3 Architectural Viewpoints

Answers and Solutions

3.7.1 Answers

3.1: Palladio currently supports view points for structural, behavioral and deployment modelling. The structural view types model component repositories and system assembly. The behavioral view types model Service Effect Specifications (SEFFs) and usage models. The deployment view types model the resource environment and allocation of assembly contexts.

3.2: The service signatures that are specified in interfaces do not depend on the components that provide corresponding services (Rhinelander 2007). Interfaces can, for example, be modelled before a component that provides them is modelled. Therefore, components should not *have* interfaces in the sense that these interfaces pertain to the components. It is sufficient to state which components provide which interface and to model these interfaces separately.

3.3: A component can require several different interfaces for several reasons: For example, it may have dependencies to different services that are also functional separated in the domain and not necessarily provided by a single component. Or it may be necessary to separate identical functionality on different layers of abstraction, e.g. high-level services specified by a protocol and their technical realization using a database.

3.4: In order to ease the reuse of components, which is the main goal of using them, we have to distinguish between components, their instantiation in the form of assembly contexts and component types. Components only specify the signatures of the services they provide and require by providing and requiring the according interfaces. If a component is instantiated, the individual properties of this instance are encapsulated in an assembly context. Component types, however, can be used to express that components provide or require interfaces that are also provided or required by other components. This makes it possible to reuse components of the same type although they may provide additional interfaces. In short, components specify *which* services are provided and

required, assembly context specify *how* this is done, and component types express which *kind* of components can replace each other.

3.5: There are the provided type and the complete type. They are used in early design stages to specify interfaces of a component as a draft for the component developers. Abstract components can be used within composite structures (e.g., as assemblies within systems) as a placeholder. This placeholder enables an iterative engineering process where architects hand-over the placeholder's specification to component developers and can substitute the placeholder with the actual component once implemented.

3.6: There cannot exist a connector concept which is independent of the used interface and can be instantiated. Assembly connectors are exclusively on the instance level and bound to the interfaces of the components, they connect. No, connectors only provide linking and message passing. Further functionality should be modeled using components.

3.7: In contrast to UML activity diagrams, SEFFs provide a specific loop construct and specific activities which are required for performance prediction for instance.

3.8: In Palladio, the component developer usually only models the inner-component behavior explicitly. Software architects can, however, derive the inter-component behavior from the component assembly, e.g., by tracing the external call actions in the SEFFs.

3.9: In the assembly context the wiring of the components is modeled, while in the deployment context the location of the assembled components (of the assembly context) on hardware is defined. This separation is important allowing to independently design the architecture from the deployment configuration on hardware.

3.10: A software architecture can be defined as the set of architectural design decisions, i.e. design decisions that realize one or more requirements (Jansen et al. 2005).

3.11: If an architectural design decision is not modelled explicitly this can make it difficult to find, follow, and revise the decision. As the architecture of a system usually involves several artefacts it is not always clear where to search for a decision that affects several elements. This can be easier for implementation code. Therefore, many but not all decisions that affect only the implementation and not the architecture of a system do not need to be modelled explicitly if they are properly documented in the code. Nevertheless, a design decision that is only explained in a code comment is hard to retrieve from other artefacts and is usually restricted to text.

3.7.2 Solutions

3.1: See Figure 3.1.

3.2: See Figure 3.2.

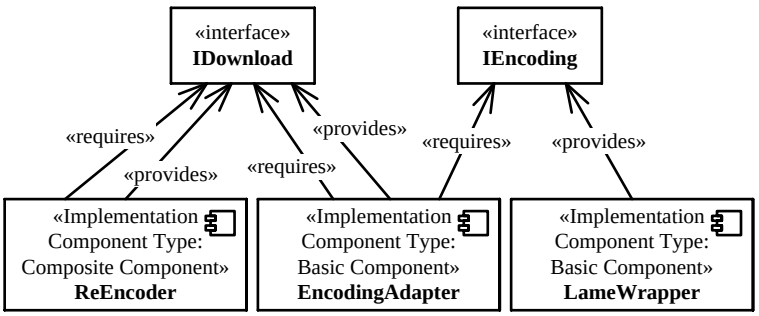

Figure 3.1: Repository Sample Solution

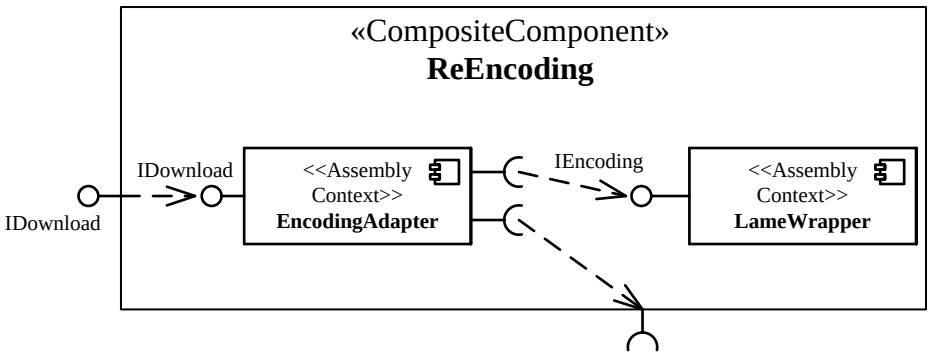

Figure 3.2: Assembly Sample Solution

3.3: See Figure 3.3 for the repository and Figure 3.4 for the system.

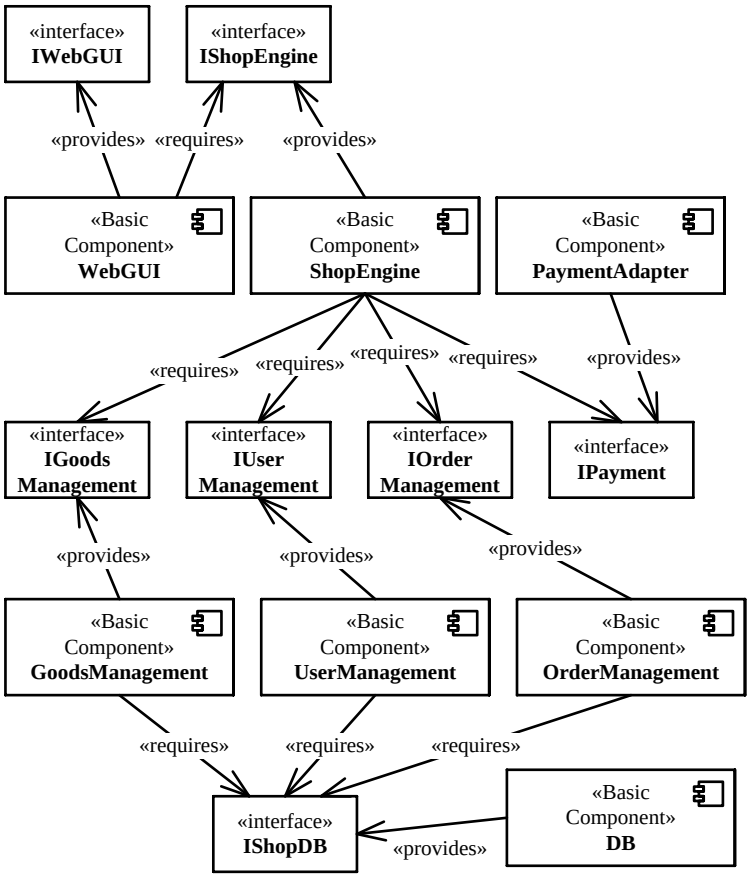

Figure 3.3: Web Shop Repository Sample Solution

3.4: A possible resource environment consists of three resource containers, each representing a server. They are connected via the same linking resource, which specifies its latency and throughput. The resource containers may specify several resources. Which resources should be modeled depends on the types of resource demands, which are expected on that resource container. The resource types most common are: CPU and hard disk. For theses resource types, processing rates and scheduling strategies are specified.

Several deployments of the assembly contexts are possible. An easy one is to deploy every one onto one server. A three tier deployment would place the WebGUI onto one server, the Database onto another and the remaining components (business logic) onto

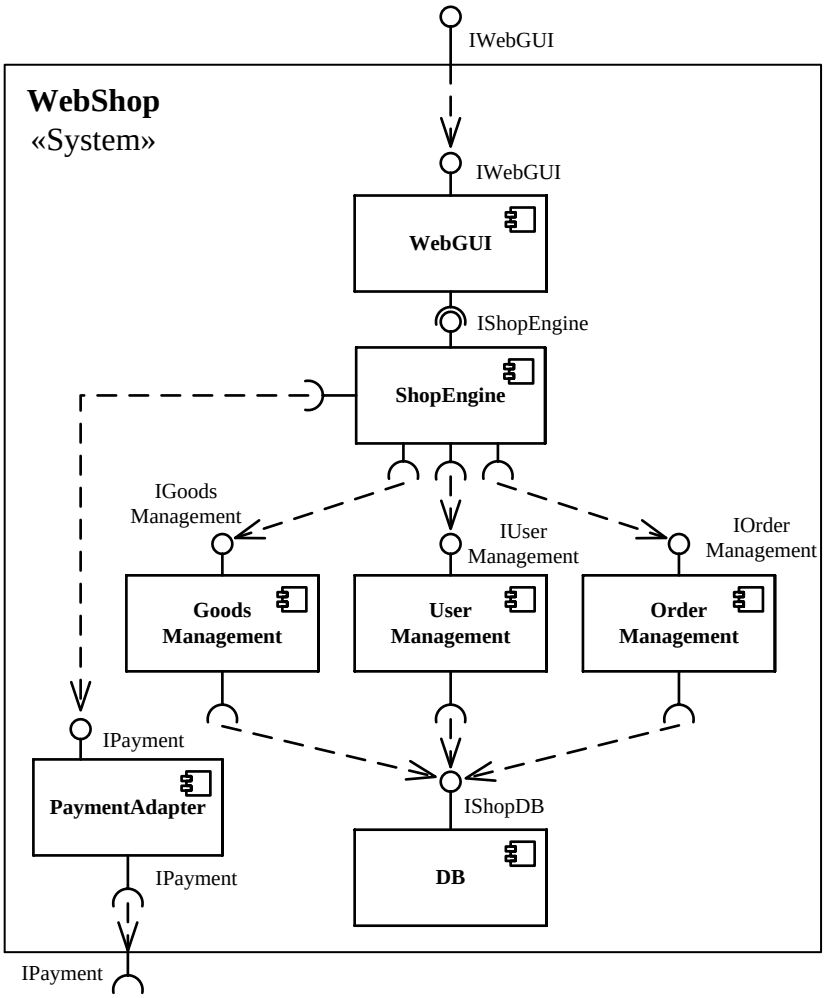

Figure 3.4: Web Shop System Assembly Sample Solution

the third server.

A quantifiable quality metric which can be observed is the response time of user interactions. Which of the two deployments is the better one, depends on the usage of the system. If there are only a few users on the system, the deployment onto one machine is to be preferred, as response times can be lower, as there is no overhead from network communication (it is also cheaper). If there are many users, the three tier deployment may be beneficial, as the load is distributed over three machines, which increases the

system capacity. Which of these alternatives is actually suited can be inspected if the workload is modeled and the system simulated. The outcome of the different deployment models can then be compared.

3.5: See Figure 3.5.

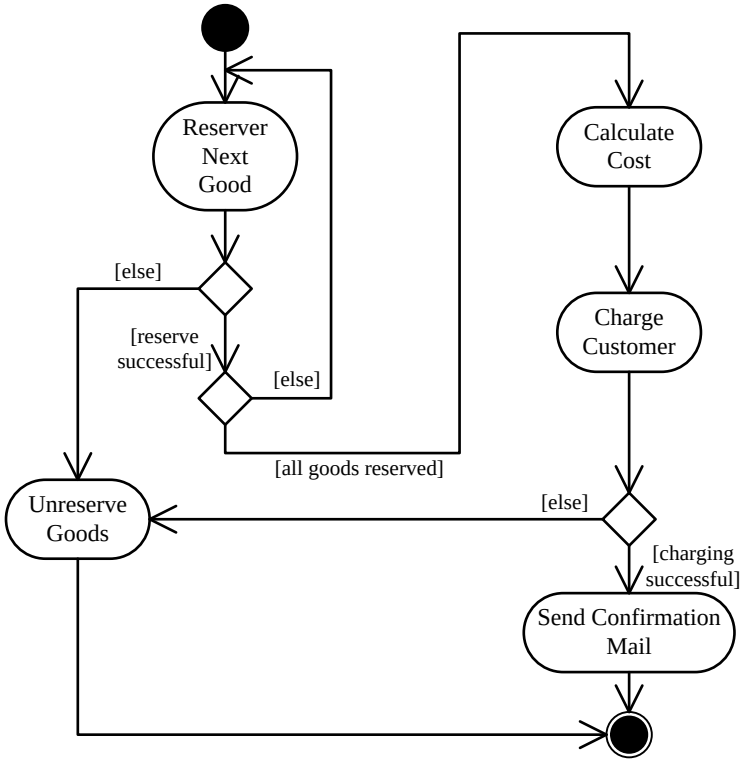

Figure 3.5: Web Shop Activity Diagram Sample Solution

4 Architectural Reuse

Answers and Solutions

4.7.1 Answers

4.1:

- *Architecture style:* An architecture style basically reflects a structure capturing architecture knowledge to be used in a certain problem context. This principle is pursued consistently through the entire software architecture and with the substantial absence of exceptions.

- *Architecture pattern:* Solution for a recurring problem that is applied on the architecture level. The solution is concerned with basic architecture concepts and crosses the boundaries of several architecture elements.

- *Reference architecture:* A reference architecture is designed for a specific domain. It standardises structural elements, their types and relations between these elements. This abstract structure can be seen as a template that can be instantiated for all systems of the domain.

4.2: Styles are more abstract, have more degrees of freedom in the realization and define an overall structure of the system. Architectural patterns are strictly defined, and many different patterns can be used in the same system.

4.3: A correct selection of style and related guidance of subsequent design decisions assures qualities and properties of the to-be built system.

4.4: The type of the style is crucial here. Layered Architecture Style is concerned with logical structural decomposition of a system, while Multi-Tier Architecture Style is concerned with the emphasis on the deployment of the system.

4.5: No, in some cases application of architectural patterns may lead to over-complicated design and even decreased quality of the system (e.g. performance bottlenecks in case of a Facade pattern).

4.6: No, a specific system architecture may omit or re-define some of the components.

4.7.2 Solutions

4.1: You need a concept allowing data collection from the sources, their processing in the cloud and delivery of the results back to the customers. Client-Server Architecture Style allows you the definition of the frontend as a thin client collecting data from the customer, and a server backend responsible for data processing. The communication could be realized via standard TCP/IP protocol. Since your company is regularly building such cloud applications, most likely you would also have at least a company-internal reference architecture available for cloud-based solutions. Thus, you would start your work from investigation if there is such an architecture available. Your would then compare the requirements to your system to the requirements definition of the reference architecture. Finally, you would instantiate the components of the reference architecture with to the required components of the system, and do the gap analysis. Such approach would allow you to reuse knowledge from the previous projects and a fast delivery to your customer.

4.2: Since you need a concept allowing to handle many variations of similar concrete software architectures, building a product line architecture provides you many advantages. SPLs allow you to create a base architecture containing your base functionality every concrete software architecture should contain (e.g. call functionality). This base architecture can be extended by further components according to the product requirements. Further extensions may for instance be a clock or a browser functionality. These extensions can be reused for each instance of the product. Thus, this allows a high level of reuse and a fast time-to-market for the concrete product.

5 Modeling Quality

Answers and Solutions

5.8.1 Answers

5.1: Goal-oriented models are usually smaller and easier to handle than general purpose models. They avoid wasting efforts for understanding and modeling parts of the system, which are irrelevant to the quality goals. Goal-oriented modeling can also help to choose appropriate methods and tools thus further optimizing the whole quality analysis process.

5.2: System stakeholders are often not clear about the quality goals themselves or are unable to formulate them in an unambiguous way. Goals may also change or get refined during the development of a system as more information becomes available. For example, it may be understood that an initial performance goal for a very fast responding system is simply too expensive to implement. Methods to assist the software architect in quality modeling are GQM, QAW, and ATAM.

5.3: If a component developer provided timing specifications or failure probability specifications with a software component, these specifications would be tied to the specific context of the component developer. For example the timing information would be based on the speed of the computer the component developer tested the component, or the failure probabilities would be based on the specific usage profile the component developer assumed during testing. A software architect using the component would use a different resource environment and usage profile, thus the specifications provided by the component developer would not be accurate. The component developer in turn cannot foresee all contexts a component will be used in or all usage profiles of future users. Factors influencing the quality properties of a component are: the underlying resource environment, the implementation of specific algorithms for a given component specification, the usage profile of a component, concurrently executing application, and the state an application or component is in while executing.

5.4: It is not possible, as the usage model is specified by a domain expert and restricted to the information of this role. The domain expert should not tie the usage model to a specific resource environment, which is in the responsibility of the software architect.

5.5: The model construct is called Component Parameter. Component developers can use it to model performance influences due to data stored in a database. For example, it is possible to model the size of a database table, which can influence the execution time of a search query. The software architect needs to override the default values given by a component developer for a specific application context.

5.6: A component developer may only specify reference to abstract resource types in an RDSEFF. These abstract references are resolved into concrete resources (e.g., 2 GHz CPU) once the resource environment model from the system deployer.

5.7: Linking Resources within a resource environment model allow expressing a network latency as well as a processing rate and a failure probability. The Palladio solvers add the latency of requests to their overall execution time and incorporate the failure probability into the overall system failure probability.

5.8.2 Solutions

5.1: The steps to model an RDSEFF are roughly as follows: you need to model an interface and add a service to it in the PCM repository. The respective component needs to be connected to this interface using a Provided Role. Then a new RDSEFF can be created for the service in the repository editor. By double clicking the RDSEFF in the repository editor, the RDSEFF editor is opened. You then need to defined at least a Start Action and a Stop Action. An Internal Action can be added to the RDSEFF and the control flow can be adjusted so that this Internal Start Action follows the Start Action and the is followed by the Stop Action.

5.2: From the user perspective creating a Palladio Usage Model is similar to creating an RDSEFF as in the former exercise. A Palladio Usage Model can be created from scratch and independently from other models. After executing the Palladio Usage Model creation wizard, a simplistic Usage Model is already generated automatically. Add a call to a component service and connect the usage model to the corresponding system model.

5.3: First, you need to add the respective parameter to the interface specification of the service (use the service from exercise 5.1). Locate the interface in the repository model and add a simple parameter of the time int and give it an appropriate name. Now you can reference the parameter from the RDSEFFs for the services. Open the RDSEFF from exercise 5.1 and locate an Internal Action. Add a resource demand to this Internal Action by right-clicking it. A dialog opens to allow entering the parameter dependency. Enter the name of the formerly created parameter, add the suffix ".VALUE" to refer to the parameter value and then specify an arithmetic operation, e.g., myParameter.VALUE ×5. Click OK to close the dialog. Now the Usage Model (e.g., the one from exercise 5.2) needs to be adjusted accordingly. Locate the Entry Level System Call for the respective service and add a Parameter Characterization to it. Specify "myParameter.VALUE = 10". Now

the value 10 can be propagated to the respective component service, so that the resource demand of $10 \times 5 = 50$ work units can be derived.

5.4: You can create a resource environment from scratch using the Palladio editor's wizard. First create the two Resource Containers with included Processing Resources, and specify the properties of the Processing Resources. Create a Linking Resource by selecting the corresponding icon from the tool box. Parametrize the Linking Resource as well. Save the model and start the Palladio editor wizard for the Allocation Model. Select the source System Model which includes the components to be deployed and select the source resource environment previously created. In the graphical editor for the Allocation Model, you can drag components into Resource Containers to specify their deployment.

6 Getting The Data

Answers and Solutions

6.6.1 Answers

6.1: System must be fully implemented and running on representative (or productive) servers. Design time analysis is not possible. Logging overhead must be low to avoid skewing of the measured data.

6.2: Middleware instrumentation does not provide insights into the internals of individual software components.

6.3: The focus of Real User Monitoring is on understanding the user behaviour, while the goal of Application Performance Monitoring is to identify and resolve performance problems during operation as early as possible.

6.4: Real User Monitoring can be used to derive user behaviour, which can be used to create usage models. Application Performance Monitoring provides insights on the application's control flow and resource demands, which can be used to specify service effect specifications.

6.5: Setting up Application Performance Monitoring (APM) entails system instrumentation. Each instrumentation probe comes with an overhead. Thus, when setting up APM the trade-off between level of detail and the monitoring overhead has to be assessed.

6.6: Data collection in virtualized enviroments is difficult because of contention effects. Such contention by other applications leads to response times that are not representative. A possible solution is to measure more fine-grained data in the hypervisor instead of end-to-end times.

6.6.2 Solutions

6.1:

(a) The methods have the following execution times:

- `doWork(2)`: *4 ms*
- `doWork(4)`: *16 ms*
- `doWork(5)`: *25 ms*
- `doWork(10)`: *100 ms*
- `doWork(15)`: *225 ms*
- `methodA()`: *(16 + 10*225) ms = 2266 ms*
- `methodB()`: *(10*29 + 100) ms = 390 ms*
- `methodC()`: *225 ms*
- `methodD()`: *(4 + 25) ms = 29 ms*
- `doService()`: *(2266 + 390) ms = 2656 ms*

Thus, the execution time of method `doService()` is *2656 ms*.

(b) The methods have the following execution times if an instrumentation probe has been injected in each method:

- `doWork(2)`: *(10 + 4) ms = 14 ms*
- `doWork(4)`: *(10 + 16) ms = 26 ms*
- `doWork(5)`: *(10 + 25) ms = 35 ms*
- `doWork(10)`: *(10 + 100) ms = 110 ms*
- `doWork(15)`: *(10 + 225) ms = 235 ms*
- `methodA()`: *(10 + 26 + 10*245) ms = 2486 ms*
- `methodB()`: *(10 + 10*59 + 110) ms = 710 ms*
- `methodC()`: *(10 + 235) ms = 245 ms*
- `methodD()`: *(10 + 14 + 35) ms = 59 ms*
- `doService()`: *(10 + 2486 + 710) ms = 3206 ms*

Now, the execution time of method `doService()` is *3206 ms*. The instrumentation code adds an overhead of *(3206 - 2656) ms = 550 ms* which is *(550 ms / 2656 ms) = 20.7%* of the pure execution time of method `doService()`.

(c) Instrumenting all methods except the method `doWork(n)`, yields a monitoring overhead less than *10%*.

- `doWork(2)`: *4 ms*
- `doWork(4)`: *16 ms*
- `doWork(5)`: *25 ms*
- `doWork(10)`: *100 ms*

- `doWork(15)`: *225 ms*
- `methodA()`: *(10 + 16 + 10*235) ms = 2376 ms*
- `methodB()`: *(10 + 10*39 + 100) ms = 500 ms*
- `methodC()`: *(10 + 225) ms = 235 ms*
- `methodD()`: *(10 + 4 + 25) ms = 39 ms*
- `doService()`: *(10 + 2376 + 500) ms = 2886 ms*

Thus, the monitoring overhead of this instrumentation is *(2886 - 2656) ms / 2656 ms = 8.66%*.

6.2:

(a) The components use two different resources (CPU and I/O). Both resources significantly contribute to the observed response times (as indicating by the resource utilization level). It is not possible to determine how much time was spent at which resource solely based on the observed response times.

(b) The throughput at each component is equal to the system throughput: $\lambda_A = \lambda_B = \lambda_C = \lambda$ The residence times at each component are (excluding time spend by waiting for other components): $T_A = 58.33ms$, $T_B = 175ms$ $T_C = 71.43ms$. You can then calculate the per-class utilizations using Equation 6.1: $U_A^{cpu} = 0.2$, $U_A^{i/o} = 0.1$, $U_B^{cpu} = 0.6$, $U_B^{i/o} = 0.3$, $U_C^{cpu} = 0.3$, $U_C^{i/o} = 0.5$. Using the Service Demand Law, you finally can determine the resource demands: $D_A^{cpu} = 10ms$, $D_A^{i/o} = 5ms$, $D_B^{cpu} = 30ms$, $D_B^{i/o} = 15ms$, $D_C^{cpu} = 15ms$, $D_C^{i/o} = 25ms$.

$$U_{i,c} = U_i \cdot \frac{R_c \cdot \lambda_c}{\sum_{d=1}^{C} R_d \cdot \lambda_d}. \tag{6.1}$$

(c) If the system would not fulfill this relationship, the assumption underlying Equation 6.1, which requires the observed response time to be proportional to the resource demand, does not hold because there are several resources contributing to the observed response time. More sophisticated estimation approaches are necessary in these situations (e.g., using optimization techniques (Liu et al. 2006), or Kalman filters (Zheng et al. 2008).

7 Answering Design Questions

Answers and Solutions

7.5.1 Answers

7.1: Questions on sizing, scalability, and load balancing are typical design questions related to performance. Questions on fault tolerance mechanisms, redundancy, design diversity, effect of physical failure, and directing quality assurance effort are typical design questions related to reliability. General questions, such as finding optimal configurations, comparing possible design alternatives, and extending legacy software systems potentially affect all quality characteristics.

7.2: Approximately 70% of the requests are completed in less than 200 seconds.

7.3: This question is tricky to answer, because the histogram only approximates the predicted response time distribution. Thus, it is impossible to know what the most likely response time is, but we can only reason in intervals. What is visible from the histogram, is, that the response time is more likely in the interval [0,20] seconds than in any other interval of width 20 seconds, because this interval is the bucket with the highest probability. The area of the histogram bars approximates the probability of the respective interval. For more detailed comparisons of the likelihood of intervals, however, it is easier to interpret the CDF.

7.4: Figure 7.1 shows the *assembly contexts* of the extended Media Store architecture. Each assembly contexts refers to a *component type* in the repository (not shown in the figure). Often, the assembly context has the same name as the component type it refers to, if this component type is used just once in the system. For example, the assembly context MediaManagement in Figure 7.1 refers to the component type MediaManagement in the repository. In this example, however, there are two assembly contexts that assemble the AudioWatermarking component type, namely the assembly context called AudioWatermarkingReplica1and the assembly context called AudioWatermarkingReplica2. Figure 7.2 shows the *allocation contexts* of this Media Store architecture. There is one allocation context for each assembly context, which is why the allocation context also usually has

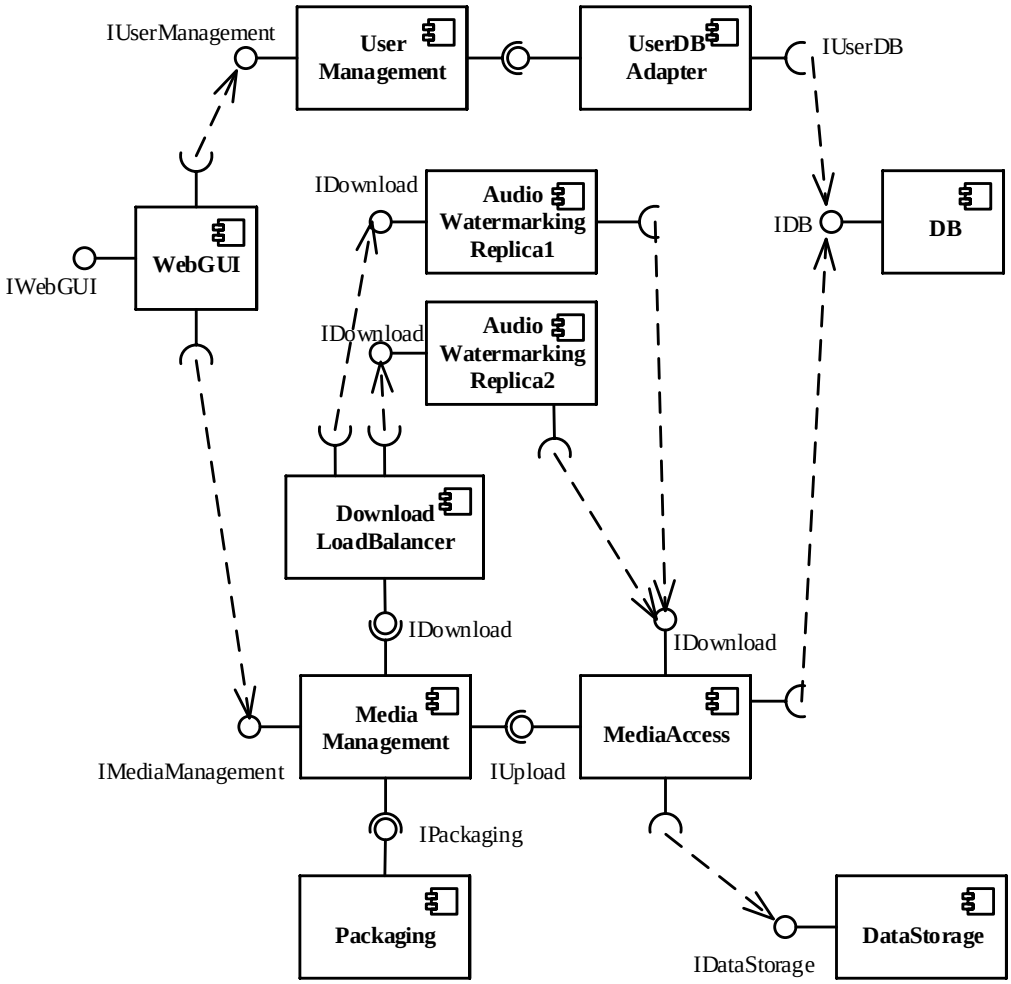

Figure 7.1: The Media Store System after Adding a Load Balancer

the same name (e.g. allocation context AudioWatermarkingReplica2 in Figure 7.2) as the assembly context (e.g. assembly context AudioWatermarkingReplica2 in Figure 7.1.

7.5: Software failure potentials, Hardware Failure Potentials, Network Failure Potentials, and System-external Failure Potentials.

7.6: Software failure potentials are modeled as failure probabilities of *internal actions*.

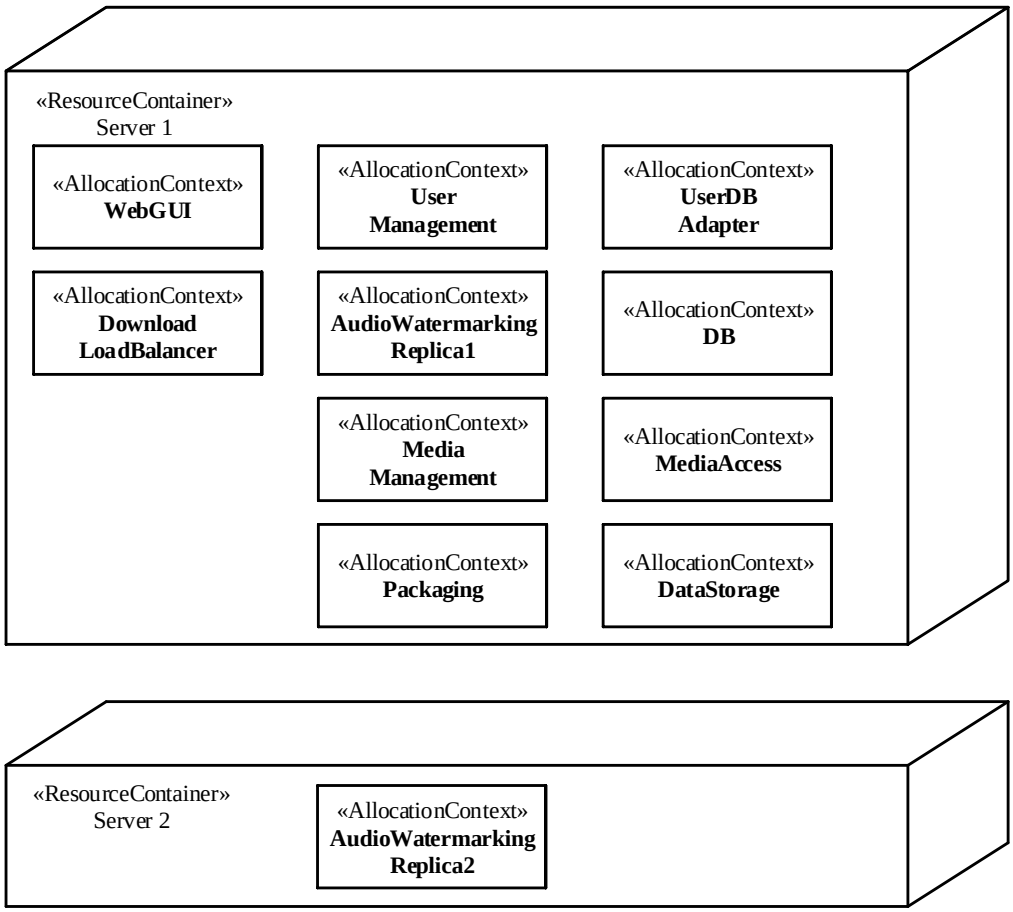

Figure 7.2: The Media Store *Allocation* after Adding a Load Balancer

Hardware failure potentials are annotated to *processing resources* as mean time to failure and mean time to repair, from which an availability value can be derived. Network failure potentials are annotated to *linking resources*. Finally, System-external failure potentials can be annotated to *required roles* of the *system*.

7.7: The analysis results show the contribution of each service or hardware resource to the overall failure probability. The largest probability here has the highest failure potential under the given usage scenario.

7.5.2 Solutions

7.1: The maximum number of users that can be served is exceeded if the response times are increasing over simulation time. The CPU becomes the bottleneck as it is the only resource in the system. One solution can be to increase the capacity of the resource that has the highest utilization to resolve the performance bottleneck.

7.2: One option to further improve the response time is to parallelize the watermarking of multiple files. For example, as each server has two cores, the AudioWatermarking component could be updated to start two threads to watermark half of the files each. Furthermore, the load balancer could split a request with multiple files and send a request for half of the files to AudioWatermarkingReplica1 and a request with the other half of the files to AudioWatermarkingReplica2. Even if this causes the same overall load in the system (or rather even has some additional overhead for calling more components), this will better use all available resources for a given request and thus reduce response time while keeping the same utilization.

7.3: Solve the Palladio model of the design alternative for reliability. The results show that the probability of failure has increased. The two reasons are that (1) additional components with failure potential have been added to the system and are used in the usage scenario and (2) an additional server with failure potential has been added to the system.

8 Under The Hood

Answers and Solutions

8.6.1 Answers

8.1: The different tools differ in their range of function, result accuracy and analysis speed.

8.2: Typically, the simulator has a workload generator, simulated users, simulated system services, and simulated resources.

8.3: The confidence of a single simulation run might be increased by running for a longer simulation time, however, simulation runs should also be repeated using different random number seeds.

8.4: In Palladio pseudo-random numbers are used to ensure reproducibility for simulation runs.

8.5: While ProtoCom is used for the generation of performance prototypes that can be used to calibrate the models on real hardware, both SimuCom and EventSim are simulators using only simulated hardware resources.

8.6: Confidence levels can be used to decide a simulation has run sufficiently long. Background: The longer simulations run, the higher is the confidence in results becomes because even unlikely execution paths tend to be executed after a long simulation time. Confidence checks are readily built into the Palladio-Bench.

8.6.2 Solutions

8.1: Imagine you are asked to build an automated design space exploration tool for Palladio models that works similar to PerOpertyx. Let us assume your tool is already capable of creating new architectural candidates, but these candidates cannot yet be evaluated and compared to each other automatically. To avoid reinventing the wheel, you plan to reuse an existing Palladio analysis tool to evaluate candidates. Consider the specific requirements of this scenario and decide for a suitable analysis tool.

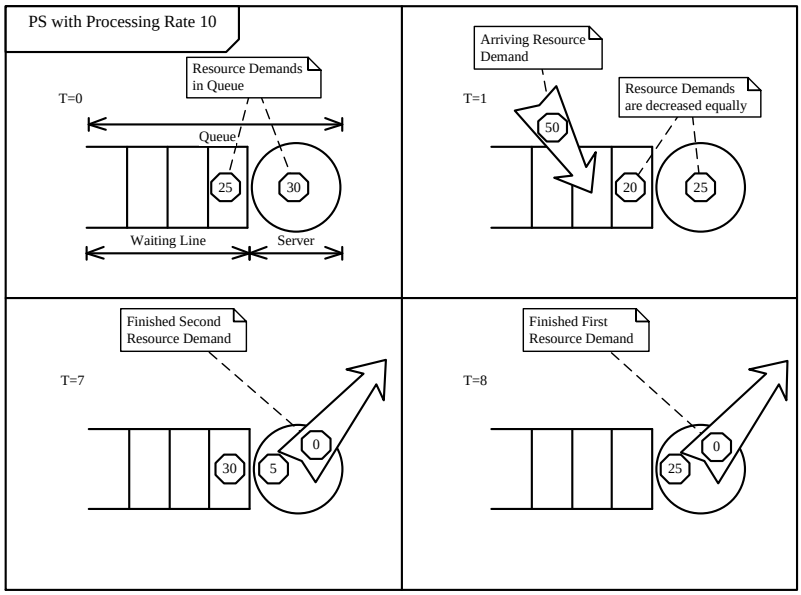

Figure 8.1: Resource demand processing using PS scheduling strategy.

8.2: Consider the scheduling examples from Figure 8.1 and Figure 8.2. If in time step $t = 1$ an additional resource demand of 5 arrives right after the demand of 50, when can we expect this job to be finished under a processor sharing policy? When is the newly arriving job finished when using a first-come, first-served policy instead?

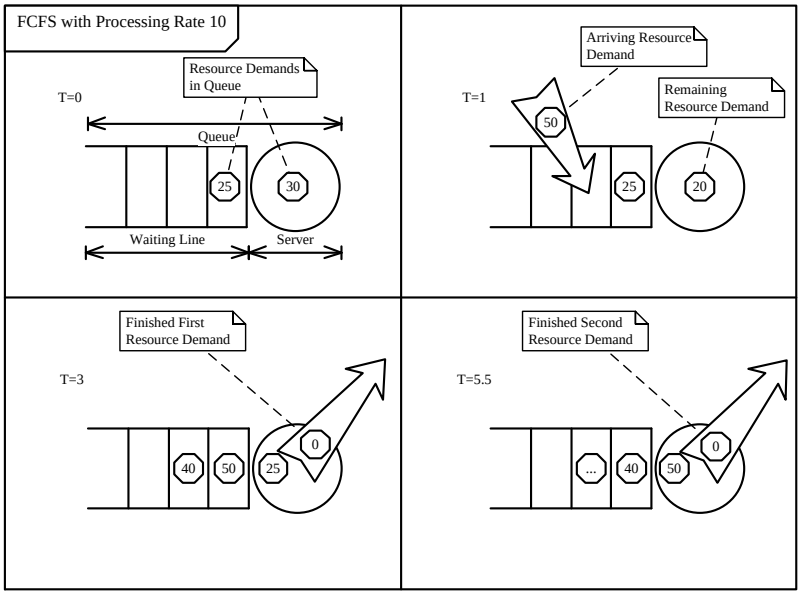

Figure 8.2: Resource demand processing using FCFS scheduling strategy.

9 Software Engineering Processes

Answers and Solutions

9.4.1 Answers

9.1: Palladio helps during planning, design, implementation, and test. During planing Palladio serves in reducing estimation risks (e.g. it is realistic to achieve certain quality goals in a given setting?). During design Palladio assists in the constructive creation and evaluation of a software design. Implementation is supported by forward engineering and code stub generation capabilities. Testing is partially supported by comparing predicted and as-is values (e.g. deviations between predicted and as-is values pointing to implementation deficiencies or non-understood design effects).

9.2: Existing assets need to be integrated and incorporated into the specification to ensure a seamless integration of system parts. Without considering existing assets early, the risk of a late failing integration increases.

9.3: Reusable components can be used in different contexts. Thus they can also be used in different design alternatives which come up during evolution iterations. For example, a universal caching component can be used in different iterations of which one might foresee a single-level caching and another iteration could foresee multi-level caching. All iterations can share the same reusable component.

9.4.2 Solutions

9.1: Planning: Define the project scope, exclude for example a grade management from the system scope. Capture the expected amount of planned users. Definition: Capture functional interfaces. Design: Identify and define components, e.g. the front end for adding courses and a component to calculate the maximum amount of assigned courses per student and plan the deployment (e.g. how to distribute the components to servers). Testing: Check the maximum amount of supported concurrent users of the front end component. Maintenance: Increase the amount of supported concurrent users by adding further server hardware and tweaking core algorithms.

10 Relation to Requirements Engineering

Answers and Solutions

10.4.1 Answers

10.1: The different types of requirements for software systems are shown in Figure 10.1. The main driver for architecture design are quality requirements. Still, all requirements may influence the architecture design.

10.2: A quality requirement is a requirement that pertains to a quality concern of the system to be, such as performance, security, or maintainability. In contrast, functional requirements are requirements concerning primarily the expected behavior in terms of reaction to given input stimuli and the functions and data required for processing the stimuli and producing the reaction. Non-functional requirements are a superset of both quality requirements and constraints. Also note that the terminology is not completely agreed upon in the requirements engineering community, so you may encounter different definitions in different sources. In particular, when working on a project, be aware that co-workers or stakeholders might have a different understanding of what a non-functional requirement (and a functional requirement) is.

10.3: Quality characteristics are the highest level of distinguishing between different quality concerns. Examples of quality characteristics are performance, reliability, maintainability, and usability. Depending on the source, different names are used for the different quality characteristics (such as efficiency or performance). Quality measures, in contrast, name more specific and measurable properties of a system. For example, the quality measure *response time* refers to the response time property of a system.

10.4: Requirements and architecture are related in two ways: (1) Requirements drive architectural design, which means that the software architect systematically designs the architecture to fulfill requirements. (2) Architectural design drives requirements engineering, which means that architecture design can provide various ways of feedback into the requirements engineering process.

10.5: Frame: the architecture frames the achievable quality by providing insight into the associated difficulty, risk, and cost. Constrain: design decisions in the architecture may

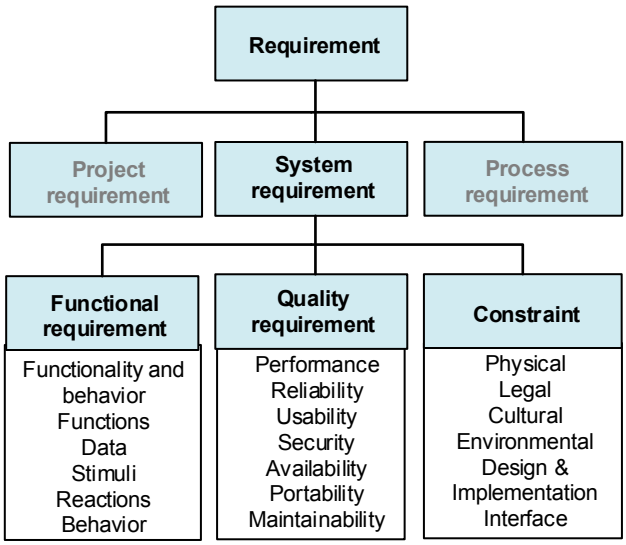

Figure 10.1: Types of Requirements (adapted from (Glinz 2007) according to (Glinz 2011))

limit or even exclude other possible requirements. Inspire: Architectural solutions may inspire new possibilities and functionality and thus new requirements.

10.6: Palladio can be used to analyze performance, reliability, and costs.

10.4.2 Solutions

10.1:

A. Kind: quality (usability), representation: qualitative

B. Kind: quality (performance), representation: quantitative

C. Kind: functional, representation: operational

D. Kind: quality (usability), representation: operational

11 Relation to Requirements Engineering

Answers and Solutions

11.4.1 Answers

11.1: *Provided roles* are mapped to port classes which implement the interfaces associated by the role.

11.2: Palladio allows for providing (and requiring) the **same** interface multiple time be the **same** component (imagine a component which has clients of the same type connected but needs to distinguish them). To cope with the potential ambiguity, **separate** port classes reflect the individual provided roles.

11.3: The forward engineering purposes are i) simulation of performance, ii) template generation for performance prediction, and iii) template generation to reduce implementation effort. The advantage of i) and ii) is the possiblity to The advantage of iii) is that manual effort of mapping the architecture to code is avoided and that the mapping from architecture to code is as the intendend mapping.

11.4: SoMoX is able to rapidly generate a software architecture from source code. This architecture can help, for instance, by i) planing tasks for developer teams, and ii) predicting the performance of the software system (if combined with the behavior detection Beagle).

11.5: Static analysis captures semantics of source code by its static semantic, while dynamic analysis actually executes source code. In order to execute source code for dynamic analysis, a running resource environment, a ready configuration, representative data (e.g. in a database), and suitable user input (either manually executed inputs, test cases or other robots) are required. Hence, in practice dynamic analysis requires more effort to setup the analysis, e.g. for preparing a distributed system.

11.6: By means of monitoring, source code can be precisely observed for certain use cases. The actual source code execution becomes visible and true performance and reliability issue might be observed. Hence, performance properties are much easier to grap using

dynamic analysis while static analysis is suitable for the static architecture. Dynamic analysis can easily limit its scope by just executing use cases of interest (e.g. the effects of inserting a single record in the Media Store frontend).

11.7: Interface violations (i.e. class-level communication of different components which bypasses component interfaces) harms the quality of component architectures. If interfaces are bypassed, programmers can hardly rely on certain communication paths which increases effort for understanding and maintaining systems. Exchanging a component's implementation becomes labour-intensive manual work since a possibly bypassing communication has to be individually handled.

11.8: ArchiMetrix supports reverse engineering by finding deficiencies in the source code. If it is executed iteratively the result of the reverse engineering will be improved. Since ArchiMetrix points out the deficiencies automatically, but not automatically repairs the source code it can be considered as a semi-automatically approach.

11.4.2 Solutions

11.1: The solution for Java source code is provided on the accompanying book's website.

11.2: The classes that can communicate directly to each other are: MediaAdapterImpl and DB Manager as well as DB Manager and Audio. All other classes communicate which each other through interfaces.

11.3: A screencast how to execute SimuCom and ProtoCom and the result of the execution is provided on the accompanying book's website.

11.4: Palladio currently supports mappings to POJOs, EJB, and different Cloud environments. The main difference, e.g., between POJOs and the other concepts is that POJOs are not using any other concept other than pure Java.

11 Bibliography

Glinz, Martin. *A Glossary of Requirements Engineering Terminology. Standard Glossary for the Certified Professional for Requirements Engineering (CPRE) Studies and Exam*. Tech. rep. available online. International Requirements Engineering Board, 2011.

– "On Non-Functional Requirements". In: *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*. IEEE Computer Society, 2007, pp. 21–26.

Jansen, Anton and Jan Bosch. "Software Architecture as a Set of Architectural Design Decisions". In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. Ieee, 2005, pp. 109–120.

Liu, Zhen et al. "Parameter inference of queueing models for IT systems using end-to-end measurements". In: *Performance Evaluation* 63.1 (2006), pp. 36–60.

Rhinelander, Richard. "Components have no Interfaces!" In: *Proceedings of the 12th International Workshop on Component Oriented Programming (WCOP 2007)*. Vol. 2007-13. Interne Berichte. Karlsruhe, Germany: Universität Karlsruhe, Fakultät für Informatik, 2007.

Zheng, Tao, C.M. Woodside, and M. Litoiu. "Performance Model Estimation and Tracking Using Optimal Filters". In: *IEEE Transactions on Software Engineering* 34.3 (May 2008), pp. 391–406.