Principles & Tactics

Architectural Thinking for Intelligent Systems

Winter 2019/2020

Prof. Dr. habil.Jana Koehler

Agenda

- Implementation of functional and non-functional requirements applying principles & tactics
- 10 principles
  - Loose Coupling
  - High Cohesion
  - Design for Change
  - Separation of Concerns
  - Information Hiding
  - Abstraction
  - Modularity
  - Traceability
  - Self documentation
  - Incrementality
- Tactics as a method to address a quality attribute

Conception of the Architecture

- Use cases, user stories & scenarios are clarified such that we can proceed with an acceptable level of risks
- The context view has been reviewed with stakeholders to reach agreement on what we will built and what we require from the environment
- We have a good understanding of the most important architectural decisions that we need to address
- The system idea has been developed

➢ We can start developing the architecture and <u>prototype</u> critical parts of the system

➢ Principles & Tactics

Tutorial Assignment 8:

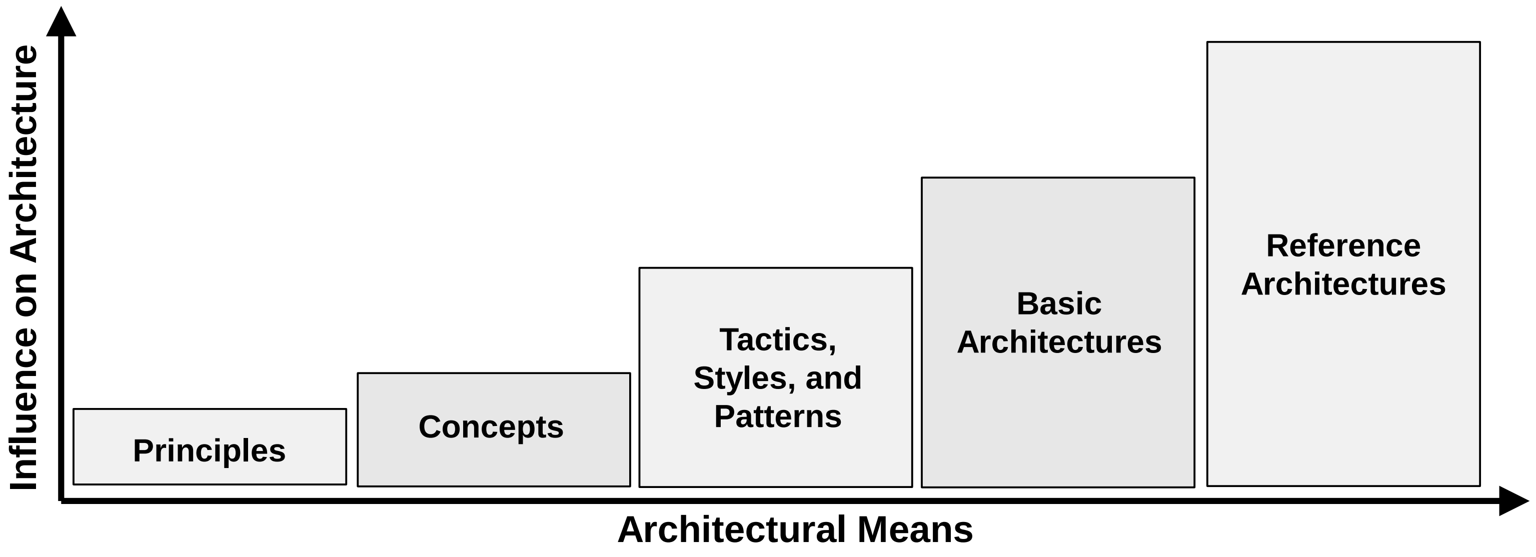
- We apply principles and tactics to further refine the architecture of our system.
- We choose the two most important architectural principles, which will guide architectural decision making.
- We also decide for specific tactics that help us to achieve the desired system qualities.

Principles

"It is only through the relationships between the components of a system that an architecture really takes effect."

Influence of Architectural Means on Architecture

- Principles, such as cohesion or coupling, provide general guidelines
- Architectural styles, tactics and patterns provide detailed solutions for concrete design decisions
- Concepts, such as object orientation or aspect orientation, help implementing principles in the software design



Architectural Principles

- Architectural principles provide proven foundations on which the architecture can be built

- 2 main objectives
  - Reduction of complexity
  - Increased flexibility/changeability by using a good system structure

- Do not say anything about how these principles are applied in a specific case

10 Basic Principles

1. Loose Coupling
2. High Cohesion
3. Design for Change
4. Separation of Concerns
5. Information Hiding
6. Abstraction
7. Modularity
8. Traceability
9. Self-Documentation
10. Incrementality

1. Loose Coupling

- Any type of dependency between two components leads to a coupling
  - Mutual calls, shared data, …

- For a given coupling, one component is the provider, the other the consumer
  - A calls B: A is consumer, B provider
  - A includes B: A is consumer, B provider
  - A writes in a message queue, from which B reads: …
- Coupling is **tight**, if changes in the provider affect the consumer(s)
- Goal must be a loose coupling where providers can change without affecting consumers

2. High Cohesion

- Describes dependencies between structures (subcomponents) <u>within</u> one component
  - Example: methods calling each other within a class

- Components should include all elements, which implement the relevant and connected behaviors of this component
  - Check: Can I understand and change a component without understanding/changing other components?
  - How easy is the component to understand?

- Encapsulate related functionality in one component

3. Design for Change

- Anticipate *foreseeable* changes in the architecture
  - e.g. from open requirements that had to be moved to a next release of a software

- Ignore unforeseeable requirements!
  - Problem of a "too flexible" architecture

- Design components based on goal hierarchies and interrelated user stories

4. Separation of Concerns

- Separate different aspects of a problem from each other and deal with each of these subproblems separately

- Each functionality is implemented in exactly one component and only there

- Break down
  - Requirements
  - organizational responsibilities
  - system into a structure of subsystem
  - complex architecture description into views
  - process of architecture creation into subprocesses

5. Information Hiding

- Only reveal those information entities to a component, the component needs to function correctly
- Hide all other information entities
- Examples
  - OOP: data fields are „private“, data access only via methods
  - Fassade pattern: shields complex systems and controls access to system components
  - Layers: layer n only uses layer n-1, does not know about other layers

6. Abstraction

- Identify important aspects, neglect unimportant details
  - Special case of information hiding
- Most widely used: interface abstraction
- Find commonalities in things that appear to be different at a first glance (entities, value objects, events, services)
- Use when negotiating functional requirements

7. Modularity

- Structure the system such that each component has a clearly defined functional responsibility

- System components easily exchangeable and self-contained
  - manageable, understandable, easy to maintain and reusable

- Achieve simple and stable architectural relationships by finding the right balance between separating concerns and self-containedness

8. Traceability

- Ability to follow structures and architectural decisions from requirements to code (remember SMART)
- How easily can your architecture be understood?
- Traceability as a key to achieve long-term viability
- Foundation to map different views to each other and achieve a consistent description of the system
  - Elementary implementation: use uniform naming conventions

9. Self-Documentation

- Every information required to understand a component or system should be a direct part of the component or system
- Reality: documentation and code get out of sync quickly
- Work in iterations across architecture – design – code
- Achieve stability in naming conventions and key structures and relationsships through domain-driven design
- Communicate!

10. Incrementality

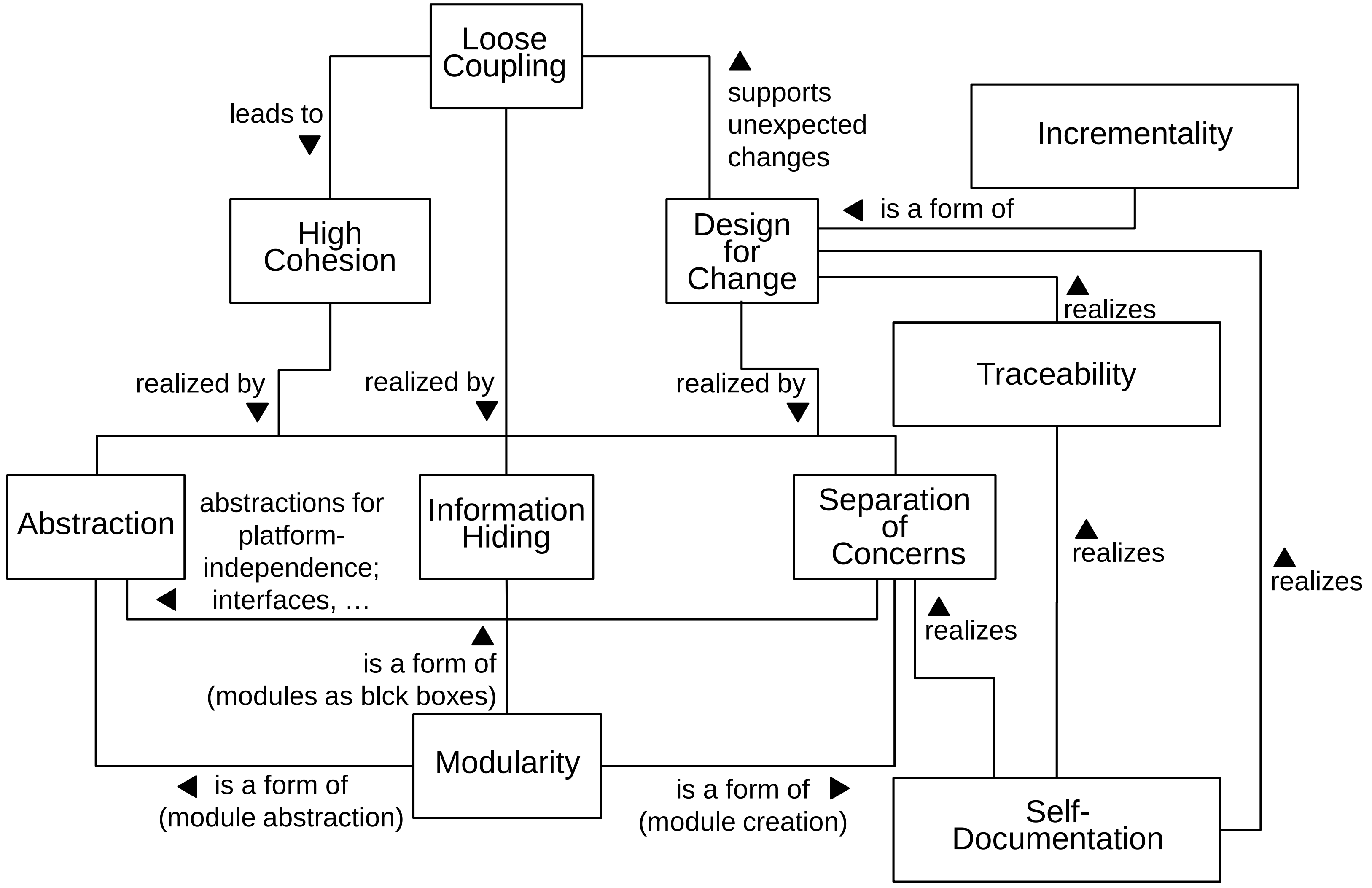
- Apply separation of concerns when developing the architecture
  - Work in phases, define milestones, review results

- Early prototyping
- Early feedback from stakeholders

- Build large systems in iterations

- Achieve piecemeal growth through good release planning
  - Goal hierarchies and scenarios help

How Principles Support Each Other

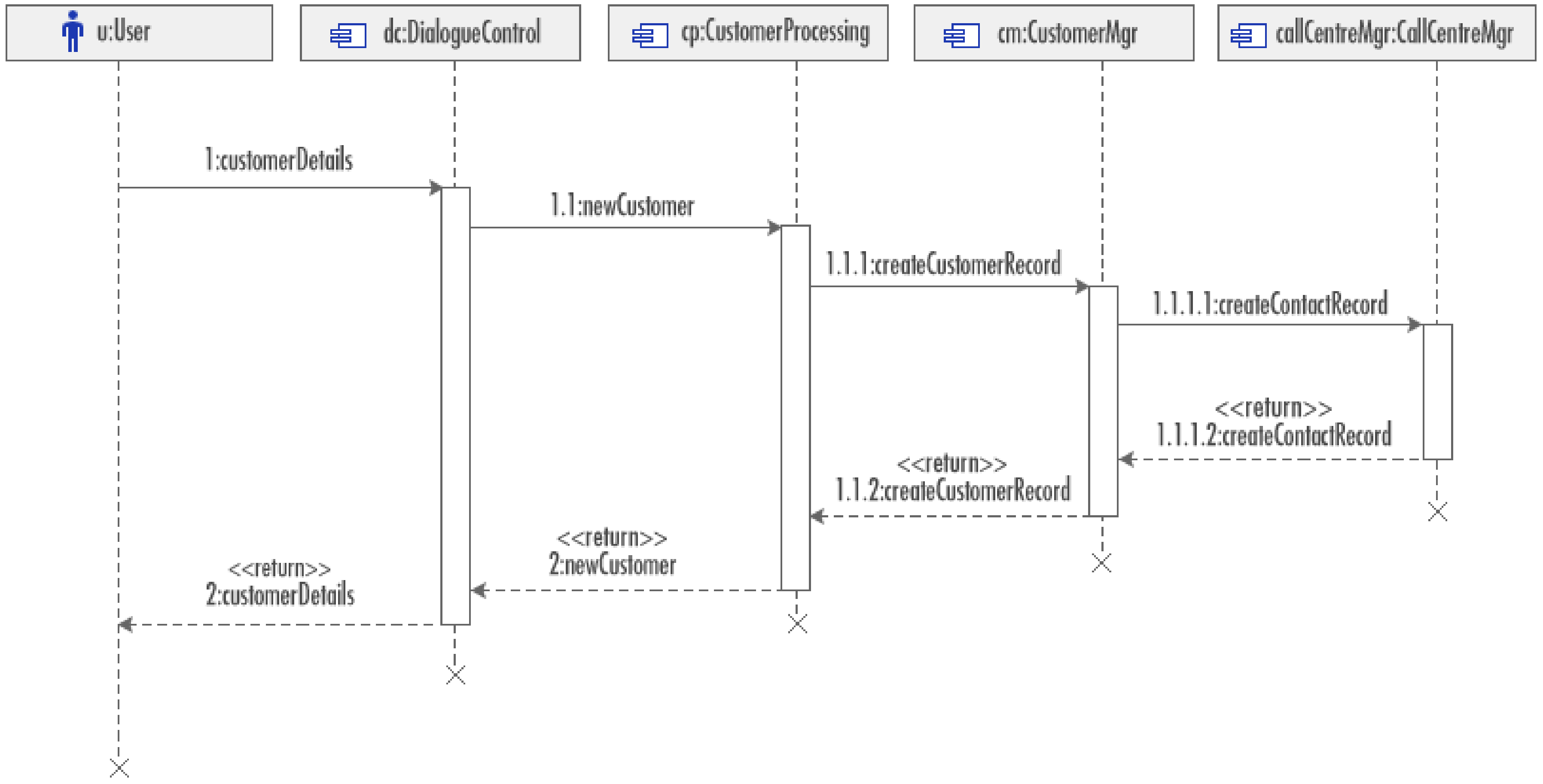


Vogel et al: Software Architektur

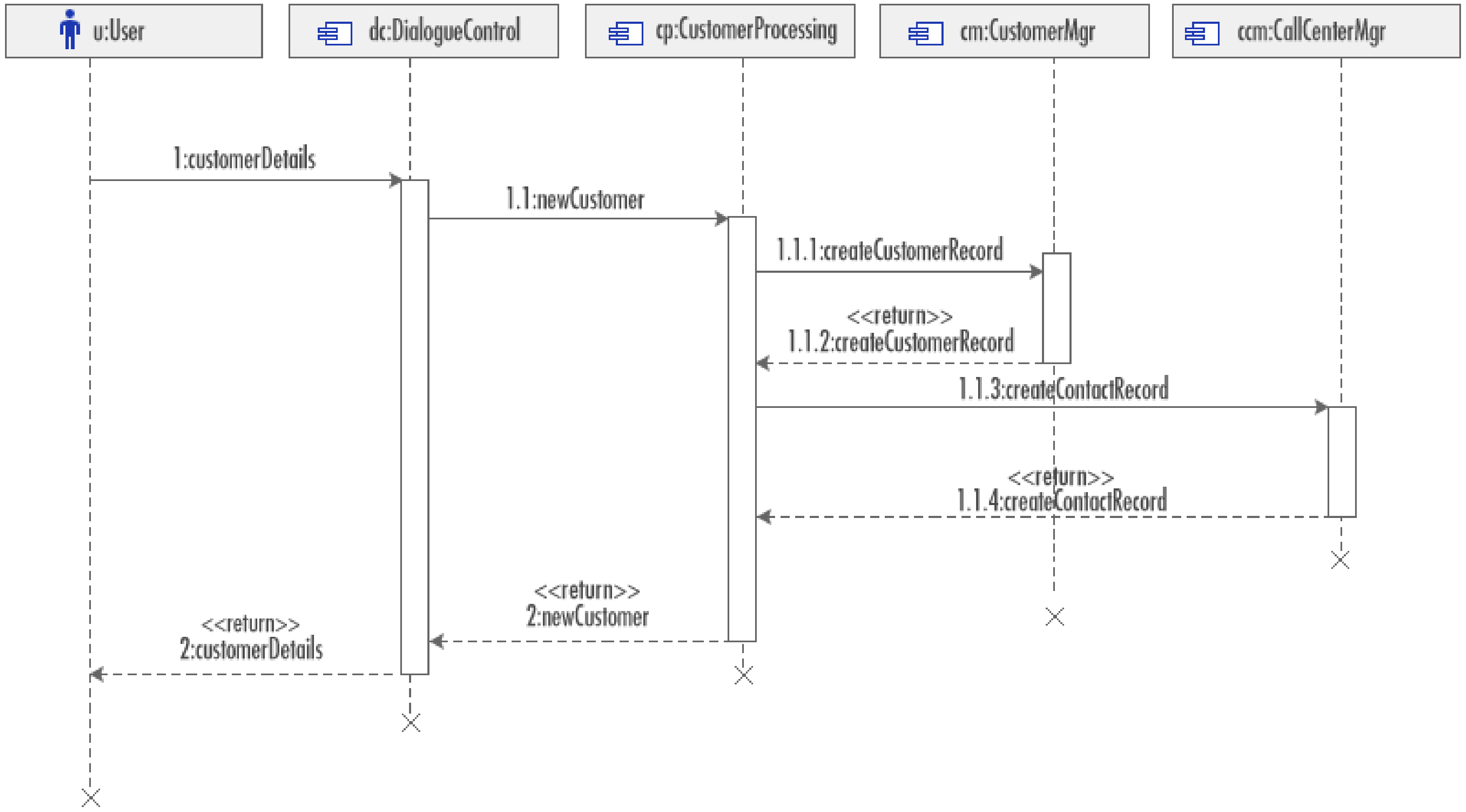
How Principles Support Each Other

- High cohesion can be achieved by abstraction, separation of concerns und information hiding
- Loose coupling and high cohesion can be achieved by modularity
- Design for change can be achieved by loose coupling, abstraction, modularity, separation of concerns and information hiding
- Abstraction helps to implement loose coupling, modularity
- Modularity combines abstraction, separation of concerns and information hiding and supports high cohesion and loose coupling
- Traceability supports loose coupling and design for change
- Self-Documentation supports design for change and traceability

How do you assess Coupling?

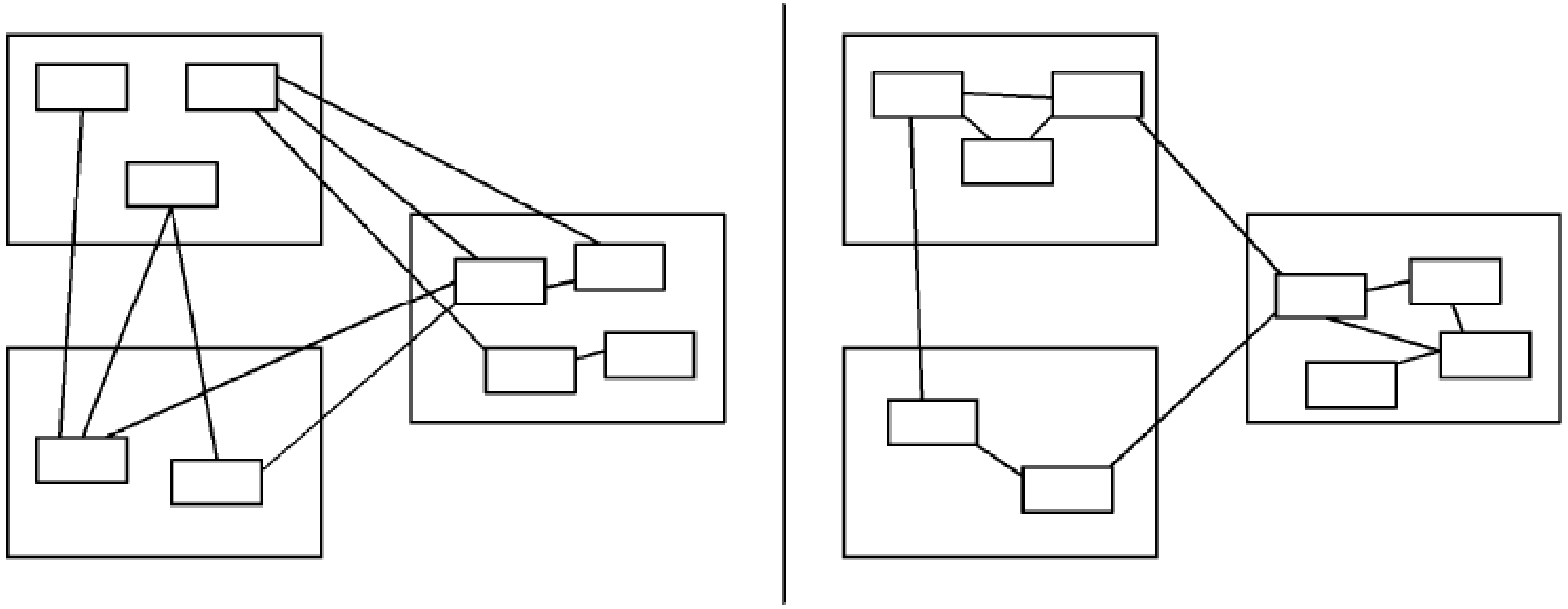

u:User
dc:DialogueControl
cp:CustomerProcessing
cm:CustomerMgr
callCentreMgr:CallCentreMgr
1:customerDetails
1.1:newCustomer
1.1.1:createCustomerRecord
1.1.1.1:createContactRecord
<<return>> 1.1.1.2:createContactRecord
<<return>> 1.1.2:createCustomerRecord
<<return>> 2:newCustomer
<<return>> 2:customerDetails

And here?

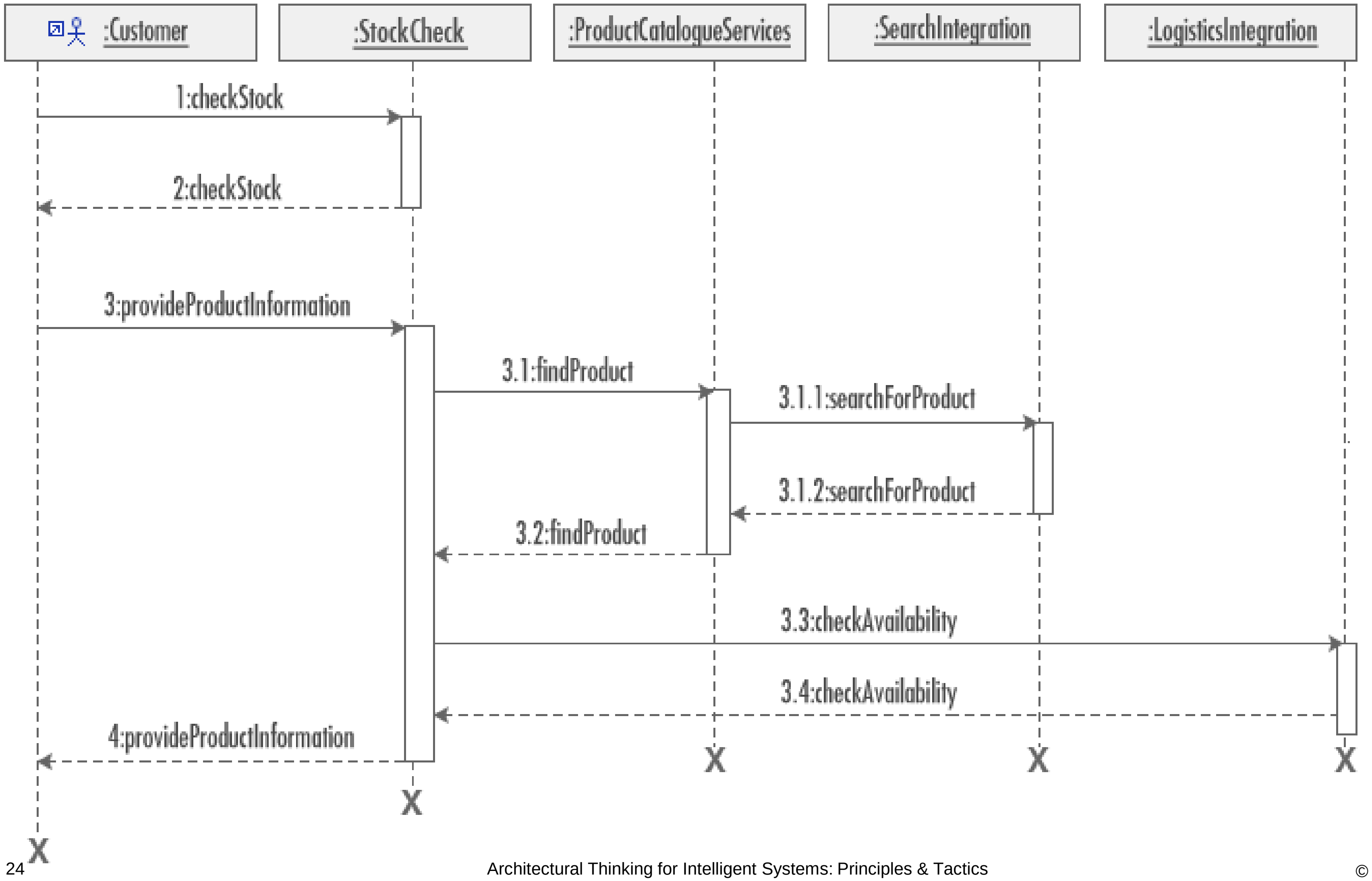


Quelle: IBM

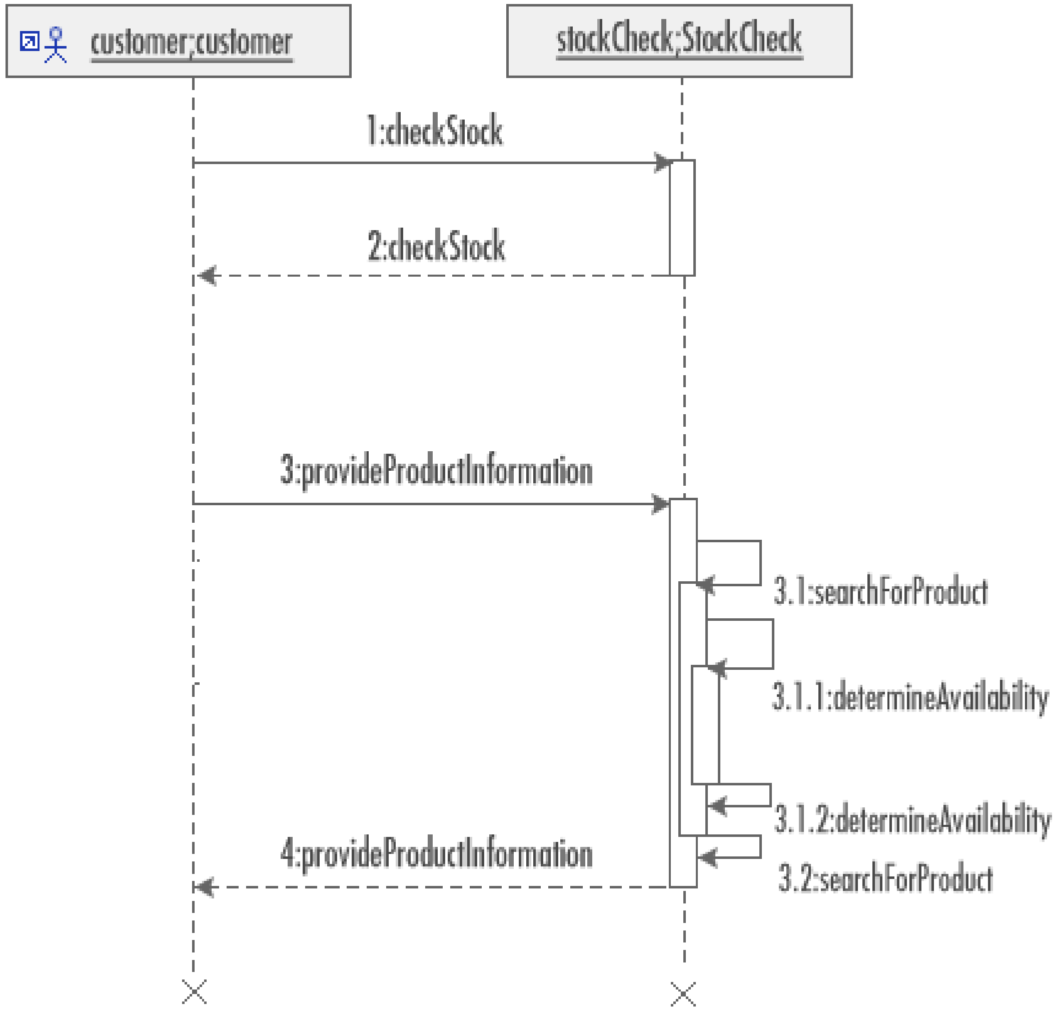
How do you assess Coupling and Cohesion in both Systems?



Cohesion of the StockCheck component?

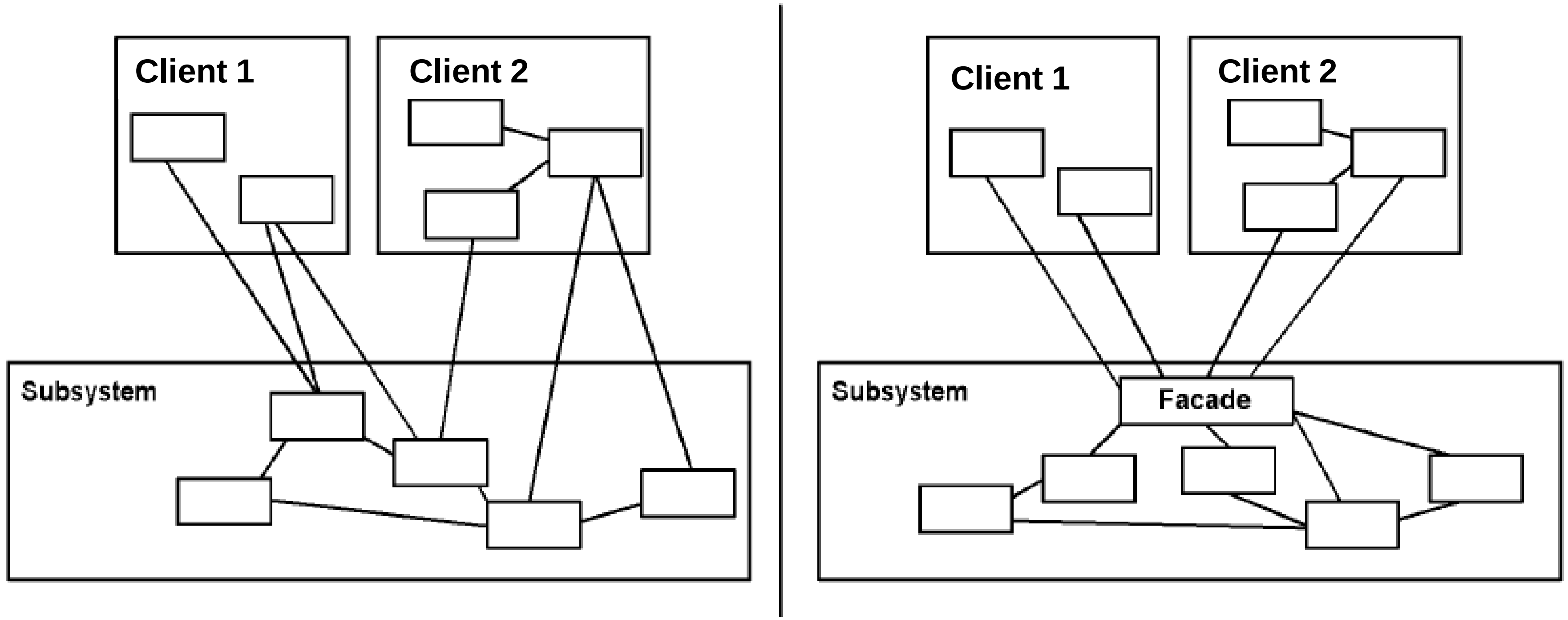


Cohesion in this Version?


customer:customer
stockCheck:StockCheck
1:checkStock
2:checkStock
3:provideProductInformation
3.1:searchForProduct
3.1.1:determineAvailability
3.1.2:determineAvailability
3.2:searchForProduct
4:provideProductInformation

Information Hiding via Facade

- Advantages and disadvantages of both systems?
- Which other principle(s) is/are recognizable here?



Interface Abstraction

- Find a good balance between general and specific interfaces
- Segregation of interfaces
  - No client should be forced to depend on methods it does not use
  - Split generic interfaces into more specific ones such that clients only need to know about methods they need
  - Keep a system decoupled, achieve loose coupling
- Design by Contract
  - Specify pre-/postconditions, invariants of an interface
  - Currently: Apache Assertions, Google Guava preconditions

Open and Closed Components

- Open for change
- Closed for access to internal details by other components
- Achieve openess through design for change
- Achieve closedness through interface abstraction and information hiding

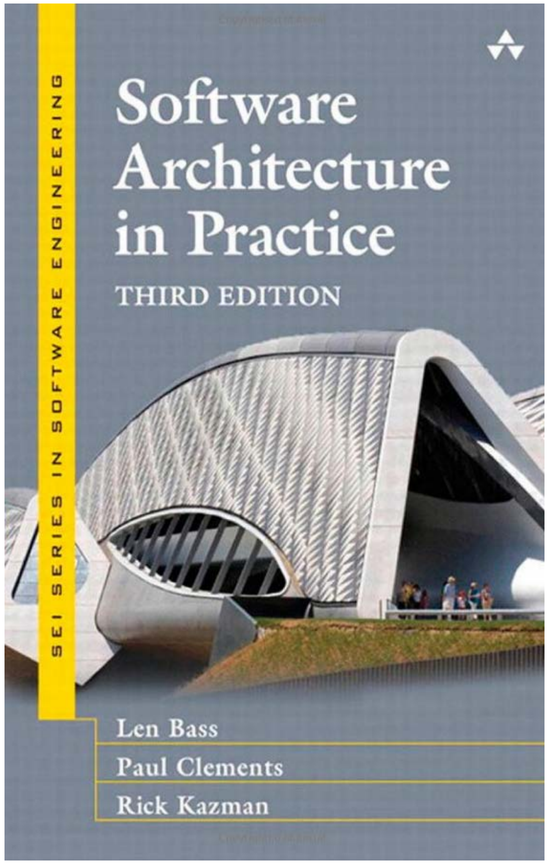
Components can be Coupled through …

- Calls
- Creation/Instantiation
- Data dependencies
- Hardware or runtime environments
- Temporal depencies

Tactics

"There are many ways to do design badly, and just a few ways to do it well."

Bass, Clements, Kazman
Software Architecture in Practice


SEI SERIES IN SOFTWARE ENGINEERING
Software Architecture in Practice
THIRD EDITION
Len Bass
Paul Clements
Rick Kazman

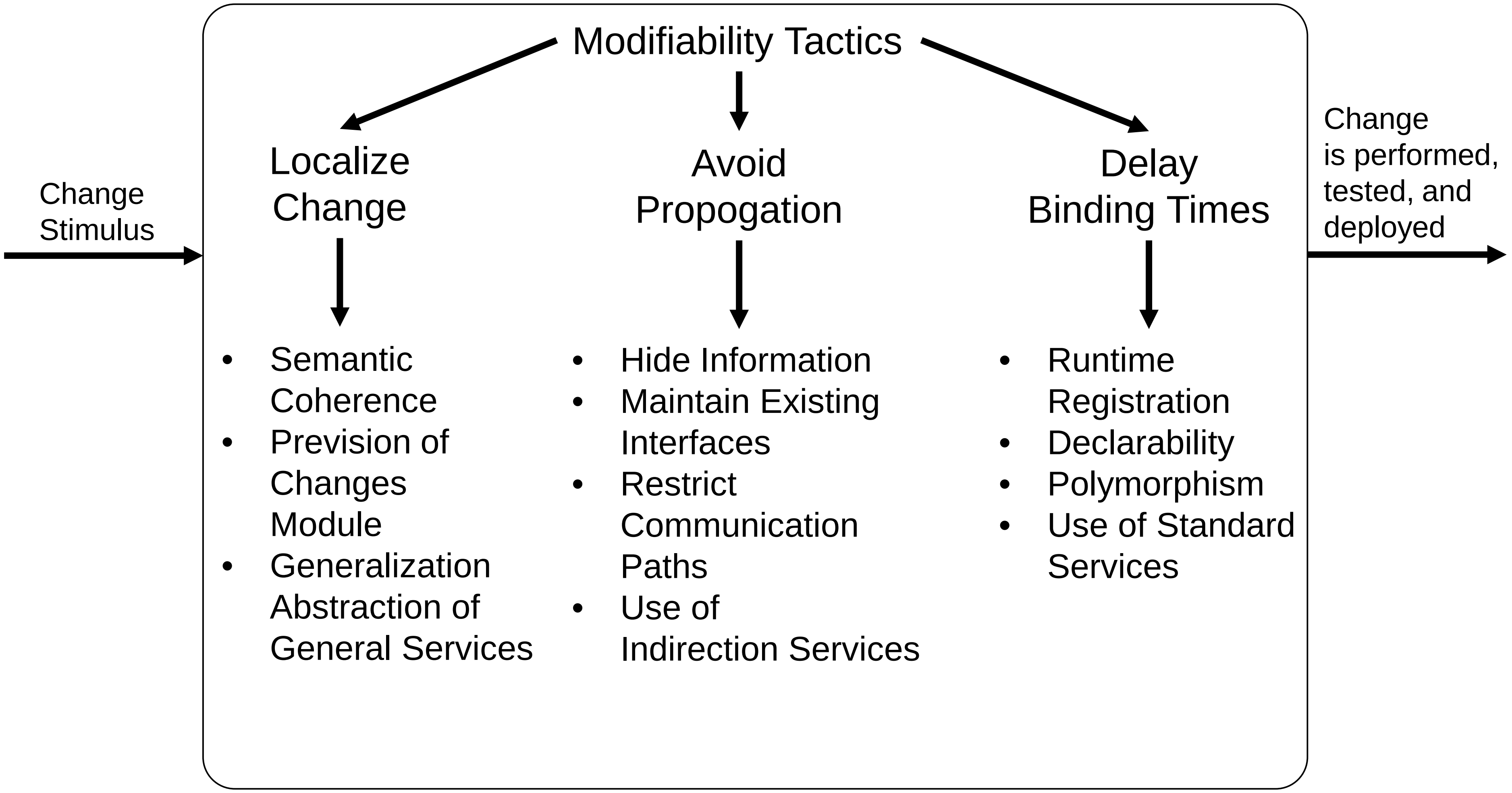
Tactics

- **A tactic is a design decision that influences the realization of the response of a quality attribute scenario**

- Specific technical solutions that help to achieve desired system qualities
  – E.g. Undo Command für Usability

- Determine the response of the system to react to a stimulus
  – Each tactic uses <u>one</u> specific structure or mechanism
  – Ignores trade-offs & compromises

Example



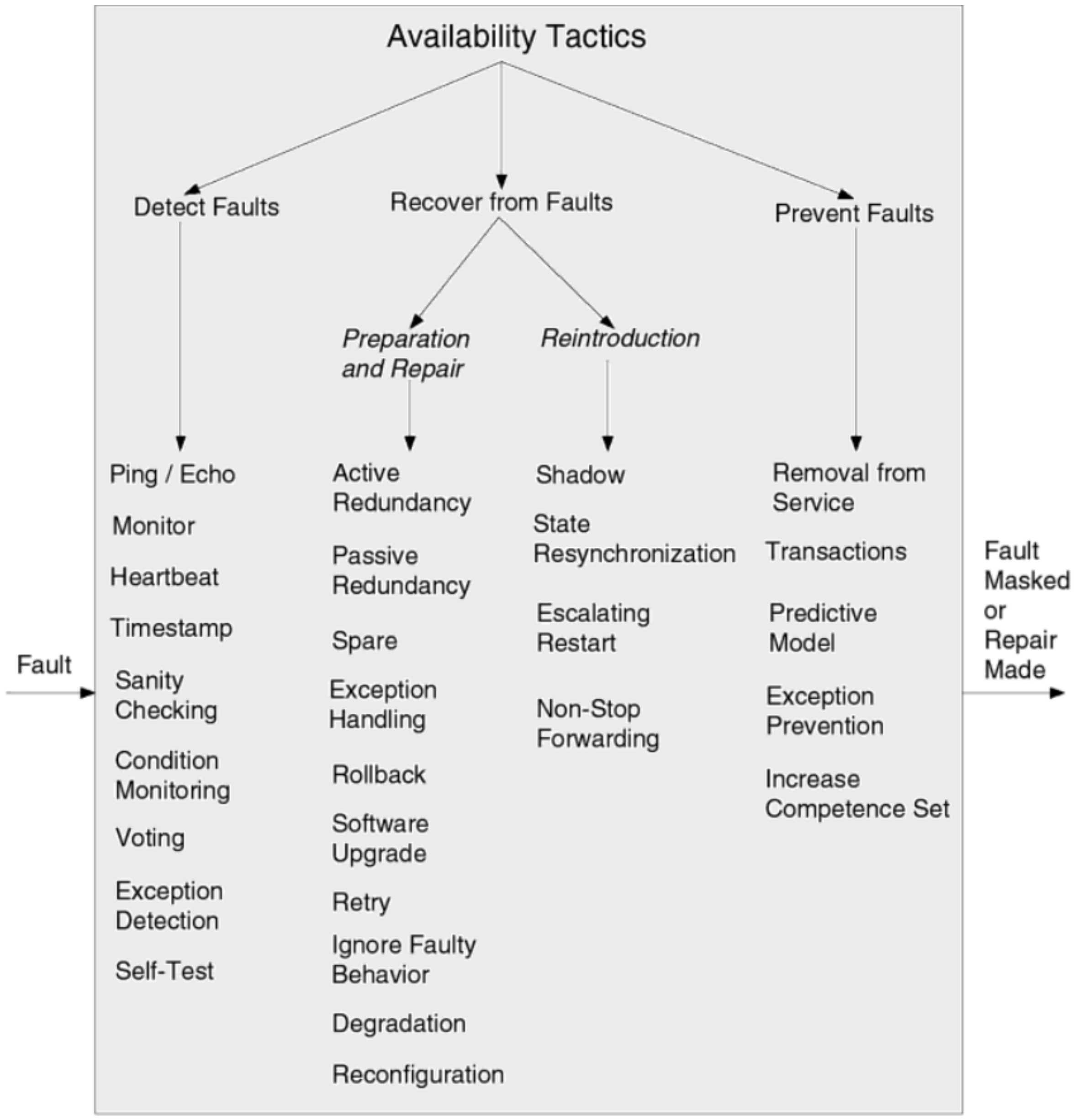
- Semantic Coherence
- Prevision of Changes Module
- Generalization Abstraction of General Services

- Hide Information
- Maintain Existing Interfaces
- Restrict Communication Paths
- Use of Indirection Services

- Runtime Registration
- Declarability
- Polymorphism
- Use of Standard Services

Taktics for Availability

- Failure: System no longer provides a service or does not provide it as specified and expected
  - the system fails and the failure can be perceived by the actors acting in/with the system
- Fault: A defect with the potential to trigger a failure
- Availability tactics focus on building systems that can withstand faults or intercept faults in such a way that they do not become failures
  - minimal tactics: limit effects of faults and enable repair


Availability Tactics
Detect Faults
Recover from Faults
Prevent Faults
Preparation and Repair
Reintroduction
Ping / Echo
Monitor
Heartbeat
Timestamp
Sanity Checking
Condition Monitoring
Voting
Exception Detection
Self-Test
Active Redundancy
Passive Redundancy
Spare
Exception Handling
Rollback
Software Upgrade
Retry
Ignore Faulty Behavior
Degradation
Reconfiguration
Shadow
State Resynchronization
Escalating Restart
Non-Stop Forwarding
Removal from Service
Transactions
Predictive Model
Exception Prevention
Increase Competence Set
Fault
Fault Masked or Repair Made

Make Decisions Specific wrt. Availability

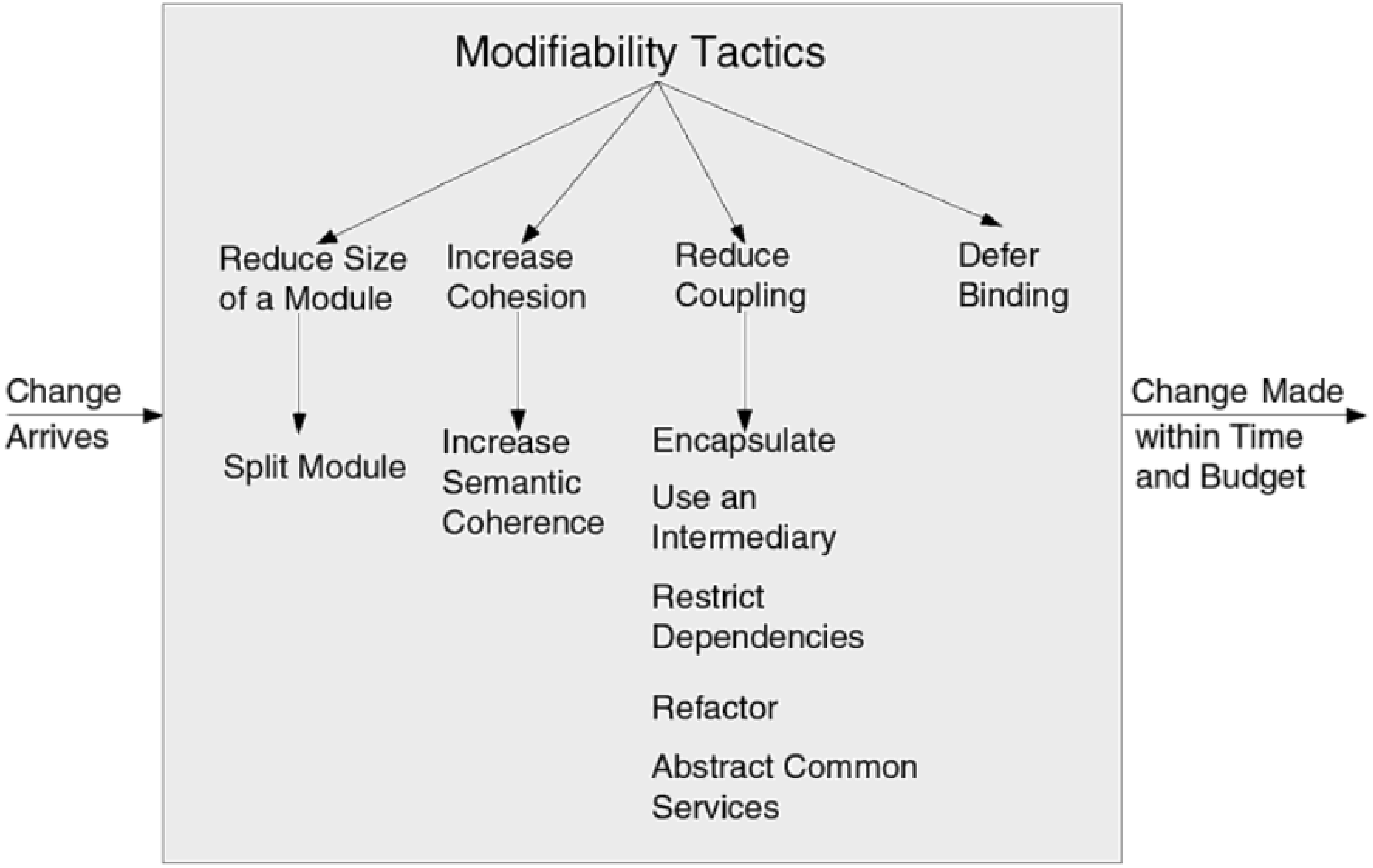
1. **Assignment of Responsibilities**
   - What must be highly available?
   - Are there any responsibilities in the system that can be used to determine faults and failures?
     - logging, notification, disabling fault causing events, be temporarily unavailable, fix/mask the fault/failure, operate in degraded mode
2. **Coordination Model**
   - Can coordination mechanisms detect availability problems (e.g. guaranteed delivery of messages?)
   - Are system parts exchangeable?
   - Does the coordination work under limited operation?

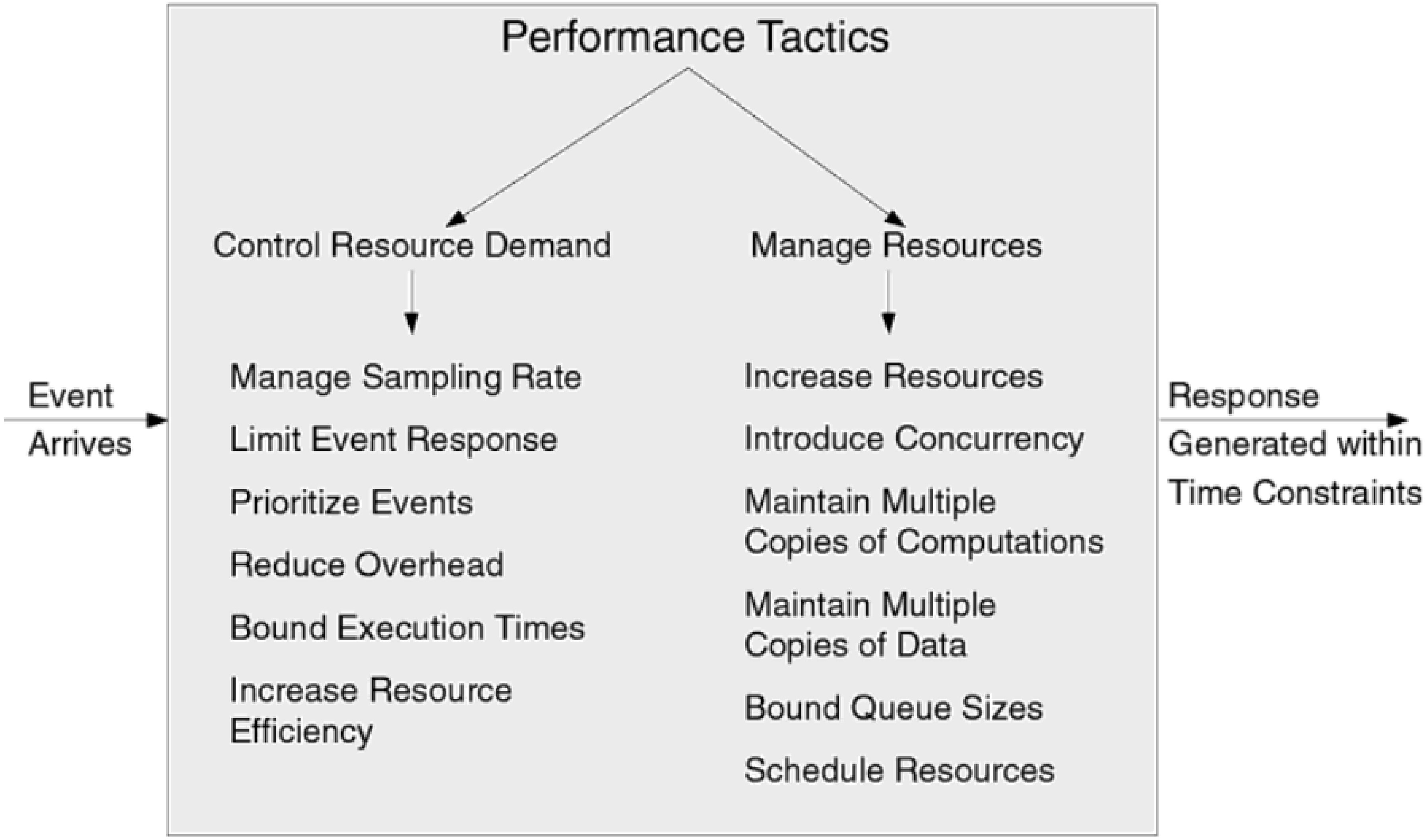
Make Decisions Specific wrt. Availability

3. **Data Model**
   - Which data sources/data operations can cause Faults/Failure?
   - Ensure that these sources/operations can be disabled, temporarily unavailable, fixed/masked
     - For example, cache write requests when a server is down and write later when server is up again
4. **Resource Management**
   - Which critical resources must function in limited operation?
5. **Mapping**
   - Is mapping between vulnerable elements flexible enough to allow for recovery?

Make Decisions Specific wrt. Availability

6. **Binding Time**
   - Is late binding used? What happens if involved components are affected by faults?
     - For example, how long can the response of a process be delayed until a fault must be anticipated?
7. **Choice of Technology**
   - What is available for logging, recovery, …
   - From which errors can the technology recover.
   - What faults can a technology bring into the system?


Modifiability Tactics
Reduce Size of a Module
Increase Cohesion
Reduce Coupling
Defer Binding
Change Arrives
Split Module
Increase Semantic Coherence
Encapsulate
Use an Intermediary
Restrict Dependencies
Refactor
Abstract Common Services
Change Made within Time and Budget


Performance Tactics
Event Arrives
Control Resource Demand
Manage Resources
Manage Sampling Rate
Limit Event Response
Prioritize Events
Reduce Overhead
Bound Execution Times
Increase Resource Efficiency
Increase Resources
Introduce Concurrency
Maintain Multiple Copies of Computations
Maintain Multiple Copies of Data
Bound Queue Sizes
Schedule Resources
Response Generated within Time Constraints

Recommendations for the Design of Component Structures

- Component functionality is well-defined and implements information hiding and separation of concerns
- DOMAIN-DRIVEN DESIGN – ENTITIES, LIFECYCLES, SERVICES
- Interfaces are well defined and “hide” change
  - Development teams can implement components independently of each other based on interface definitions
- Tactics and patterns are used
- Architecture must never depend on a specific product/tool

Recommendations for the Design of Component Structures

- Data producing and data consuming components are clearly separated from each other
- Relationship between development components and runtime components are understood (not nec. 1-1)
- Processes should be easily modifiable, if necessary at runtime
- Only a few types of interaction should be present in the system
- The same type of interaction should be implemented in the same way
- Resource conflicts are recognized and their resolution is clearly defined

Architecture Definition II

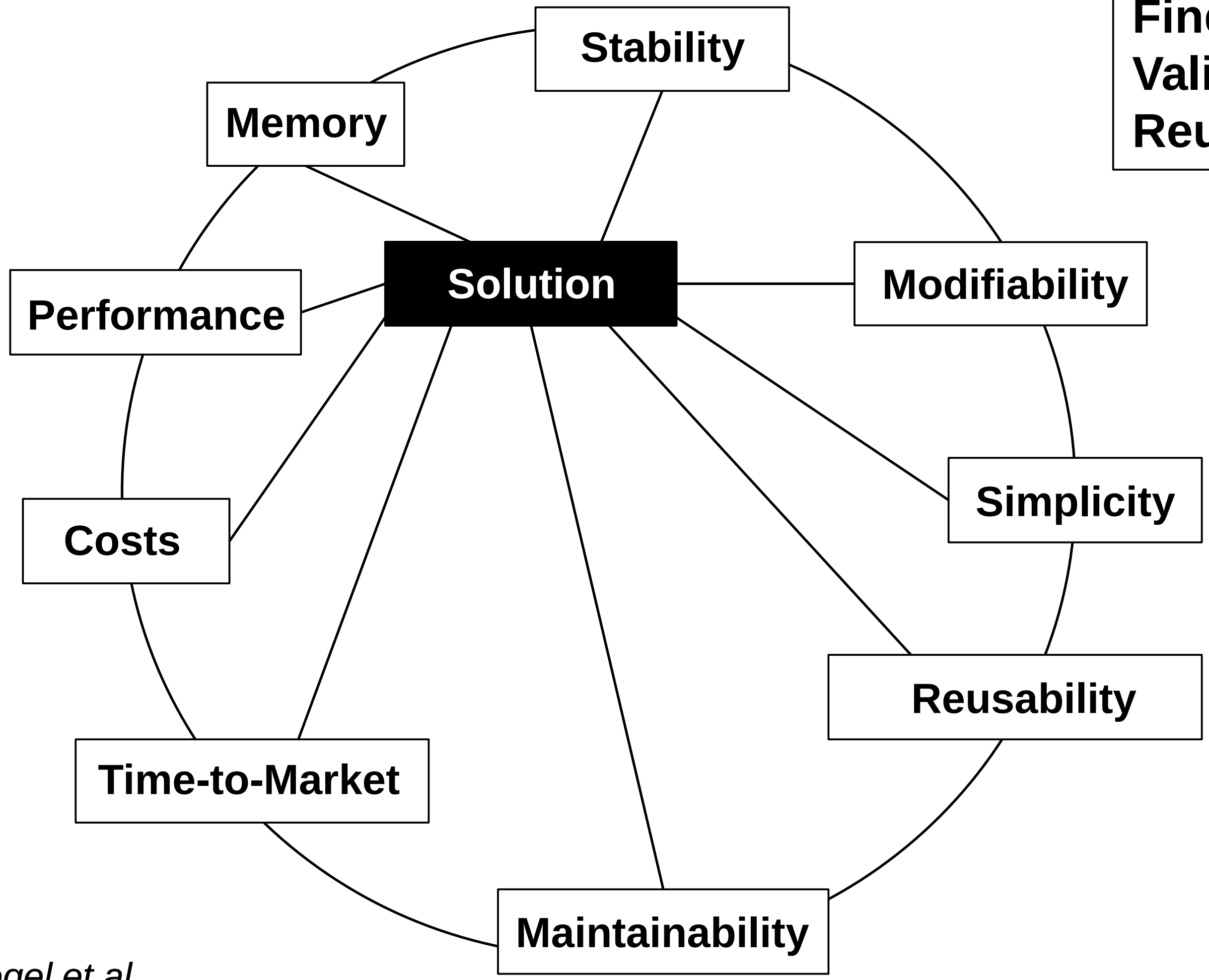
A software system's architecture is the set of principal <u>design decisions made</u> about the system.

Taylor, Medvidovic, Dashofy
Software Architecture - Foundations, Theory and Practice
Wiley 2010

For further definitions see
http://www.sei.cmu.edu/architecture/start/community.cfm

➢ Architecture means making decisions

Quality Attributes as Forces on Solution

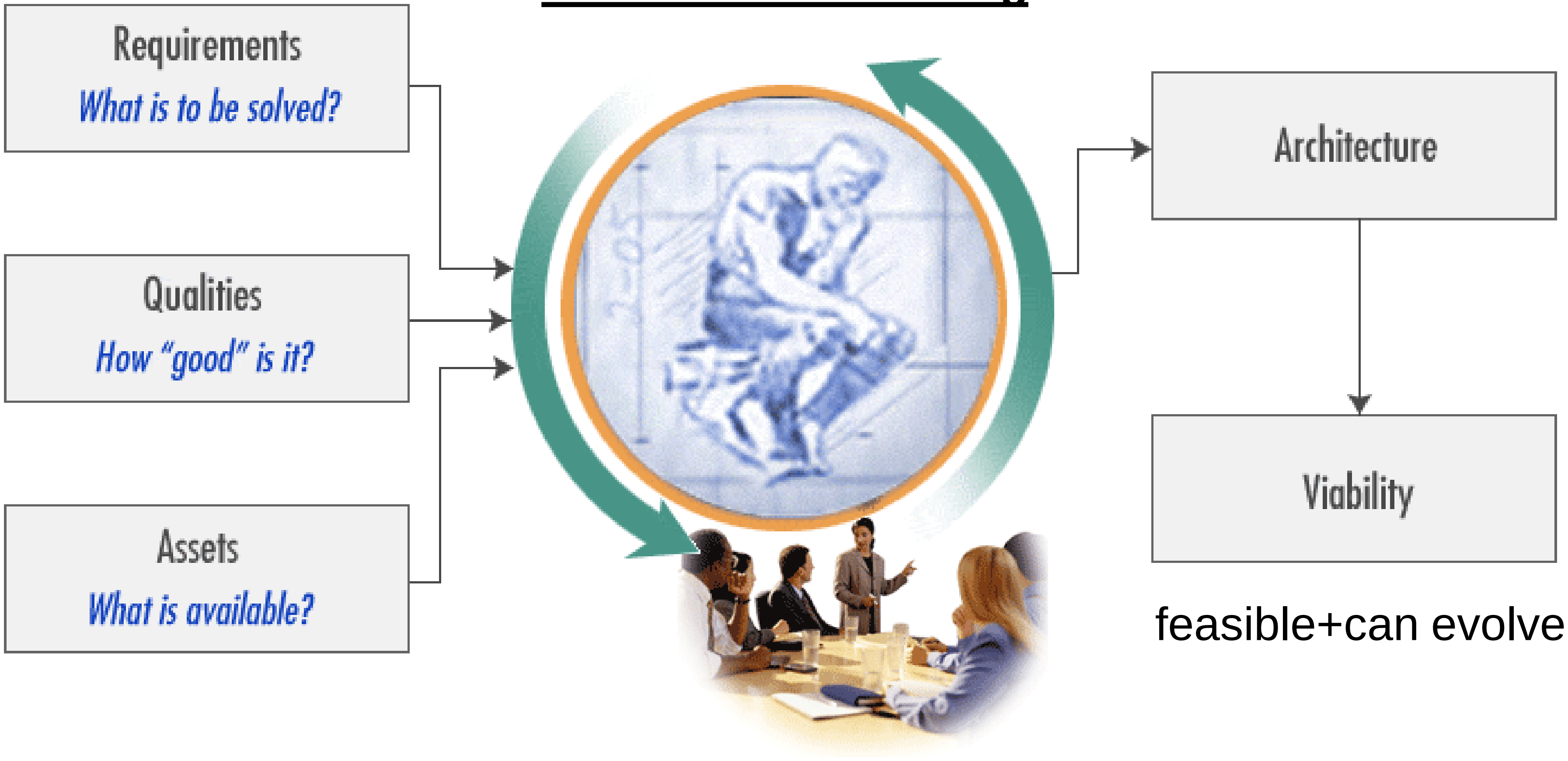


Understand forces
Find compromises
Validate early
Reuse proven solution ideas

Vogel et al.

Architectural Thinking

Input **Process of architectural Thinking** **Output**



feasible+can evolve

Source: IBM Architectural Thinking

Conway's Law

Conway's law is an adage named after computer programmer Melvin Conway, who introduced the idea in 1968; it was first dubbed Conway's law by participants at the 1968 *National Symposium on Modular Programming*.

It states that organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations
—M. Conway

Conway, Melvin E. (April 1968)
"How do Committees Invent?", *Datamation* **14** (5): 28–31

Summary

- Principles and tactics as a basis to find a system structure implementing requirements
- Principles can support each other, must be made specific for a given architecture
- Each tactic only supports a single quality attribute
- Work iteratively based on a close interaction with the development team, actively support agile development
- Architectural decisions are the responsibility of the architect
- Base decision on a prioritized list of quality attributes
- Constantly create and revise views for stakeholders
- Evaluate architectures early and repeat evaluation

Working Questions

1. How do you define architecture from the point of view of the work process and the architect's work products?
2. Explain how quality attributes act as forces in solution finding.
3. What is an architectural principle? Give examples of principles for a good architectural design.
4. Explain the relationship between 2 given architectural principles.
5. What are tactics?
6. What do tactics NOT deal with?
7. Explain examples of tactics for a quality attribute.