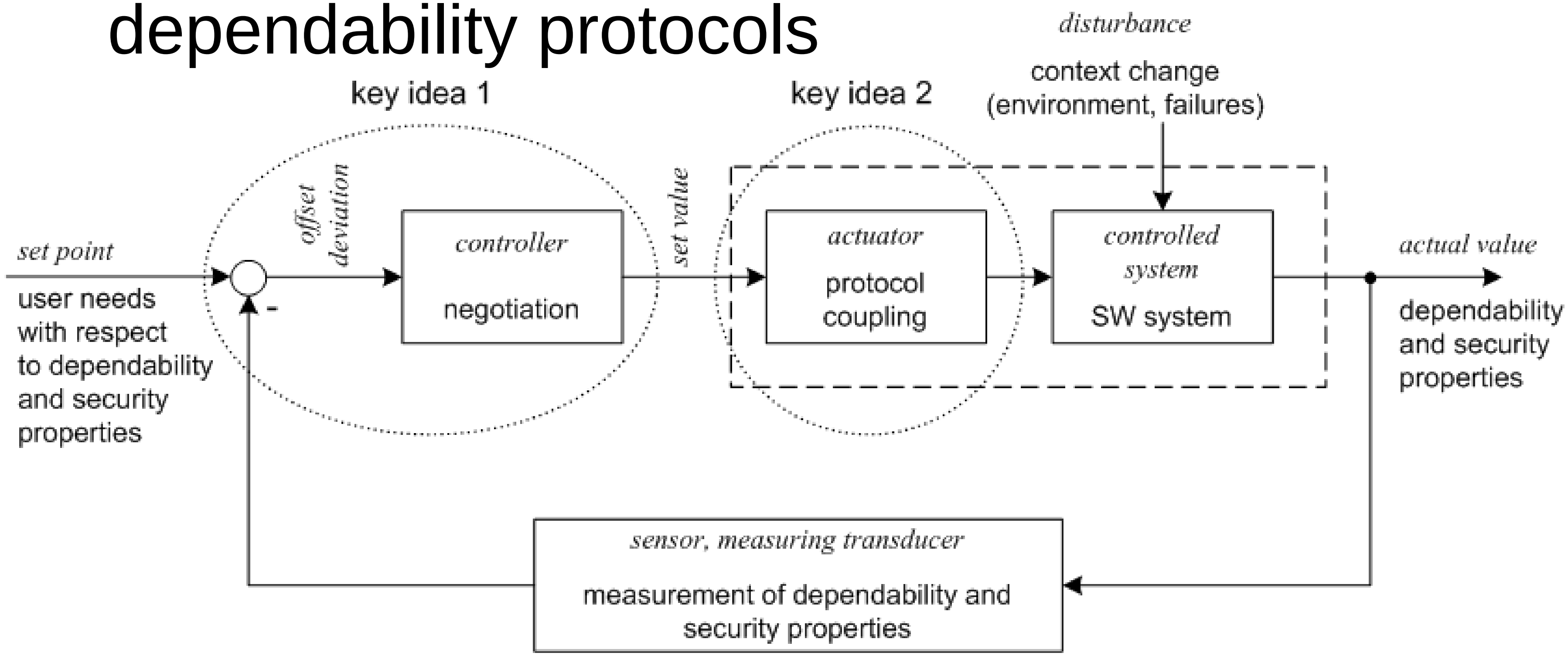
- Short-term adaptivity to react to observed, or act upon expected (temporary) changes
- Often termed „autonomous“, „self-*“, or „software agility“
- Control-loop approach:
  - Monitoring
  - Diagnosis (analysis, interpretation)
  - Re-configuration (repair)
- BUT: focus on system‘s components contradicts complexity theory

Adaptive Coupling

- Complexity theory demands focus on structure and interaction rather than properties of the individual constituents
- Relationships of differing strengths → mixture of tightly and loosely coupled parts
- → overall system properties are also determined by the strength of coupling
- → inner loop provides adaptivity by controlling the strength of coupling

Inner loop (short-term adaptivity)

- properties are balanced by negotiation between infrastructure and application
- explicit control of coupling mechanisms, e.g., run-time selection and reconfiguration of dependability protocols



Forms of coupling

coupling type	tightly–coupled	loosely–coupled
temporal	synchronous	asynchronous
referential	explicit binding, partition	discovery, space–based
constraint validation	per instance, operation, or transaction	postponed, triggered, or background task
constraint management	implicit	explicit
constraint decision	boolean (valid/violated)	imprecise, negotiation
system health detection	deterministic	gradual — "good enough"
system repair	reactive repair	proactive repair or homeostasis
update propagation	synchronous, eager	lazy or probabilistic (epidemic)
replica placement	full replication	partial and statistical replication
replica consistency	1–copy–serializability	ϵ–serializability
atomicity	roll–back	undo or compensation
isolation level	serializable	phantom read, dirty read, . . .
locking	strict	best effort reservation
consensus	deterministic	probabilistic

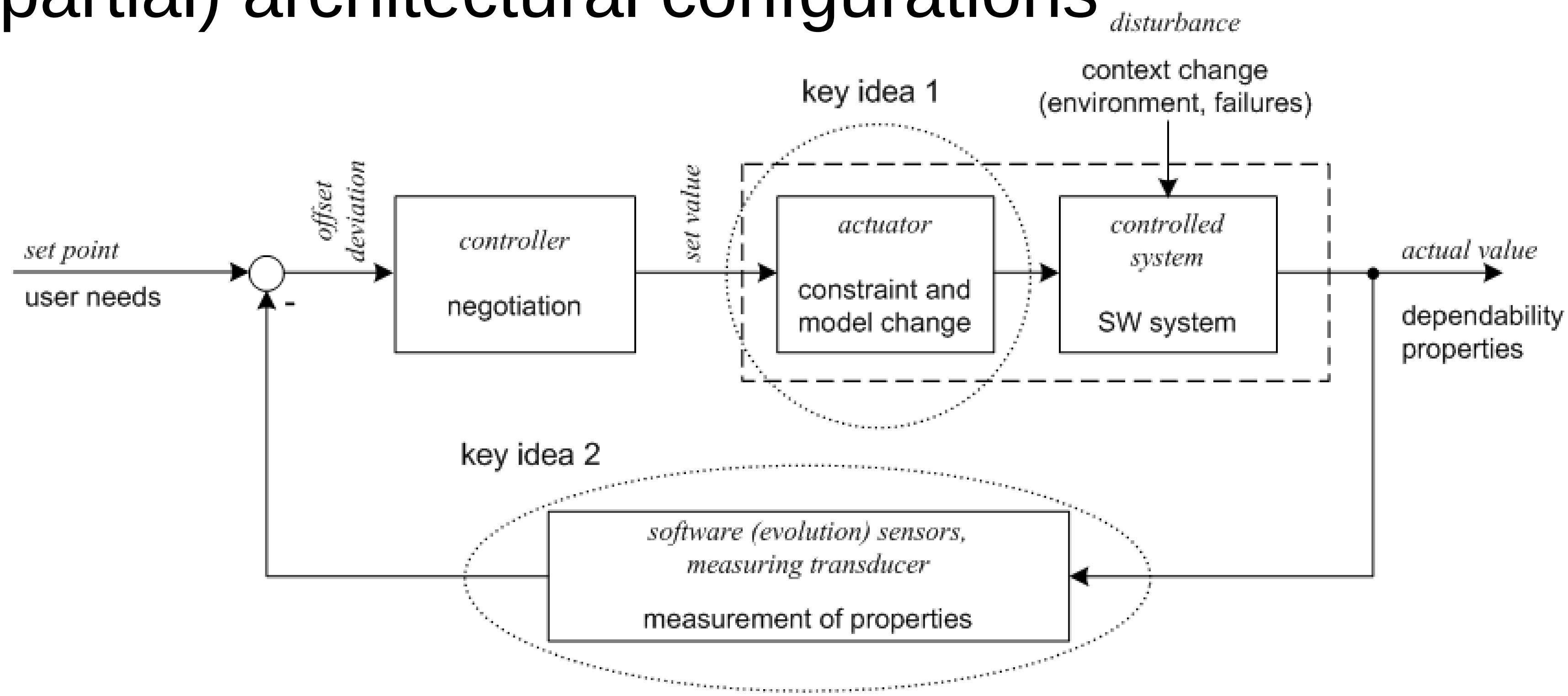
Long-term evolution

- regulate emerging behaviour (policies)
- evolvement of user needs and context

- → change the system‘s design while running!
- requires run-time accessible and processable requirements and design-views, e.g.
  - constraints
  - models („UML virtual machine“)
  - (partial) architectural confgurations

Outer loop (long-term evolution)

- measurement of properties (incl. history)
- negotiation of needs
- explicit manipulation of requirements/design: constraints, models („UML virtual machine“), (partial) architectural confgurations



Run-time software development

- requires middleware support
  - stored in repositories
  - accessed via reflection
  - aspect-oriented programming (dynamic aspects)
  - protocols for meta-data exchange
- → convergence of software development tools with middleware services („re-engineering running software“)
- → new challenges: e.g., run-time testing and verification

Constraint management

- Predicates, that stem from requirements
- Lifecycle:
  - informal during analysis
  - formal during design (e.g., UML+OCL)
  - tangled with implementation code
- Can be a problem:
  - checked in different places
  - requirements traceability and verification
  - design-by-contract principle (heterogeneous composition)
  - run-time control (e.g., activation/de-activation)

Distributed constraint validation

- Constraint validation itself becomes subject to node and link failures
- Possibly stale copies may be used for validation → consistency threat
- Potential inconsistencies may be accepted: Integrity is (temporarily) relaxed to increase availability
- Negotiation:
  - static (deployment or run-time)
  - dynamic (run-time: application call-back or user intervention)
- Requires explicit management of constraints and consistency threats

Loosely-coupled validation

- Explicit run-time constraints allow to decouple constraint **validation** from business activity
  - Asynchronous *validation* at any time (continously, triggered)
  - Check-out/check-in (e.g., in mobile systems)
  - Asynchronous *negotiation* and *reconciliation* (decoupled from system health set-points)
- Explicit run-time constraints allow to decouple constraint **activation** from business activity
  - Deactivate/revoke constraint to „heal“ the system
  - Introduce new constraints
  - Alternate constraints for different system *missions*

Inconsistency Management

- Explicit run-time constraints → decouple validation/activation from (degraded) business activity
- Explicit constraint management supports system maintenance and evolution: Deploys a smooth way of re-design without service interruption or re-compilation
- Performance impairment often acceptable
- Inconsistency management (large-scale)
- Constraint-in-the-small vs. constraint-in-the-large
  - imprecise, require negotiation
  - part of heterogeneous and dynamic composition
  - undergo continous evolution

Overview

- Dependability challenges
- Control loop: Adaptivity and evolution
- The SOA potential

How to actually implement this?

- different pace of change
- complemetary approaches

- share the need for
  1. Reconfiguration of the architectural coupling, including strength of coupling
  2. Measurement and negotiation of properties
  3. Run-time processable requirements and design artifacts (meta-data) → information sharing between application and infrastructural service

- Can SOA address these needs?

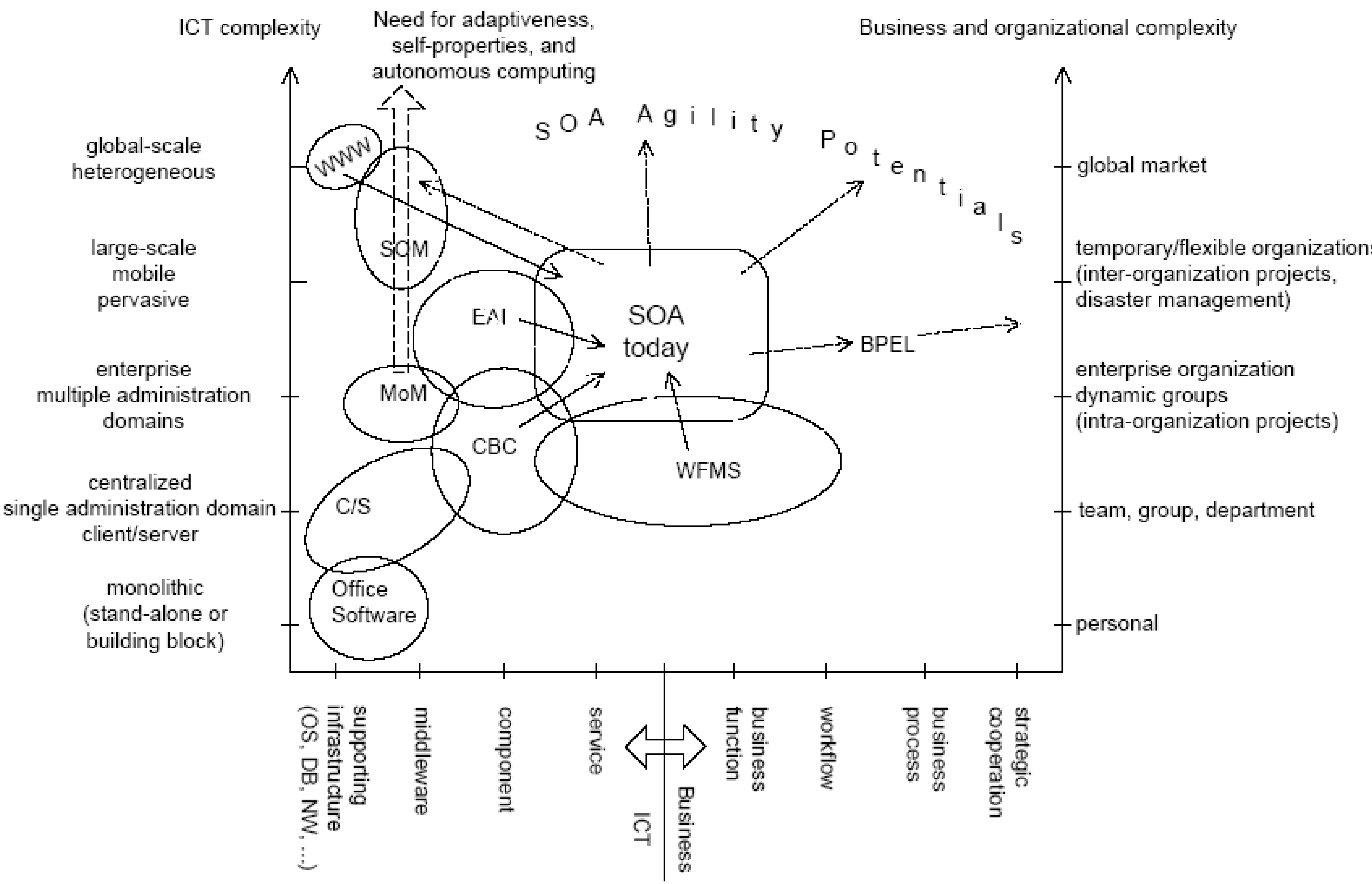
SOA is an evolution, not a revolution

- EAI – Enterprise Application Integration (MoM) (note: Was an argument for CBSE as well)
- WfMS – Workflow Management Systems →BPEL
- CBSE – Components are not obsolete!
  → WS provide a *virtual* component model
- WWW – *Loose* coupling: Heterogeneous, flexible, and dynamic orchestration
- Re-use (note: Was an argument for CBSE, Middleware, ...)
- Interface management (note: ...)
- *Business* integration („business goals with IT“)

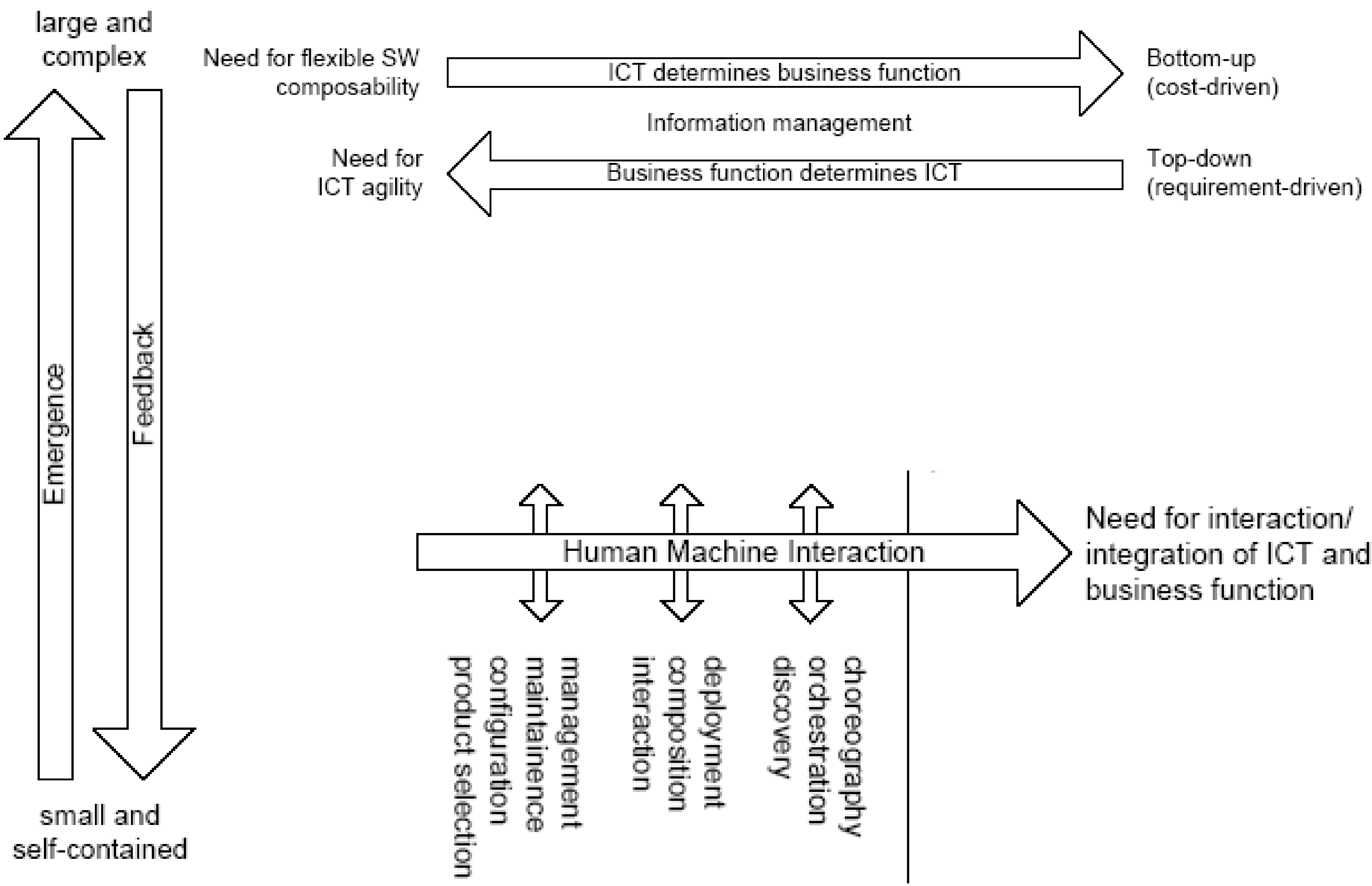
Related *WS* Standards and Concepts

- WS-Coordination: Consensus, e.g. WS-Transactions and WS-BusinessActivity
- Discovery: UDDI did not work, alternative approaches are investigated and discussed
- WS-MetaDataExchange: Important means for run-time adaptation
- Service Oriented Middleware?
  - particular challenge for *end-to-end* properties
  - but natural support for *vertical integration*
- Service Replication: The wheel need not be re-invented

A framework for business integration



Dimensions of complexity



Summary

- SOC addresses *some* needs for adaptive dependability (coordination, meta-data)
- There are many complementary approaches (e.g., WS-Reliability,...), but none widely adopted yet.
- In some cases, SOC is „yet another technology (wrapper)“ where the *wheel need not be re-invented* (e.g., replication)
- There are new research challenges, in particular SLA for end-to-end properties
- → future research is needed: SOM, actual realization of interface-“promises“

What SOA can do for Software Dependability

Karl M. Göschka
Karl.Goeschka@tuwien.ac.at
Vienna University of Technology

System Life Cycle

- Development Environment: physical world, human developers, development tools, production and test facilities → development faults.
- Use Environment: physical world, administrators, users, providers, infrastructure, intruders.
- Use phase: service delivery, service outage, service shutdown.
- Maintenance: repairs and modifications (iterative development process).
- Design-time/run-time convergence