Hogeschool van Arnhem en Nijmegen
HAN University of Applied Sciences

SOFTWARE ARCHITECTURE en de "Quality Without a Name"

Christian Köppe, HAN University of Applied Sciences

Is dit een goede straat/goed gebouw?

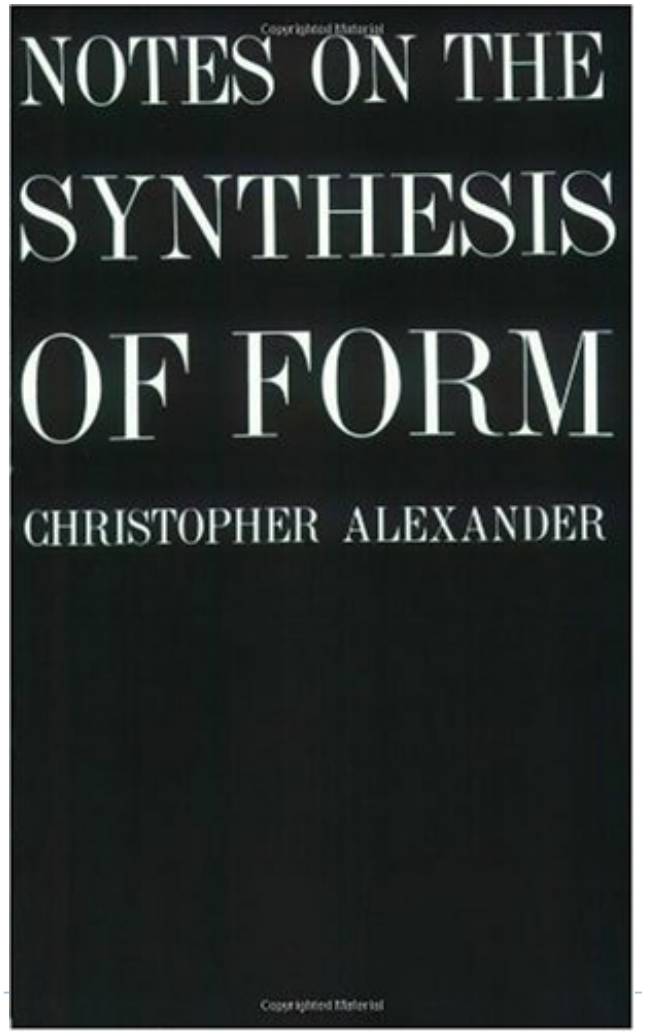
Welke “voelt” beter?

Welke “voelt” beter?

Hogeschool van Arnhem en Nijmegen
HAN University of Applied Sciences

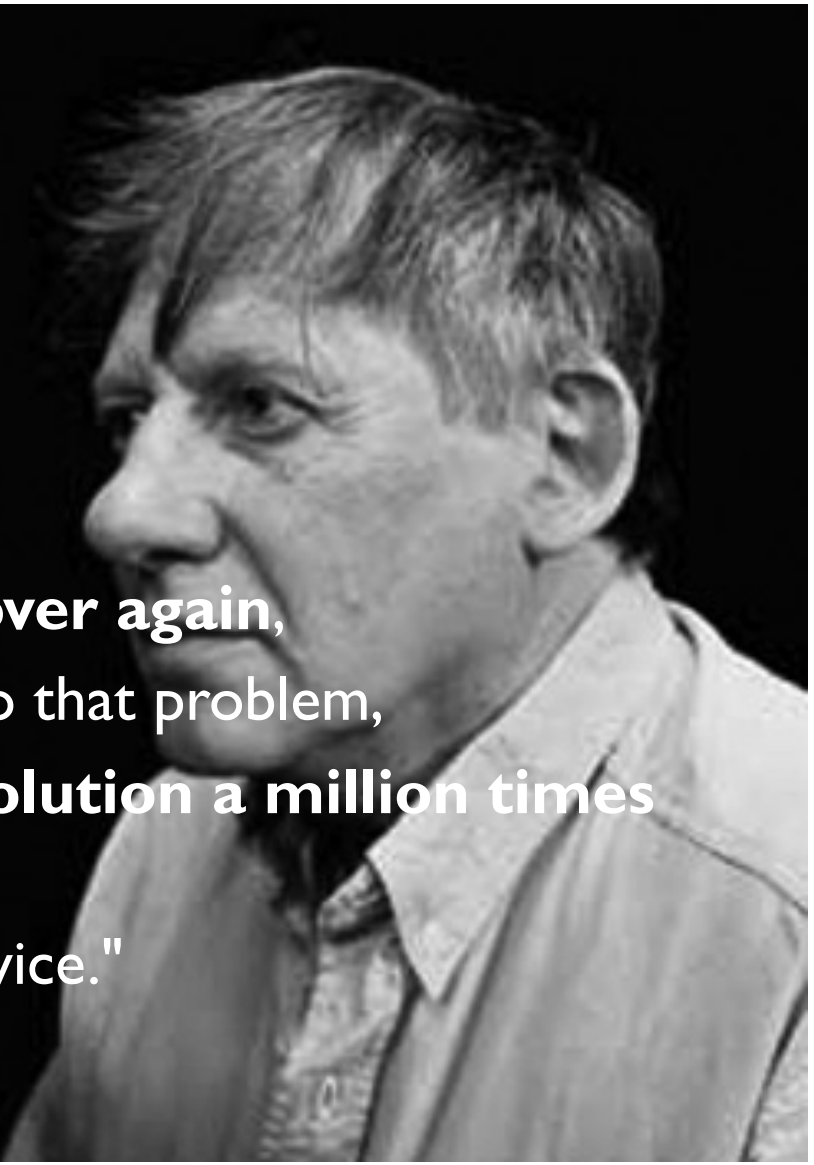
Welke “voelt” beter?

The Quality Without a Name (QWAN)



1977 “A Pattern Language”

Architect Christopher Alexander



"Each pattern describes
a **problem** which occurs **over and over again**,
and then the **core of the solution** to that problem,
in such a way that you can **use this solution a million times over**,
without ever doing it the same way twice."

Pattern-parts

- Name
- Problem
- Solution

This is not all!

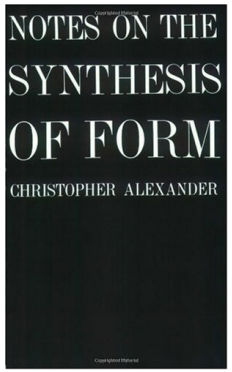
Pattern-parts

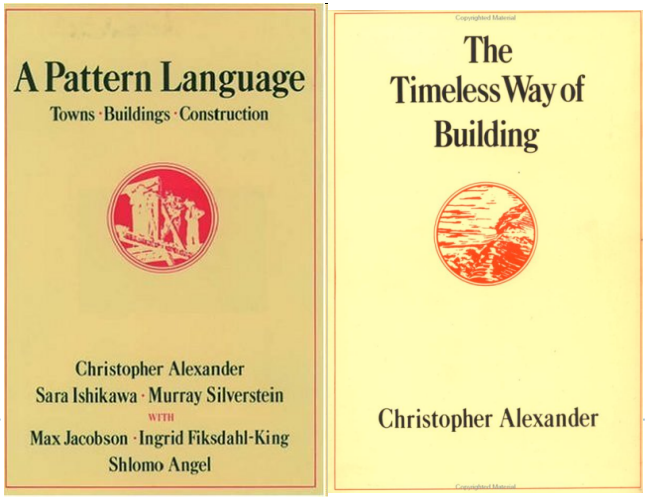
- Name
- Context
- Problem
- Forces
- Solution
- Resulting context/ consequences
- Related patterns
- Examples

- Especially this "extra" information makes patterns valuable and easily applicable.

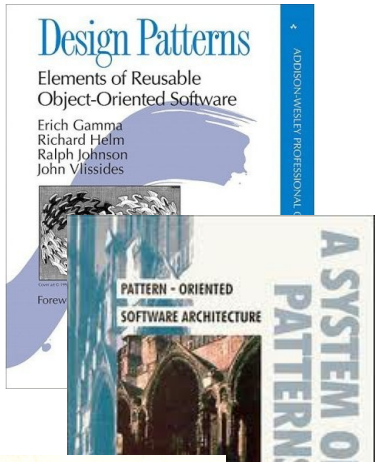
Evolution of Pattern Applications

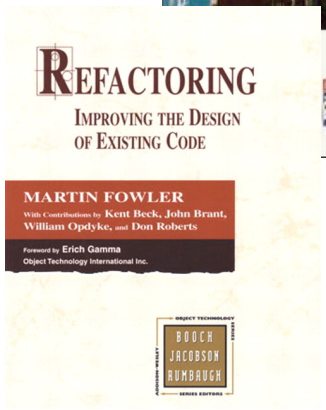
architecture





ICT





Waarom überhaupt SA?

Waarom niet alleen functionaliteit realiseren??

-> The Big Ball of Mud!

Foote, B., & Yoder, J. (1997). Big Ball of Mud. In *Pattern Languages of Program Design* (pp. 653–692). Addison-Wesley.

Waarom überhaupt SA?

Software architectuur belangrijk om

- Complexiteit te beheersen en
- (vooral) kwaliteitseigenschappen te realiseren.

Wat zijn kwaliteitseigenschappen?

- ???
- ???
- …

Quality Without a Name: ISO 9126

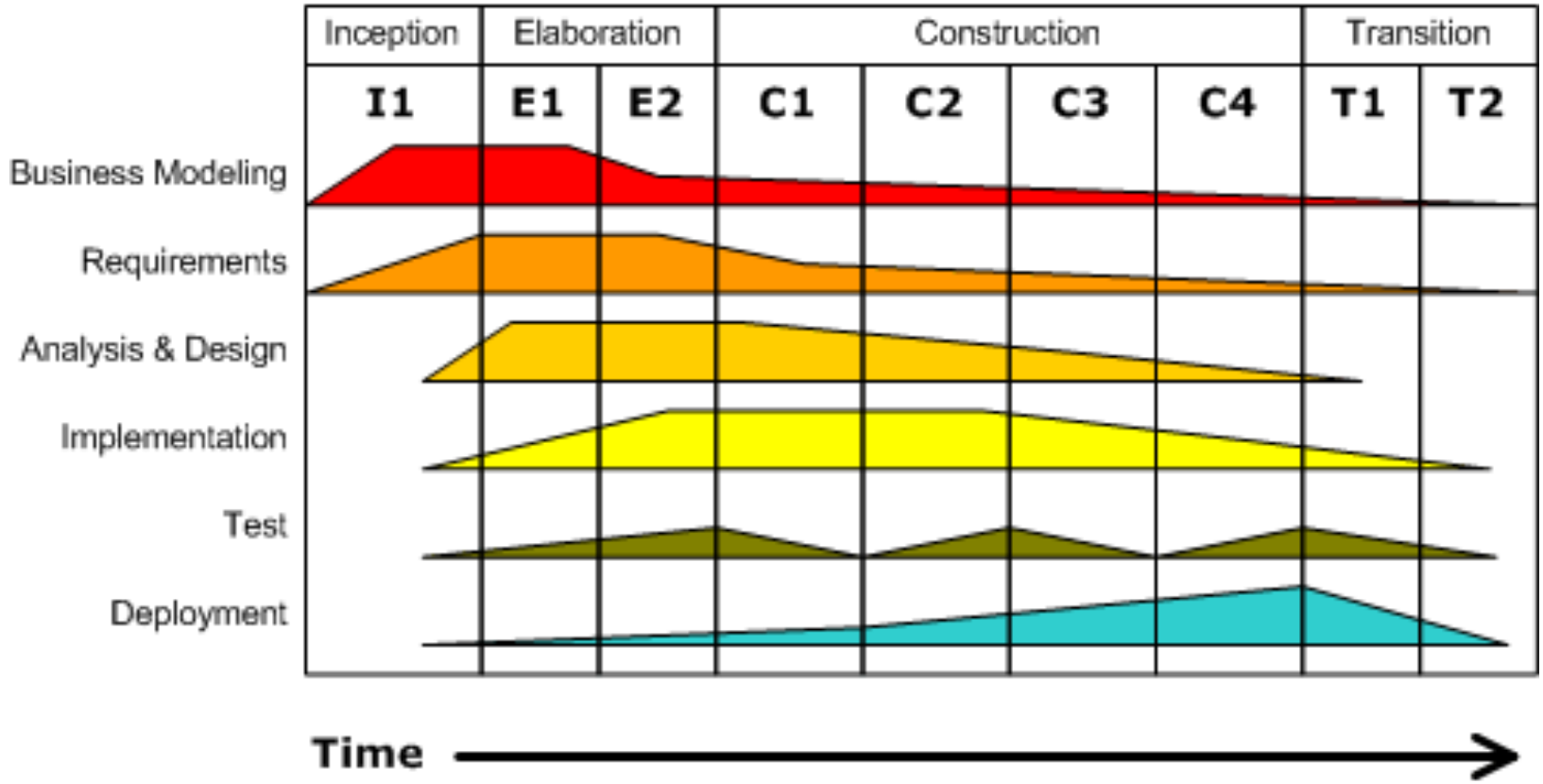
- **Functionality** - A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.
  - Suitability, Accuracy, Interoperability, Security, Functionality Compliance
- **Reliability** - A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.
  - Maturity, Fault Tolerance, Recoverability, Reliability Compliance
- **Usability** - A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.
  - Understandability, Learnability, Operability, Attractiveness, Usability Compliance
- **Efficiency** - A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.
  - Time Behaviour, Resource Utilization, Efficiency Compliance
- **Maintainability** - A set of attributes that bear on the effort needed to make specified modifications.
  - Analyzability, Changeability, Stability, Testability, Maintainability Compliance
- **Portability** - A set of attributes that bear on the ability of software to be transferred from one environment to another.
  - Adaptability, Installability, Co-Existence, Replaceability, Portability Compliance

Zie http://en.wikipedia.org/wiki/ISO/IEC_9126

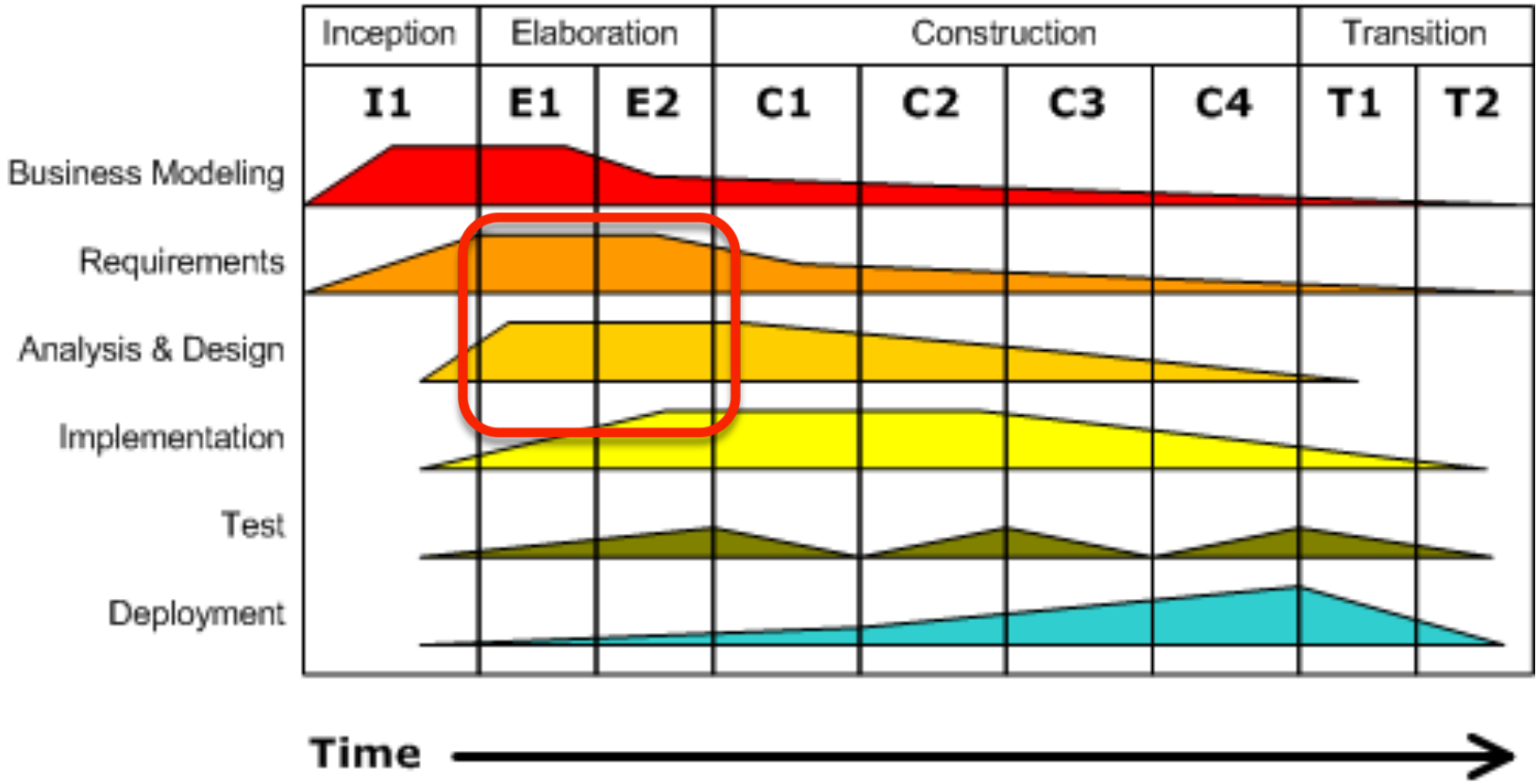
The bigger picture

- Hoe komen we erachter welke QA's belangrijk zijn?

  -> System Requirements!

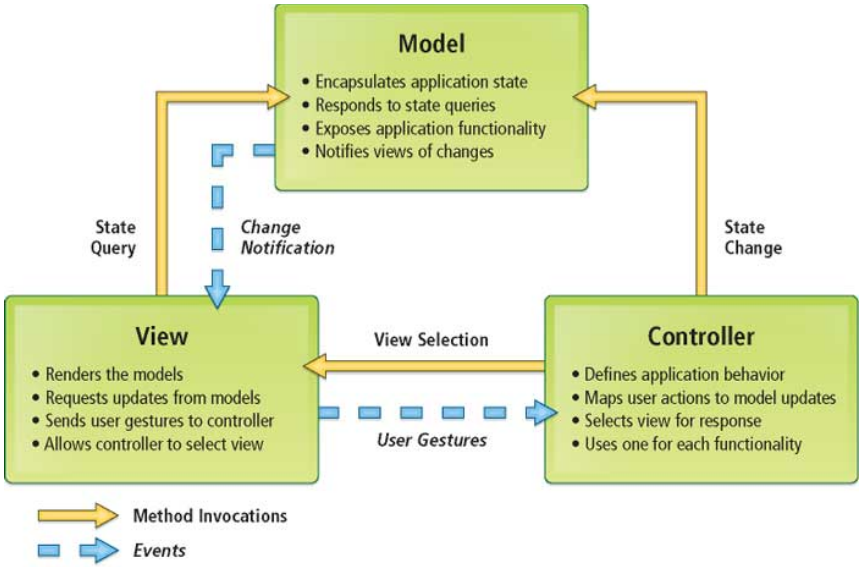


De vertaalslag: patterns!



- Architecturally significant requirements -> Patterns

Voorbeeld: MVC

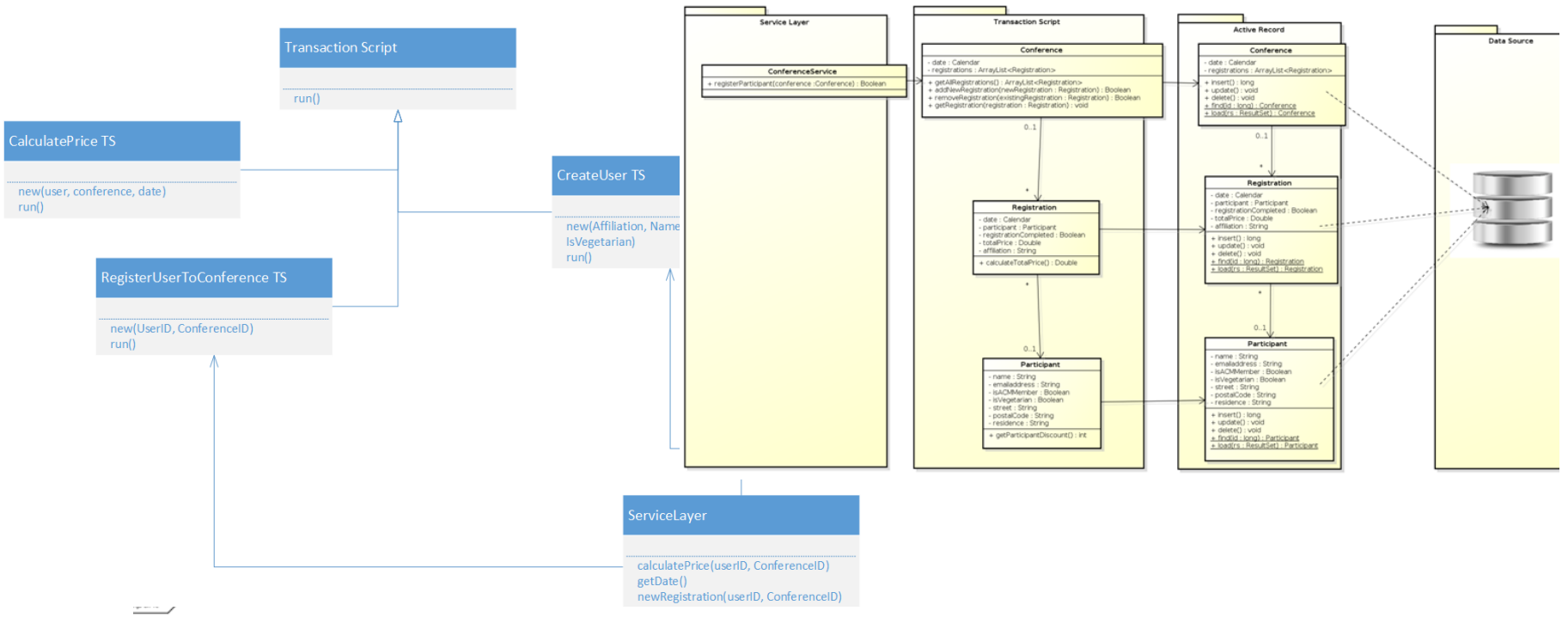


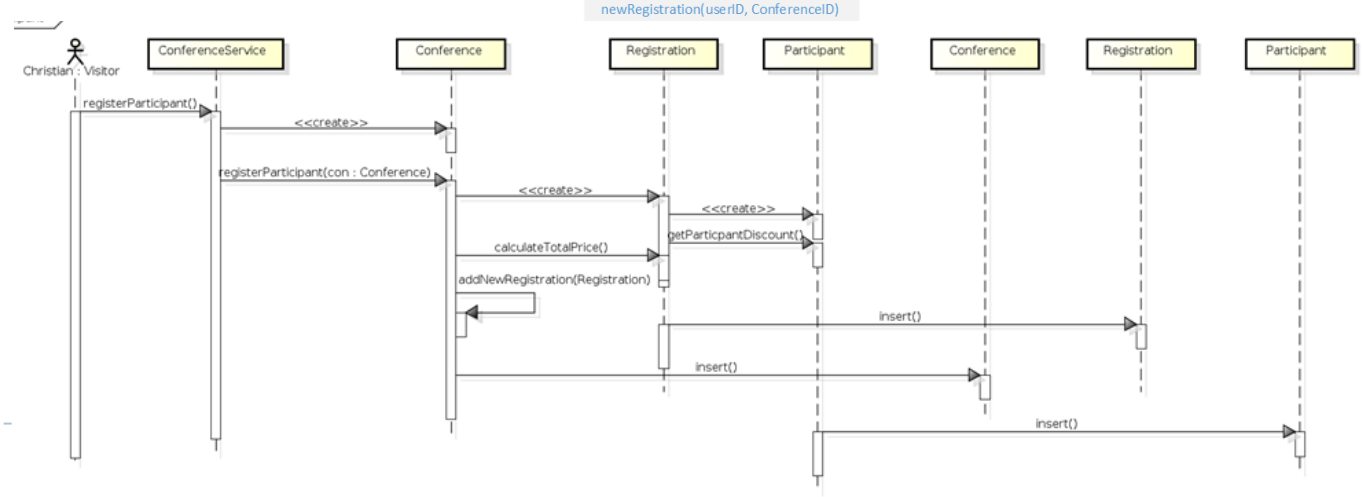
- Wat zijn hier de voorbeelden van bereikte kwaliteiten?
- Altijd van toepassing?
  - Gevaarlijk en soms helemaal niet nodig (KEEP IT SIMPLE, STUDENT)

Een oude oefening

- For the PLoP conference a new registration system is needed.
- It should include the minimal functionality of a conference registration:
  - Price is $600
  - Early registration (with $100 discount, until sep 15)
  - 10% discount for ACM members
  - Vegetarian meal option
  - Name, affiliation, address, emailaddress, veggie-opt and correct price need to be stored
- Make a design for that system!

De resultaten

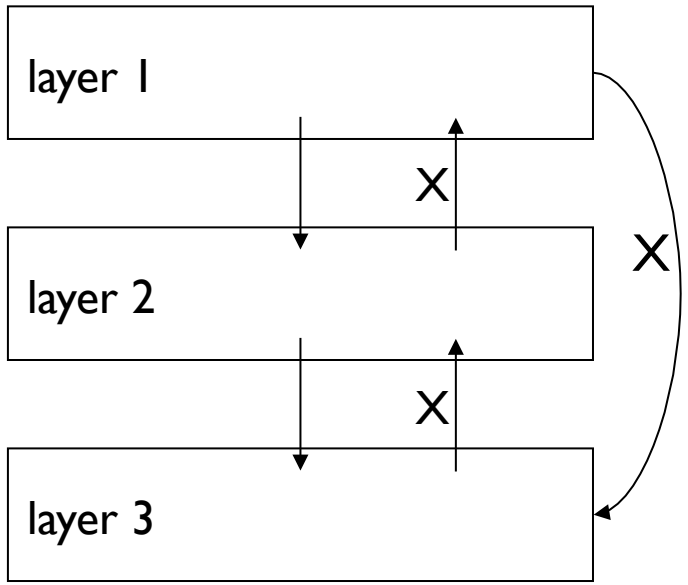




Decisions and Justification

- Belangrijk onderdeel in SAD
- Patterns toepassen is altijd een trade-off tussen kosten en baten
- Soms voordelig voor een criterium en nadelig voor een ander

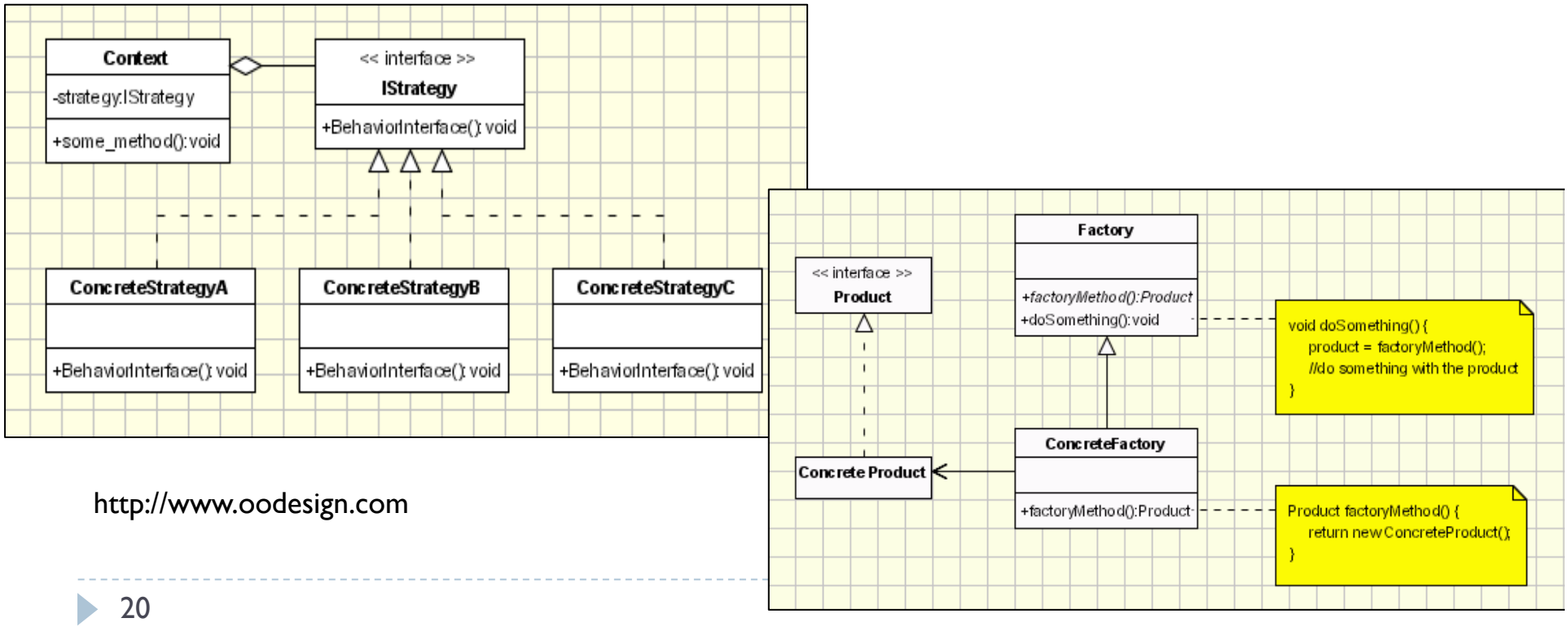
- LAYER Pattern



Context, Problem, and Consequences First

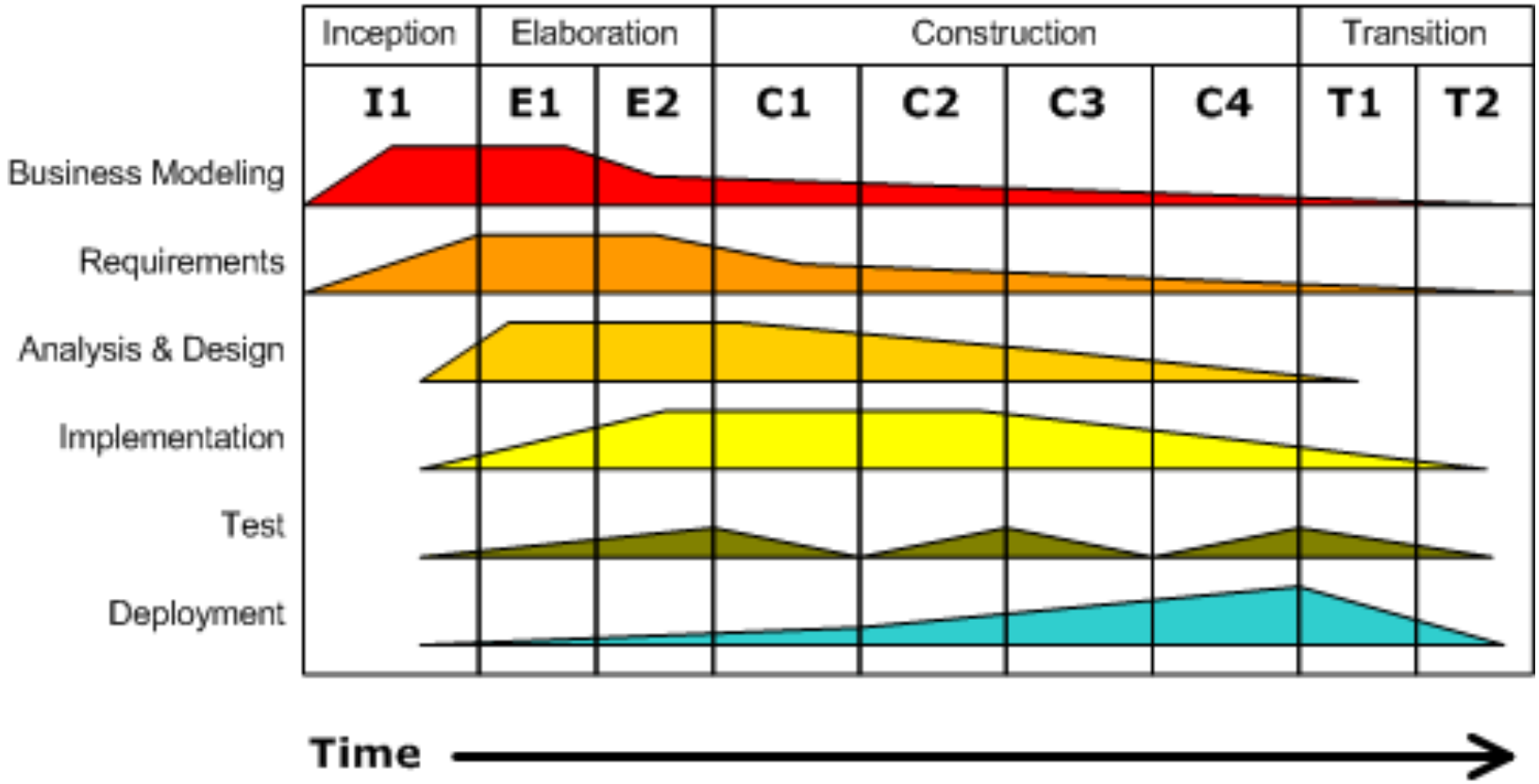
- Dus: per system/context/situatie kijken welke pattern/s daadwerkelijk bijdragen aan een verbetering in het geheel

- Soms zijn hier ook combinaties aan patterns helpend



http://www.oodesign.com

Alles is goed (met RUP)?



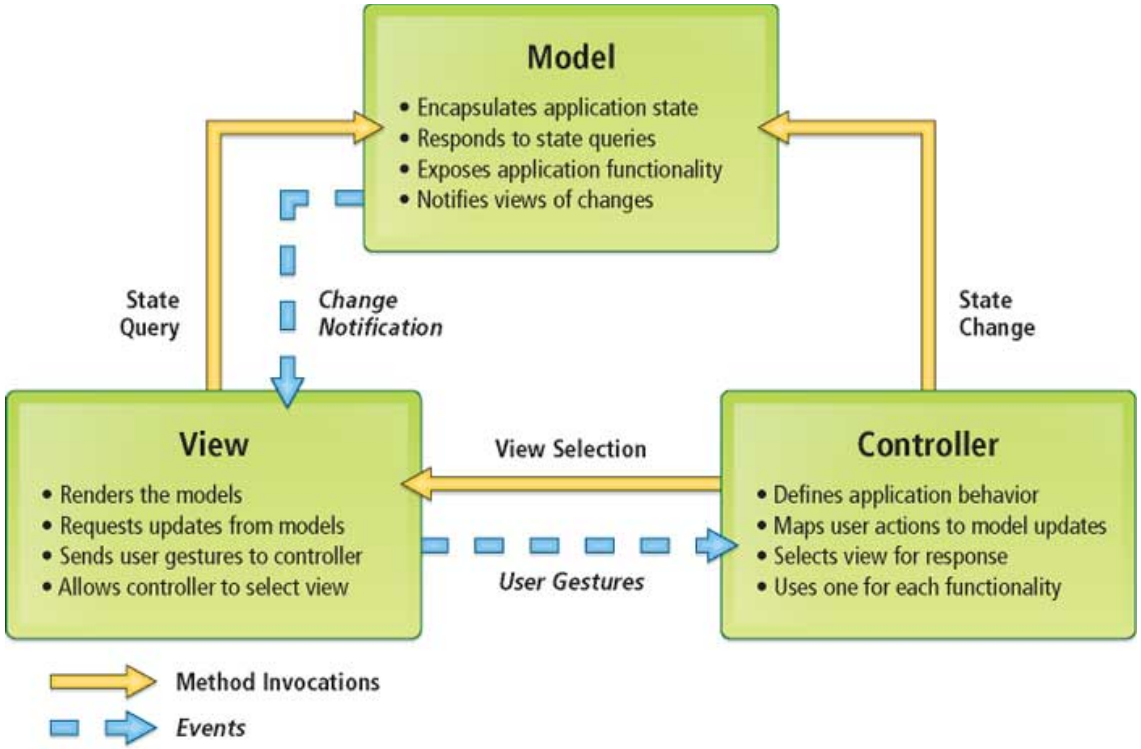
Software Entropy

Lehman's laws (among others):

1. A computer program that is used will be modified.
2. When a program is modified, its complexity will increase, provided that one does not actively work against this.

- Refactoring for structural improvement
- Follow intended architecture
  - (and update it when necessary)

What does “follow architecture” mean?



Software Architecture Compliance Checking

Materiaal van Leo Pruijt, HU

HOGESCHOOL UTRECHT

Software Architecture & Practice

Intended ↔ Implemented Architecture

At least two versions of a Software Architecture exist…

- A system's **intended architecture** captures the design decisions made prior to the system's construction
  - It is the *as-conceived* or *prescriptive* architecture
  - To answer all requirements: functional and non-functional
- A system's **implemented architecture** describes how the system has been built
  - It is the *as-implemented* or *descriptive* architecture
- Unfortunately there often is a substantial difference between the *intended architecture* and the *implemented architecture*…

Reasons for differences… 1/2

Differences between intended architecture and implemented architecture are often caused through evolution:

- When a system evolves, ideally its intended architecture is modified first
- In practice, the system – and thus its implemented architecture – is often directly modified
- This happens because of
  - developer sloppiness
  - perception of short deadlines which prevent thinking through and documenting
  - lack of documented intended architecture
  - need or desire for code optimizations
  - inadequate techniques or tool support

Taylor, Medvidovic & Dashofy (s.d.) *Basic Concepts, Software Architecture Lecture 3*. http://www.softwarearchitecturebook.com/svn/main/slides/ppt/03_Basic_Concepts.ppt

Reasons for differences… 2/2

- Reasons why intended and implemented architecture differ…
  - Architectural drift
  - Architectural erosion

- *Architectural drift* is the introduction of principal design decisions into a system's implemented architecture that
  - are not included in, encompassed by, or implied by the intended architecture
  - but which do not violate any of the intended architecture's design decisions

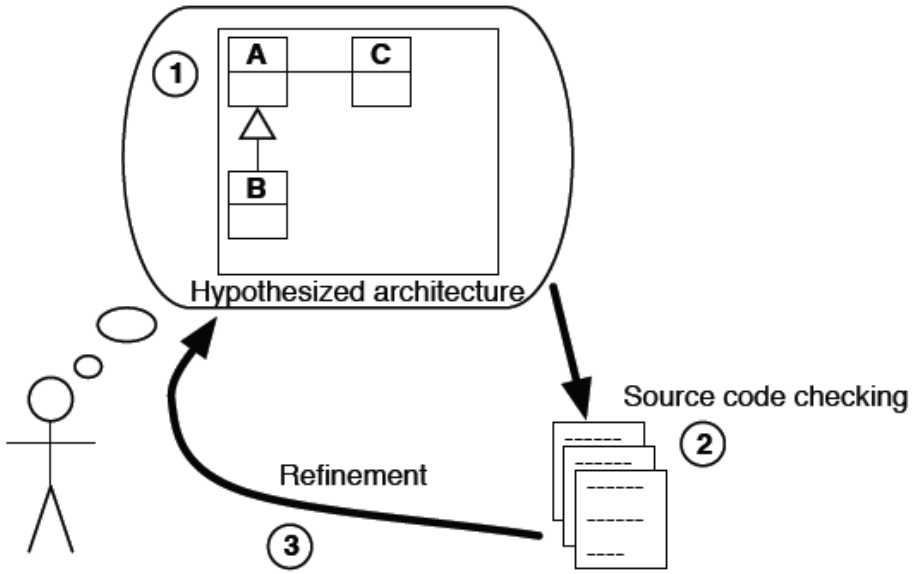
- *Architectural erosion* is the introduction of architectural design decisions into a system's implemented architecture that violate its intended architecture

Taylor, Medvidovic & Dashofy (s.d.) *Basic Concepts, Software Architecture Lecture 3*. http://www.softwarearchitecturebook.com/svn/main/slides/ppt/03_Basic_Concepts.ppt

Software Architecture Compliance Checking (SACC)

- SACC is an approach to bridge the gap between the high-level models of architectural design and the implemented program code, and to prevent architectural erosion.
- *Architecture compliance* is "a measure to which degree the implemented architecture in the source code conforms to the planned software architecture".
- The terms architecture compliance and its synonym architecture conformance are both used in literature.

Top-down process



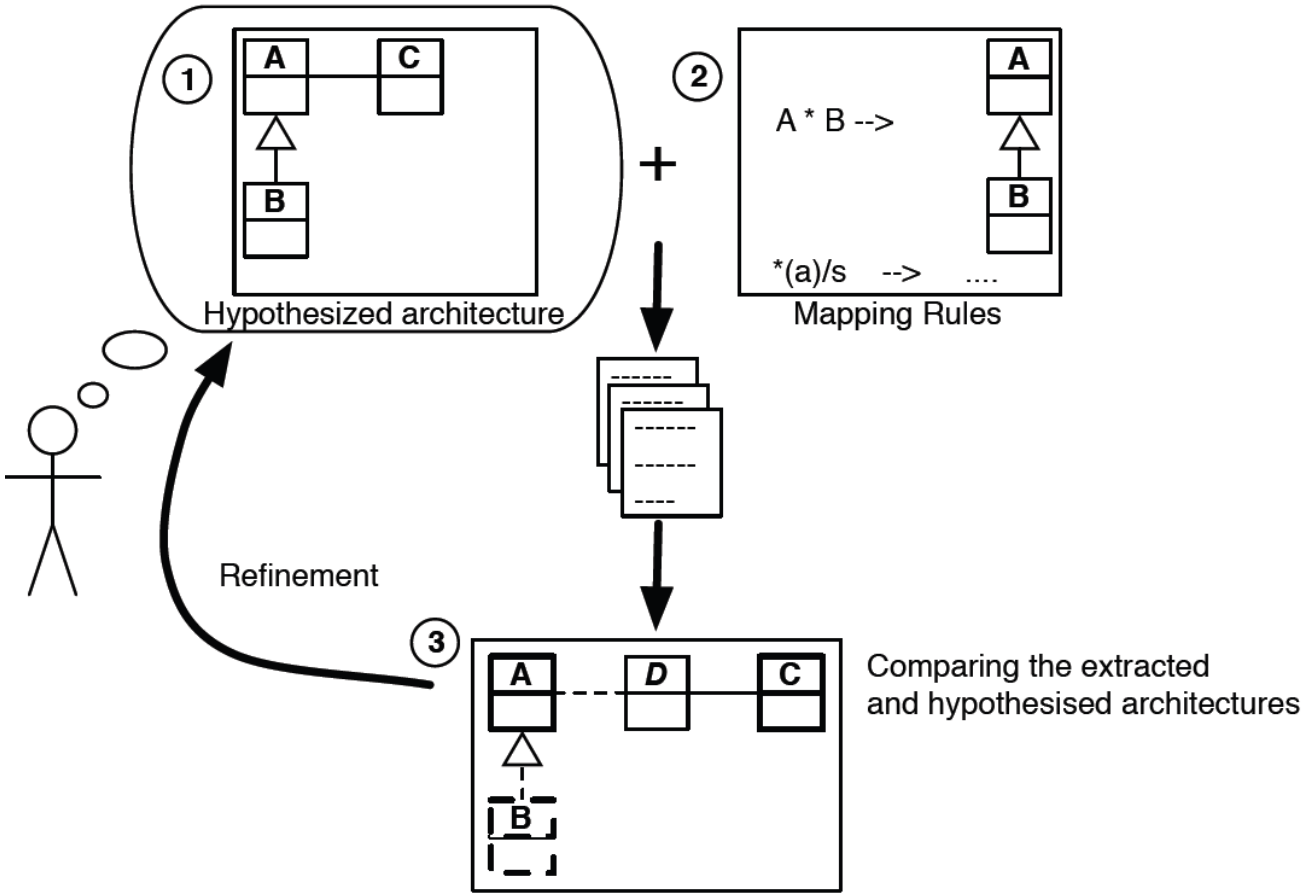
Violation checking conform the Reflection Model

a) Define the intended architecture: modules & rules

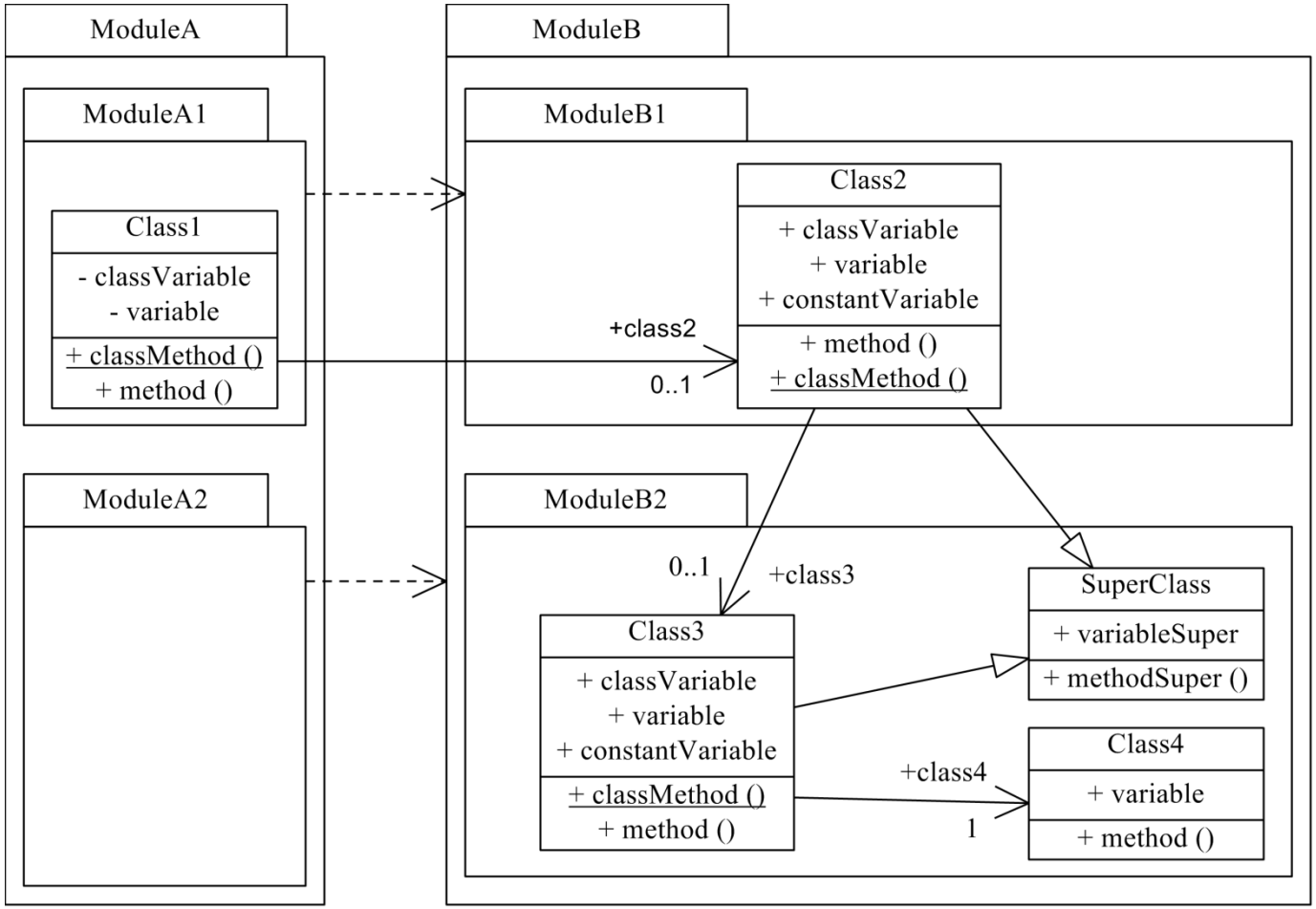
b) Map source code units to the modules

c) Compare the intended and implemented architectures

d) Report violations against the intended architecture



Example: Intended Architecture



At *model level* (in UML), a dependency is shown by a dashed arrow: ---------->

Violating Dependencies in Source Code

Examples of violating dependencies within Class1:

Category	**Dependency Types**	**Example Code** (from Class1)
Import	Class import	import ModuleB.ModuleB2.Class3;
Type declaration	Instance variable Parameter Return type Exception Type cast	private Class3 class3;
Method call	Instance method Class method Constructor	variable = class3.method();
Variable access	Instance variable Class variable Constant variable Local variable Enumeration	variable = class3.variable;
Inheritance	Extends class Extends abstract class Implements interface	public class Class1 extends SuperClass { }
Annotation	Class annotation	@Class3

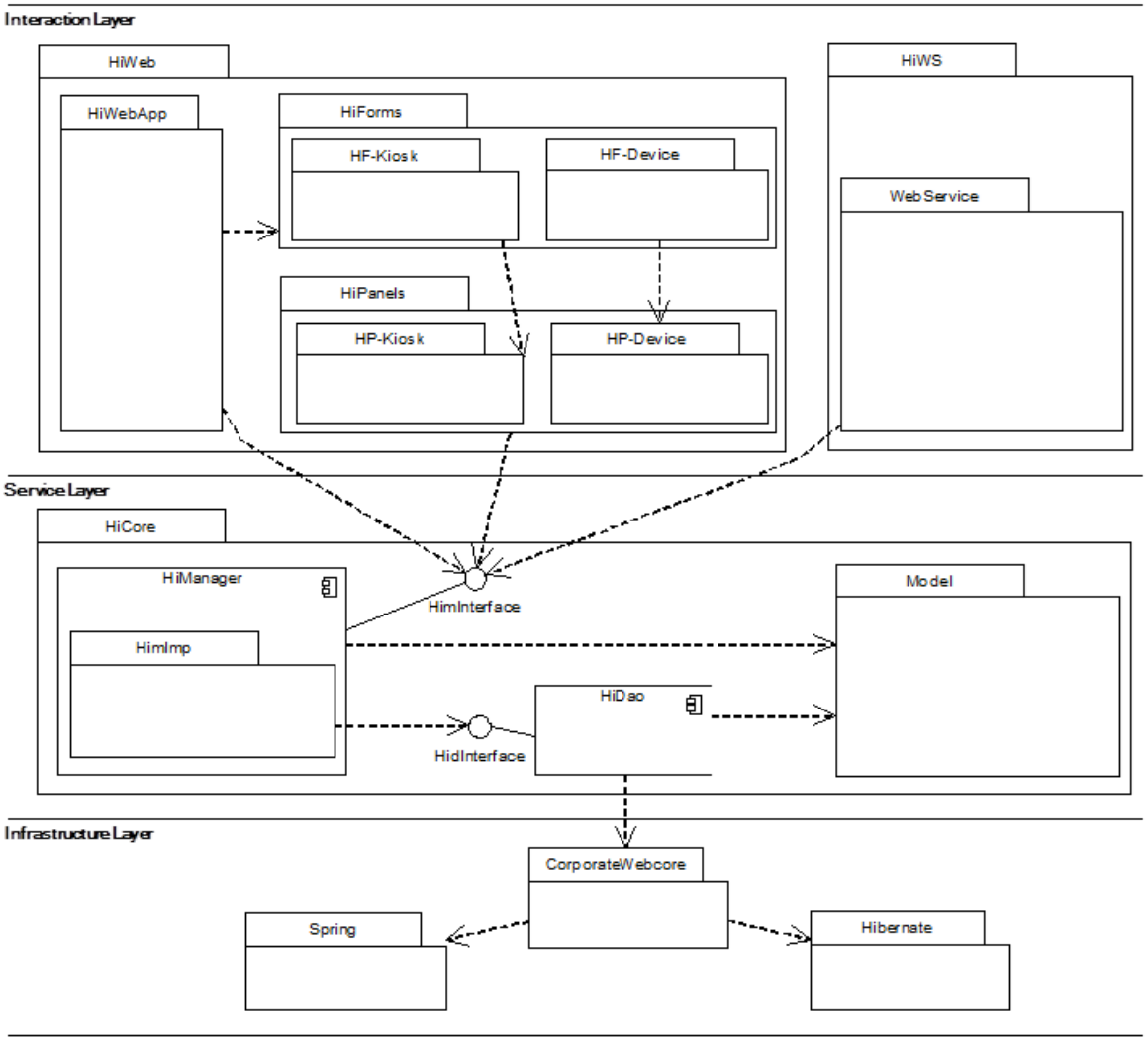
Semantically Rich Modular Architecture

Different Types of Modules and Rules may be present in a SA!

Module Types

- *Logical clusters* represent units in the system design with clearly assigned responsibilities, like subsystem "HiWeb".
- *Layers* are modules with a hierarchical level and constraints on the relations between the layers.
  - Which constraints?
- *Components* are autonomous modules with *Facades* and specific rules to enforce information hiding.
  - How can information hiding be enforced?
- *External systems* are sytems or libraries used by the core system.

Example: Intended SA of a system of the Schiphol Group



What is present?

- Modules?
- Module Types?
- Rules?
- Rules Types?

Rule Types

Modular architectures may contain rules of different types

- Each rule type characterizes another kind of *constraint* on a module.
- These constraints are categorized as follows:
  - Property rule types
  - Relation rule types

Property Rules Types

Property rule types constrain a certain characteristic of the elements included in the module and their sub modules.

Type of Rule	Description (D), Example (E)
Naming convention	D: The names of the elements of the module must obey the specified standard. E: HiDao elements must have suffix DAO in their name.
Responsibility convention	D: All elements of the module must adhere to the specified responsibility. E: HiForms is responsible for presentation logic only.
Visibility convention	D: All elements of the module have the specified or a more restricting visibility. E: HiManager classes have package visibility or lower.
Facade convention	D: No incoming usage of the module is allowed, except via the facade. E: HiManager may be accessed only via HimInterface.
Inheritance convention	D: All elements of the module are sub classess of the specified super class. E: HiDao classes must extend CorporateWebCore.Dao.GenEntityDao.

Relation Rules

Relation rule types specify whether a module A is allowed to use a module B

- use a module = depend on a module

Type of Rule	Description (D), Example (E)
Is not allowed to use	D: No element of the module is allowed to use the specified to-module. E: HiPanels is not allowed to use HiWS.
Back call ban (specific for layers)	D: No element of the layer is allowed to use a higher-level layer. E: Service Layer is not allowed to use the Interaction Layer.
Skip call ban (specific for layers)	D: No element of the layer is allowed to use a layer more than one level lower. E: Interaction Layer is not allowed to use the Infrastructure Layer.
Is allowed to use	D: All elements of the module are allowed to use the specified to-module. E: HiWebApp is allowed to use HiForms.
Is only allowed to use	D: No element of the module is allowed to use other than the specified to-mod.(s) E: HiForms is only allowed to use HiPanels.
Is the only module allowed to use	D: No elements outside the from-mod.(s) are allowed to use the specified to-mod. E: HiDao is the only module allowed to use CorporateWebcore.
Must use	D: At least one element of the module must use the specified to-module. E: HiDao must use CorporateWebcore.

SACC with Lattix's Dependency Structure Matrix (DSM)

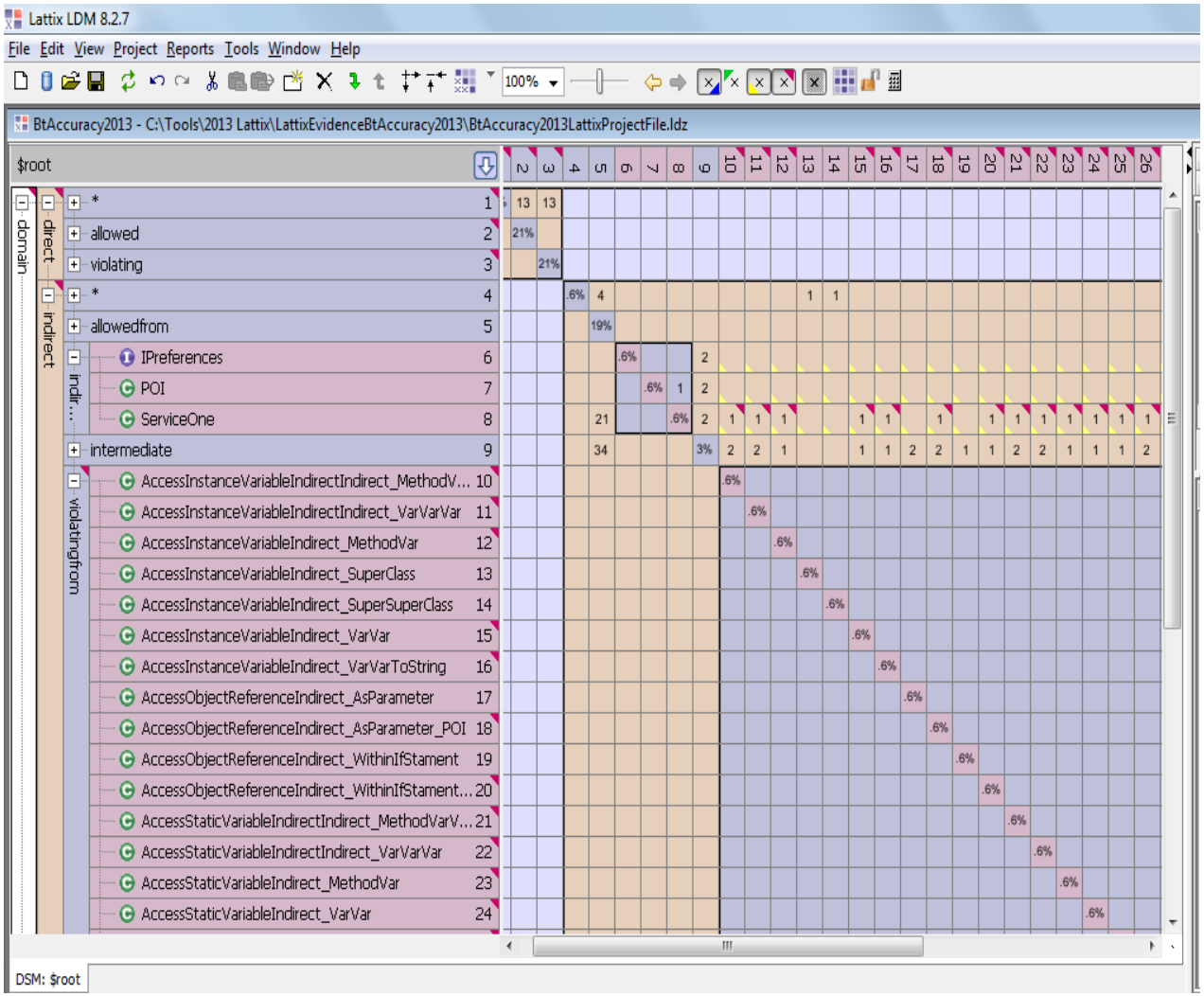
Hogeschool van Arnhem en Nijmegen
HAN University of Applied Sciences

Procedure:

1. Modules are already mapped to code units.
2. Select a module and define a rule
   - Rule language
   - No reflection
3. Violations are reported as red triangles

Example of Violations:

- Module 10-12 use module 8 (ServiceOne)

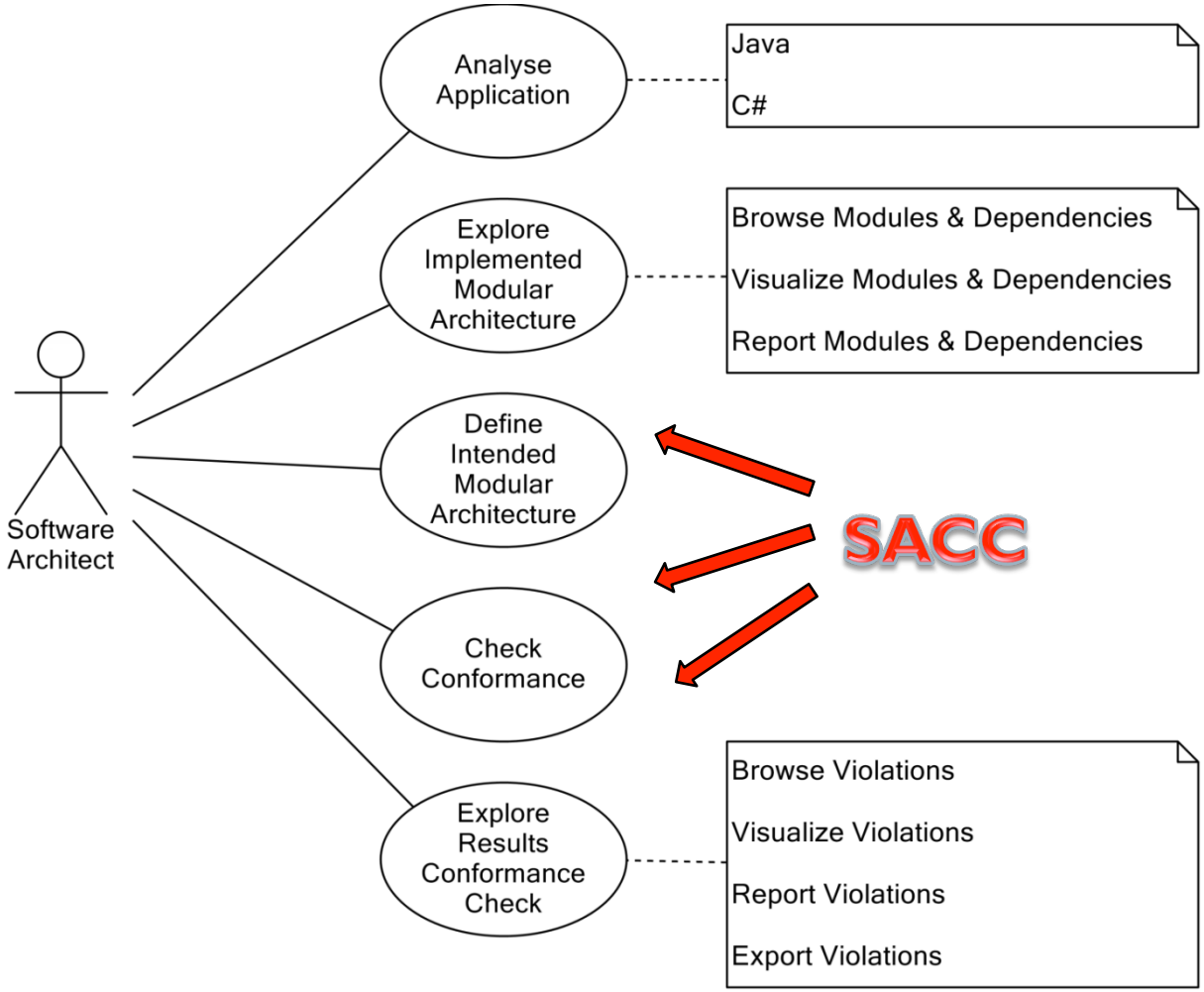


Hogeschool van Arnhem en Nijmegen
HAN University of Applied Sciences

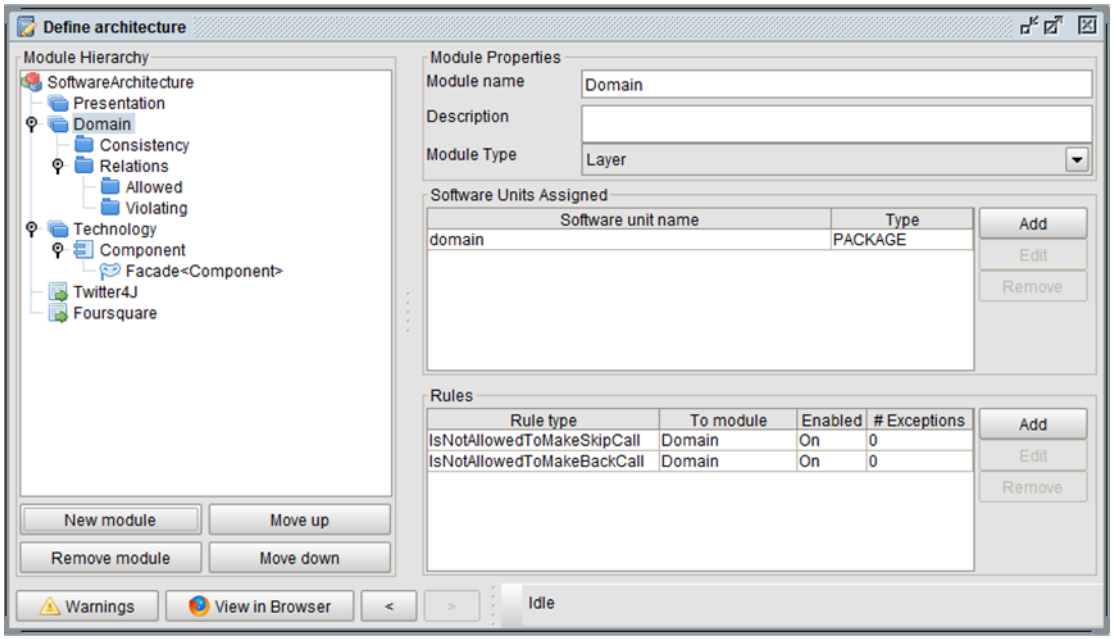
HUSACCT Supports SACC

Procedure:

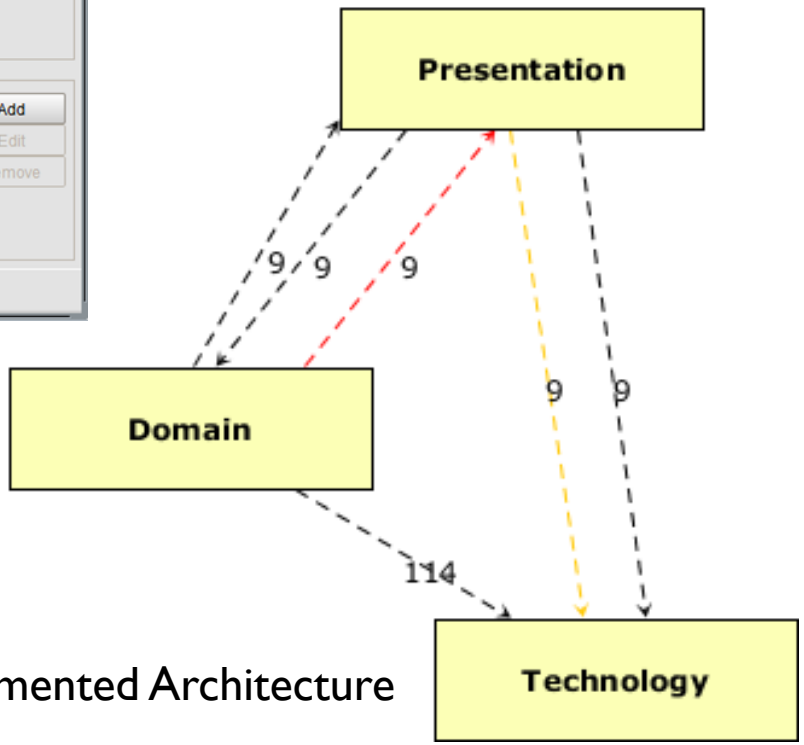
1. Define: Specify the intended architecture: modules & rules
2. Define: Map source code units to the modules
   - After analysis
3. Validate: The rules of the intended architecture are checked in the implemented architecture
4. Validate: Browse, visualize or report or export the violations



SACC with HUSACCT



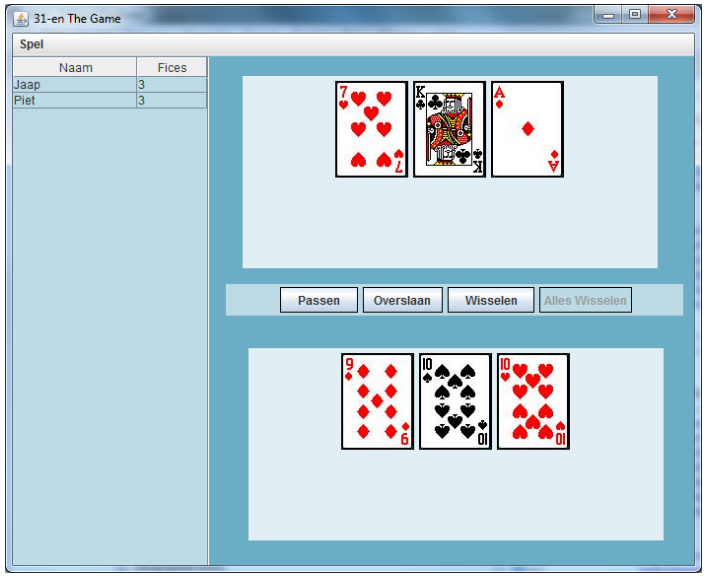
Intended Architecture



Implemented Architecture

HUSACCT tool, manual and video, are attainable via: http://husacct.github.io/HUSACCT/

Voorbeeld SA Compliance Checking: Game31

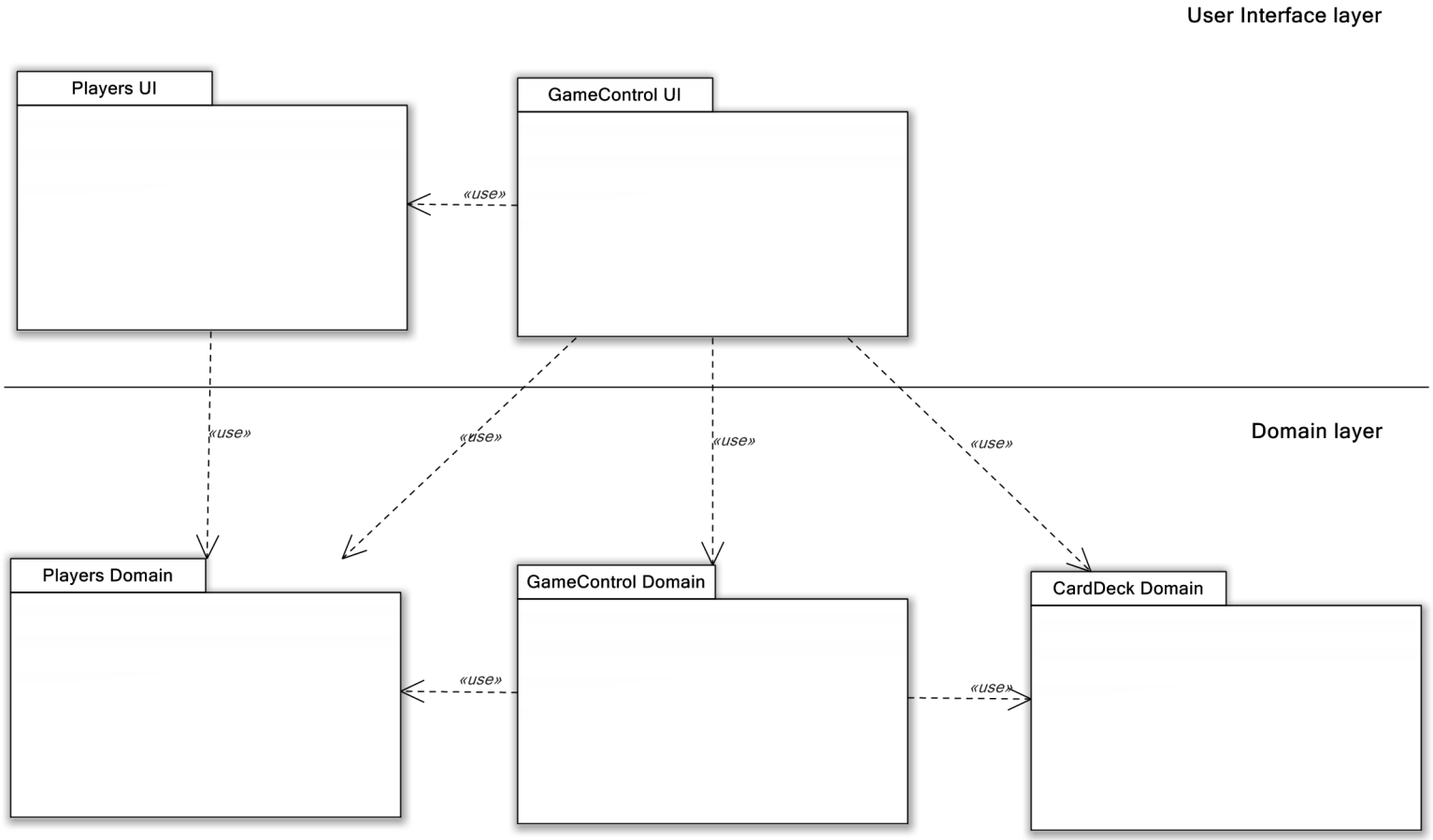


We will perform one check

1. Intended Architecture ↔ Implemented Architecture
   - The Intended Architecture aims at:
     - Modules with high cohesion within a module (one knowledge area)
     - Modules with low coupling between the modules
     - Maintainability, reusability of modules for other card games

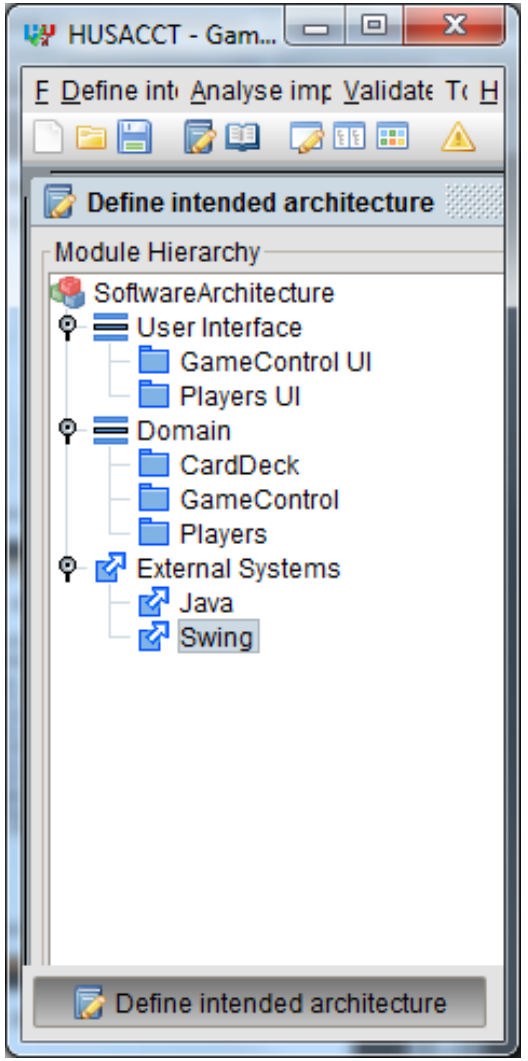
Intended Architecture ↔ Implemented Architecture

Intended Architecture



Define Intended Architecture in HUSACCT: Modules

1. Menu: Define Intended Architecture
2. Add the following modules:
   - User Interface, type = Layer
     - GameControl UI, type = Subsystem
     - Players UI, type = Subsystem
   - Domain, type = Layer
     - CardDeck, type = Subsystem
     - GameControl, type = Subsystem
     - Players, type = Subsystem
   - External Systems, type = ExternalLibrary
     - Java, type = ExternalLibrary
     - Swing, type = ExternalLibrary



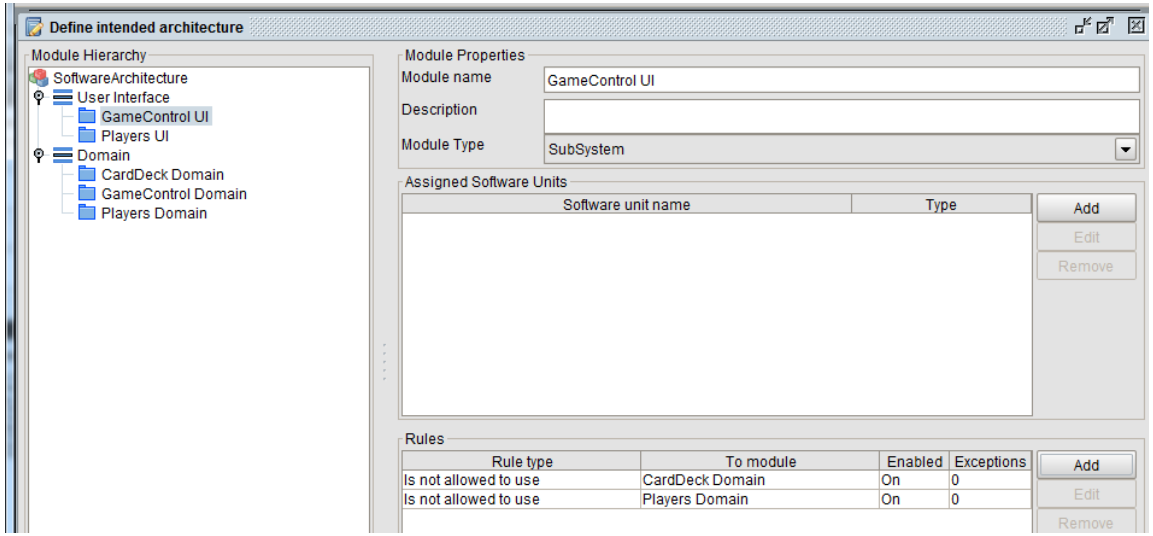
Define Intended Architecture: Rules

Automatically added

1. Layer Domain is not allowed to back call layer User Interface

Add manually

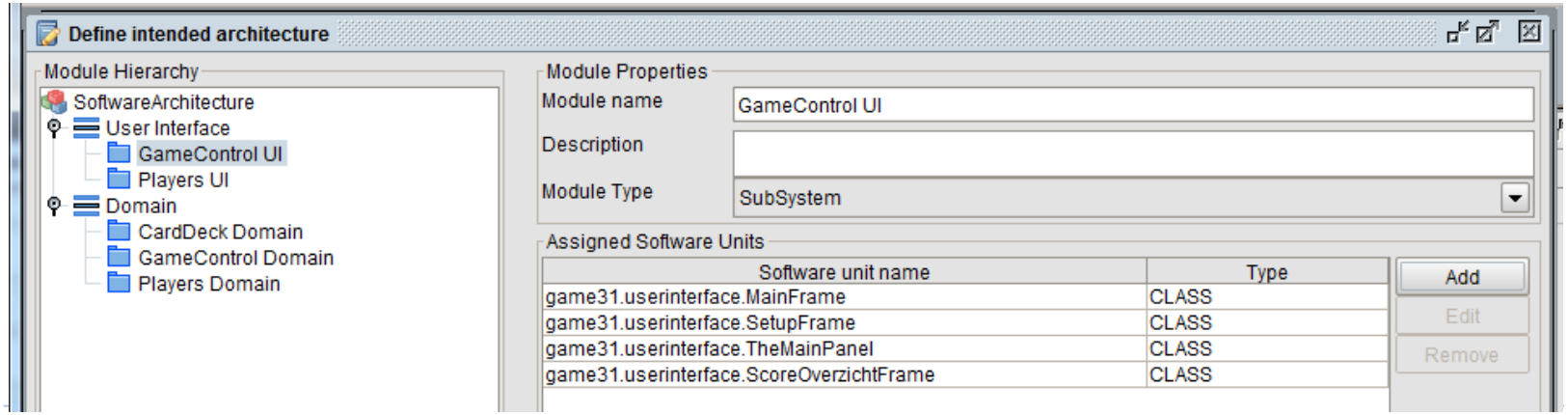
2. Players UI is only allowed to use Players Domain
3. Players Domain is not allowed to use CardDeck Domain
4. CardDeck Domain is not allowed to use Players Domain
5. GameControl UI is the only module allowed to use GameControl
6. User Interface is the only module allowed to use Swing



Define Intended Architecture: Assign SUs

Assign software units (in the implemented SA) to modules in the intended SA

1. User Interface: game31.userinterface
   1. GameControl UI: MainFrame, TheMainPanel, ScoreOverzichtFrame, SetUpFrame
   2. Players UI: setupSpelers
2. Domain: game31.domein
   1. CardDeck: KaartSpel, KaartStapel, Kaart
   2. GameControl : Spel, SpelRonde, Deelname, Tafel, Hand
   3. Players: Speler, HumanSpeler, ComputerSpeler, Pot
3. External Systems: -
   1. Java: xLibraries.java
   2. Swing: xLibraries.javax.swing



Check Conformance

Menu: Validate Conformance → Validate Now

- Study the results

1. Which rules are violated?
2. Which rules are not violated?
3. One rule has a very high number of violations
   1. Select the rule and study the violations
   2. Are all violations in line with the intention of the rule?
4. Add an exception to a rule
   1. View: Define Intended Architecture
   2. Select User Interface.Players UI
   3. Select the rule 'is only allowed to use' → Edit
   4. Add Exception: Players UI 'is allowed to use' ExternalSystems
   5. Save
   6. Check the conformance again
   7. Are there still violatations against this rule?

Create an Intended Architecture Diagram

Menu: Define intended architecture → Intended architecture diagram

- Include Violations and External libraries: Options → Show
- Study the results

1. Which violations are the most problematic ones?

Create a detailed diagram

- Select all three top-level modules concurrently
- Zoom in
- Make the model readable: resize frames, move modules

2. Which violations are the most problematic ones?

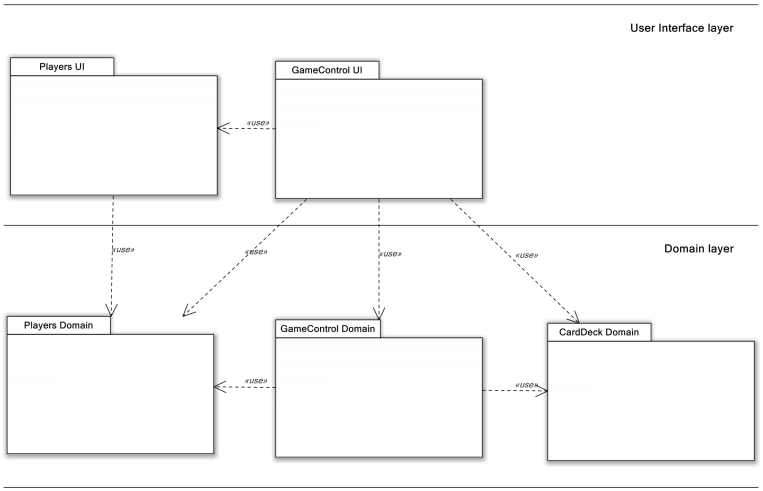
Select the dependency arrow between Domain.GameControl and External Systems.Java

- Study the dependencies
- Which unreported violations are visible to the intention of a defined rule?

3. What to do, to get them reported as violations?

Voorbeeld: Conclusions

- The implemented architecture (see the conclusions of the SA reconstruction):
  - Is hard to maintain
  - Offers limited reuse opportunities
- It will take a lot of work to reconstruct Game31 conform the intended architecture
  - Consequently, it is not often done
  - So, developers have to cope with the implemented architecture for the rest of the system's lifetime
- It wouldn't have taken a lot of work to construct Game31 conform an intended architecture
- So:
  1. Design an architecture
  2. Check the conformance regularly
  3. Adjust the architecture, if needed



Conclusions

- Kwaliteit heeft een aantal namen (ISO 9126…)
- Belangrijk in de hele levenscyclus

- Moet actief aan worden gewerkt om doelen van architectuur te bereiken/te behouden
- Patterns helpen/ondersteunen, maar zijn geen silver bullets!

Bedankt!

www.koeppe.nl