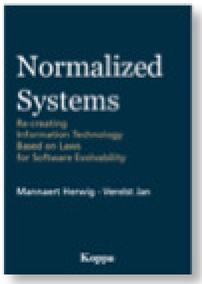


In Search of Design Patterns for Evolvable Modularity

Prof. dr. Herwig Mannaert

IARIA Computation World, Athens, 2009

Universiteit Antwerpen

Contents

Contents

- Evolvability in Software Systems
  - Challenges
  - Solutions
  - Reflections
- In Search of Evolvable Modularity
  - Principles
  - Elements
  - Reflections
- Conclusions

The Business Challenge

- The **Agile Organization**
  - Continually scans its ecosystem
  - Reacts quickly to opportunities and is innovative
- Has 2 Characteristics
  - **Complexity**
    - Multi-channel vs. single channel
    - Diversify offerings/Additional services
  - **Change/Evolvability/Flexibility**
    - "These things are changing so fast it's invention in the hands of the owner." (Hansen et al., 2007)

The ICT Challenge – Part 1

- Modular structures of Information Systems in this complex, quickly changing environment, need to be:
  - Very flexible
  - Reliable (even mission-critical)
  - Totally secure
  - User friendly
  - Portable
  - Preferably affordable !
  - ...

The ICT Challenge – Part 2

- Complexity
  - Compare JEE/.NET to COBOL
  - …
- Change
  - Structured Development – 70’s
  - Object Oriented Development – 80’s
  - Component-Based Development – 1995-
  - Service-Oriented Development – 2000-
  - The Next Hype…

The ICT Challenge – Part 3

The Law of Increasing Complexity
Manny Lehman

"As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it."

Proceedings of the IEEE, vol. 68, nr. 9, september 1980, pp. 1068.

Lehman's Law

- Suggests
  - Interpretation 1
    - Even if we can (ever) offer the desired levels of evolvability in information systems, these evolvability levels automatically decrease over time, unless ever increasing perfective maintenance is performed
  - Interpretation 2
    - That the marginal cost of a change to a system, is ever increasing over time
  - Interpretation 3
    - That systems
      - Require ever higher budgets
      - Require ever larger IT-departments
      - Eventually have to be replaced anyway, thereby effectively writing off the complete investment in the development and maintenance of the system !
  - Interpretation 4
    - As change increases, complexity increases, so the two main challenges seem interrelated

Contents

The Good News - Modularity

- Very complex systems already exist, for example, in hardware, telecommunications, space industry.
- They are based on proven engineering concepts such as:
  - Modularity
  - Standards

Modularity in Software

- Modularity has been the basis of Information Systems Design since the ’60s
  - Has proven its relevance in the past => no hype !
  - And will probably play a decisive role in the future
- Independent of programming language, packages, frameworks, even paradigms !

The Promise of Modularity

"expect families of routines to be constructed on *rational principles* so that families fit together as **building blocks**"

uit: McIlroy, *Mass Produced Software Components*,
1968 NATO Conference on Software Engineering, Garmisch, Germany.

Modules – Advantages

Complexity Reduction

Reuse

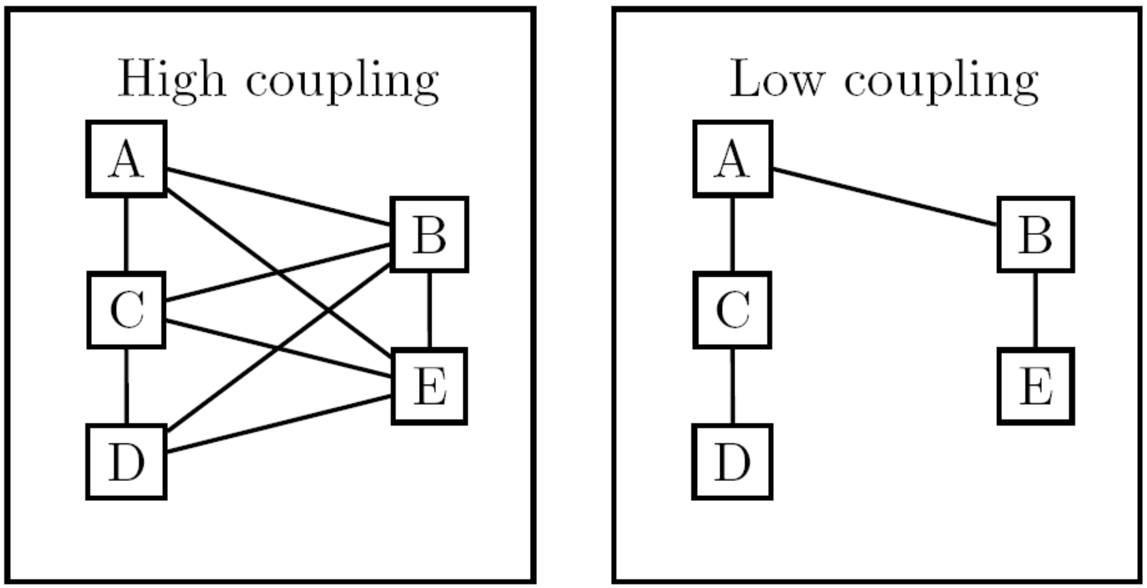
Evolvability

Modularity - Constructs

- Modules are implemented in constructs, which are becoming increasingly powerful
  - Functions/procedures,
  - Objects,
  - Components
  - Services
  - Aspects
  - …
- We *are* making progress !

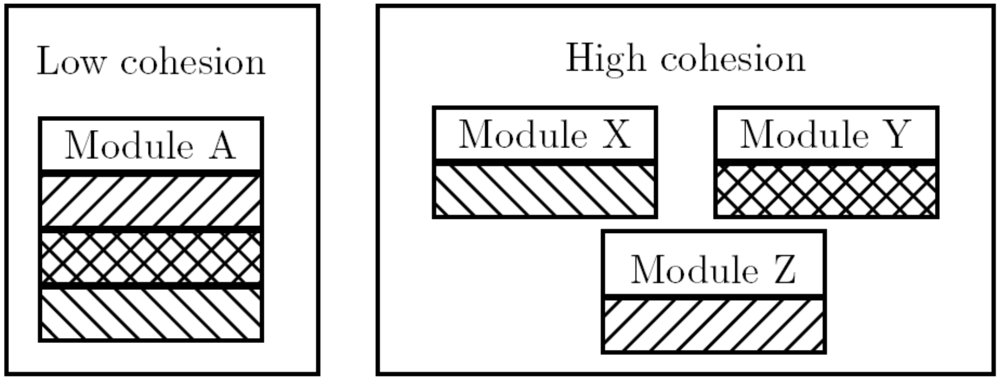
Designing Modules - Coupling

- **Coupling** is a measure of the dependencies between modules



Designing Modules - Cohesion

- **Cohesion** is a measure of how strongly the elements in a module are related



- Good design=

  Low coupling and **high** cohesion!

Contents

The Dream: Doug Mc Ilroy

“expect families of routines to be constructed on *rational principles* so that families fit together as **building blocks**”

uit: McIlroy, *Mass Produced Software Components*,
1968 NATO Conference on Software Engineering, Garmisch, Germany.

The Reality: Manny Lehman

The Law of Increasing Complexity
Manny Lehman

"As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it."

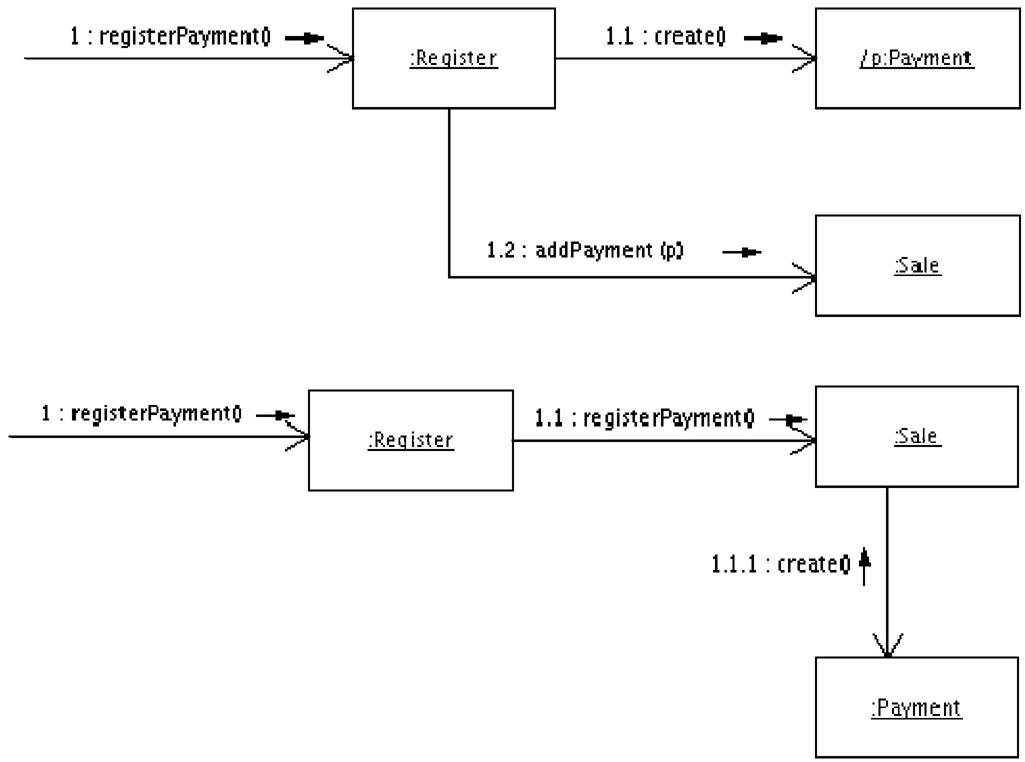
Proceedings of the IEEE, vol. 68, nr. 9, september 1980, pp. 1068.

Better constructs,

but how to design evolvable modular structures with them?

Low coupling and high cohesion. Everybody knows this. The question is how to do this.

Example: Minimize Coupling !



The Problem – Part 1

- Different opinions about ‘good’ design
  - “Low coupling” is too vague !
  - “Information hiding” was formulated by Parnas in 1972, but still needs to be refined
  - Philippe Kruchten (2005): “We haven’t found the fundamental laws in software like in other engineering disciplines”
- Limited, unsystematic application of ‘good’ design
  - Technical difficulties
  - Project Management difficulties

The Problem – Part 2

- Modularity in other disciplines, like hardware and space, is static modularity. It does not accomodate continuous changes.

  We need evolvable modularity.

- Design, the mapping from functional requirements to constructive primitives, is a complex activity.

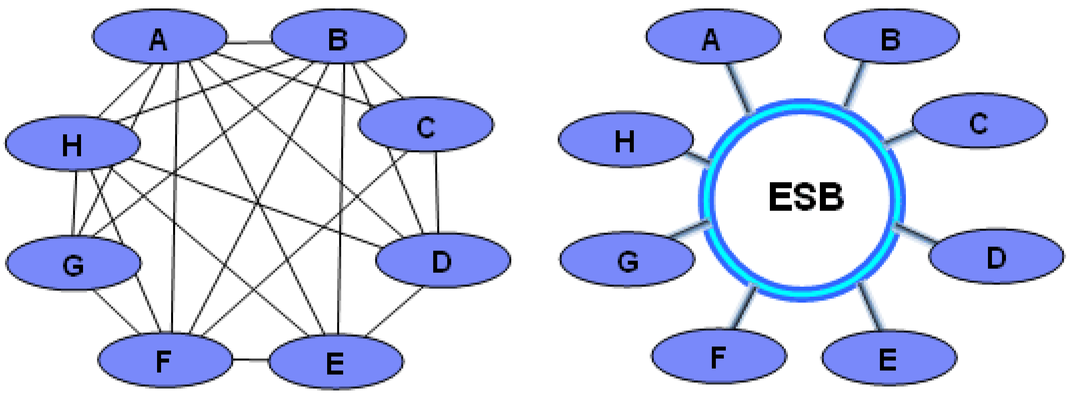
  It cannot be done on a 1-1 basis.

The Theories – Part 1

- Stability in System Dynamics:
  - In systems theory, the dynamic evolution is studied based on a differential/difference equation
  - A system is stable if and only if:
    - a bounded input results in a bounded output
    - it has poles in the left plane or inside the unit circle:
  - For a first order model, **stability ←→ a<0**:
    - $dy(t)/dt = x(t) + ay(t) \leftrightarrow Y(s)/X(s) = 1/(s-a)$
    - $y[k+1]-y[k] = x[k] + ay[k] \leftrightarrow Y(z)/X(z) = 1/(z-(1+a))$
  - This means that the increment cannot have a positive contribution from the size of the system

Example: Enterprise Service Bus

- The effort to include an additional component may or may not vary with the system size



Impact = N **Impact = 1**

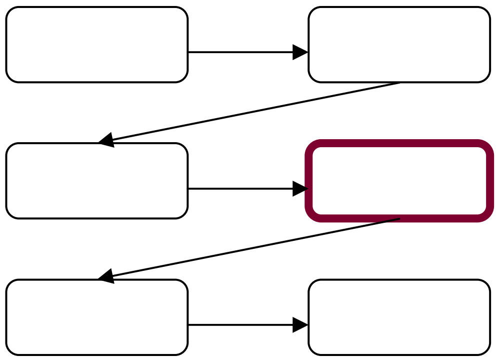
Source: http://nl.wikipedia.org/wiki/Enterprise_Service_Bus

The Theories – Part 2

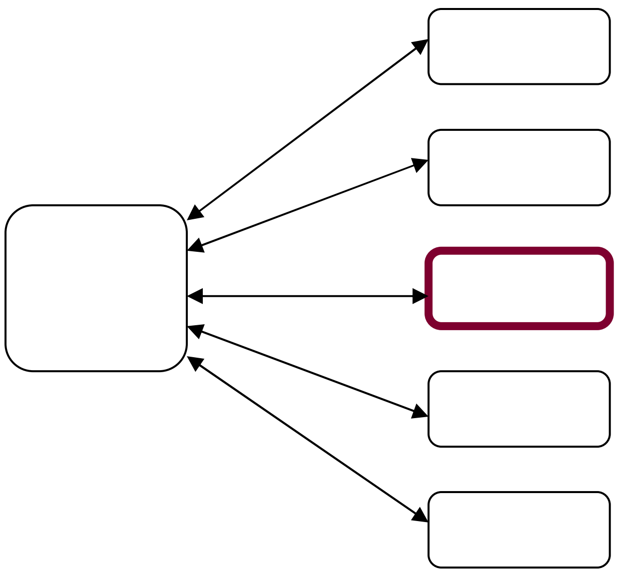
- Entropy in Thermodynamics:
  - In thermodynamics, the dynamic evolution is represented by the entropy of a system
  - A system will always increase its entropy, which basically represents the irreversibility in nature
  - In statistical thermodynamics, Boltzmann defined entropy as the number of possible microstates for a given macrostate, such as:
    - a number of coins with or without partitions
    - gas container with or without partitions
  - In information theory, Shannon defined entropy similarly as the expected value of uncertainty:
    - $-\sum_i p(x_i) \log(p(x_i))$

Example: Workflow Controllers

- The effort to debug a system after adding another component may or may not increase



Uncertainty = N



Uncertainty = 1

Contents

- Evolvability in Software Systems
  - Challenges
  - Solutions
  - Reflections
- In Search of Evolvable Modularity
  - Principles
  - Elements
  - Reflections
- Conclusions

Basic Information Systems Model

- Context:
  - Technology environment: language/package/… acting as omnipresent background technology
  - Software entities: instantiations of constructs
- Primitives:
  - Data entity: entity with attributes and/or links
  - Action entity: entity representing operation at a modular level, containing one or more tasks
  - Task: chunk of code performing a functionality, considered to be a *change driver*
  - External technology: presence of entities of another technology environment, *implies a task !*

Model and System Stability

- Y[k]: the number of all software entities at k, including the various versions
- X[k]: the number of (versions of) software entities to be added to the system at k
- Y[k+1]: the number of all software entities at k+1 when the system works again properly
- Stability: the output function Y stays bounded for every bounded input function X
- aY[k] = combinatorial effects

Model and System Entropy

- Macrostate: an observable output and or state of the information system
- Microstate: the whole of all states and results of all software entities of the running system
- Partitions: software entities that externalize the system state of control and/or workflow, i.e. transactions

Stability and Normalized Systems

- Systems theoretic stability: bounded input results in bounded output for infinite time
- Software stability: bounded amount changes results in bounded impacts for infinite time
- *Assumption of unlimited system evolution*: number of all primitives and all dependencies between them become unbounded
- *Normalized systems*: information systems that are stable wrt defined set anticipated changes
- *Postulate*: Information systems need to be stable wrt defined set of anticipated changes

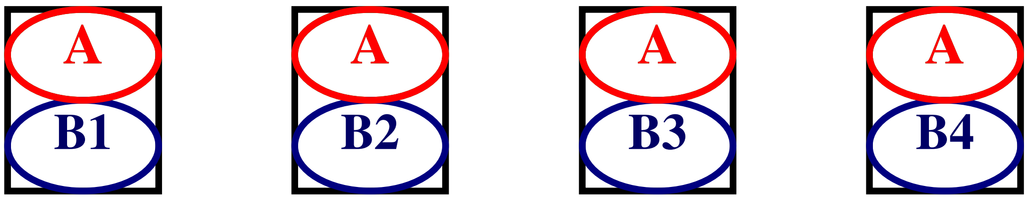
Basic Information Systems Model

- Changes:
  - Additional data entity
  - Additional data attribute
  - Additional action entity, incl. receive/call existing
  - Additional task version, incl. mandatory/new state
- Assumptions:
  - Unlimited number of entities
  - Unlimited number actions receiving 1 data entity
  - Unlimited number actions calling 1 action entity
  - Unlimited number of versions of 1 task

Separation of Concerns

- An action entity can only contain a single task
- Proof (RaA):
  - Action entities E_i combine A with version B_i
  - Additional mandatory version of A (change 4)
  - Number impacts E_i unbounded (assumption 2)
- Manifestations:
  - Multi-tier architectures
  - External workflow systems
  - Separating cross-cutting concerns
  - Use of messaging, service, integration bus

SoC: Multiple Version Task

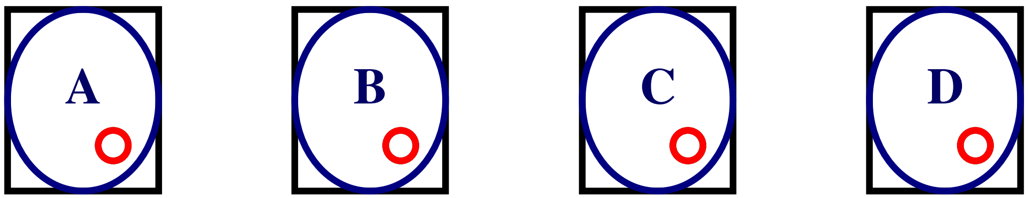


SoC: Applying Theories

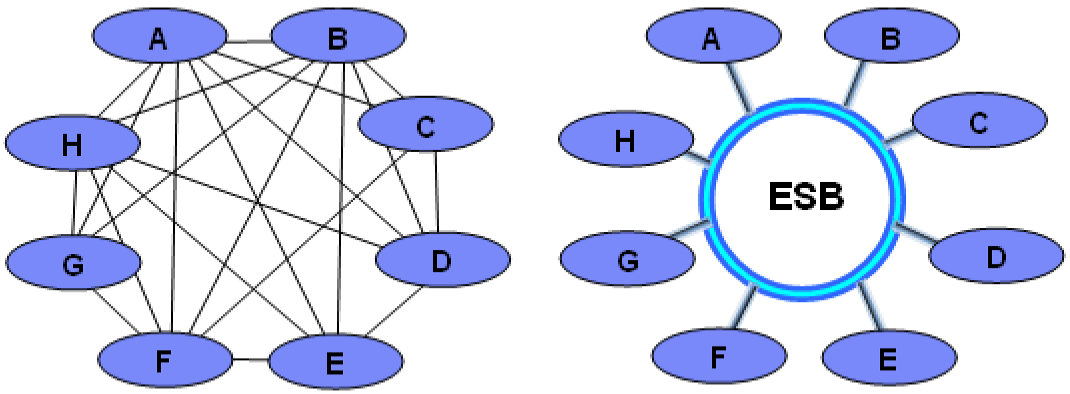
- Stability:
  - X[k]: new version of task A
  - aY[k]: the various new versions of the entities due to the multiple versions of B

- Entropy:
  - Macrostate: successful outcome of A+B
  - Microstates: the possible versions of the combined entity that may have been used and that may hide that A is not working properly in another version

SoC: Non-Encapsulated Task



Example: Enterprise Service Bus



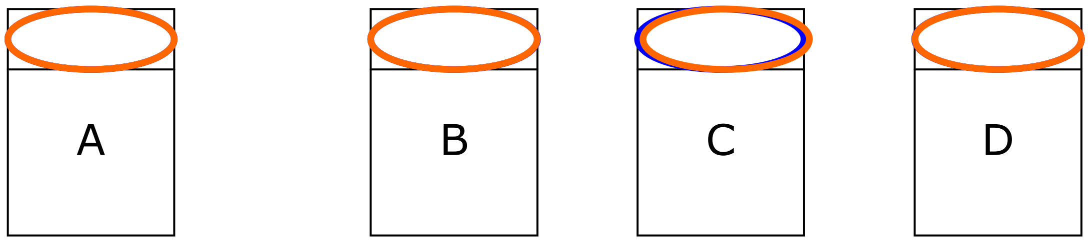
Impact = N **Impact = 1**

Source: http://nl.wikipedia.org/wiki/Enterprise_Service_Bus

Data Version Transparency

- Data entities received/produced by action entities need to exhibit version transparency
- Proof (RaA):
  - Action entities E_i receive D
  - Additional attribute in D (change 2)
  - Number impacts E_i unbounded (assumption 2)
- Implementations:
  - XML / Web Services at run-time
  - OO / JavaBeans at compile-time
  - Tag-Value pairs in legacy systems

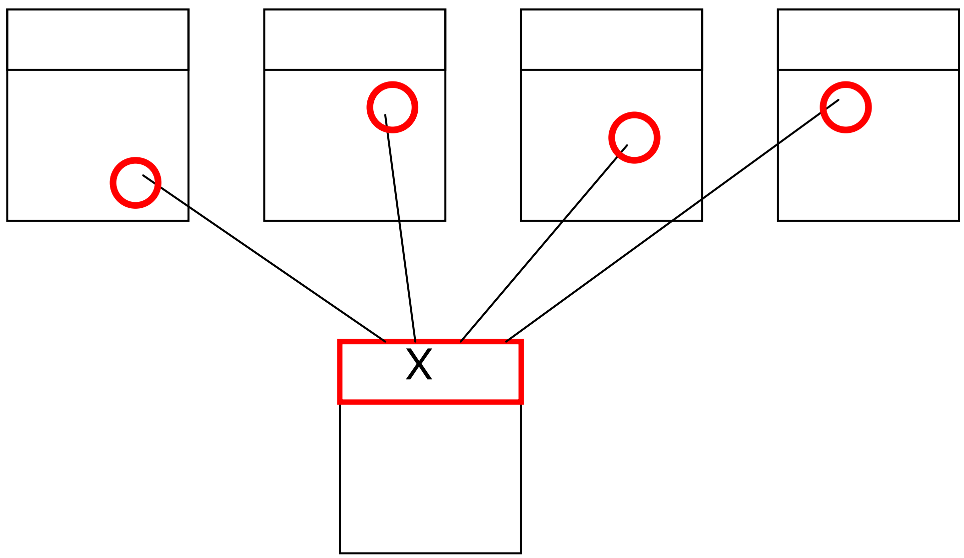
DvT: Multiple Version Data



Action Version Transparency

- Action entities called by other action entities need to exhibit version transparency
- Proof (RaA):
  - Action entities E_i call A
  - Additional version of A (change 4)
  - Number impacts E_i unbounded (assumption 3)
- Implementations:
  - OO facade patterns
  - Procedural wrapper functions

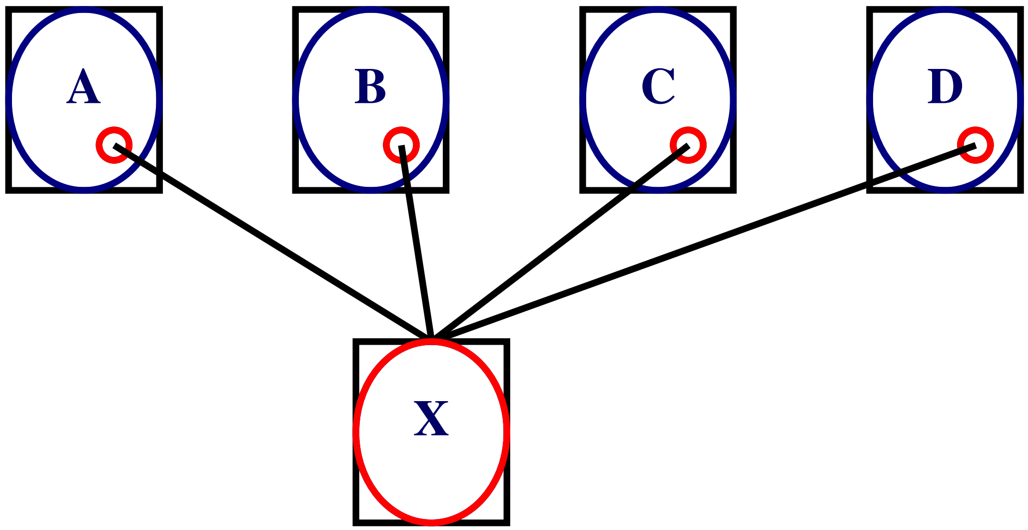
AvT: Changing the Interface



Separation of States

- The calling of an action entity by another action entity needs to exhibit state keeping
- Proof (RaA):
  - Action entities E_i calling action entity A
  - Additional version of A new state (change 4)
  - Number impacts E_i unbounded (assumption 3)
- Implications:
  - Stateful workflow systems
  - State related to instance of data entity
  - No stateless synchronous pipelines allowed
- Manifestation: async communication systems

SoS: Non-Encapsulated State



SoC: Applying Theories

- Stability:
  - X[k]: new version of task X
  - aY[k]: the various new versions of the tasks A, B, C, and D that cope with the new condition
- Entropy:
  - Macrostate: unsuccessful outcome of A+X
  - Microstates: the fact that A might have failed, or X, or A might have reacted in an inappropriate way to the failure of X

Normalized Systems Principles

- Presented principles solve the vagueness in identifying combinatorial effects:
  - Until now, no clear principles
    - ➔ subjectivity, ad hoc
  - McIlroy: “to be constructed on *rational principles”*
- Conclusion
  - Omnipresent CE ➔ No *evolvable* modularity !

Contents

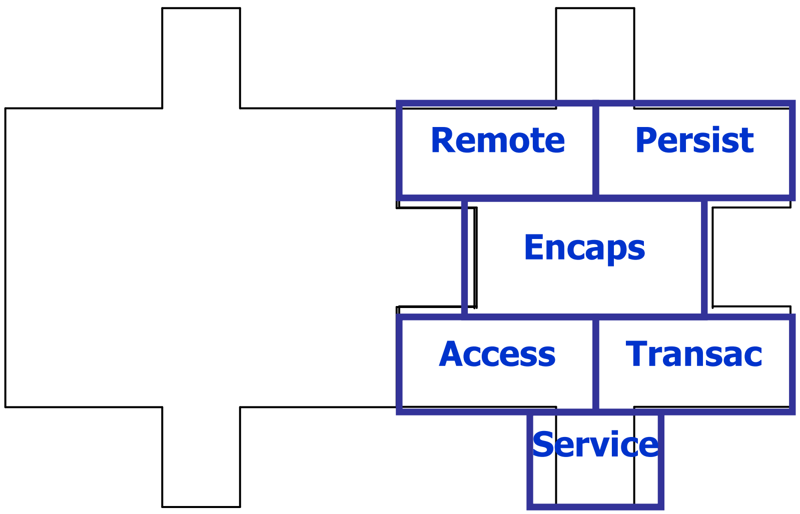
Towards Elements

- Dealing with CE
  - Mostly implicitly
    - The principles reflect well-known heuristic knowledge of designers
  - Mostly manual (even refactoring is only semi-automatic)
    - Without generally-accepted principles or laws
    - Without systematic application of ‘good’ design
    - …
  - And will remain manual even with improved constructs, which have been and will continue to be developed/improved gradually over time
  - Proposed principles can be applied, but even this is a manual approach…

Normalized Systems Elements

- The proposed solution =
  - Structure through Encapsulations, called Elements
    - A Java class is encapsulated in 8-10 other classes, dealing with cross-cutting concerns, in order to deal with the anticipated changes *without CE*, and fully separating the element from all other elements.
    - Every element is described by a "detailed design pattern". Every element builds on other elements.
    - Every design pattern is executable, and can be expanded automatically.
  - Realizing the core functionality of Information Systems
- Application = *n* instantiations of Elements

Normalized Systems Elements

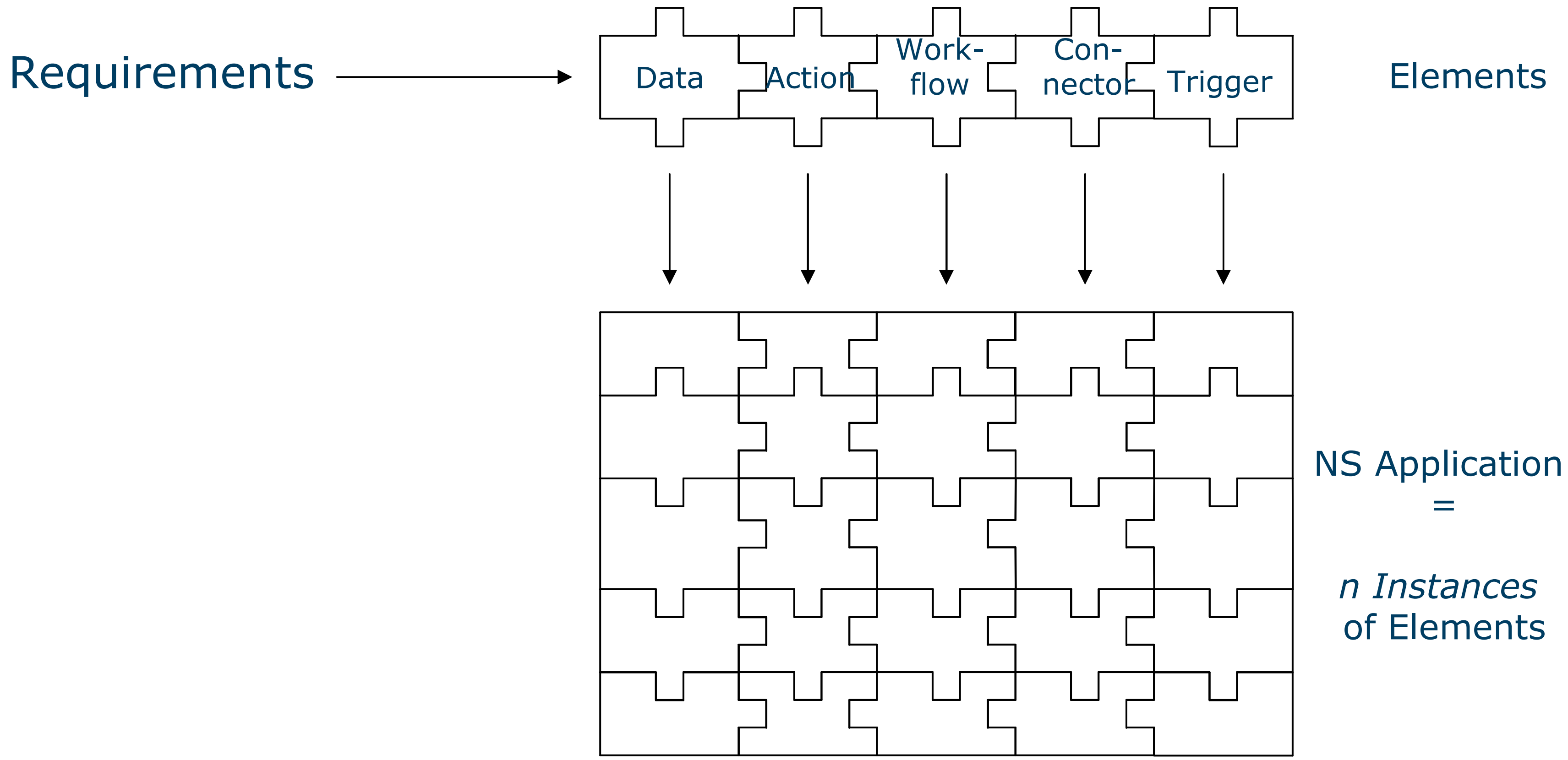


Choosing the Elements

- The same old primitives we have been using for more than 6 decades:
  - Data (registers, structs, records)
  - Actions (instructions, functions, procedures)
  - Connectors (IO commands)
  - Workflow (controllers, main programs)

- The elements should bridge the gap between antropomorphism and separation of concerns

Building NS Applications



Universiteit Antwerpen

Contents

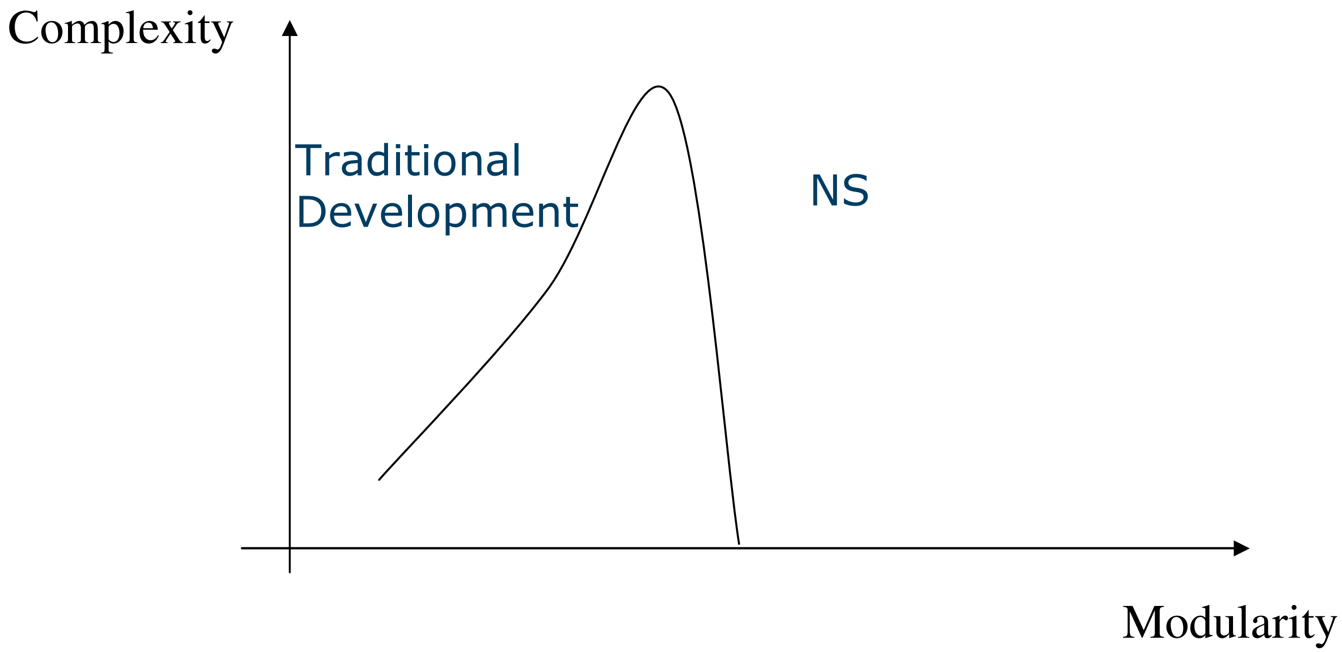
Normalized Systems Elements

- Characteristics
  - *Ex ante, proven* evolvability
    - Wrt anticipated changes
    - Changes in packages, frameworks, programming languages...
  - True Black Box, as the inside of an instantiation of the element is 'known', and therefore does not require black box inspection by the user.
    - McIlroy: "safely to regard components as *black boxes*"
  - True Black Box realizes Reuse
    - McIlroy: "families fit together as *building blocks"*
    - One cannot reasonably expect that modules can be systematically reused, when there are no generally-accepted principles for dealing with coupling and hundreds of developers are concurrently working on the same information system…
  - True Black Box controls Lehman
    - Any degradation does not affect other elements

NS Elements

- Proposed elements offer *evolvable* modularity
  - Infinite and controlled evolution of information systems
  - System-theoretic bounded input/bounded output
  - No Lehman, but McIlroy !
  - *Ex ante, proven* evolvability
- Evolvable modularity is based on Structure
  - Extremely fine-grained modular structure
  - Extremely systematic application of the principles
  - NOT on advanced code generation
- And leads to Determinism

The Cost of Modularity



Implications on Constructs

- The OO class is an unprotected construct:
  - Allows data and action encapsulation
  - Allows many concerns and combinatorial effects
  - *Does not enforce* any evolvability constraints
- More recent augmentations are consistent:
  - JavaBean component model
  - Server-side component models
  - Service oriented architectures
  - Aspects for cross-cutting concerns

Other Issues: Performance

- Stability theorems
  - No stateless sync pipelines are allowed
  - Calling an action needs to exhibit state keeping
  - No stateless sync pipelines are allowed
  - An action can only contain a single task
- OLTP commandments
  - Do not lock a system resource too long
  - Use transactions to clean up your mess
  - Reuse resources across clients
  - Come in, do your work, and get out
  - Deal with large number of small things

Other Issues: Testing / Docs

- In order to obtain stable building blocks, we propose the encapsulation of software entities into higher-level stable elements according to structures implied by the stability theorems.

- This structured composition of entities into the higher-level elements can be described as “design patterns”, that are detailed, unambiguous, and parametrized. Therefore:
  - Both unit and integration testing of such a stable building block should become a trivial thing.
  - The complete and unambiguous documentation of the building block should consist of the documentation of this design pattern and the expansion parameters.

Contents

Conclusions

- The Challenges are Complexity and Evolvability
- The Answers are Modularity and Determinism
  - High-quality IT: advanced modular structures of *proven* evolvability are needed to realize McIlroy and withstand Lehman !
    - Leads to required levels of determinism not offered by current methodologies, architectures, patterns…, that do not eliminate CE.
  - Low-quality IT: vague and unsystematic approaches to evolvability should be replaced by systematic approaches !
  - Progress has been made, but no magical solution !
- Normalized Systems
  - Principles are constraints on modular structures.
  - Stable Information Systems should be composed of Elements, complying at all times with the principles.

Thank you for your attention !

For more information:
herwig.mannaert@ua.ac.be