Software Architecture & Quality

Lukas Bach - lbach@outlook.de - lukasbach.com

1 Introduction

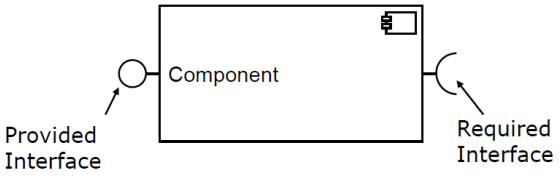
Software architecture: fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution. Result of a set of design decisions relating the system structure (e.g. components…)

Quality Metrics

- Performance
- Reliability
- Security

Potential others: Functionality, Reliability, Usability, Efficiency, Maintainability, Portability.

1.1 Components and Interfaces



Benefits of components:

- Encapsulation (comprehension, redundancy, maintainability)
- Reuse (Unplanned/Component repositores or Planned/Product Lines)
- Reconfiguration
- Efficiency
- Engineering Approach (systematic construction)

2 Architectural View Points

Architectural design decisions are made early on (requirement-oriented) or during later phases, when the influence of execution environments becomes clearer (code-oriented).

- Why explicit models of Software Architecture?
  - Implicit models might not exist, if no decisions were made
    - Reuse, COTS, planned future reuse
  - Communication between stakeholders and developers
  - Analysis of quality impact
- Factors influencing the Architecture
  - Requirements
  - Re-Use
  - Organization (Conway's Law: Organization produce designs copying their internal communication structure)
- Benefits of Software Architecture:
  - Stakeholder communication
  - System analysis
  - Large-scale reuse
  - Project planning

2.1 View-based Modelling

2.1.1 Model

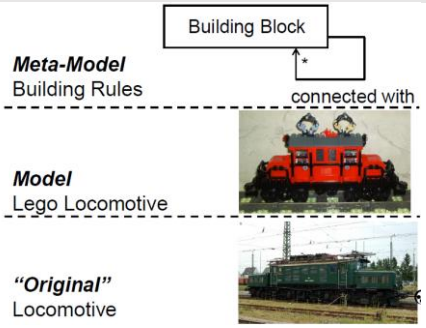
A model is an: (i) Abstraction of the modelled entity, (ii) With a given abstraction aim, (iii) Leaving out details not relevant to the abstraction, (iv) To simplify analysis/comprehensibility/…

- *Representational Feature* (Abbildungsmerkmal)
  Models are always models of something, i.e. depictions of natural or artifical originals, which themselves can be models.
- *Reduction Feature* (Verkürzungsmerkmal)
  Models do not capture all attributes of the originals.
- *Pragmatic Feature* (Pragmatisches Merkmal)
  Models are not assigned to their original, but rather fulfill their substitution function
  - For particular subjects,
  - In particular intervals of time, or
  - Under restriction to particular mental/actual operations

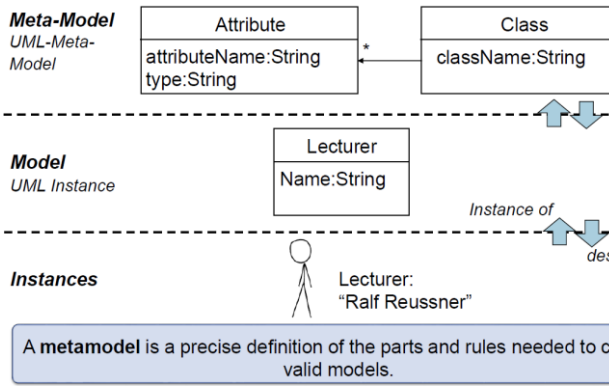
Benefits of Models: Analyzability, Cope with complexity, Usable by domain experts, reuse of domain abstractions, increased communication efficiency, consistent documentation.

- Extensions of legacy software systems
- Performance predictions at design time (Palladio)
- Analytical solvable
- Simulation (faster than real execution)

2.1.2 Meta-Model



Example:



2.1.2.1 Syntax and Semantics

A Metamodel covers an abstract syntax, at least one concrete syntax, and static and dynamic semantics.

- *Abstract syntax*: Elements and their relations independent of representation
  e.g. UML diagram
- *Concrete syntax*: Representation of model-Instances, e.g. on storage devices
  e.g. Java implementation
- *Static semantics*: Semantics evaluatable without executing the model
- *Dynamic semantics*: Description of behaviour of the model

Domain specific language (DSL): Defined by an abstract syntax and a mapping to at least one concrete syntax. The abstract syntax describes language structure, the concrete syntax defines the textual/graphical depiction of encoded elements of the abstract syntax (02-24).

From my SWT2 recap:

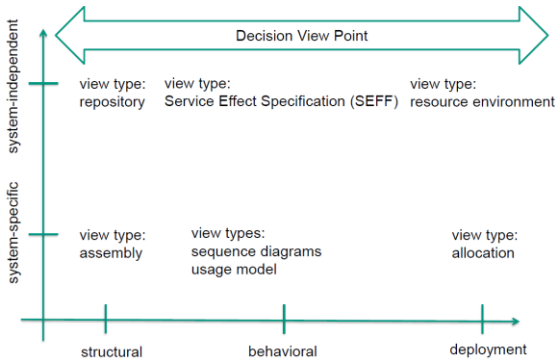
- *Abstract Syntax*
  - Describes contructs of which the model consists, as well as their properties and relations between them. Independent of concrete depiction of these.
  - E.g. UML class diagram
- *Concrete Syntax*
  - Constructs, properties and relations specified in the abstract syntax. At least 1, arbitrarily many concrete syntaxes are required.
  - E.g. Data instanziation
- *Static Semantics*
  - Modelling rules and restrictions that are hard to express in abstract syntax. Using specialized languages for restriction-definitions as constraints.
  - E.g. OCL contraint
- *Dynamic Semantics*
  - Describes the meaning of the constructs. Natural language specs.

2.1.3 View Points, View Types, Views

- *View point*: Conceptual viewpoint to define certain concern. Specific to certain stakeholders.
  - E.g. set of diagrams/behavior diagrams
- *View type*: Set of meta-classes whose instances a view can display. Concrete syntax + mapping to abstract meta-model syntax. On level of metamodel.
  - E.g. diagram/activity diagram
- *View*: actual set of elements and their relationships. On level of model.
  - E.g. specific diagram instance

2.2 Palladio Component Model (PCM)

Modelling Language (DSL) used by Palladio Approach.



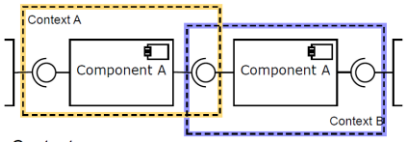
2.2.1 Structural View Points

Dependency structure of systems and the specification of their building blocks.

- Component repository: system independent set of data types, interfaces and components.
  - Used by architect to retrieve components, specify new interfaces and to abstract components.
  - Used by developers to create and deposit concrete components.
- Black/Grey/White Box components
  - Black box: No internal information
  - Grey box: Abstract view of internals, e.g. UML
  - White box: Implementation given
- Component vs Classes/Objects: Components deployment context changes after compilation. Component may consist of several classes (more on 02-50ff).

TODO more on components F02-51_67.

- Assembly: Specifies inner structure of composite entity.
  - Entities: Assembly Context, Assembly Connectors, Delegation Connectors, Systems
  - Assembly Context: The same component type can be used multiple times in different "contexts"



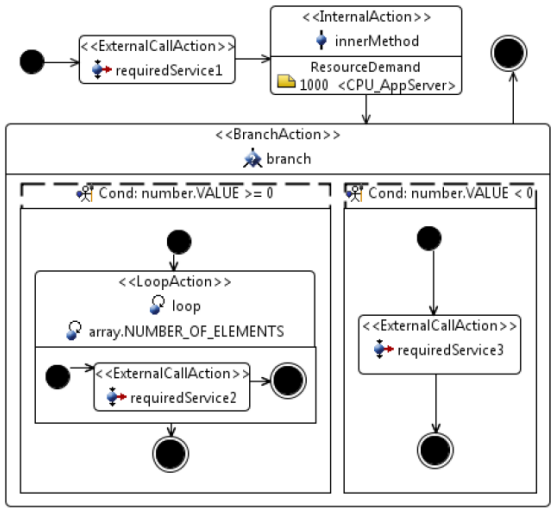
  - Connectors cannot be 1st class members

2.2.2 Behavioral View Points

- Inner-component behavior: Internal behavior of one component independent of others
- Inter-component behavior: Interaction between components, abstracting from internals

2.2.2.1 Inner-component

Service Effect Specification (SEFF): describes inner-comp. behavior. Specifies relationships between provided and required services of comp and abstract information of inner wiring.



2.2.2.2 Inter-component

E.g. sequence diagrams (components, messages, combined fragments).

2.2.2.3 Scenario Behavior

User scenarios describing interaction between user types and modelled software system. User types can be humans or other systems. Derived from requirement documents or specified by domain experts. See activity diagrams.

2.2.3 Deployment View Point

- Creating deployment instance = putting implementation instance into a context.
- Specification of execution environment, allocation of resources (e.g. 100mbit/s)

2.2.4 Decision View Point

Decision taken during development and evolution are modelled as decision viewpoints. Can also come from changed requirements or constraint.

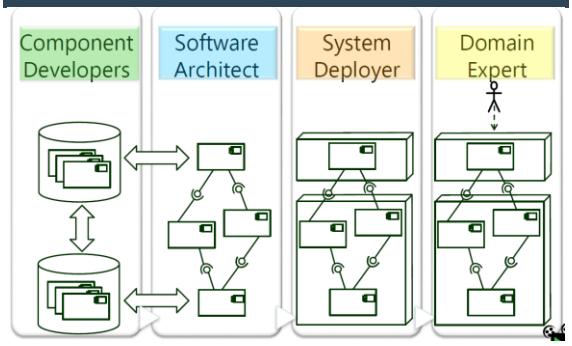
Decision viewpoint connects all other viewpoints.

- *Existence decisions*: Why was element introduced?
- *Ban/Nonexistence decision*: Why was element excluded?
- *Property decisions*: system qualities not linked to particular architecture elements
- *Executive decisions*: related to organizational aspects, e.g. why use Java EE

3 Palladio (Supplementary)

Performance prediction for component-based software architectures. Allows creating PCM model instances with graphical editors.

3.1 Roles



Detailed Role Descriptions: F03_27-50.

3.2 Contexts

- Assembly Context (Architect)
  - Binding of Components (horizontal composition)
  - Encapsulation of composite components (vertical composition)
- Allocation Context (Deployer)
  - Allocation to hardware resources
- Usage Context (Domain Expert)
  - User Arrival Rate, #Users, Request Probabilities, …

4 Architectural Reuse

Levels of Reuse

- Reuse of components (Implementations)

- Architecture (Design Patterns)
- Architectural Patterns
- Architectural Styles (system-wide constraints. Not mixable)

Reuse of components and interfaces is more fine-grained that reuse of architecture.

4.1 Component and Interface Reuse

4.1.1 Component Identification

Result of divide-and-conquer decomposition. Components should be

- entities of *reuse* (in same product line, product family, different context…)
- entities of *update/maintenance* (due to technical/requirement changes, interoperability, …)
- entities of *dynamic reconfiguration* (swapped due to software updates or memory limits…)
- entities of *distribution* (on different computers due to parallelization, fault-tolerance, design…)
- entities of *delivery/accounting* (different versions, specialized functionality, …)

4.1.2 Component Reuse

Reusable component 2-3 times more expensive.

4.1.2.1 Palladio Submodels

F04_20ff. Every level removes one unbound context for QoS.

- Specification & Implementation level
  - All unbound contexts
- Assembly level
  - Removes context “External Services”
- Allocation level
  - Removes context “Allocation”
- Runtime level
  - Removes context “State”
- Execution level
  - Removes context “Usage Profile”
  - No unbound QoS contexts left

4.2 Architectural Style Reuse

Architectural style: Named collection of architectural design decisions, applicable to given development context, constrain decisions specific to the system in the context, eliciting beneficial qualities in resulting systems.

Examples: OO design, modular design, event-based.

Characteristics: TODO F04_28

4.2.1 Dimensions

Never mix within one dimension, but one style per dimension is okay. Dimensions:

- Primary decomposition
  n-tier architecture or microservices
- Secondary composition
  e.g. n-tier within one microservice
- Interaction/communication
  e.g. events, sync. Messages, call-and-return

4.2.2 Software Blood Groups

Software categories according to reuse.

- *0*: independent of tech and application
  e.g. Regex, Strings
- *A*: independent of tech
  e.g. enterprise information system
- *T*: Independent of application
  e.g. software using technical APIs
- *AT*: based on tech and application
  Not reusable or maintainable. To be seperated
- R: Representation software
  Transfer between A and T.

+	0	A	T	R
0	0	A	T	R
A		A	AT	./.
T			T	./.
R				R

4.2.3 Event-Based

Asynchronous message exchange. Queueing/Buffering required. Questions like (wrong events? Full buffer? Interrupts?) have to be answered.

Drawbacks: Behavior hard to extrapolate, non-deterministic, bad scalability.

4.2.4 Microservices

Services are independently deployable and scalable, can be written in different languages and managed by different teams.

Decomposition according to functional units rather than technical domains as in N-tier architecture.

Often follow Conways law, each service modelling a team.

4.3 Architectural Pattern Reuse

Collection of architectural design decisions applicable to recurring design problems, parametrized for different contexts of the problems.

Concerns the organization of the whole system rather than local impacts of *design patterns*.

More concrete, explicit, prescriptive and mixable than *architectural styles*.

Benefits:

- Fine-grained decision
- Restricted design space
- Warranty for quality
- Improved maintenance and evolution

Drawbacks:

- Architectural pattern misuse
- False implementation
- Performance and maintenance problems

4.3.1 Peer-to-Peer

No hierarchy, cyclic dependencies. All components are peers in flat organization. Communication via shared channel.

4.3.2 Client-Server

Hierarchy. Composed of components (clients, servers or both) and mostly sync. connectors.

Pro: Straightforward data distribution, effective usage of networked systems, scalable.

Con: No shared data model, redundant management per server, no central register of services.

Fat vs. Thin Client: F04_63.

4.3.3 Pipeline

Acyclic, specialized roles. Stream of data passed through.

Leverages many design principles (divide and conquer, low coupling, high abstraction, reusability, reuse, flexibility, testability, …)

4.3.4 Blackboard

Central data-oriented repository of shared data.

4.3.5 Coordinator

Hierarchy with peers, acyclic.

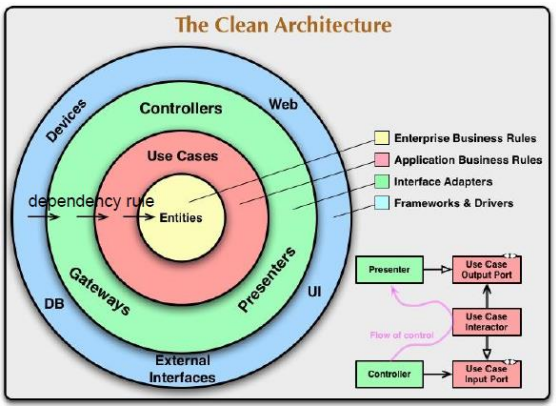
4.4 Reference Architecture Reuse

A way of standardization of software architectures for a specific domain. Template for the system design, applicable for all systems of that domain.

Implicit variation points compared to PL architectures: I can implement a reference architecture in various ways, reusing previous design decisions.

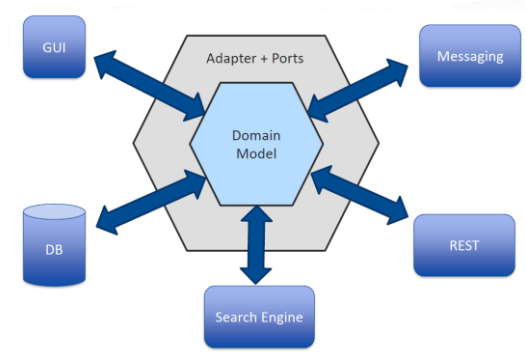
Example:

- OSI stack
- clean architecture
  - leads to testable systems and independence of frameworks, UI, DBs and external agencies
  - Dependency rule: must always point inwards



Interface Inversion: F04_80.
TODO F04_81-84.

- Hexagonal architecture



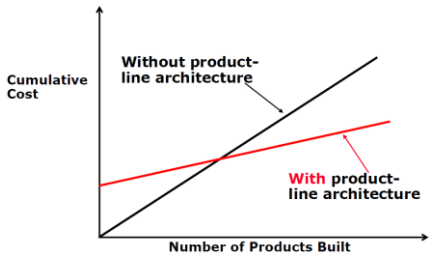
4.5 Product Line Architectures

Developing different very similar products.

Product line: Set of software-intensive systems that share a common managed set of features.

Difference to reference arch: PL arch is an *explicit model of variability*, variation points are made explicit.

Example implementations: Framework or Application with Plugins.



Variation points delay design decisions.

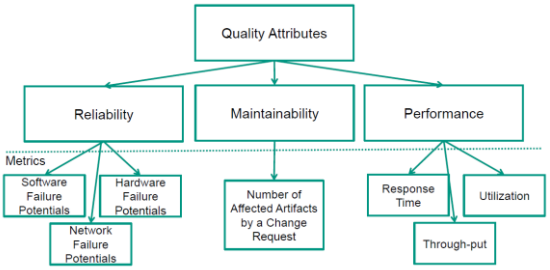
Binding can happen pre- (PL) or post-delivery (configurable product)

Approaches:

- Proactive (Mainstream):
  - SPL-Engineering as primary phase
  - Automated generation of software products
  - *High initial effort* and organizational costs, unused functionality is implemented
- Reactive
  - Product-Engineering as primary phase
  - Low initial effort, evolution and maintainability

5 Modelling Quality

5.1 Software Quality



- Intrinsic attribute: Definition of property A involves property B
  - E.g. System is “available” if reaction time <1s
- Extrinsic attribute: Change of property A affects property B in architecture C
  - E.g. large-grain components improve performance, but reduces maintainability, in one specific architecture
  - i.e. architecture influences quality

5.1.1 Maintainability

Motivation: Lots of artifacts affected by change requirements, e.g. implementation, build config, tests…

KAMP: Karlsruhe Architectural Maintainability Analysis. Overview on F05_32-50.

- Creating Target Model
- Modelling Change Request
- Calculating Structural Change Propagation
- Updating Context Annotation
- Deriving Architecture-based Task List
- Deriving Task Lists with respect to Work Areas

5.1.2 Performance

- Timeliness: Response time and throughput
- Efficiency: Resource utilization and costs

Degree to which a system fulfills a given non-functional property, like responsiveness or dependability.

5.2 Performance Evaluation

Goals:

- Platform Selection
- Platform Validation
- Evaluation of Design Alternatives
- Performance Prediction
- Performance Tuning and Optimization
- Scalability and Bottleneck Analysis
- Sizing and Capacity Planning

Approaches:

- Educated Guess
  - Quick and cheap, but inaccurate and risky. Only simple situations.
- Performance Measurement
  - Load-tests and benchmarks. Accurate and realistic, but expensive and time-consuming.
- Performance Modeling and Analysis/Prediction
  - Cheaper and quicker than load-testing, appliable at design stage. But complex to build, accuracy depends on how representative models are. Models may differ from actual design.

5.3 Performance Modelling – Queueing Theory

Queueing Theory leverages *Analytical Models*.

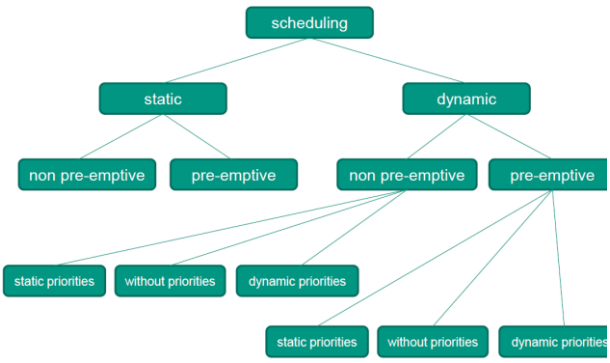
5.3.1 Terminology

- Arrival rate: #request arrivals per timeunit
- Interarrival time: Timebetween successive arrivals
- Service rate: Max #requests processable per timeunit by server
- Throughput: #requests processed per timeunit
- Queuing delay: Time waited by request in waiting area before being serviced

- Response time: Total amount of time request spends in queue, i.e. sum of queuing delay and service time.
- Workload types:
  - Open workload: Fixed frequency of arbitrary requests
  - Closed workload: Fixed number of circulating users
- Workload classes
  - Class of workloads with similar usage behavior
- Flow Equilibrium Assumption: assume that the rate of request arrivals equals rate of request departures.

5.3.2 Scheduling Strategies

- First Come First Served
- Last Come First Served
- Round Robin
  - Each request served short periods of time
- Processor Sharing
  - all requests simultaneously
- Infinite Server
  - Enough servers s.t. no queues are formed
- Random
- Priority Queuing
- Shortest Job First



5.3.3 Typical Assumptions

TODO F05_92-93

5.3.4 Operational Laws

Equations describing average service behavior of a queue, but not on scheduling or inter-arrival-/service-times.

- Service Demand Law
  - $Di, c = \frac{U_{i,c}}{X_c}$ computes resource demand D of workload class c at resource i, for utilization $U_{i,c}$ of resource due to workload class, and throughput X_c of workload class.
- Forced Flow Law
  - $Xi = V_i \times X_0$ computes throughput X_i of resource i for average number of visits V_i made by requests to that resource, and the system throughput X_0.
- Little's Law
  - $N = X \times R$ computes average number of customers in a black box, for average depature rate X (i.e. throughput) and average residence time in the box R.
  - Applied to any black-box containing a set of components.

5.4 Performance Modelling – Palladio

Palladio leverages *Design models*.

Resource Demanding Service Effect Specification (RDSEFF): Extension of SEFF for performance prediction.

- Parametrized, behavioral abstraction and quality specification for a component.
- Describes resource utilization of component and how required services are called.
- Elements: F05_111-112
  - Internal Call, External call, emit async event, resource demanding behavior, branch, loop, fork, acquire, release, …

5.4.1 Constructing a Palladio Performance Model

Visualization on F05_114-141.

5.5 Putting the Pieces together

Combine analytical models (e.g. Queueing networks) and design models (e.g. Palladio), run simulations/analysis on that to get response time, throughput, utilization, …

6 Getting the Data

- Information required about system structure, behavior, execution environment, usage
- Middleware: software layer between OS and application in business information systems
  - Infrastructure for application layer
  - Middleware can monitor response times etc.
  - Monitoring overhead should be low

6.1 Data Collection

- Different Palladio roles need to quantify influences on quality (e.g. component dev: failure rates and resource demands, deployer: hardware availability, …)

- Data becomes more available during development
  - Design: Only diagrams, use cases, …
  - Development: Measurable system parts
  - Operation: Measurable performance
- Model Calibration: Enrich system model with quantitative data like resource demands. Prerequisite for conducting performance analysis
- Accuracy vs Precision: F06_14

6.2 Real User Monitoring vs Application Performance Monitoring

- Real User Monitoring (RUM): Mainly web-based apps, understand user behavior. Inject JS to get info on behavior and experience.
  - E.g. Google Analytics
- Application Performance Monitoring (APM): Internal aspects, monitoring services to identify performance problems and get info of resource utilization and component interaction.
  - E.g. App Dynamics

Both applied during operation (real use or deployed in representative environment). Constraints: Measurement overhead must be negligible in production env, and legal perspective must be considered (user monitoring).

Data needs to be aggregated, correlated and interpreted.

Use workload classes during interpretation (representative request sequence) to find user sessions with similar usage behavior.

6.3 Resource Demand Estimation

6.3.1 Data Collection

Working prototype required, otherwise you can just make educated guesses or look at comparable apps.

Approaches:

- Execution in test environment, custom load
  - Simplified data interpretation, but system has to be available in test env, and only applicable during low resource utilization (TODO?).
- Collect monitoring data during prod operation
  - Challenging estimation of resource demand. Estimations become obsolete with switching software/hardware.

Mapping between Palladio and Workload classes: TODO F06-34.

6.3.2 Estimating Resource Demand

Approaches:

- Measure Response times
  - Assumes that considered resource dominates overall response time, and queue waiting time is insignificant
  - Very easy, but only applicable during low resource utilization, and platform specific.
- Apply Service Demand Law, one request type at a time
  - Assumes only one workload class
  - Easy to measure total resource utilization, but system needs to available in test env, task-specific monitoring often not available, different workload classes hard to partition (cannot capture different transactions)
- Apply Service Demand law, multiple request types at a time
  - Assumes $U_{i,c}$ can be determined.
  - Applicable also for request/transaction mixes, but partial utilization $U_{i,c}$ hard to derive (system only provides total utilization, profilers have high overhead).
- ByCounter and Microbenchmarks
  - Count bytecount operation and bytecode operation benchmarks.
  - Platform independent resource demands, but Bytecodes required, and hard to capture complex middleware systems.

Task: F06_45.

6.4 Estimating Failure Probabilities

- Failure probabilities quantify reliability
- Design Phase:
  - Estimate component sizes based on experience, high failure prob for high sizes
  - Estimated bounds by developers per comp.
  - Measure on similar systems
- Development Phase:
  - Code metrics (LOC, McCabe) on implement.
  - But often complexity uncorrelated with failure rates
- Operation Phase:
  - Statistical Analysis of Bug Tracker DB
- Also applicable: Static code analysis, test case execution, analyzing bug DB.

6.5 Common Pitfalls

- Wrong Resource Demands
  - Time intervals include many other influences, don't reflect resource consumption correctly.
- Inaccurate Timers

  - E.g. clock ticks provided by CPU on multi-CPU environment
- Missing Resources
  - Memory, passive resources like thread pools (TODO?)
- No Validation
  - Does model describe system performance accurately?
- Too Many Details
  - Harder to isolate root cause, increases complexity of collecting informations and maintaining the model

7 Answering Design Questions

7.1 Design Questions

Palladio was designed to answer performance and reliability questions.

7.1.1 General Design Questions

- Which designs is the best?
- What is an optimal configuration for the System?
- How to extend a legacy software system?
  - Extend with wrappers & adapters, or modify?

7.1.2 Additional Performance Design Questions

- How to optimally size the system?
  - Capacity planning/sizing, avoid wasted resources, prevent bottlenecks due to insufficient capacities
- How scalable is the system?
- How to balance the load on system resources?
  - Utilize resource evenly

7.1.3 Additional Reliability Design Questions

- Which Fault Tolerance mechanism shall the system exhibit?
  - Against bugs or physical failure
- How much redundancy shall the system exhibit
- Useful to introduce design diversity?
  - TODO?
- How to deal with physical failures?
- Which components contribute the most to the reliability of the system?
  - Direct QA effort into most critical component

7.2 Understanding the Results

7.2.1 Understanding Performance Results

- Analysis results (from performance simulation)
- Service results
  - Services are located at system boundary or within system internals. Store response times of simulated requests and as experienced by simulated users.
  - Visualize with
    - Histograms
    - Cumulative distribution function (CDF, maps response time to probability that the actual time is below that)
- Resource results
  - Utilization for CPU, Memory, HDD. Waiting/Holding time for passive resources.

Core Causes:

- Bottlenecks
  - Can appear at processing- and passive resources. Hardware bottlenecks easy to identify, software ones not so much.
- Long paths
  - Sequence of actions and calculations too long
  - Reduce required processing in path, make parts async/parallel, reduce latencies by reducing remote communication

7.2.2 Understanding Reliability Results

Software reliability: TODO F07_52

- Software failure: During processing of user request due to software fault. Caused by bugs in software or dependencies.
- Hardware failure: CPU is unavailable while access, or other resources.

7.3 Tactics for Improving Quality

Table of improvement tactics on F07_64-70.

7.3.1 Performance Tactics

7.3.2 Reliability Tactics

7.4 Automatically Searching for Better Architectures

TODO lecture video?

8 Under the Hood

8.1 Quality Analysis Tools

- Solving an analysis model: Gather quality metrics for the system presented by the analysis model

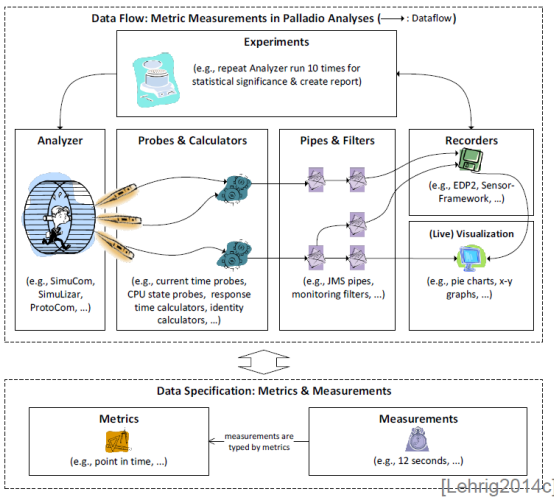
- System is first modelled with Palladio, then analyzed by analysis tool

8.2 Performance Simulation of Palladio Models

- Simulation run:
  - *Workload generator* continuously spawns *simulated users*
    - Workload/#users can be parametrized
    - Closed workload: Sustain constant population size
    - Open workload: Spawn users at inter-arrival times
  - Users put simulated system under load by *sending requests*
    - Simulate usage scenarios
    - Usage scenarios can also include probabilistic branches
  - *Simulated services* consume simulated resources to handle requests
    - Simulated request starts at system's provided interface and follows simulated component behavior
    - Invoked internal actions issue resource demand accordingly
  - Concurrent system calls *compete for simulated resource* accesses
    - Processing resources (CPU, HDD) and passive resources (DB connection pools, semaphores)
  - This competition for scarce resources is referred to as *resource contention*
- Scheduling Policies are also simulated

8.2.1 Gathering Measurements

Palladio uses the Quality Analysis Lab (QuAL) for gathering measurements from different analysis tools.



[Lehrig2014c]

Specification of metrics and measurements:

- Metric descriptions: characterize measured metrics, like response times. Document capture type (real numbers) and unit (seconds).
- Measurements: Store values at given point.

Data flow for metrics:

- *Analyzer* runs the environment to be measured, e.g. simulation or performance prototype
- *Probes* measure values for a given analyzer
- *Calculators* attach to probe sets to enrich measurements by the investicated metric and measuring point, and specify which measurements are of interest outside of the analyzer
- *Pipes and Filters* transfer and filter measurements by calculators to data sinks like recorders
- *Recorders* store measurements, e.g. DB, XML, …
- *(Live) Visualization* presents recorded measurements (from Recorder) or live monitoring data (from Pipes) graphically
- *Experiments* used to conuct analyzer runs and create new measurements

8.3 Performance Analysis Tools

Description of SimuCom, EventSim, ProtoCom, F08_37-52.

- SimuCom generates testable implementations from components and SEFF. Process-oriented and generative.
- EventSim interprets loaded Palladio model during simulation. Event-oriented and interpretive. Faster than SimuCom, but fewer features.
- ProtoCom transforms palladio models into executable performance prototypes.

9 Case Studies

See slides, three applications of the topics.

10 Wrap Up

See slides 09_18ff.

11 Clean Architecture (Andrena)

11.1 Reactive Manifest

- Existing architecture:

  - SPA web app, >10 services, 2 replicas/service, API Gateway in between, one DB per service, sync. REST via HTTP and async. Messages.
- Domain driven design (DDD) for specifying microservices.

11.2 Domain Driven Design

- Many different domains during design, development, … Domains in different kinds:
  - Recruiting (Supporting kind)
  - Vertrieb (Supporting kind)
  - Matching (core kind)
  - Abrechnung (generic kind)
- Avoid anemic domain models:
  - Hardly any behavior, just bags of getters/setters
- Map bounded contexts to microservices
- Map problem space to subdomain, and solution space do bounded context.

11.3 Sync/Async communication

- Synchronous service communication
  - + Easy, comparable with method invocations
  - + Results immediately available
  - + Intuitive program flow
  - + Easy debugging
  - –High coupling
  - –Error handling required if service unavailable
  - –Load balancing non trivial
- Asynchronous service communication
  - +Multiple receives can share processing
  - +Can repeat failed messages
  - +Can collect failed messages
  - +Sender does not have to know who receives messages
  - +It's okay if the receiver is unavailable for some time
  - +Easily persistable communication
  - –Inconsistence for some time is unavoidable

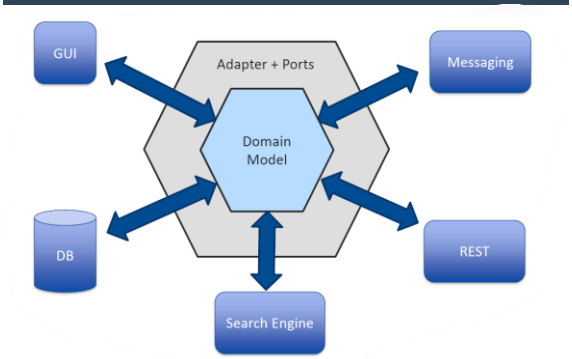
11.4 Using data of different services

Store data redundantly on multiple services, or accept higher coupling between services and store it in additional service?

11.5 SOLID

- Single Responsibility
- Open Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

11.6 Hexagonal architecture



Adapter divides domain code from infrastructure code.