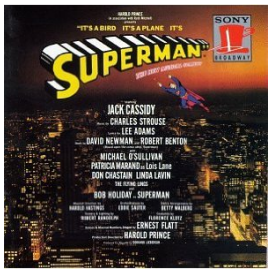
VIEWS AND VIEWPOINTS OF SOFTWARE ARCHITECTURE

BY PANKAJ KAMTHAN

1. INTRODUCTION



It's a Bird...It's a Plane...It's Superman
(Broadway, 1966)

The meaning of things lies not in the things themselves, but in our attitude towards them.
— Antoine de Saint-Exupéry

The notion of software architecture view (or, simply, view) and software architecture viewpoint (or, simply, viewpoint) are critical to understanding software architecture, and are explored in this document.

CONVENTION/NOTATION

The term **'system'** in the following **subsumes** the term **'software system'** as well as the term **'software-intensive system'**, unless otherwise stated.

2. MOTIVATION FOR VIEWS AND VIEWPOINTS

Without views, you end up with a **single, all-encompassing model** that tries (and usually **fails**) to illustrate **all** of the aspects of your system.
— Nick Rozanski and Eoin Woods

The notions of view and viewpoint are motivated by the **software engineering principle** of **separation of concerns** [Ghezzi, Jazayeri, Mandrioli, 2003].

There are a number of compelling reasons for studying views and viewpoints, including the following:

- Proper Delivery and Deployment
- Understanding Software Systems via Understanding Human Systems

2.1. PROPER DELIVERY AND DEPLOYMENT

The sketch shown in Figure 1—using amusement as basis and inspired by [Brittan, 1980]—has become an **exemplar** to illustrate the significance of **several aspects of software engineering**: the importance of **proper communication among stakeholders**, both individually and collectively; the importance of **clarity in software requirements**; the importance of **prototyping and feedback**; the importance of **documentation**; and so on.

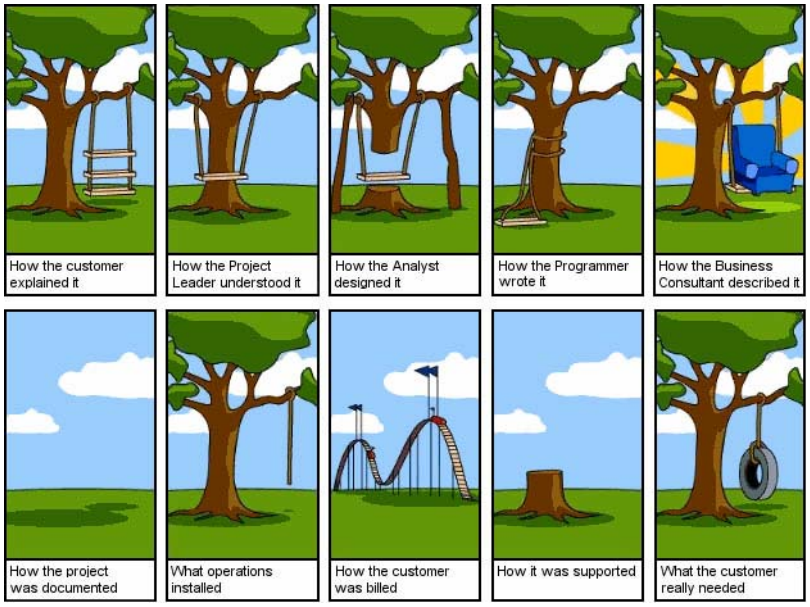


Figure 1. From customer's aspiration to engineer's realization. (Source: How Projects Really Work[1].)

It is evident that in this sketch, the **customer, project leader, analyst, programmer, and business consultant** had **different views and different viewpoints** of the **same thing**.

[1] URL: http://www.projectcartoon.com/ .

2.2. UNDERSTANDING SOFTWARE SYSTEMS VIA UNDERSTANDING HUMAN SYSTEMS

UNDERSTANDING HUMAN SYSTEMS

The human body is a **large and complex** system. It has a number of **organ systems**, as shown in Figure 2, each from a different view (or perspective), from a different viewpoint, addressing different concerns, by different stakeholders (say, doctors).

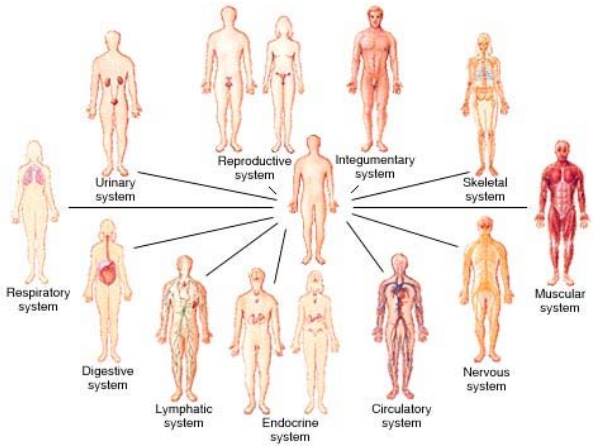


Figure 2. The different systems in a human body. (Source: Google Images.)

For example, **a cardiologist, neurologist, and dermatologist** have a **different view of the structure** of human body; **a kinesiologist and a psychiatrist** have a **different view of the behavior** of human body.

A **general practitioner**, as the name implies, has a **general view** of the human body. A general view is of limited use in non-trivial medical situations. In other words, a single view may not be sufficient.

UNDERSTANDING SOFTWARE SYSTEMS

In software engineering, the interest is in **large and complex** systems[2]. It is necessary for the stakeholders to understand such systems.

However, the **structure** of most software systems is far **too large** and their **behavior** far **too complex** to be grasped **all at once by a single person**.

[2] There are different 'measures' of size, different 'meanings' of 'large', and different kinds of complexities, including structural, behavioral, and communicational, complexity. However, the exactness is not relevant for the discussion or the argument.

Therefore, it is preferable to identify, restrict, and **focus** the attention on one aspect (or a small number of aspects of) the system.

3. VIEWS, VIEWPOINTS, AND SOFTWARE ARCHITECTURE DESCRIPTION

Figure 3 shows the relationships between **stakeholders, concerns, software architecture description, views, and viewpoints**.

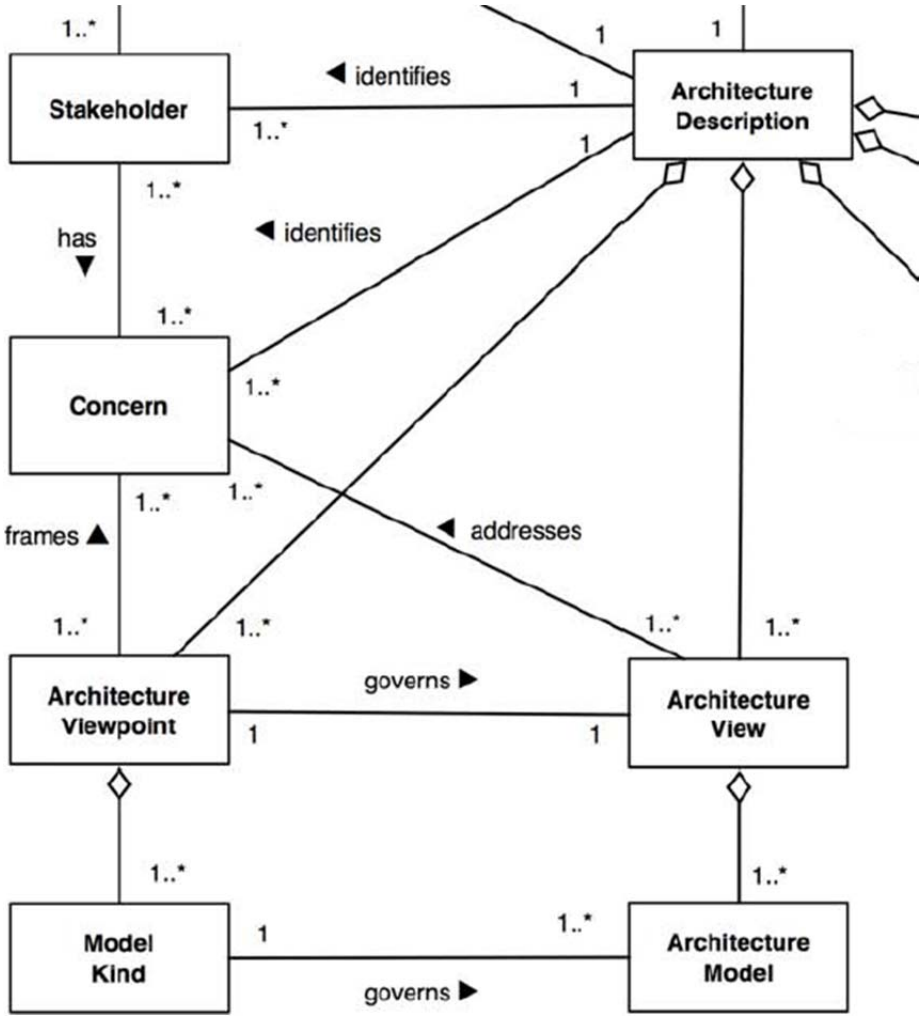


Figure 3. The placement of view and viewpoint in the conceptual model of architectural description. (Source: [ISO/IEC/IEEE, 2011].)

OBSERVATIONS

- The software architecture needs to address **diverse competing interests** that emanate from **different stakeholders**. The **selection of viewpoints** is typically based on **consideration and concerns of the stakeholders of the software architecture**.

- A view and a viewpoint are both part of software architecture description.

- A viewpoint **frames** a concern in the sense of highlighting it.

- The software architecture description is usually based on **more than one viewpoint**.

- A view is composed of **one or more models**.

4. DEFINITION OF VIEW

There are number of definitions of view, including the following:

Definition [View] [IEEE, 2000]. A representation of a system from the perspective of a **related set of concerns**.

Definition [View] [Wikipedia]. A specification for the system at a **particular level of abstraction** from a given viewpoint.

A view is a **projection** of a system. It follows from the definition of view that there can be **multiple projections** of a system.

4.1. EXAMPLES OF VIEWS

There are a **number of possible views** of a given software architecture. Indeed, the collection of views could be considered as software architectural knowledge [Babar, Dingsøyr, Lago, Vliet, 2009, Section 2.2]. The following are some **possible examples**:

- Product Manager View
- System Engineer View
- Requirements Engineer View
- User View
- Implementer View (Programmer View, Database Engineer View)
- Tester View

5. DEFINITION OF VIEWPOINT

There are number of definitions of viewpoint, including the following:

Definition [Viewpoint] [IEEE, 2000]. A specification of the **conventions** for **constructing and using** a view.

5.1. EXAMPLES OF VIEWPOINTS

There are a **number of possible viewpoints** of a given software architecture. The following are some **common examples**:

- Commerce Viewpoint [Hilliard, 2001]
- Requirements Viewpoint [May, 2005]
- Allocation Viewpoint [Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord, Stafford, 2010]

6. THE RELATIONSHIP BETWEEN VIEWPOINT AND VIEW

The relationship between viewpoint and view is **symbiotic**.

A view **conforms** to a viewpoint. For example, Use Case View corresponds to the Requirements Viewpoint.

According to the ISO/IEC/IEEE 42010 Frequently Asked Questions[3]:

> An architecture viewpoint **documents** the **conventions for constructing, interpreting and analyzing** a particular kind of **view**.

6.1. DIFFERENCE BETWEEN VIEWPOINT AND VIEW

According to [ISO/IEC/IEEE, 2011]:

> A viewpoint is a way of looking at systems.
>
> A view is the **result** of **applying a viewpoint** to a particular system-of-interest.

[3] http://www.iso-architecture.org/ieee-1471/faq.html .

According to the ISO/IEC/IEEE 42010 Frequently Asked Questions[4]:

> **An architecture viewpoint is a way of looking at a system.**
>
> **An architecture view is what you see when looking at a system from a chosen viewpoint.**

REMARKS

The word 'see' is being used in the sense of **perception**, not sight. For example, it should be possible for a person who is **visually disabled** to be able to 'see' a system.

There are a couple of analogies that can highlight the **differences** between view and viewpoint [ISO/IEC/IEEE, 2011].

THE RELATIONSHIP BETWEEN VIEWPOINT AND VIEW: ANALOGY 1

In programming languages, a program is one instance of applying a programming language to a specific problem.

In a similar manner, a viewpoint specifies the conventions (such as notations, languages, and types of models) for constructing a certain kind of view. That viewpoint can be applied to **many** systems. Each view is one such application.

In short, in the style of C++:

```
view : viewpoint :: program : programming language
```

THE RELATIONSHIP BETWEEN VIEWPOINT AND VIEW: ANALOGY 2

In **cartography**, a legend defines the conventions used in preparing a map (such as its scale, colors, and other symbology) to **aid** readers in interpreting that map **as intended**. To **be expressive** and to **be interpreted correctly**, every map should have a legend, as shown in Figure 4.

[4] URL: http://www.iso-architecture.org/ieee-1471/faq.html .

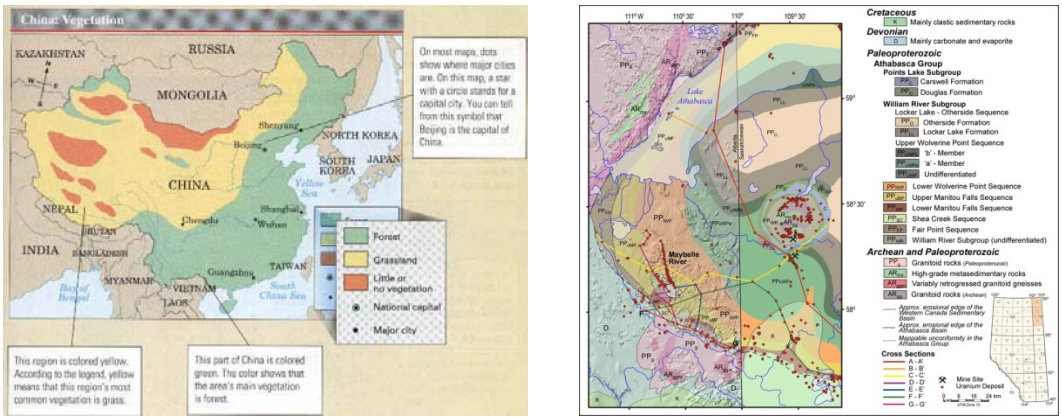


Figure 4. A natural vegetation map and a geological map of China. (Source: Google Images.)

In short:

`view : viewpoint :: map : legend`

In a similar manner, **every** architecture view should have an architecture viewpoint specifying the conventions for interpreting the contents of the view.

7. SOFTWARE ARCHITECTURE VIEWPOINT MODELS

There are **several models for software architecture viewpoints**, each with a **different emphasis**, usually for **different types of software systems**.

For example, the **4+1 View Model of Software Architecture** is focused on **design**, and the **Siemens Four View Model of Software Architecture** is focused on **embedded and real-time software systems**.

REMARKS

- There are **surveys** on software architecture viewpoint models [Garland, Anthony, 2003; May, 2005; Sangwan, 2015, Chapter 6].

- In [Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord, Stafford, 2010, Page 23], a **historical tour** of software architecture viewpoint models is given.

- There are ways for **defining new viewpoints** [Hilliard, 2001; Koning, Vliet, 2006]. In [Shames, Skipper, 2006], **space systems architectures** are studied, and a collection of minimal, normative set of viewpoints is given.

7.1. EXAMPLES OF SOFTWARE ARCHITECTURE VIEWPOINT MODELS

The following are some common software architecture viewpoint models, listed **chronologically**:

- The Reference Model for Open Distributed Processing (RM-ODP)
- **The 4+1 View Model of Software Architecture**
- The Siemens Four View Model of Software Architecture
- The Garland-Anthony View Model of Software Architecture
- The SEI View Model of Software Architecture
- **The Rozanski-Woods View Model of Software Architecture**

8. THE 4+1 VIEW MODEL OF SOFTWARE ARCHITECTURE

The 4+1 View Model of Software Architecture [Kruchten, 1995], as currently defined, has five views:

- Logical View
- Process View
- Implementation View
- Deployment View
- Use Case View

It has four interrelated views **projected upon** (and **tied together**) by one view, as illustrated in Figure 5.

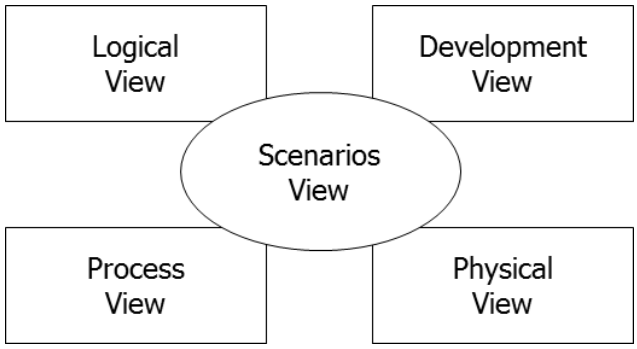


Figure 5. The 4+1 View Model of Software Architecture: Version 1 [Kruchten, **1995**].

The 4+1 View Model of Software Architecture has been adopted by the **Rational Unified Process (RUP)** [Kruchten, 2004]. The 4+1 View Model of Software Architecture has **evolved** to that shown in Figure 6.

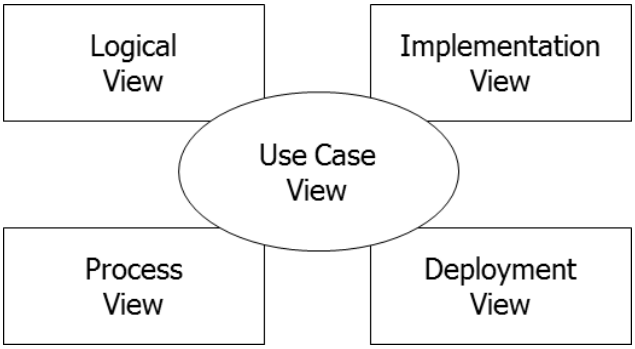


Figure 6. The 4+1 View Model of Software Architecture: Version 2 [Kruchten, **2004**].

LOGICAL VIEW

Purpose: This view focuses on **functionality** inspired by abstractions in the **problem domain**. It is responsible for the **conceptual organization** of layers and high-level functionality of components in each layer.

Stakeholders: End User, Maintainer.

Representation: UML Package Diagram; UML Component Diagram.

Example: An ATM CustomerInterface is related to a Keypad, Display, Speaker, and a ReceiptPrinter component.

PROCESS VIEW

Purpose: This view focuses on (concurrent) tasks performed by software at run-time. It is responsible for **process, threads, and control**.

Stakeholders: Integrators.

Representation: UML Class Diagram (+ Process/Thread Stereotypes); UML Interaction Diagrams.

Example: The ATM process is related to a CustomerInterface thread and a DeviceController thread.

IMPLEMENTATION VIEW

(This is also known as **development view**.)

Purpose: This view focuses on the actual **source code, data files, and executables**.

Stakeholders: Programmers.

Representation: UML Package Diagram; UML Component Diagram; UML Composite Structure Diagram.

Example: The source code for an ATM component.

DEPLOYMENT VIEW

(This is also known as **physical view**.)

Purpose: This view focuses on physical deployment of processes and components to processing nodes. It is responsible for the **physical network configuration (topology)** between nodes.

Stakeholders: Systems Engineers.

Representation: UML Deployment Diagram.

Example: The hardware capabilities of an ATM node.

USE CASE VIEW

Purpose: This view presents **architecturally significant** use cases that help highlight main architectural decisions and choices.

Stakeholders: Requirement Engineers, Testers.

Representation: UML Use Case Diagram; Other.

Example: Improvise.

REMARKS

Remark [Selecting Views]. Even though all **views are optional**, most are recommended. However, **not all views are needed** in all cases. For example, if there is a single program (one 'process'), then the **process view** can be **eliminated**. However, the use case view is **never** eliminated.

Remark [Mobile Health]. The 4+1 View Model of Software Architecture has been used to describe the software architecture of **Emergency Remote Pre-Hospital Assistance (ERPHA)**, a Mobile Application for assisting paramedics while attending victims of traffic accidents [Lavariega, Avila, Gómez-Martínez, 2015].

Remark [Limitations of 4+1 Views]. The 4+1 view model of architecture **does not address** all desirable views of possible interest. For example, in its original definition, there is **no data view**[5]. Furthermore, it also **does not** address feasibility, maintainability, usability, or safety concerns.

9. THE ROZANSKI-WOODS VIEW MODEL OF SOFTWARE ARCHITECTURE

The Rozanski-Woods View Model of Software Architecture [Rozanski, Woods, 2005] has six views:

- Functional View
- Information View
- Concurrency View
- Development View
- Deployment View
- Operational View

FUNCTIONAL

Purpose: This view describes the system's functional elements, their responsibilities, interfaces, and primary interactions.

This view is the **cornerstone** of most architecture designs, and is often of **interest to many stakeholders**. It **drives** the shape of other system structures such as the information structure, concurrency structure, deployment structure, and so on.

[5] This view, if included, would focus on a **persistent** data schema.

It also has a significant impact on the system's quality attributes such as its ability to change, its ability to be secured, and its runtime performance.

Concerns: Functional Capabilities, External Interfaces, Internal Structure, Design Philosophy.

Stakeholders: Acquirers, Assessors, Communicators, Developers, System Administrators, Testers, Users.

Representation: UML Component Diagram; Other (**Boxes-and-Lines Diagram**).

Example: Figure 7 illustrates a UML Component Diagram for a shopping system, **WebShop** [Rozanski, Woods, 2005]. It is **annotated** by UML Note, and **extended** via stereotypes and constraints.

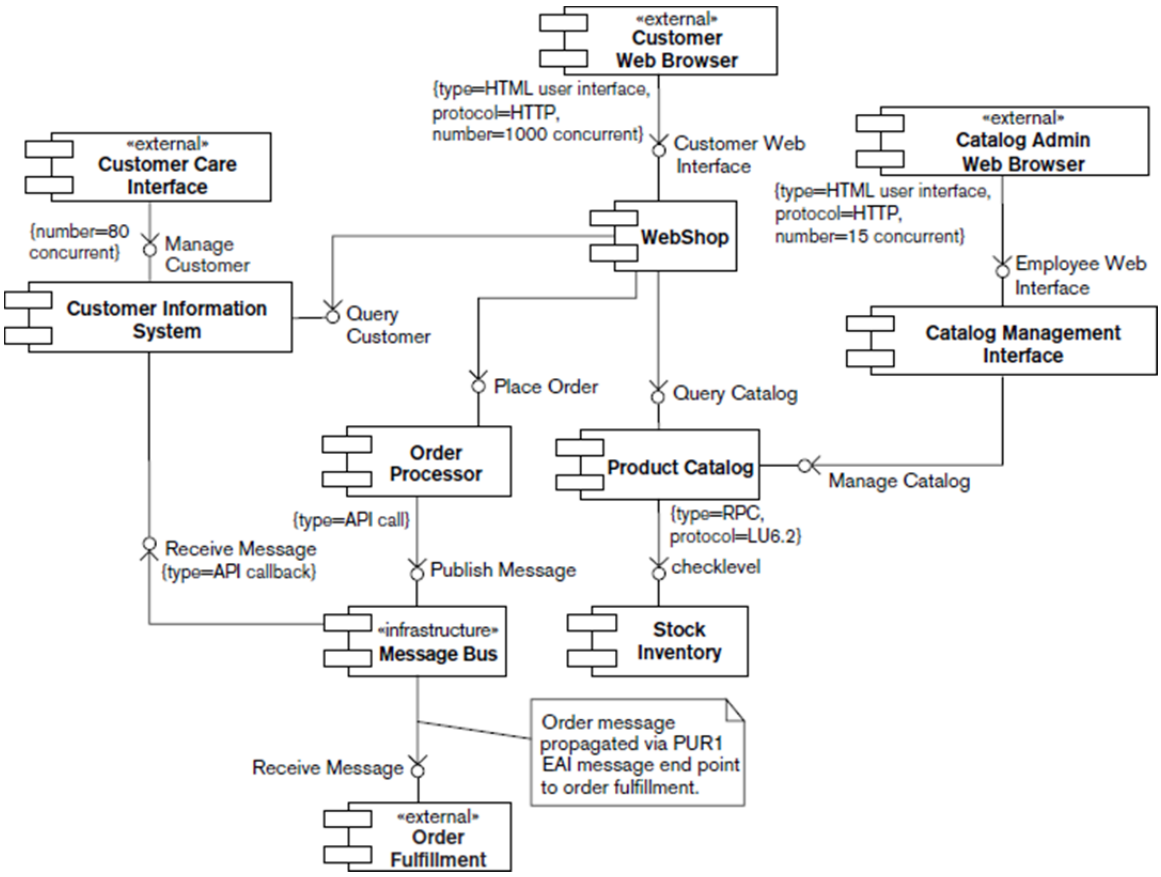


Figure 7. A UML Component Diagram for WebShop. (Source: [Rozanski, Woods, 2005].)

Figure 8 illustrates a UML Component Diagram for a medical system, **Diabetes Share System (DSS)**.

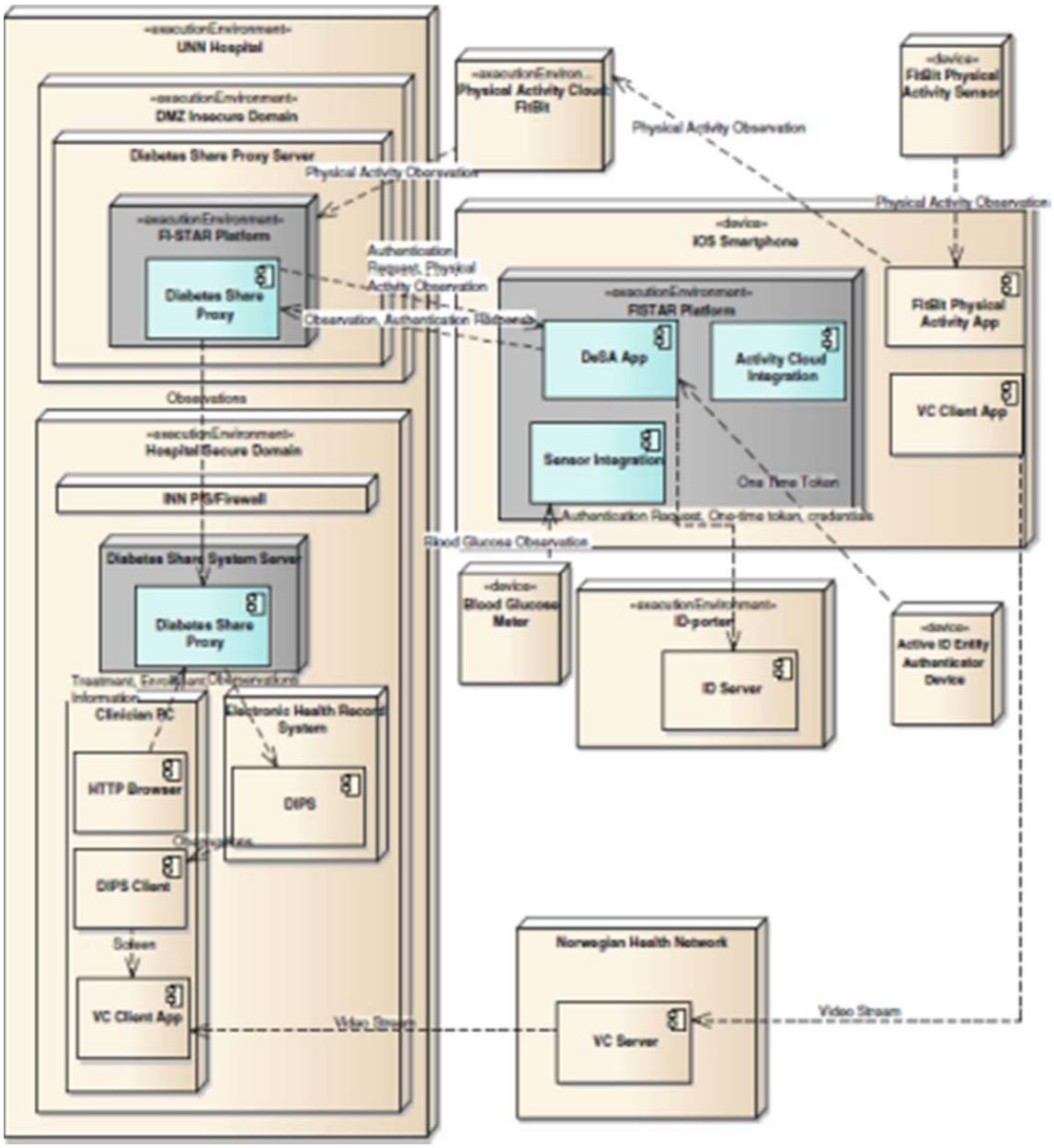


Figure 8. A UML Component Diagram for Diabetes Share System (DSS). (Source: [Brost, Hoffmann, 2015].)

INFORMATION

Purpose: This view describes the way that the architecture stores, manipulates, manages, and distributes information.

The ultimate purpose of virtually any computer system is to **manipulate information in some form**, and this viewpoint develops a complete but high-level view of **static data structure and information flow**.

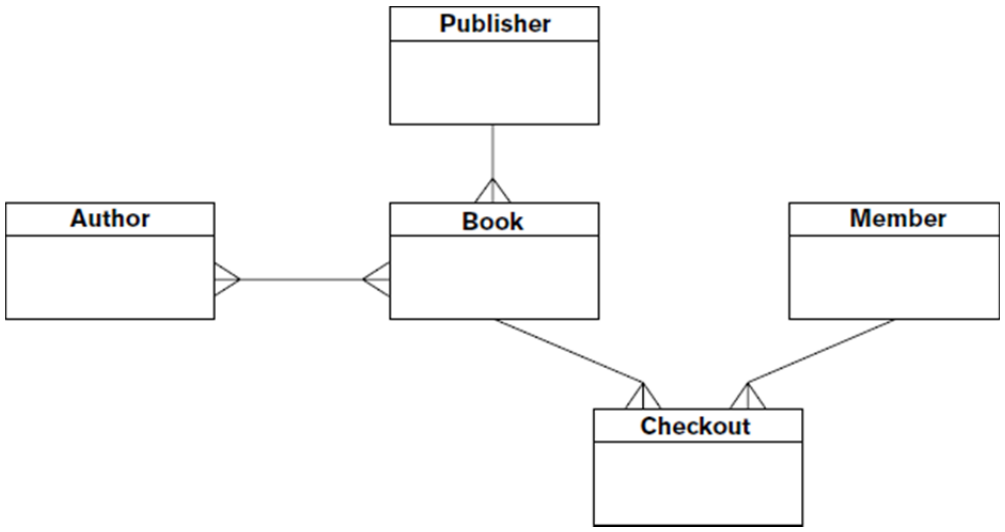
The objective of this analysis is to answer the significant questions around content, structure, ownership, latency, references, and data migration.

Concerns: Information Structure and Content; Information Flow; Data Ownership; Timeliness, Latency, and Age; References and Mappings; Transaction Management and Recovery; Data Quality; Data Volumes; Archives and Data Retention; Regulation.

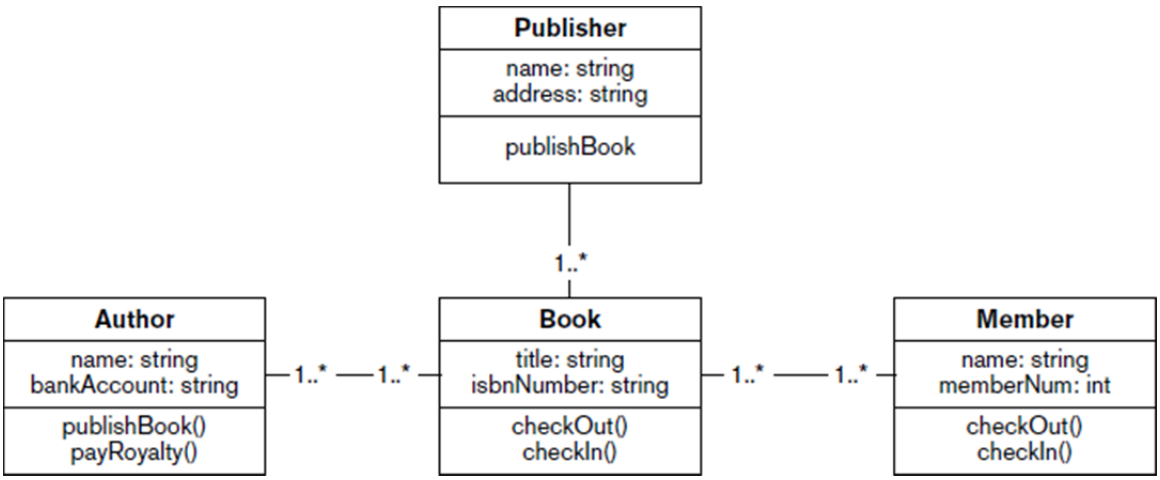
Stakeholders: Users, Acquirers, Developers, Maintainers.

Representation: ER Diagram; UML Class Diagram; UML State Machine Diagram; Other (Box-and-Lines Diagram).

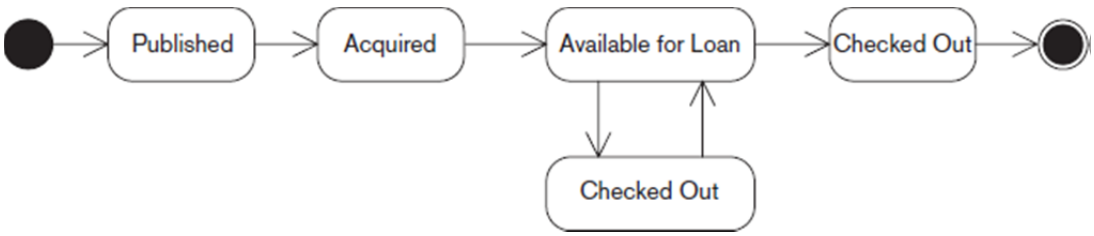
Example: Figure 9 illustrates an ER diagram, UML Class Diagram, and a UML State Machine Diagram, respectively, for a **library system**.



(a)



(b)



(c)

Figure 9. The different diagram types, including, but **not** limited to, UML diagram types, that illustrate information view of a library system. (Source: [Rozanski, Woods, 2005].)

REMARKS

- To **avoid** treading into micro-architecture design, the details of operations and properties in classes in Figure 5(b) should be kept to a minimum.

- The second **`Checked Out`** state in Figure 5(c) should be **relabeled** to**, say**, **`Out of Commission`**.

CONCURRENCY

Purpose: This view describes the concurrency structure of the system, and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is **coordinated and controlled**.

This entails the creation of models that show the **process and thread structures** that the system will use and the **interprocess communication** mechanisms used to coordinate their operation.

Concerns: Task Structure, Mapping of Functional Elements to Tasks, Interprocess Communication, State Management, Synchronization and Integrity, Startup and Shutdown, Task Failure, Reentrancy.

Stakeholders: Developers, Testers, Administrators.

Representation: UML; Other (Boxes-and-Lines Diagram).

Example: There is **limited native support** for concurrency in UML, but that can be enhanced using UML Notes and stereotypes. Figure 10 shows an example of UML being used for a concurrency model. This model shows how the system is implemented by using three processes (**a client, a statistics service, and a statistics calculator**) along with a process group to implement a DBMS instance.

There is little or no architecturally significant thread design in this model. The concurrent activity between the **`Statistics Accessor`** and **`Statistics Calculator`** components needs to be coordinated because they are in different processes, and that is achieved using a mutex[6].

[6] A **mutex (or mutual exclusion)** refers to the problem of ensuring that **no two processes or threads** can be in their **critical section** at the **same time** [Wikipedia]. (A **critical section** refers to a **period of time** when the process accesses a **shared resource**, such as **shared memory**.)

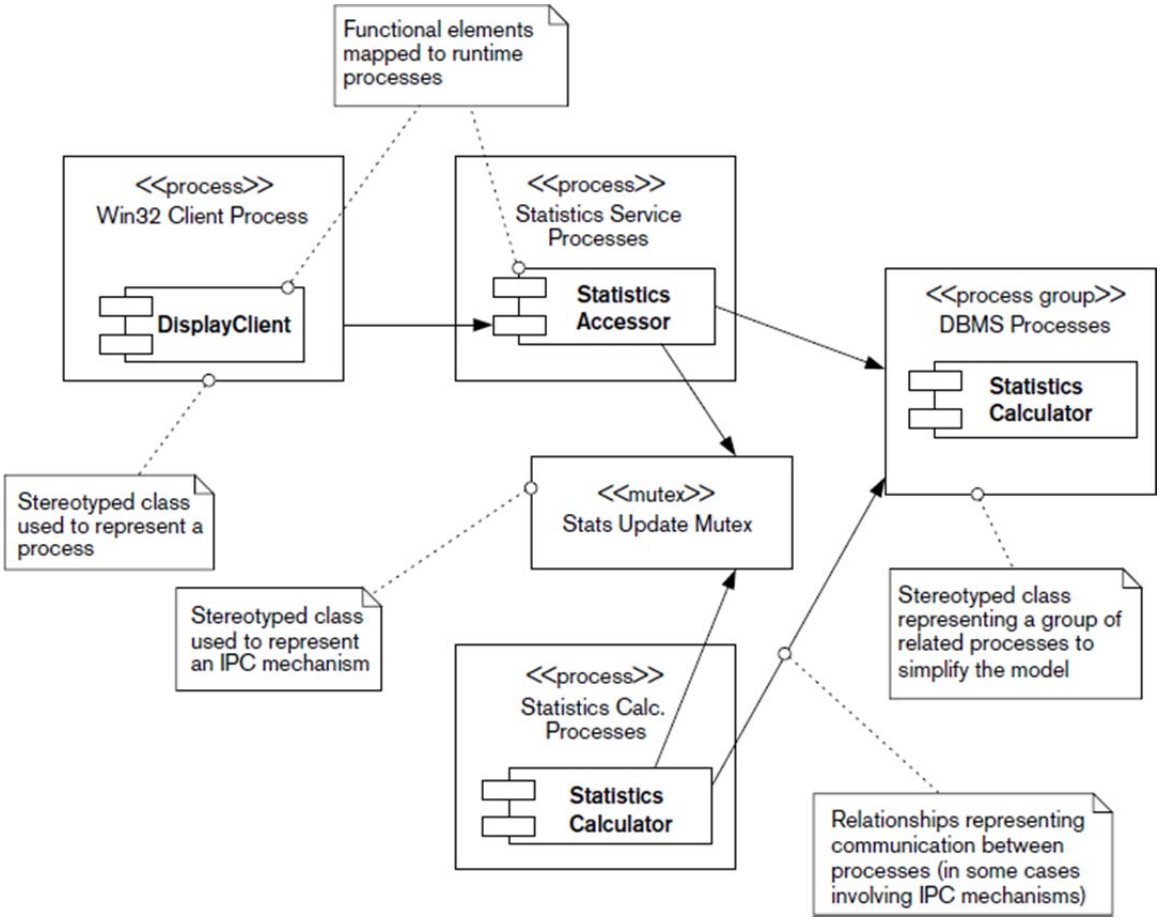


Figure 10. The different constructs in an extension of UML to illustrate a service that provides statistical data analysis. (Source: [Rozanski, Woods, 2005].)

DEVELOPMENT

Purpose: This view describes the architecture that supports the **software development process**.

This view communicates the aspects of the architecture of interest to those stakeholders involved in **developing, testing, and maintaining** the system.

Concerns: Module Organization, Common Processing, Standardization of Design, Standardization of Testing, Instrumentation, Codeline Organization.

Stakeholders: Software Developers, Testers, Maintainers.

Representation: UML Package; UML Component Diagram; Other (**Boxes-and-Lines Diagram**).

Example: Figure 11 shows a module organization with three layers, each layer being represented by a **stereotyped package**. The system's modules are shown as UML packages within the layers.

The model shows that **the domain layer depends on the utility layer**, and **the utility layer, in turn, depends on the platform layer**. In other words, the domain-layer components can access **only** the utility-layer components, and so on.

However, in this system, **non-strict layering** has been used because all of the domain-layer components depend on facilities provided by the Java Standard Library component rather than accessing its facilities via intermediate utility components.

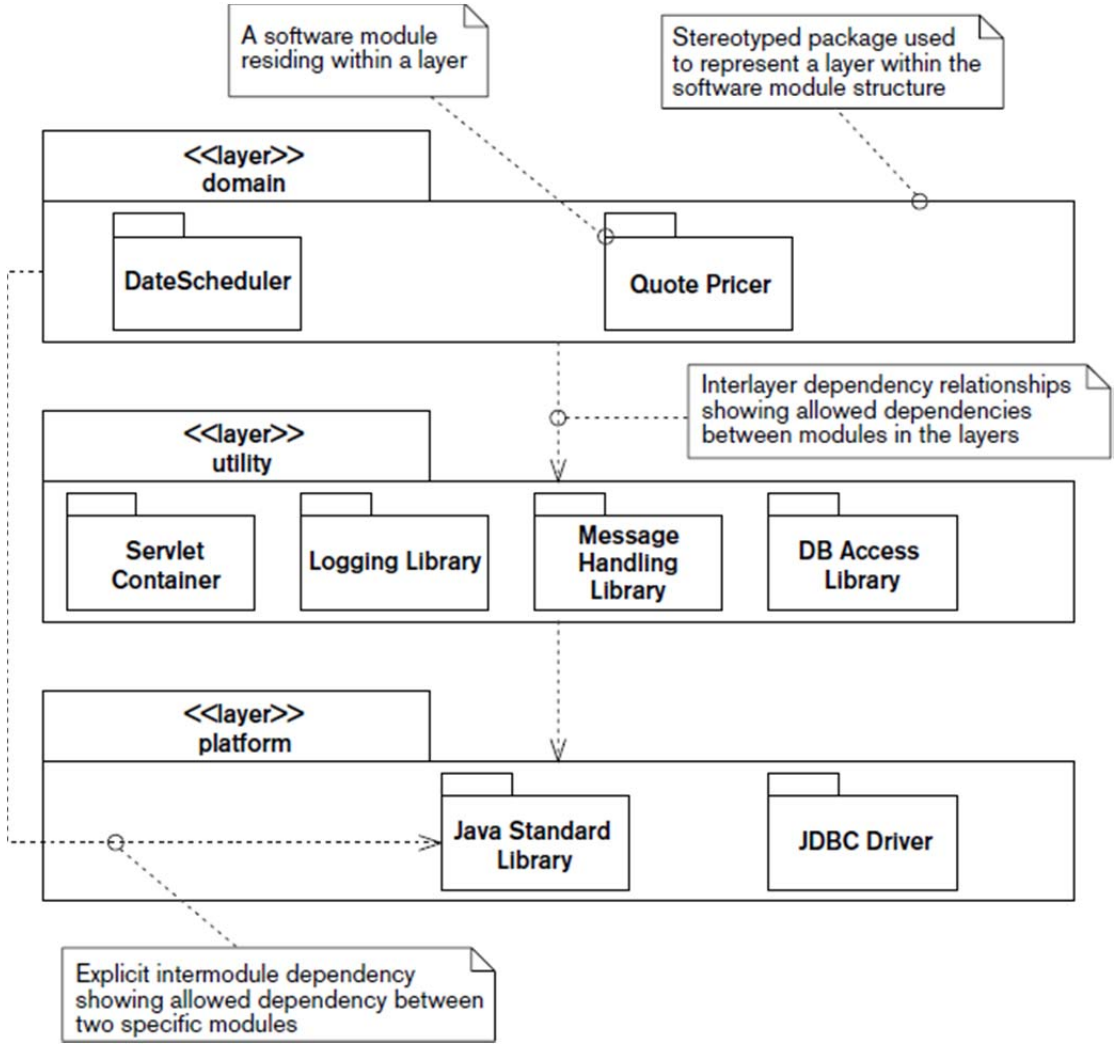


Figure 11. The different UML constructs to illustrate organization of modules in layers. (Source: [Rozanski, Woods, 2005].)

DEPLOYMENT

Purpose: This view describes the environment into which the system will be deployed, including capturing the dependencies the system has on its runtime environment.

This view captures the **hardware environment** that the system needs (primarily the **processing nodes, network interconnections, and disk storage facilities** required), the technical environment requirements for each element, and the mapping of the software elements to the runtime environment that will execute them.

This view is important for **DevOps** [Lwakatare, Kuvaja, Oivo, 2015].

Concerns: Types of Hardware Required, Specification and Quantity of Hardware Required, Third-Party Software Requirements, Technology Compatibility, Network Requirements, Network Capacity Required, Physical Constraints.

Stakeholders: System Administrators, Developers, Testers, Communicators, Assessors.

Representation: UML Deployment Diagram; Other (Boxes-and-Lines Diagram).

Example: Figure 12 shows a simple runtime platform model that maps functional elements to processing nodes. It is **annotated** by UML Note, and **extended** via tagged values.

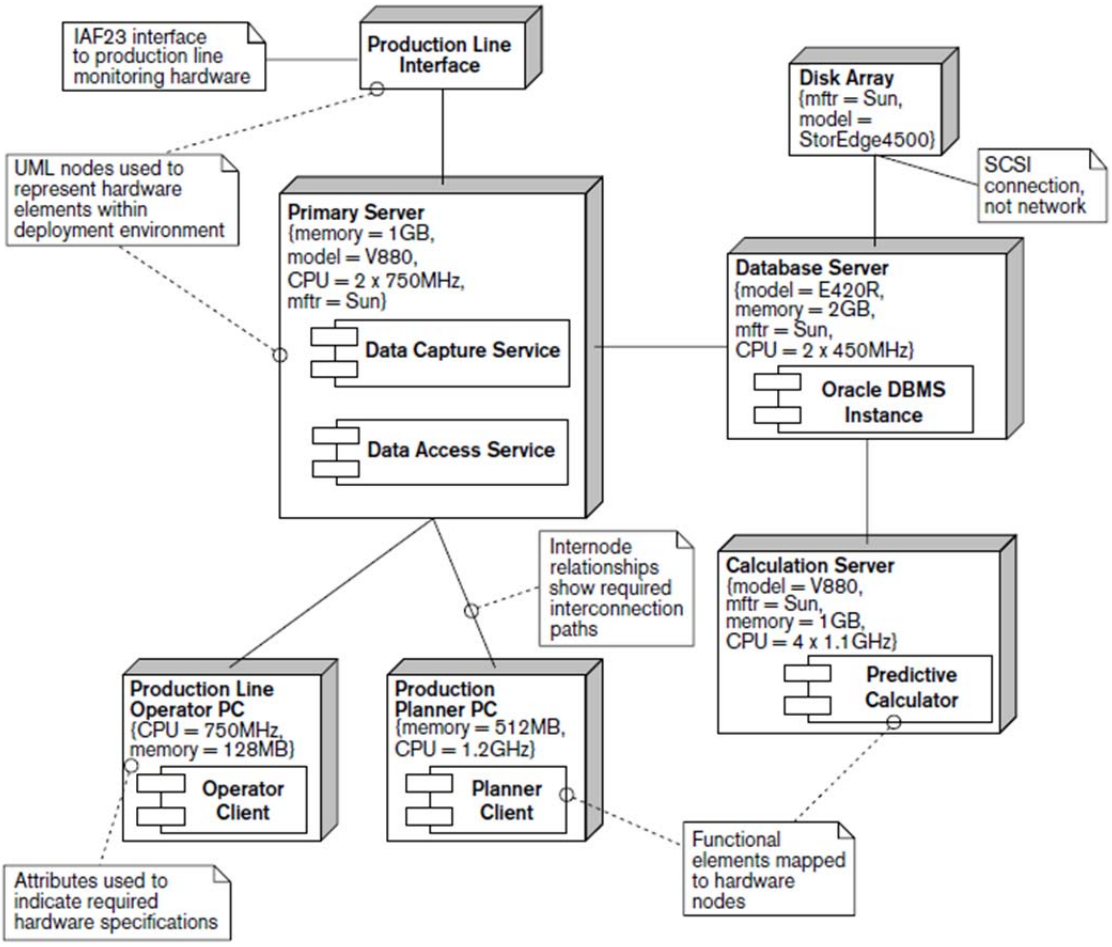


Figure 12. A UML Deployment Diagram illustrating the hardware environment and the mapping of the software elements to the runtime environment. (Source: [Rozanski, Woods, 2005].)

OPERATIONAL

Purpose: This view describes how the system will be **operated, administered, and supported** when it is running in its production environment.

For many systems, **installing, managing, and operating** the system is a **significant task** that must be **considered and planned** at **design time**. The aim of this view is to identify **system-wide strategies** for addressing the **operational concerns** of the system's stakeholders, and to identify solutions that address these.

This view is important for **DevOps** [Lwakatare, Kuvaja, Oivo, 2015].

Concerns: Installation and Upgrade, Functional Migration, Data Migration, Operational Monitoring and Control, Configuration Management, Performance Monitoring, Support, Backup and Restore.

Stakeholders: System Administrators, Developers, Testers, Communicators, Assessors.

Representation: Boxes-and-Lines Diagram.

Example: UML does **not** have **native support** for representing the operational view. Figure 13 shows the operational view of a data migration system. It includes a large **database of taxpayers** that is **migrating into a new system**. The data must be extracted, sorted, and loaded into the new system. Finally, indexes must be created on the new system. The solid arrows denote time.

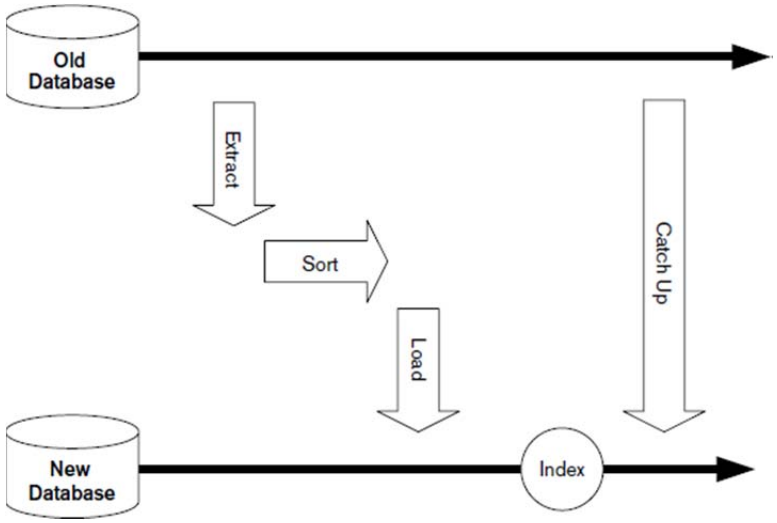


Figure 13. The operations involved in the migration of a tax system. (Source: [Rozanski, Woods, 2005].)

9.1. THE RELATIONSHIPS BETWEEN VIEWS

It is evident that all of the views in the Rozanski-Woods View Model of Software Architecture are **interrelated**. However, in practice, there are **strong dependencies** only between **some** of the views.

Figure 14 shows the most important of these dependencies. The relationships illustrate a **strong dependency**: if something changes at the end of the arrow, a change will probably be required at the start of the arrow.

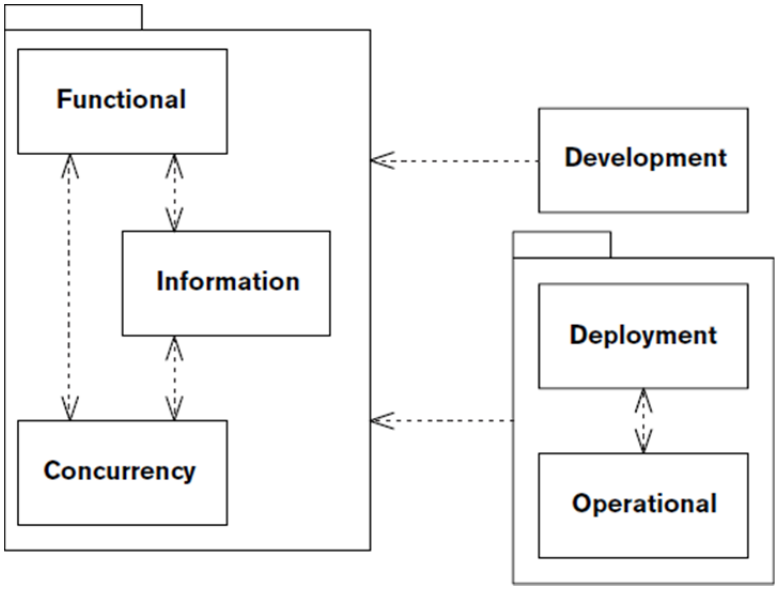


Figure 14. The primary relationships between views in the Rozanski-Woods View Model of Software Architecture. (Source: [Rozanski, Woods, 2005].)

10. QUALITY OF VIEWS

The motivation behind discussing the quality of views is that a view in and of itself may or may not be useful or reliable.

There are certain **desirable quality attributes** of a view, including the following:

- Utility
- Consistency

UTILITY

A view must be **useful**, at least to the stakeholders that it is aimed for.

The **set of views** must, as an ensemble, **cover all concerns** of **all the stakeholders**. (For example, this is the **aim** of the views in the Rozanski-Woods View Model of Software Architecture [Rozanski, Woods, 2005].)

Indeed, this is one approach for **selecting necessary views**.

CONSISTENCY

A view must be **internally and externally consistent** [Rozanski, Woods, 2005, Chapter 22].

- **Internal Consistency.** The purpose of **internal consistency** is achieving consistency in a **single view**.

- **External Consistency.** The purpose of **external consistency** is achieving consistency across **multiple views**. In other words, the specification of one view **should not contradict** that of another view. For example, in Figure 14, Information View should not specify anything that contradicts a statement in the Development View.

ACKNOWLEDGEMENT

The inclusion of images from external sources is only for non-commercial educational purposes, and their use is hereby acknowledged.

REFERENCES

[Babar, Dingsøyr, Lago, Vliet, 2009] Software Architecture Knowledge Management: Theory and Practice. By M. A. Babar, T. Dingsøyr, P. Lago, H. van Vliet. Springer-Verlag. 2009.

[Brittan, 1980] Design for a Changing Environment. By J. N. G. Brittan. The Computer Journal. Volume 23. Number 1. 1980. Pages 13-19.

[Brost, Hoffmann, 2015] Identifying Security Requirements and Privacy Concerns in Digital Health Applications. By G. S. Brost, M. Hoffmann. In: Requirements Engineering for Digital Health. S. A. Fricker, C. Thümmler, A. Gavras (Editors). Springer International Publishing. 2015. Pages 133-154.

[Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord, Stafford, 2010] Documenting Software Architectures: Views and Beyond. By P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford. Second Edition. Addison-Wesley. 2010.

[Garland, Anthony, 2003] Large-Scale Software Architecture: A Practical Guide using UML. By J. Garland, R. Anthony. John Wiley and Sons. 2003.

[Ghezzi, Jazayeri, Mandrioli, 2003] Fundamentals of Software Engineering. By C. Ghezzi, M. Jazayeri, D. Mandrioli. Second Edition. Prentice-Hall. 2003.

[Hilliard, 2001] Viewpoint Modeling. By R. Hilliard. The Twenty Third International Conference on Software Engineering (ICSE 2001). Toronto, Canada. May 12-19, 2001.

[ISO/IEC/IEEE, 2011] ISO/IEC/IEEE 42010:2011. Systems and Software Engineering -- Architecture Description. The International Organization for Standardization (ISO)/The International Electrotechnical Commission (IEC). 2011.

[Koning, Vliet, 2006] A Method for Defining IEEE Standard 1471 Viewpoints. By H. Koning, H. van Vliet. Journal of Systems and Software. Volume 79. Issue 1. 2006. Pages 120-131.

[Kruchten, 1995] The 4+1 View Model of Architecture. By P. B. Kruchten. IEEE Software. Volume 12. Issue 6. 1995. Pages 42-50.

[Kruchten, 2004] The Rational Unified Process: An Introduction. By P. Kruchten. Third Edition. Addison-Wesley. 2004.

[Lavariega, Avila, Gómez-Martínez, 2015] Software Architecture for Emergency Remote Pre-hospital Assistance Systems. By J. C. Lavariega, A. Avila, L. G. Gómez-Martínez. In: Mobile Health: A Technology Road Map. S. Adibi (Editor). Springer International Publishing. 2015. Pages 453-472.

[Lwakatare, Kuvaja, Oivo, 2015] Dimensions of DevOps. By L. E. Lwakatare, P. Kuvaja, M. Oivo. The Sixteenth International Conference on Agile Processes, in Software Engineering, and Extreme Programming (XP 2015). Helsinki, Finland. May 25-29, 2015.

[May, 2005] A Survey of Software Architecture Viewpoint Models. By N. May. The Sixth Australasian Workshop on Software and System Architectures (AWSA 2005). Brisbane, Australia. March 29, 2005.

[Rozanski, Woods, 2005] Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. By N. Rozanski, E. Woods. Addison-Wesley. 2005.

[Sangwan, 2015] Software and Systems Architecture in Action. By R. S. Sangwan. CRC Press. 2015.

[Shames, Skipper, 2006] Toward a Framework for Modeling Space Systems Architectures. By P. Shames, J. Skipper. The Ninth International Conference on Space Operations (SpaceOps 2006). Rome, Italy. June 19-24, 2006.