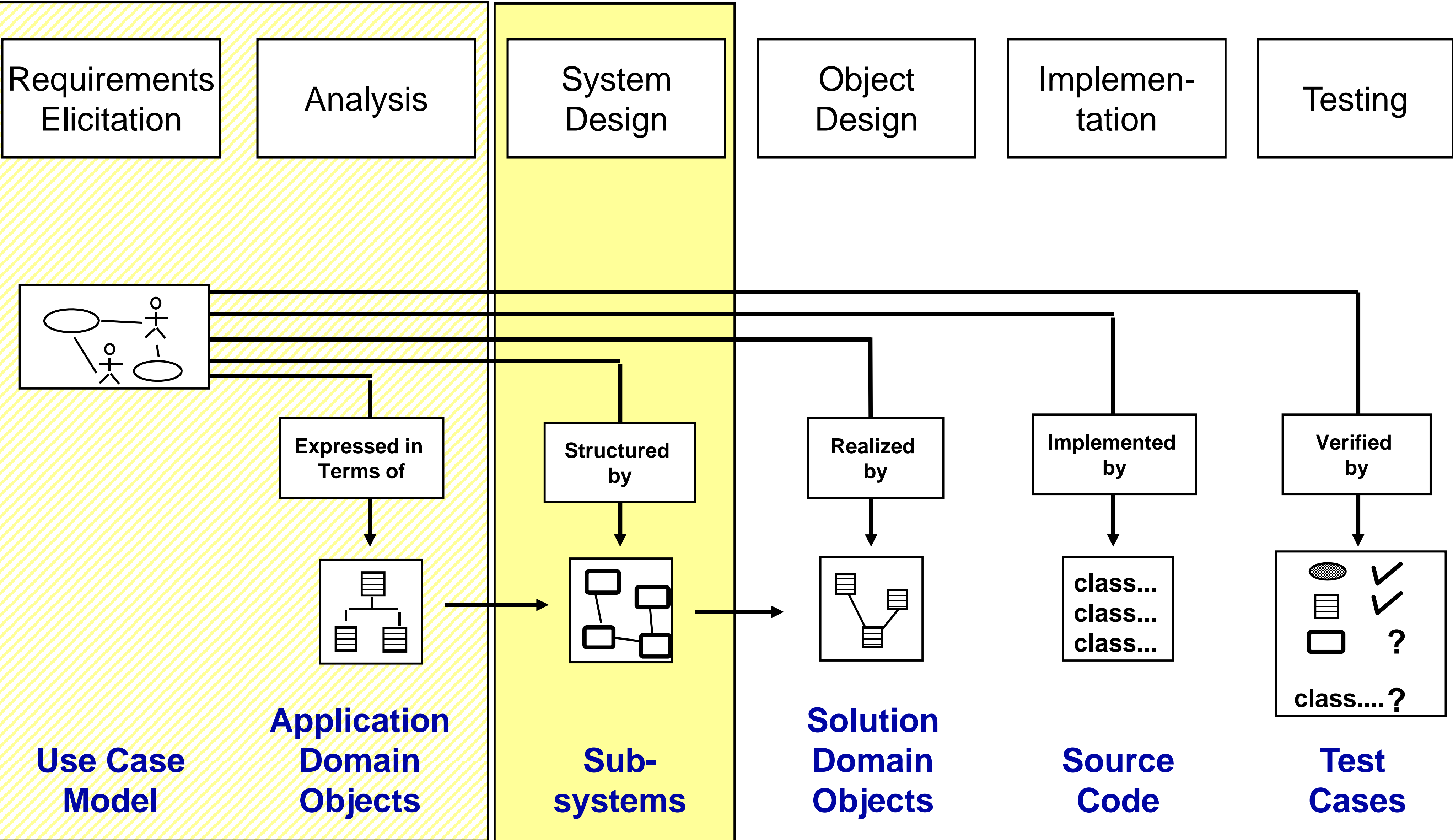
Chapter 9, System Design – Addressing Design Goals

Object-**O**riented
Software **C**onstruction

Armin B. Cremers, Tobias Rho, Daniel Speicher &
Holger Mügge
(based on Bruegge & Dutoit)

Software Lifecycle Activities

...and their models



Overview
Four Sessions on System Design

b-it

- System Design I – Software Architecture
  - System Design Concepts
  - Subsystem Decomposition
- System Design II – Architecture Organization
  - Hardware/Software Mapping
  - Persistent Data Management
  - Global Resource Handling and Access Control
  - Software Control
  - Boundary Conditions

Today

- System Design III - Addressing Design Goals
  - Tactics for Performance, Security, Availability, Modifiability

Non-functional requirements (NFR)

b-it

- Non-functional requirements define the overall qualities or attributes of the resulting system
- Non-functional requirements place restrictions on the product being developed, the development process, and specify external constraints that the product must meet.
- Some non-functional requirements can be rephrased as functional requirements
- Project management issues (costs, time, schedule) are often considered as non-functional requirements as well

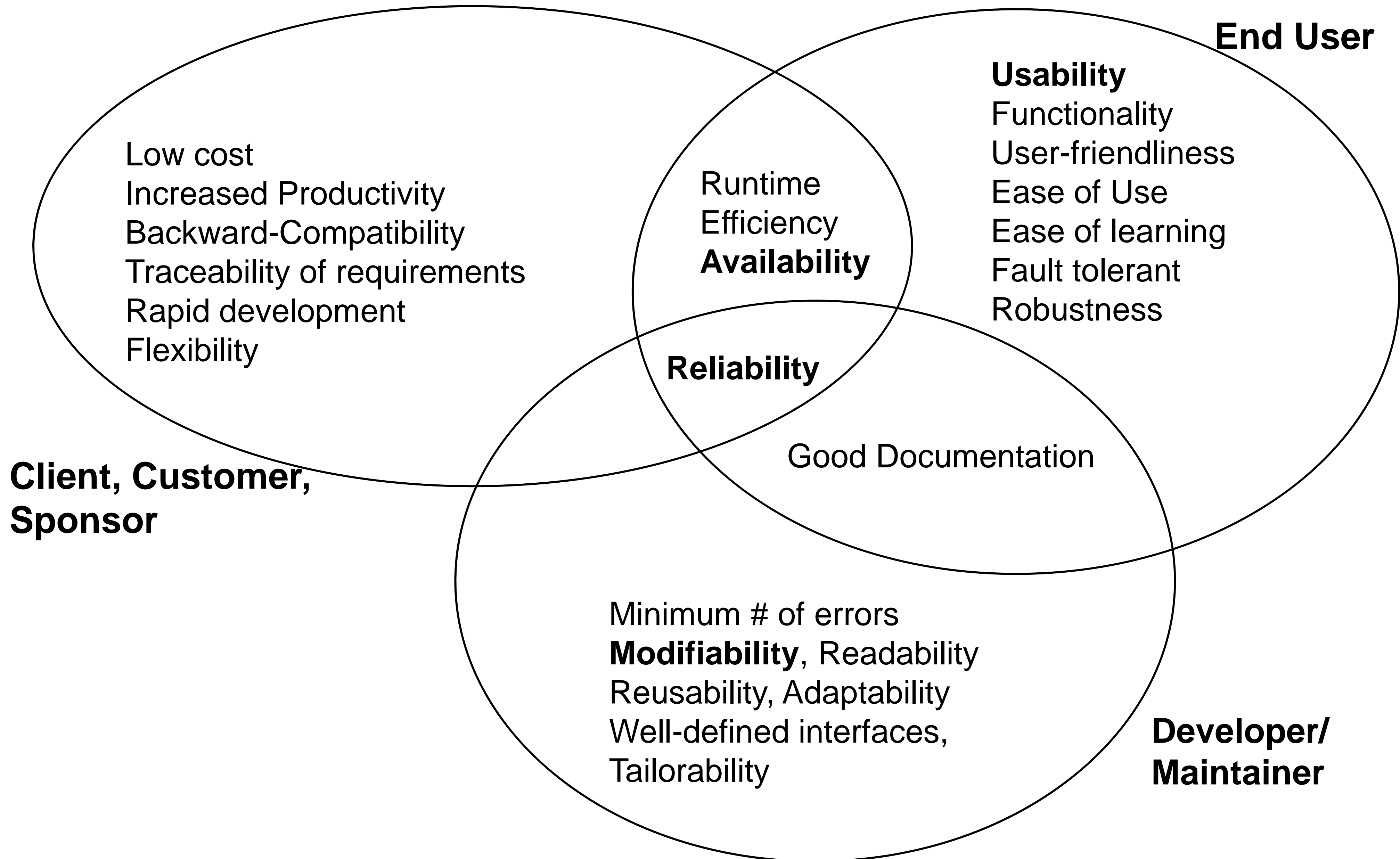
  → not handled in this session

List of Non-Functional Requirements

b-it

- Reliability
- **Modifiability**
- Maintainability
- Understandability
- Adaptability
- **Availability**
- Efficiency
- Portability
- Traceability of requirements
- Fault tolerance
- **Performance**
- Cost-effectiveness
- Robustness
- High-performance
- Good documentation
- Well-defined interfaces
- **Usability**
- Reuse of components
- Rapid development
- Minimum # of errors
- Readability
- Ease of learning
- Ease of remembering
- Ease of use
- Increased productivity
- Low-cost
- Flexibility
- **Tailorability**

Relationship Between Roles & NFRs



Insertion of NFRs in Use Cases
Textual Representation

b-it

- Normally informal and high-level representation during the requirements phases:

  Example (Performance):
  The response time shall be fast enough to avoid interrupting the user's flow of thought

- A bit more accuracy is useful:

  Example (Performance):
  Any interface between a user and the automated system shall have a maximum response time of 2 seconds

- Quantitative and accurate non-functional requirements can be verified and tested during:
  - Implementation (unit testing)
  - Prototyping (during requirements)
  - Testing (acceptance)

From non-functional requirements towards design goals

b-it

- Interpreting nonfunctional requirements as concrete design goals for the chosen software architecture
- Turning high-level description towards more meaningful design statements

- Achievement of design goals relies on fundamental design decisions or so-called **tactics**

- Much more technical background and experience is required

- Next: Presentation of tactics:
  - Performance, Availability, Modifiability, Usability

Availability Tactics Properties

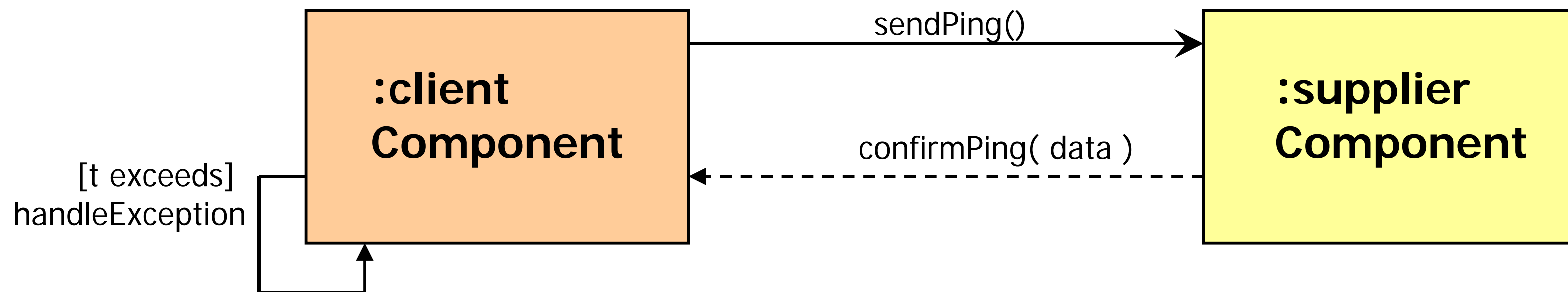
b-it

- Availability quantifies
  - the total up-time of a system as requested or expected by end-users.
  - quantifies the total allowable failure rate of a system (i.e. when it no longer delivers a service consistent with the specification)

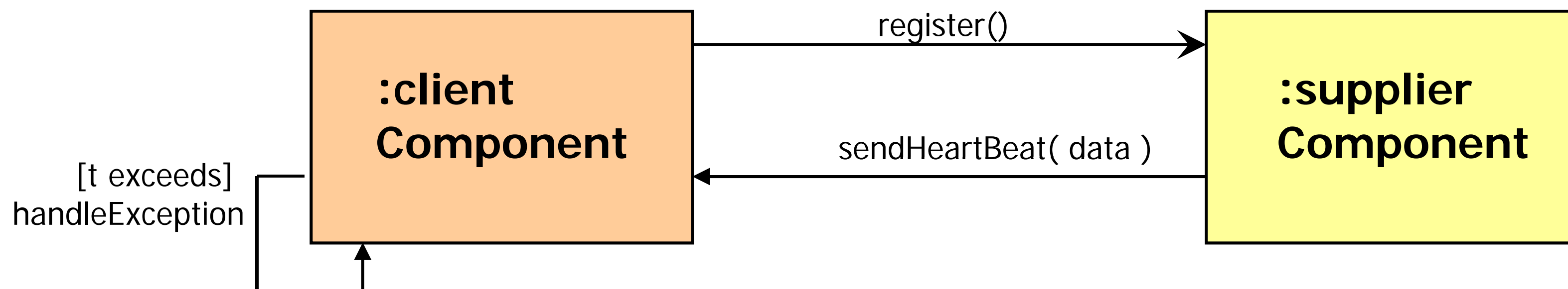
- Areas of concern:
  - Failure Detection: How are failures detected?
  - Failure Handling: How can occurred failures be handled?
  - Failure Prevention: How can failures be avoided?
  - Failure Notification: What kinds of notification are needed for delegating the event of a failure?

Availability Tactics
Fault Detection

Ping/echo



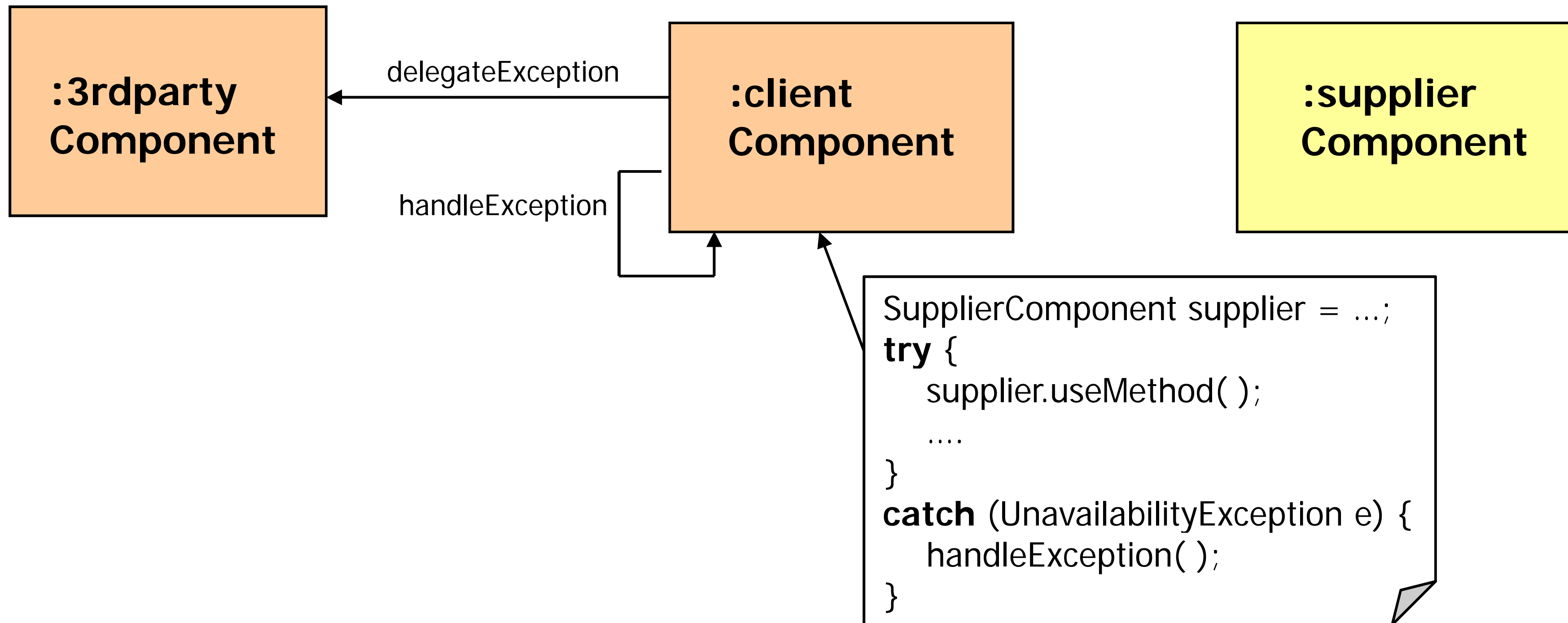
Heartbeat



- If the confirmation (heartbeat) is not received after a pre-defined time, the supplier component is said to be failed
- Local registry in the client component needed
- Usually used for distributed architectures

Availability Tactics
Fault Detection

Exception Handling



- Checked vs. unchecked Exceptions
- Chain of Responsibility Design Pattern (Exception Delegation)
- Usually used for local architectures, but also for distributed ones (cf. Sun’s RMI technology)

Exceptions in Java – Unchecked

b-it

- Runtime Exceptions
  - the special case for exceptions (subtype RuntimeException)
  - are not declared
  - may occur in any call instead of a normal result
  - caller may or may not catch them (call in try-block)
  - if not catched the program/thread terminates and information about the exception is printed to error output stream
  - are used for unexpected exception
  - are used when the caller typically can not repair the cause

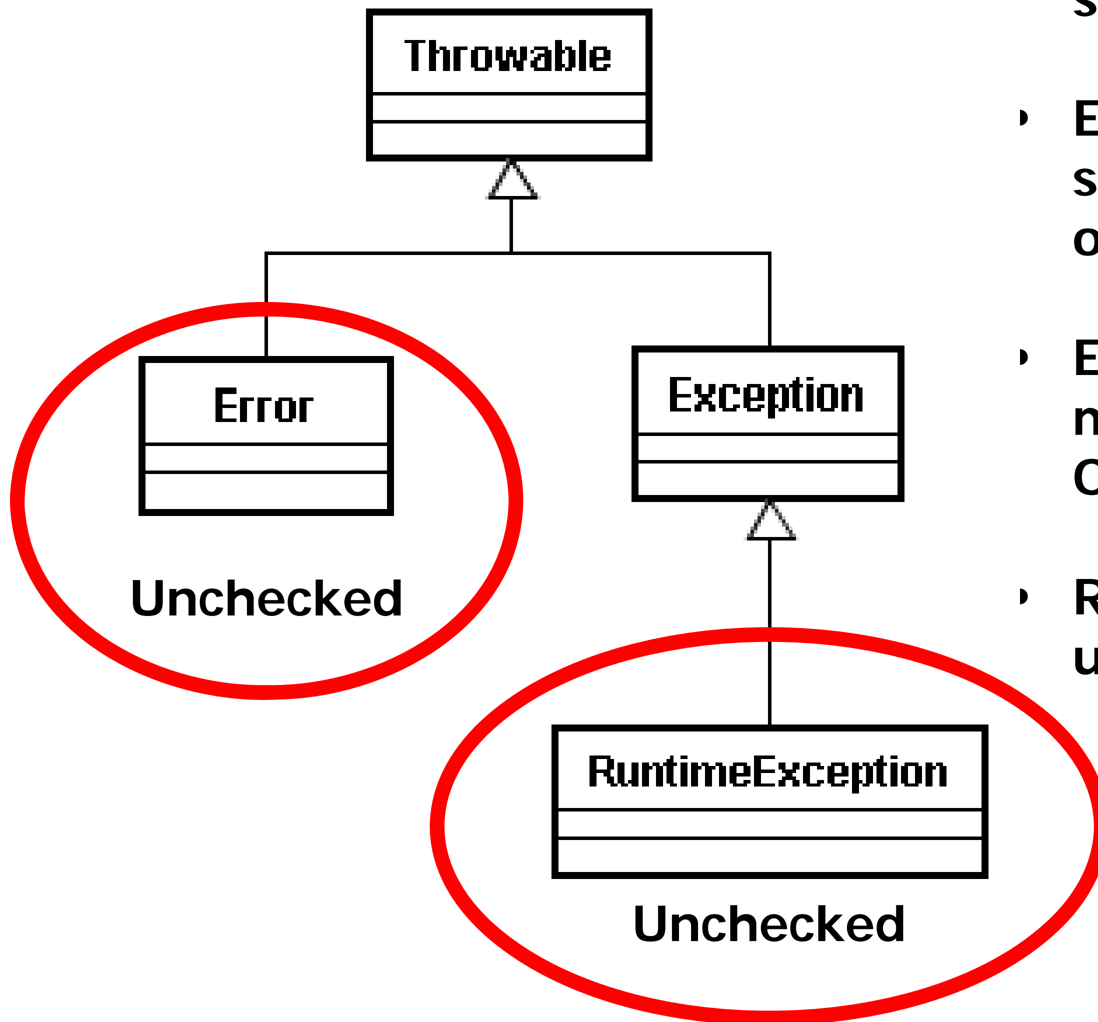
- Examples
  - ArrayIndexOutOfBounds
  - ClassCast

Exceptions in Java – Checked

b-it

- Statically checked Exceptions
  - the standard case of exceptions
  - are declared in the method signature
  - can occur only when declared => can be checked at compile time
  - caller must either catch or throw them further up the call stack
  - are used for “expected” exceptions
  - are used when the caller probably can help or react meaningful
- Example
  - ClassNotFound
  - IOException

Java Exception Class Hierarchy



- Throwable:
  superclass of all errors and exceptions
- Exceptions:
  signal abnormal conditions that can often be handled by some catcher
- Error:
  more serious problems, such as OutOfMemoryError
- RuntimeException:
  unchecked Exceptions

Using Exceptions in Java (1/2)

What about this?

```
try {
   // do something that can throw an exception.
}
catch (ClassNotFoundException cnf_exc) {
}
```

Using Exceptions in Java (2/2)

… and what about this?

```
try {
   // do something that can throw an exception.
}
catch (Exception e) {
   log(e); // logs to a persistent error log
}
```

Criteria on using Exceptions

Condition	Contingency	Fault
Is considered to be	A part of the design	A nasty surprise
Is expected to happen	Regularly but rarely	Never
Who cares about it	The upstream code that invokes the method	The people who need to fix the problem
Examples	Alternative return modes	Programming bugs, hardware malfunctions, configuration mistakes, missing files, unavailable servers
Best Mapping	A checked exception	An unchecked exception

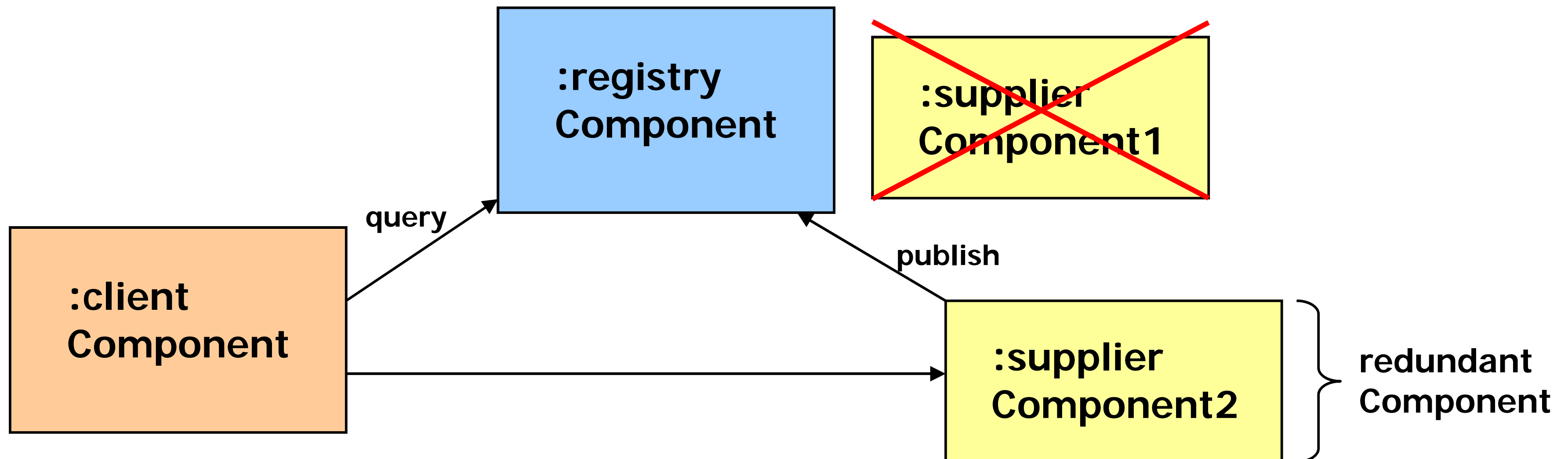
Links about Exception Handling

b-it

- Bruce Eckel: (use only unchecked exceptions!)
  - http://www.mindview.net/Etc/Discussions/CheckedExceptions
- Sun: (always use checked exceptions!)
  - java.sun.com/docs/books/tutorial/essential/exceptions/index.html
- Barry Ruzek: (differentiated view)
  - http://dev2dev.bea.com/pub/a/2006/11/effective-exceptions.html

Availability Tactics
Fault Handling

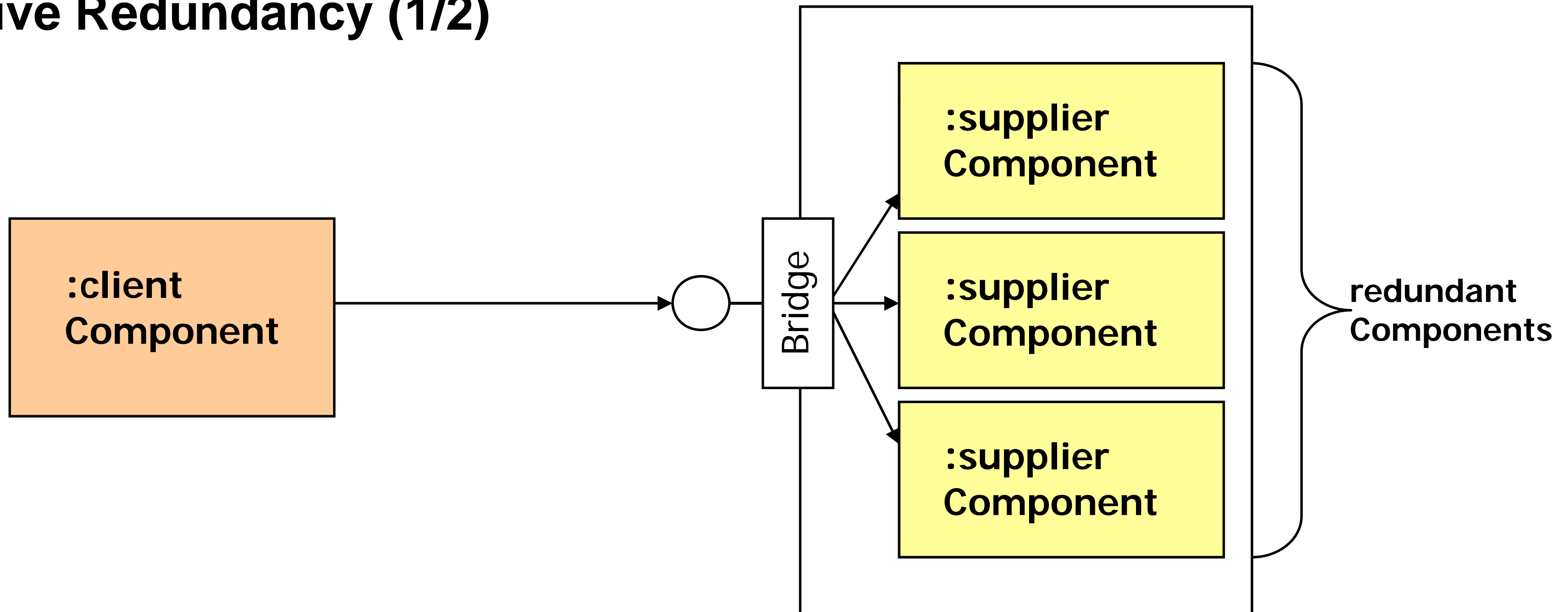
Client-sided switch to alternative component



- Upon failure, a client **locates** an alternative supplier component which has an **equivalent** service
- Often mastered in combination with a global directory (SOA)
- No coordination between supplier components, **state** of old supplier component is lost
- Scenario: Architecture with many **state-less** services (e.g. printer service, atomic transactions)

Availability Tactics
Fault Handling

Active Redundancy (1/2)

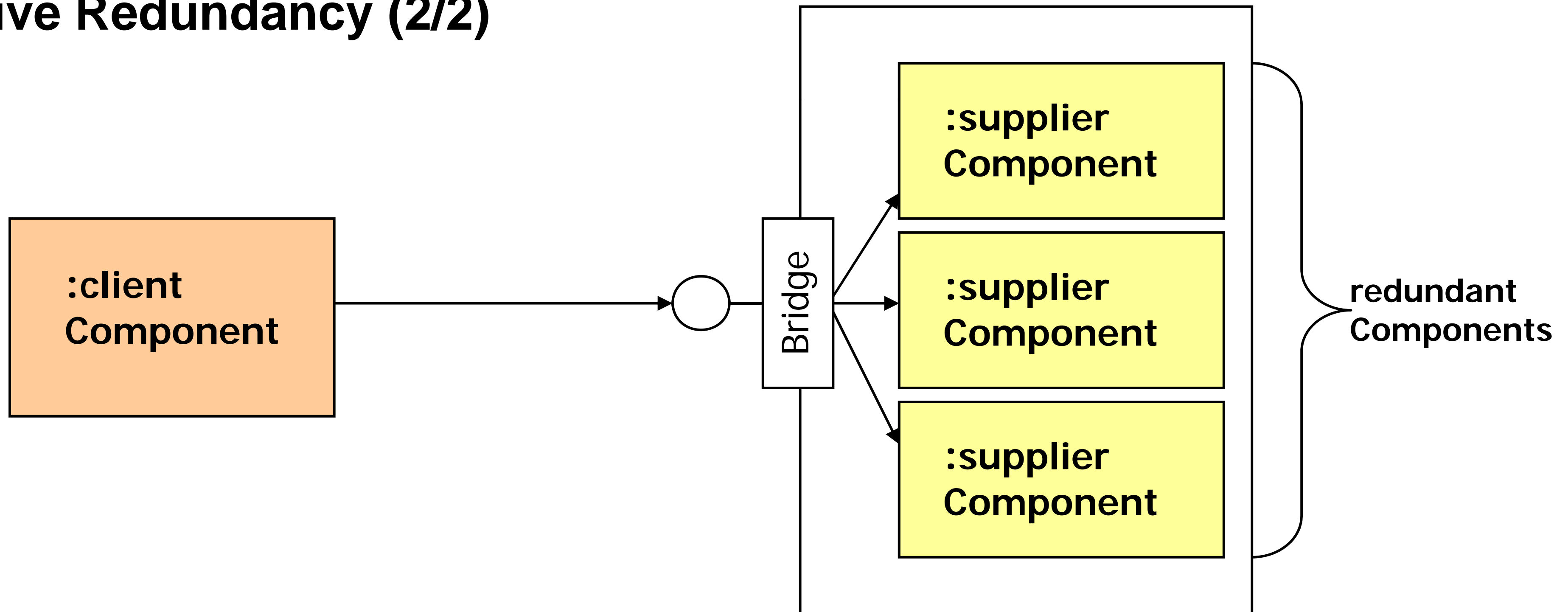


- Request to a service is delegated by “Bridge” to all redundant supplier components
- Response is produced in parallel by all redundant components, “Bridge” takes the first as the response for the client

→ **All components are in the same state**

- Fault Handling: Rely on redundant component

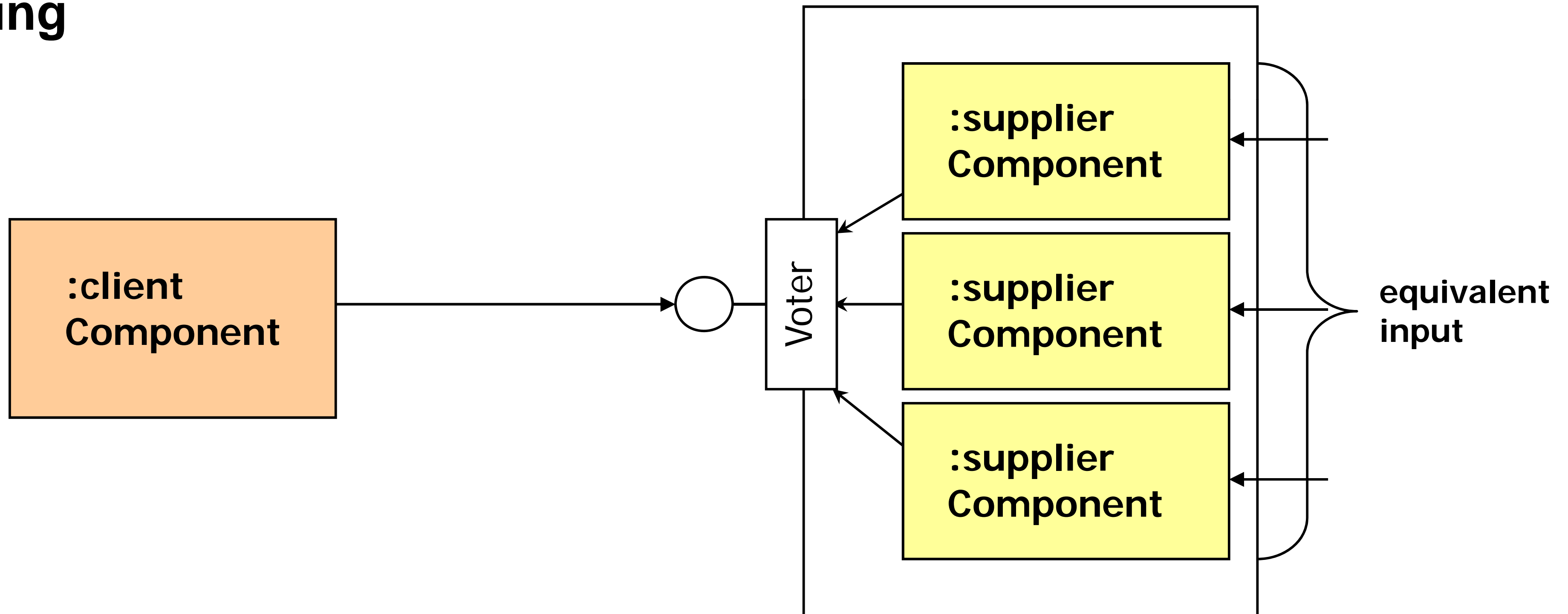
Active Redundancy (2/2)



- Bridge can be part of the client
- Scenario: Architecture with **stateful** services, with quick responses even when a failure occurs
- Drawbacks: massive resources needed, communication overhead, reliable communications paths needed

Availability Tactics
Fault Handling

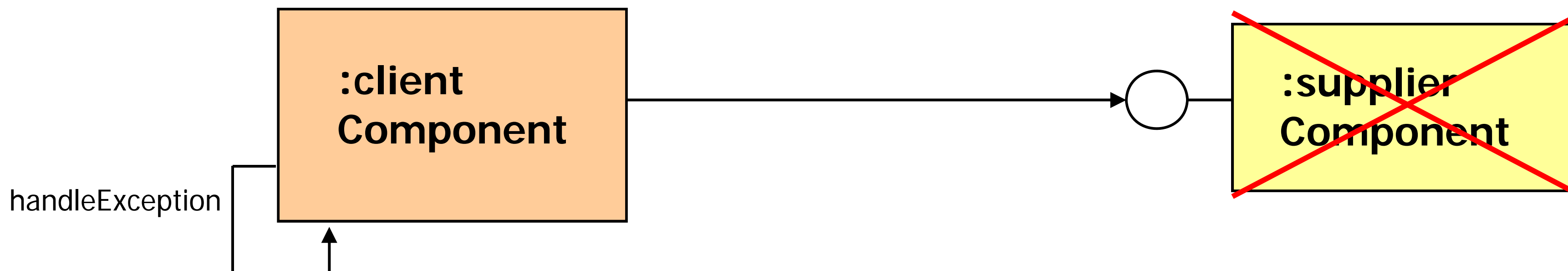
Voting



- Voter is capable of detecting faulty behavior of a single component. Upon detection, it skips the answer and waits for the next result.
- Detection of faulty algorithm and failure of components
- Scenario: control systems with sensors as inputs

Availability Tactics
Fault Handling

Handling failure of non-redundant components



- So far: all handling mechanisms assumed redundant components

→ Redundancy cannot always be guaranteed or is not desired

- Failure of non-redundant component: many possible options (*context-dependent*):
  - Integration of user to enhance decision making
  - Local replacement or adaptation of component
  - Discard of functionality
- Scenario: Architecture holding crucial, secure data as a singleton

Modifiability Tactics
Properties

- Supportability requirements are concerned with the ease of changes to the system after deployment. Classes:
- Adaptability
  - The ability to change the system to deal with additional application domain concepts (adaptivity = autonomous adaptation)
- Tailorability
  - End-Users are able to make modification to a system during runtime

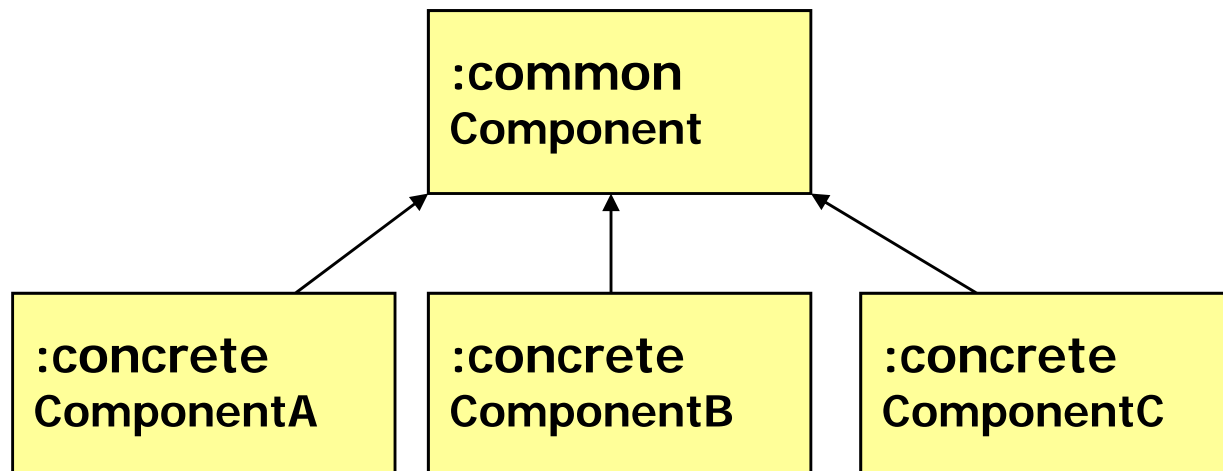
- Areas of concern:
  - Localize and anticipate changes
  - Defer design decision
  - Tailorability
  - Policy-based agreements (cf. Performance)

Modifiability Tactics
Localize Changes

Coupling – Reduction of **consequences**

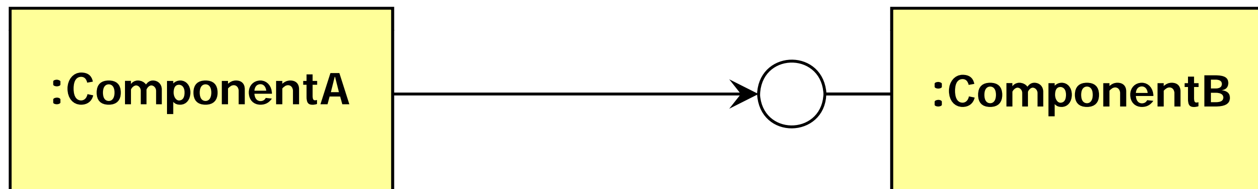
- Changes to a component should not affect directly dependent components
- Coupling metrics can be used to localize changes (cf. chapter 7)

Identify Common Components – Reduction of **costs**:



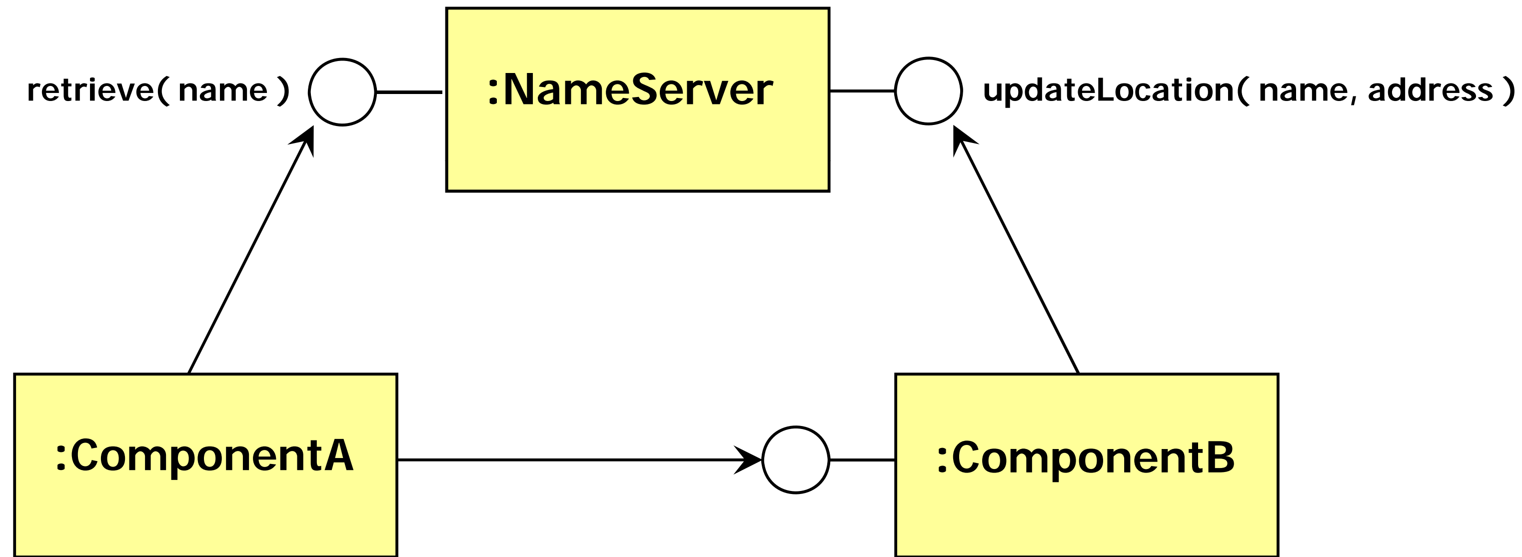
- Modification to a common component only need to be done once rather than in each concrete component, where the service of that component is used (trade-off to coupling)

Modifiability Tactics
Dependency Types



- Semantics of data and service
  - Pre- and Postcondition (cf. Object Design)
- Sequence of Data
  - Reception of data in a fixed order
- Location
  - Runtime location of B must be consistent with the assumptions of A (e.g. fixed address)
- Quality of Service

Modifiability Tactics
Location Tactics



- A name server (index) enables the location of A to be changed without affecting component B
- B must maintain its current location
- A can retrieve the current location of B with a specific key:

  retrieve ("ComponentB") = 141.26.14.5
- Scenario: distributed architectures with mobile services, unfixed addresses (DHCP)

Modifiability Tactics
Anticipate changes

b-it

- Anticipate which component should be modified due to
  - New technology
  - New implementation for an algorithm
  - New subcomponent by a new vendor
  - New error types

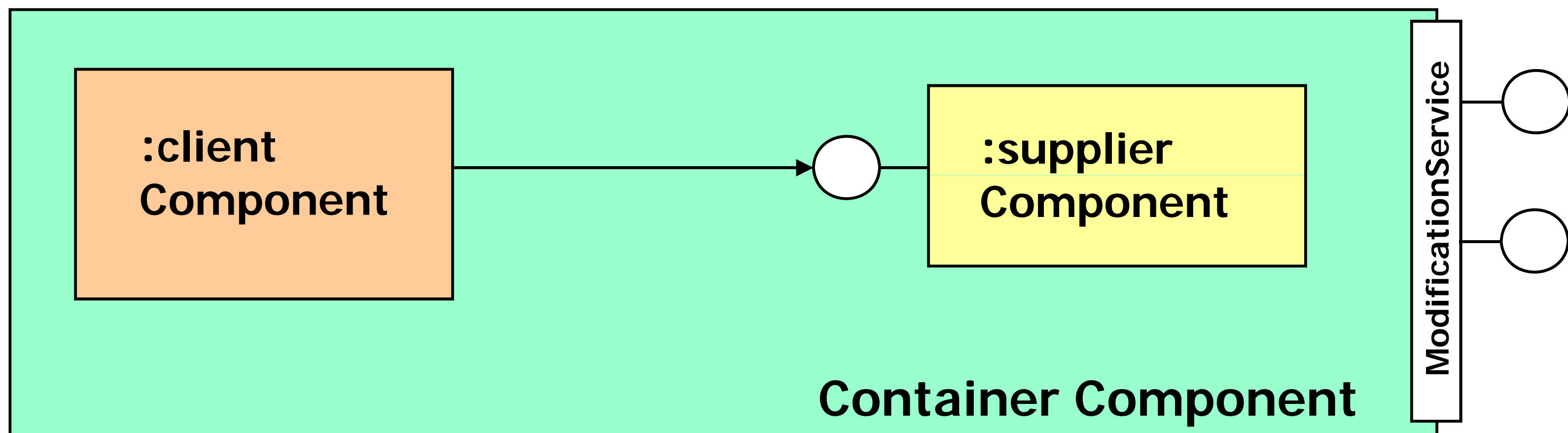
- Application of design patterns
  - Example: Strategy Pattern for inserting and using different concrete algorithms at runtime (algorithms accord to a common abstract abstraction (strategy))

Modifiability Tactics
Defer design decision

- Conventional development: all design decisions are made by developers during development
  - Result: monolithic architectures, no modification possible *after deployment*
- Defer design decisions: Shift design decisions to the deployment time of the architecture
  - Non-developers (end-users) or administrator are able to modify the architecture
  - Deployment time: start- and runtime of the software
  - Result: open, flexible architecture
- **Cost**: Additional infrastructure to support late design decision
- Easy example: Configuration files
- Most common technique: late binding (→ defer binding time)

Modifiability Tactics
Defer design decision

Runtime Registration of Component (Hot deployment)

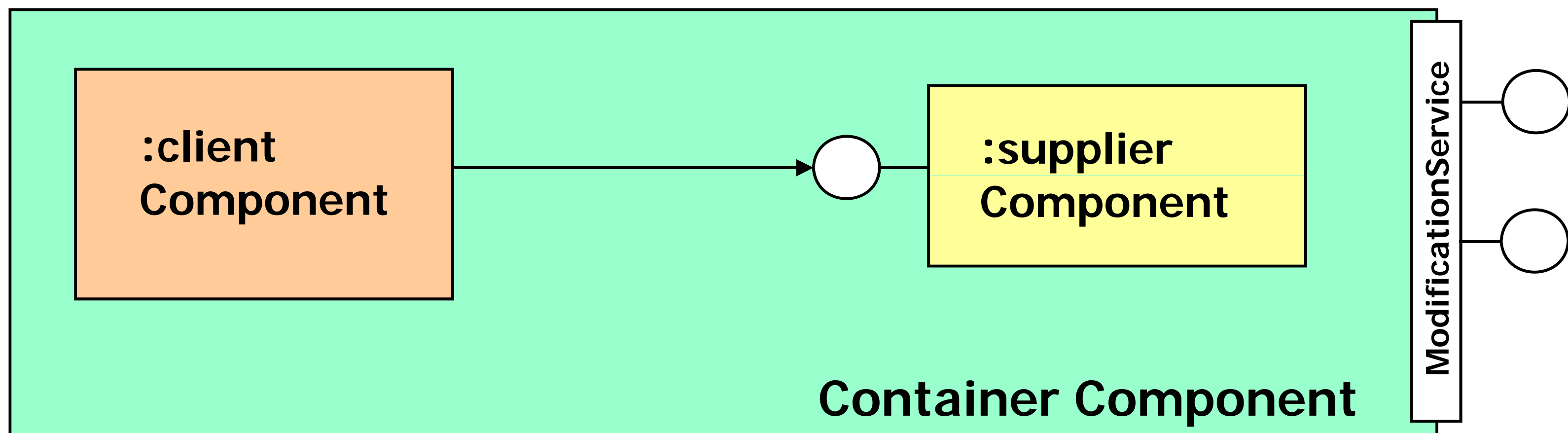


- Additional Container (Runtime Environment) component allows for:
  - Loading new components
  - Defining new bindings
  - Deleting existing components
  - Configuration of single components
  - Configuration of non-functional requirements (e.g. security)
  - Configuration of implicit services (e.g. persistency of data, transactional behavior) … tbc …

Modifiability Tactics
Defer design decision

b-it

Runtime Registration of Component (Hot deployment)

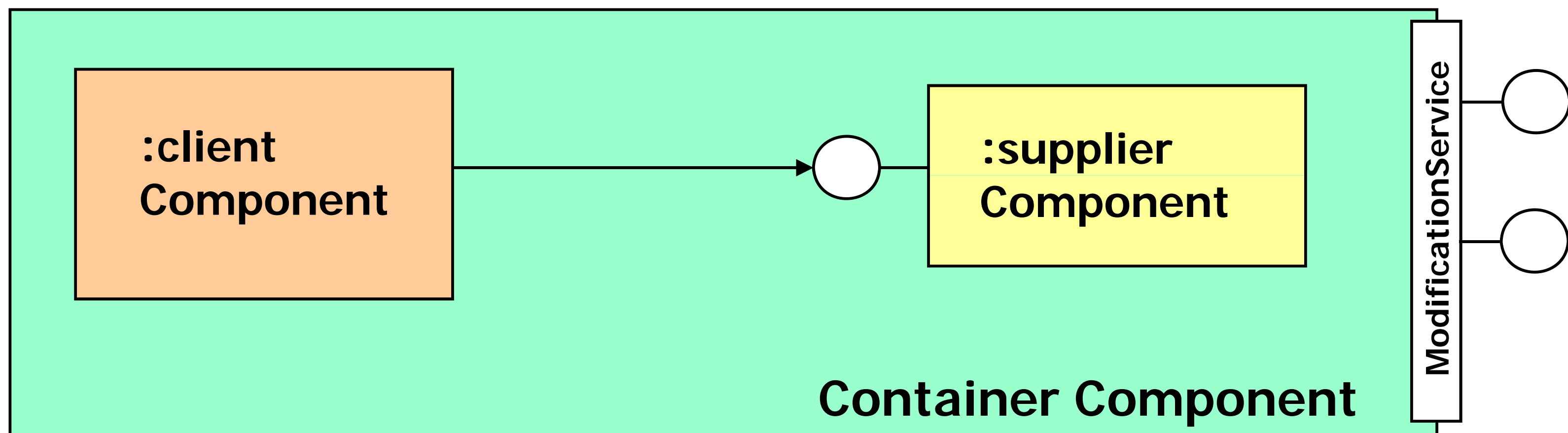


- Defining workflow structures
- Defining exception handling

- Technical concerns
  - Adoption of dynamic object binding (polymorphism) → cf. chapter 2
  - Design Patterns
  - Class loading mechanisms (e.g. in Java)
  - Reflection (runtime analysis of classes)

Modifiability Tactics
Defer design decision

Runtime Registration of Component (Hot deployment)



- Storage concerns
  - Storage of any design decision in flat files (deployment descriptor)
  - Easy to read and to manipulate
- Examples:
  - Enterprise Java Beans (EJB)
  - CORBA Component Architecture (CCM)

Modifiability Tactics
Tailorability

- Fields of application are differentiated and dynamically changing
  - New tasks and working contexts arise
  - Instant modification often necessary

- Tailorability is defined
  - changing aspects of an application‘s functionality
  - in a persistent way
  - during the use of an application (at runtime)
  - by end-users or local experts

Modifiability Tactics
Tailorability

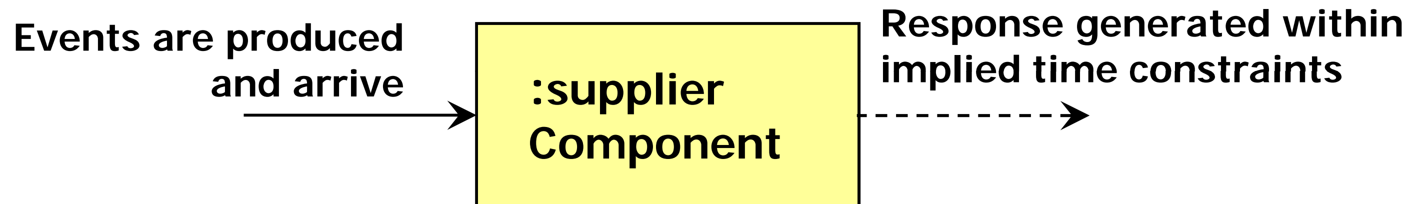
- Problem:
  - User do not have experience with tailoring
- Goal:
  - Provide tailoring routines on different levels of complexity
- End-user with little experience can adopt to less complex routines
- More experienced may use more sophisticated routines
- Proposed Levels:
  - Customization
    - Modification of presentation objects among a set of pre-defined configuration options
  - Integration
    - Creation or the combination of (existing) program behavior that results in new behavior
  - Extension
    - Adding completely new behavior (→ re-implementation)

Performance Tactics
Properties

b-it

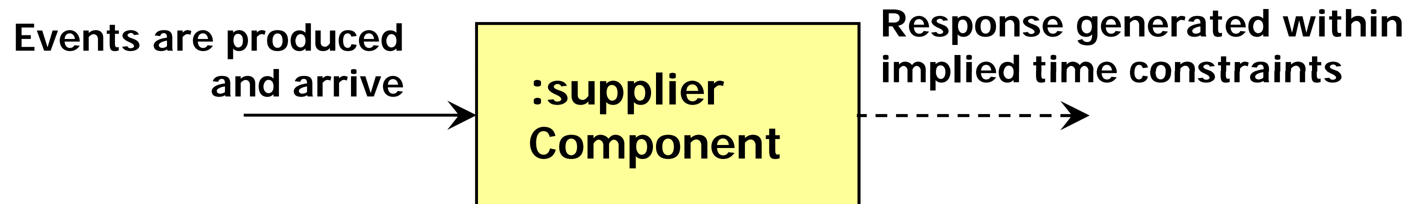
- Performance requirements concern the *speed of operation* of a system
- Types of performance requirements:
  - Response requirements (how quickly the system reacts to a user input)
  - Throughput requirements (how much the system can accomplish within a specified amount of time)
- Areas of concerns
  - Reduction of computational overhead
  - Scheduling of events
  - Policy-based regulations

Performance Tactics
Reduction of computational overhead (1/2)



- Increase computational efficiency of supplier component
  - Inspection of chosen algorithm
- Manage events rate from the perspective of the clients
- Place a limit on execution time that is used to respond to an event
  - Scenario: iterative data processing: limit the number of iterations
- Introduce efficient caching/queue structures
  - Fix a maxim number of events in such structure
- Provide more efficient communication path
  - Example: avoid marshalling of data (cf. RMI vs. direct object interaction)

Performance Tactics
Reduction of computational overhead (2/2)



- Bypass high layers in a layered architecture (open architecture!)
- Introduce concurrency when events can be processed in parallel
- Establish and maintain multiple copies of input events (data)
  - Degree of concurrency can be increased
  - Data consistency must be ensured
- Increase available resources
  - Faster processors, additional resources
  - Cost vs. performance trade-off
- Scheduling of events to reduce contention for a resource

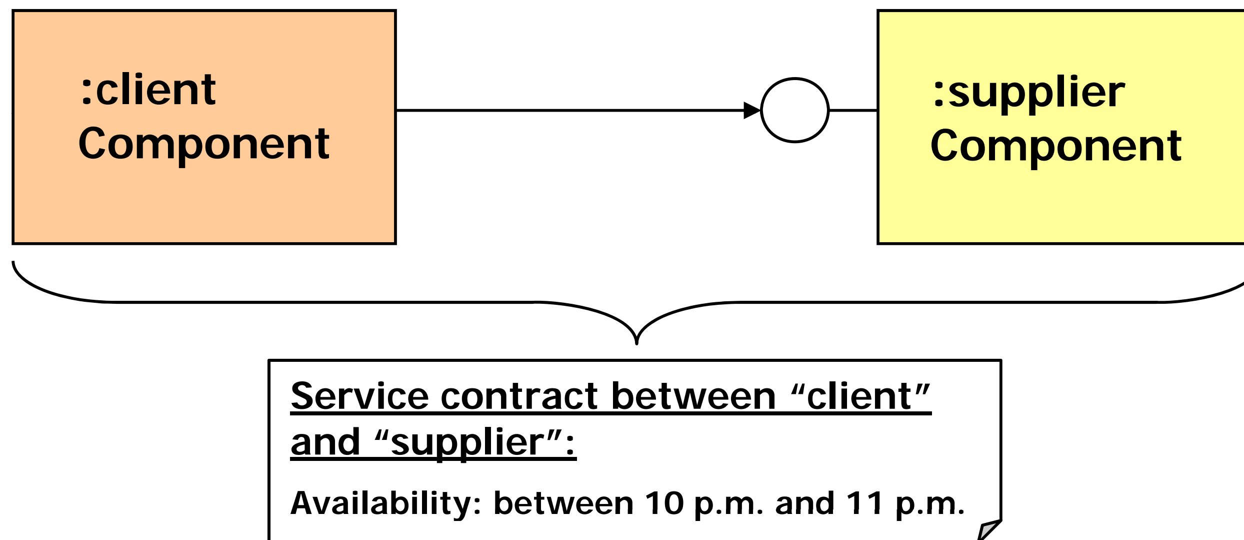
Performance Tactics
Scheduling of events

b-it

- A scheduler manages the assignment events to a resource
  - Resource: CPU, buffer, network
- Two different parts:
  - Priority assignment of events
  - Dispatching of events to a resource

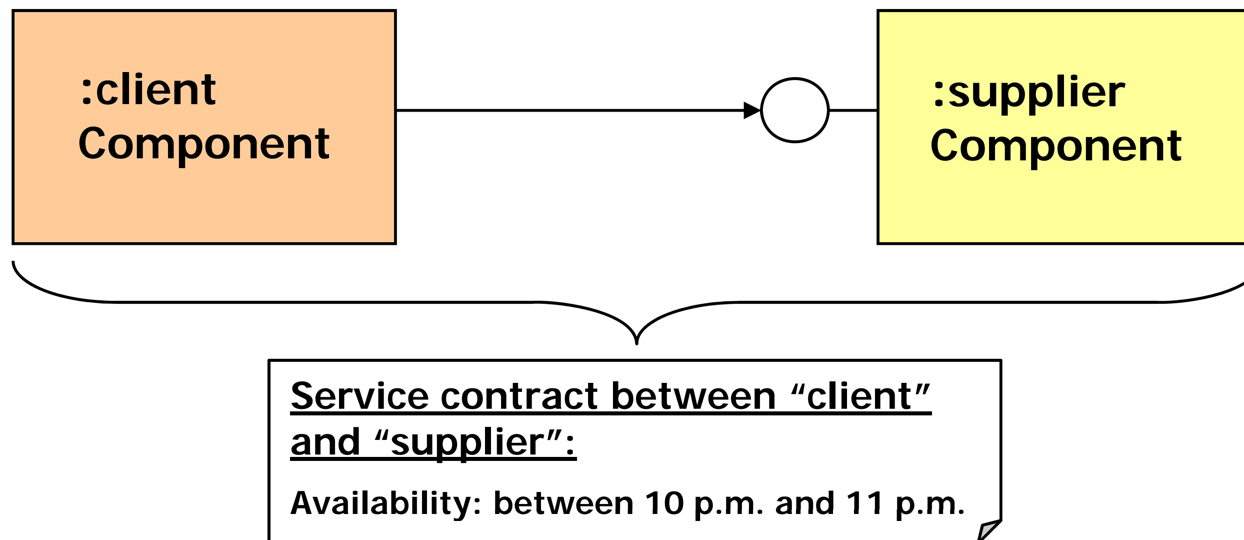
- Prominent Scheduling implementations
  - First-in/First-out (FIFO)
    - No Assignment of priorities, all events are equal
  - Fixed-priority scheduling
    - Semantic importance (assign priority to events w.r.t. to domain characteristic)
    - Deadline monotonic (assign higher priority to events with shorter deadlines)
    - Rate Monotonic (assign higher priority to events with shorter periods)

Performance Tactics
Policy-based Regulations



- A policy regulates the interaction between client and supplier in terms of a contract (**contract-based collaboration**)
  - SOA: Service Level Agreement (SLA)
- Different issues can be defined
  - Availability of a service
  - Guaranteed performance of a service
  - Modification concerns (e.g. notification before an adaptation occurs)

Performance Tactics
Policy-based Regulations



- The violation of a contract can limit the trustworthiness of a provider for future applications
- Reputation and trust models (→ Peer-to-Peer architectures, ebay)
- Establishment of contracts based on social negotiation between users
- No real standards have emerged so far

Usability Tactics
Properties

- Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of support the system provides to the user
- Usability is the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of system or component

- Areas of concerns
  - Runtime aspects (Nielsen's heuristics)
    - Support user during system execution
  - User interface design

Usability Tactics
Principles according to Nielsen

b-it

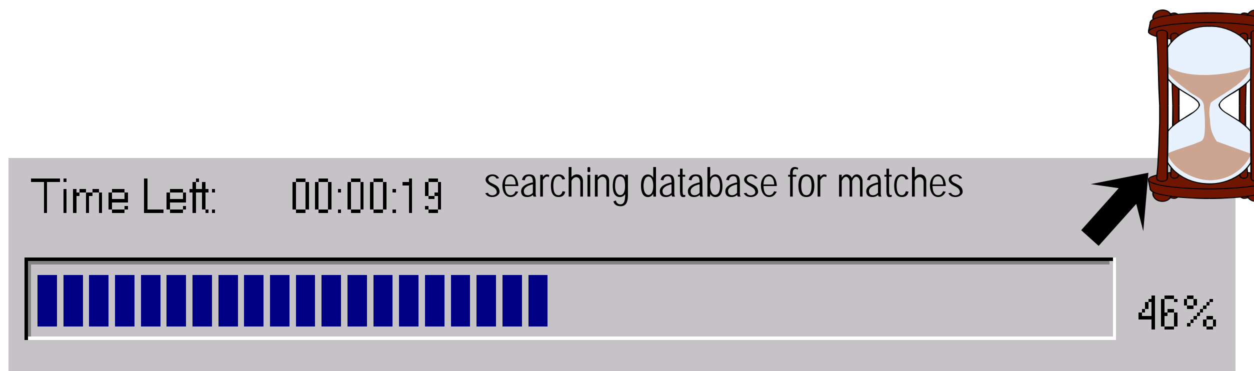
- Visibility of system status
- Match between system and the real world
- User control and freedom
- Consistency and standards
- Recognition rather than recall
- Aesthetic and minimalist design
- Help users recognize, diagnose, and recover from errors
- Help and documentation

- Called "Nielsen's heuristics of usability", 1994
- http://www.useit.com/

Usability Tactics
Principles according to Nielsen (1/3)

Visibility of system status

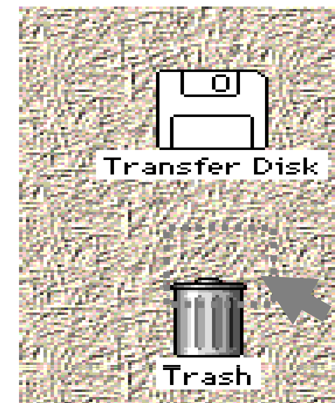
- The system should always keep users informed about what is going on, through appropriate feedback within reasonable time
- For example, pay attention to response time
  - 0.1 sec: no special indicators needed
  - 1.0 sec: user tends to lose track of data
  - 10 sec: max duration if user to stay focused on action
- for longer delays, use percent-done progress bars



Usability Tactics
Principles according to Nielsen (2/3)

Match between system & real world

- The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms.
- Follow real-world conventions, making information appear in a natural and logical order
- Bad example (old) Mac and Amiga desktop
  - Dragging disk to trash
  - logically should delete it, not eject it



Usability Tactics
Principles according to Nielsen (3/3)

b-it

User control & freedom

- Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue.
- Support undo, cancel and redo
- Wizards
  - “Surf” between states
  - For infrequent tasks (e.g., modem configuration.)
  - Not for common tasks
  - Good for beginners
    - have 2 versions (WinZip), no fixed paths

Usability Tactics
User model

- During system execution, it must rely on some information of the user (user model). Contents:
  - User knowledge of the system or environment
  - User Preferences (e.g. style of interface, response time)
  - Location based information (e.g. residence, coordinates)
  - Domain related preferences (e.g. hobbies, activities)

- Scenario: Location-based services

Usability Tactics
User interface design

- Many guidelines available
  - Layout manager (e.g. Java)
  - Styles: MVC

- Basic message: separate user interface from the rest of the application
  - Reason: User interface is expected to change frequently during development and deployment
  - Separation: localize changes

- More information:
  - http://www.cs.uni-bonn.de/III/lehre/vorlesungen/SWT/RE05/slides/10_Interactive%20Systems.pdf

Summary

- Implementing non-functional requirements as design goals into an architecture is no trivial task
  - Incorporation of many technical issues
  - Many trade-off situations
  - Many experience needed
  - Dependencies to many related disciplines

- Key success factor for a successful architecture design