



Software Engineering Project A

SEPA - SWE40001/EAT40003

Week 5 – Software Requirements & Design

1 SEPA

CRICOS 00111D
TOID 3069

This week is all about...

- Software Requirements
- Software Architecture & Design
- Peer Reviews

Software Requirements



- Business Goals
- Requirements Elicitation and Analysis
- Functional Requirements
- Non-Functional/Quality Requirements
- Quality of Requirements

... Problem Domain

Why do we build Software Systems?

When discussing application development, the first question we ask has nothing to do with budget, time frame, scope, or any number of qualifications.

The first question is always, “How close will the product be to your core business output?”

Why do we build Software Systems? Cont.

Why is this so important? Core business value always has software requirements, either in terms of direct output or making that output more efficient.

The idea that software is key not only to business functions, but to the business itself, is at the core of any deliberation.

Why do we build Software Systems? Cont.

- To keep/retain customers
- To increase customer base
- To improve customer relationships
- To improve productivity/efficiency (within organization/team)
- Technology used no longer supported/outdated
- Changes in environment (legal, social)

In other words . . .

To achieve some “Business” Goals!

Categories of Business Goals

They fall into five broad categories:

1. Reduce cost of ownership: development, maintenance, deployment, operation.
2. Improve the quality of the system(s) compared with its predecessors with respect to performance, modifiability, security, reliability etc.
3. Improve the capabilities/functionality offered by the system compared to its predecessors.
4. Improve organization's market position.
5. Improve external confidence in either the organization or the system.

☞ *Software systems generally have more than one goal!*

Business Goals (cont.)

Priority of goals need to be specified:

- Some goals are “nice to have”, others “need to have”, some “absolutely critical”
- Developers sometimes have to “push back” or make trade offs. Knowing priority gives insights.

Source of goals need to be specified:

- Some goals are *inherent* to the system being developed
- Some goals are a result of *market analysis*
- Some goals are *arbitrary* - could cause problems!

Problem vs Solution

- When we build software we are often provided with a lot of information.
- If we take all of this information we loosely classify it either in the *Problem Space* or *Solution Space*
- Some information (unfortunately) lies in between... beware of information “too close” to the Solution Space when describing the problem!
- Separating the Problem space from Solution space is an important high-level concept.

Problem vs Solution (cont.)

Problem Space

The Problem space is where all the customer needs that you'd like your product to deliver live. Customers are also not likely to serve you their problem space needs on a silver platter.

Solution Space

Any product that you actually build exists in solution space, as do any product designs that you create. Solution space includes any product or representation of a product that is used by or intended for use by a customer.

Problem vs Solution (cont.)

The development team's (first) job is to unearth the client needs and define the problem space.

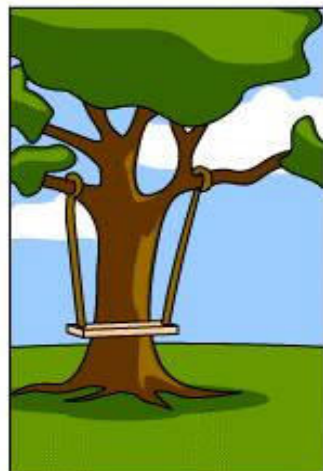
- The reality is that clients are much better at giving you feedback in the solution space (than thought).
- The feedback you gather in the solution space actually helps you test and improve your problem space hypotheses.



The many views of Software ...



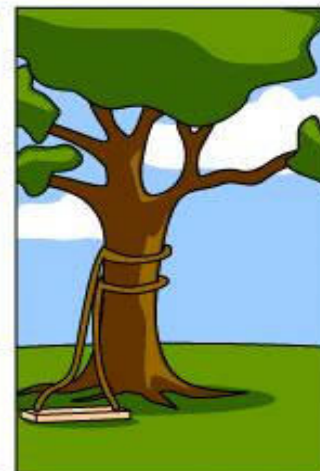
How the customer explained it



How the Project Leader understood it



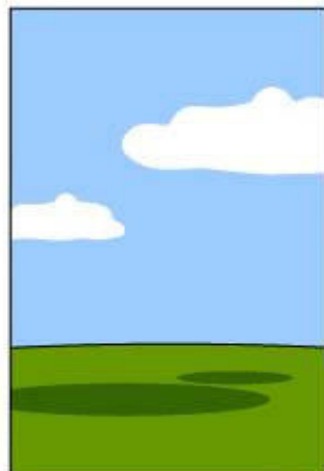
How the Analyst designed it



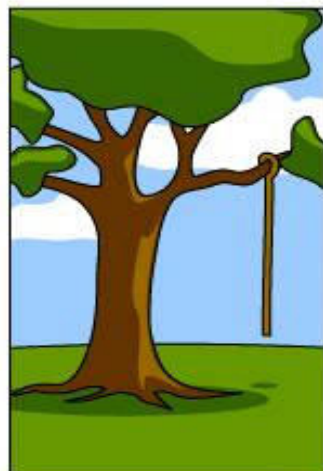
How the Programmer wrote it



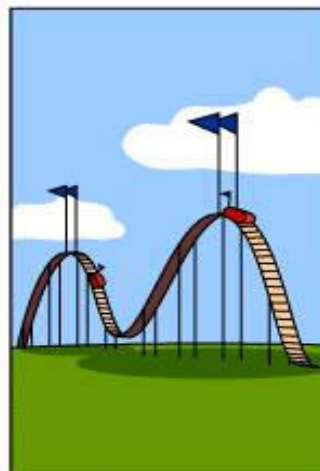
How the Business Consultant described it



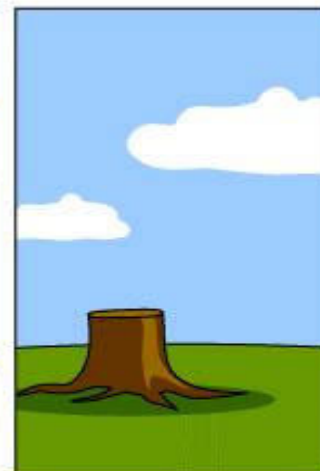
How the project was documented



What operations installed



How the customer was billed



How it was supported



What the customer really needed

The Role of Requirements Analysis

- Requirement analysis is the process of determining user expectations for a new or modified product and is vital for effective software testing
- It lays down the basic foundation for various stages of the SDLC.
- Without a thorough needs analysis and a clear direction, even the most efficient system architecture, well-written code, or comprehensive testing will prove ineffective.

The Role of Requirements Analysis (cont.)

- The first, most important, step of any project is “Requirement Analysis” where requirements should be
 - scripted, actionable, quantifiable, traceable, and
 - directly linked to the business needs or opportunities.

Functional and Non-functional Requirements

A typical Requirement Analysis process should identify testable requirements through frequent interactions with various stakeholders (Client, Business Analyst and Technical Leads etc.)

- Domain requirements (business entities ...)
- Functional requirements
- Non-functional requirements

Functional and Non-functional Requirements (cont.)

Functional requirements describe system services or functions

e.g.

- Compute sales tax on a purchase
- Update the database on the server ...

☞ In essence, anything that can be directly expressed in code...

Functional and Non-functional Requirements (cont.)

Non-functional requirements are constraints on the services and/or the development process

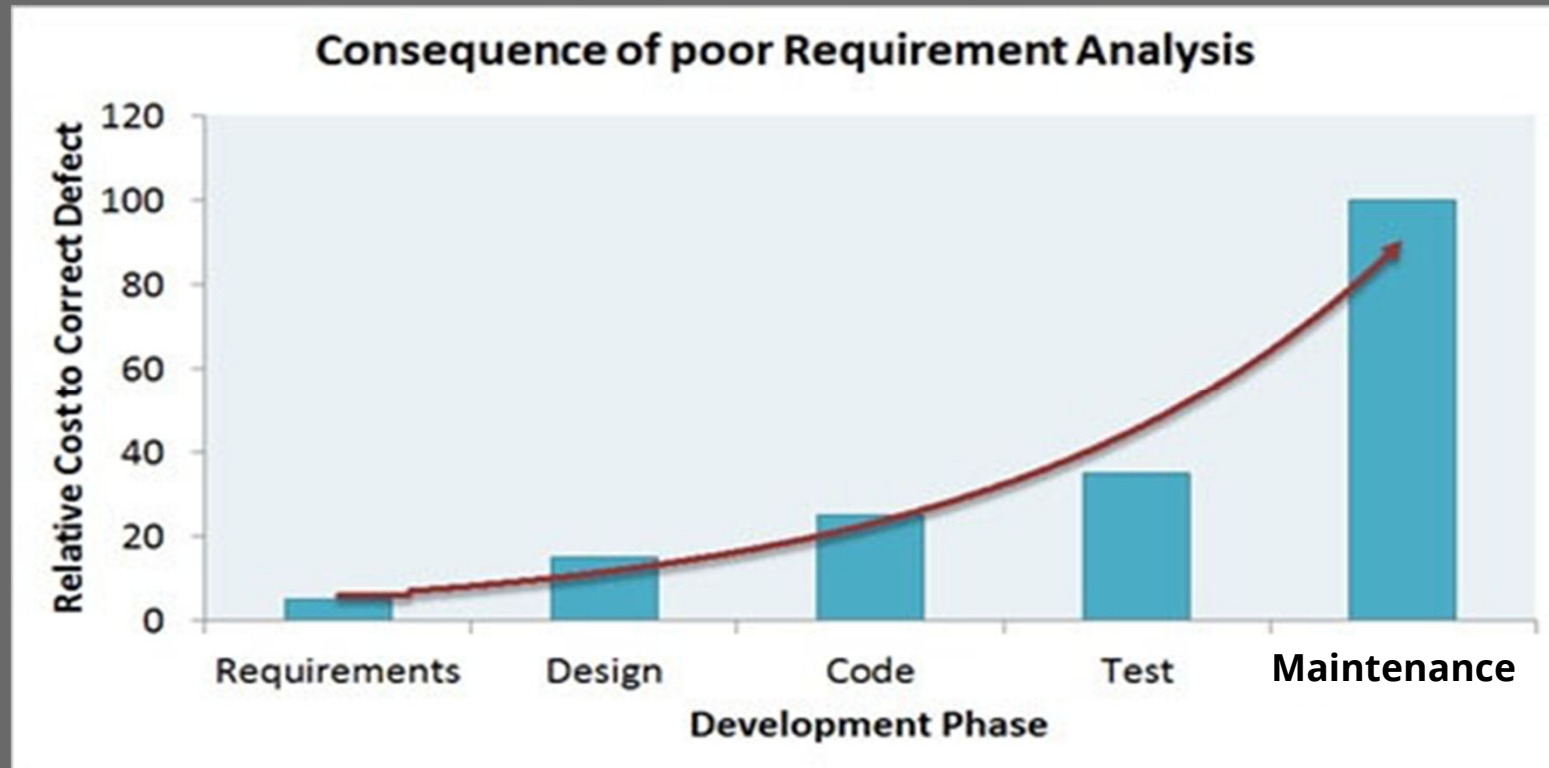
e.g.

- also known as **quality** requirements
- “Reducing cost of deployment” cannot be directly expressed in code...
- “User friendliness” ??

Domain requirements stem from the application domain of a system

- may be functional or non-functional

Importance of good Requirements Analysis



Poor requirement gathering = more maintenance

But what (other) types of requirements do exist?

☞ *Functional vs. non-functional is only one (coarse-grained) way to distinguish requirements.*

Other Types of Requirements

Data requirements:

- System state: Common states,
- Input/output formats, Persistent Data

Other deliverables:

- Documentation
- Install, convert,
- train

Other Types of Requirements

Managerial requirements:

- Delivery time
- Legal
- Development
- process

Helping the stakeholder:

- Business goals
- Definitions
- Diagrams

Functional Requirements

Traditional “feature” requirements

Roster planning, Midland Hospital

R47. It must be possible to attach a duty type code (first duty, end duty, etc.) to the individual employee.

R475. The system must be able to calculate the financial consequences of a given duty roster - in hours and in money terms.

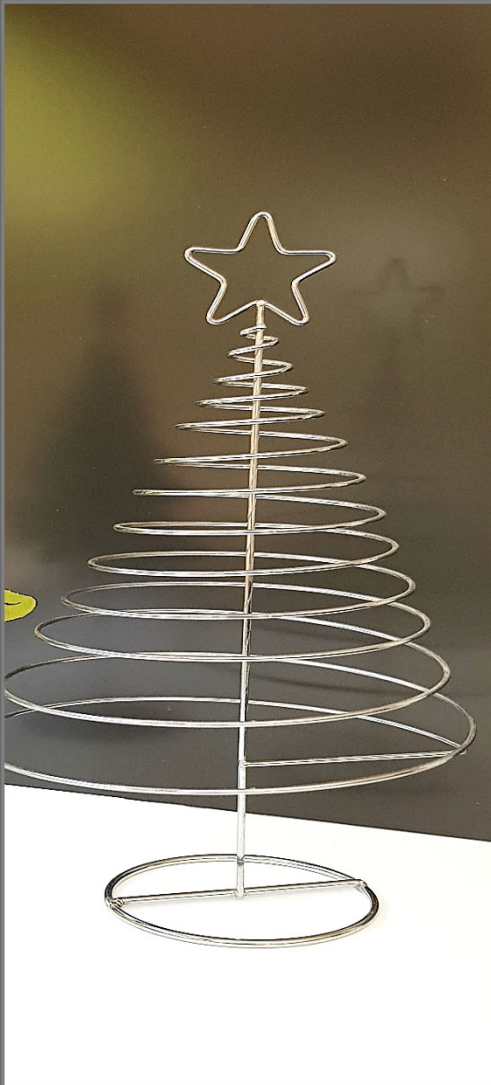
R479. The system must give notice if a duty roster implies use of a temporary worker for more than three months.

R669. The system must give understandable messages in text form in the event of errors, and instruct the user on what to do.

What is “wrong” with these kinds of requirements?

Traditional “feature” requirements

- *Requirements are too vague*
- *Not specific enough*



Users Perspective

Users often tend to think about systems in terms of *"features"*

- E.g., "the system must provide a function to do X"
- Get users to tell you **stories** involving those features.
 - How do users anticipate these features to be used *in combination* with other features.
- Scenarios and Use cases can assist in order to get requirements **complete and consistent!**

Use Case

- It's the interaction between the user and a software system.
 - It's different from a business process, which might capture all the things that that user would do to achieve a bigger picture goal or outcome in the organization.
- *"A use case is a set of scenarios tied together by a common user goal"* – Martin Fowler
 - Every use case must be tied to a *specific goal*
 - A use case is directly or indirectly invoked by an *actor*

Use Case

- “A *use case* is the *specification* of a *sequence of actions*, including *variants*, that a system (or other entity) can perform, *interacting with actors* of the system.”

Scenarios

- Users *interact* with a computer system to complete a “task” (or) achieve a “goal” (or have some fun...)
- These interactions can be captured as a set of *scenarios* (or) stories
- Example: “*Buy a product*” scenario for an online store:
 - ☞ *Mary browses the catalogue and adds desired items (chocolate) to the shopping basket. When she has finished shopping, Mary provides her address for delivery, credit card information and then confirms the transaction. The system confirms her payment and e-mails a transaction record.*

But...

Use cases and scenarios are still too much focused on the “solution domain” instead of identifying the real problem to be addressed.

Tasks & Support (approach)

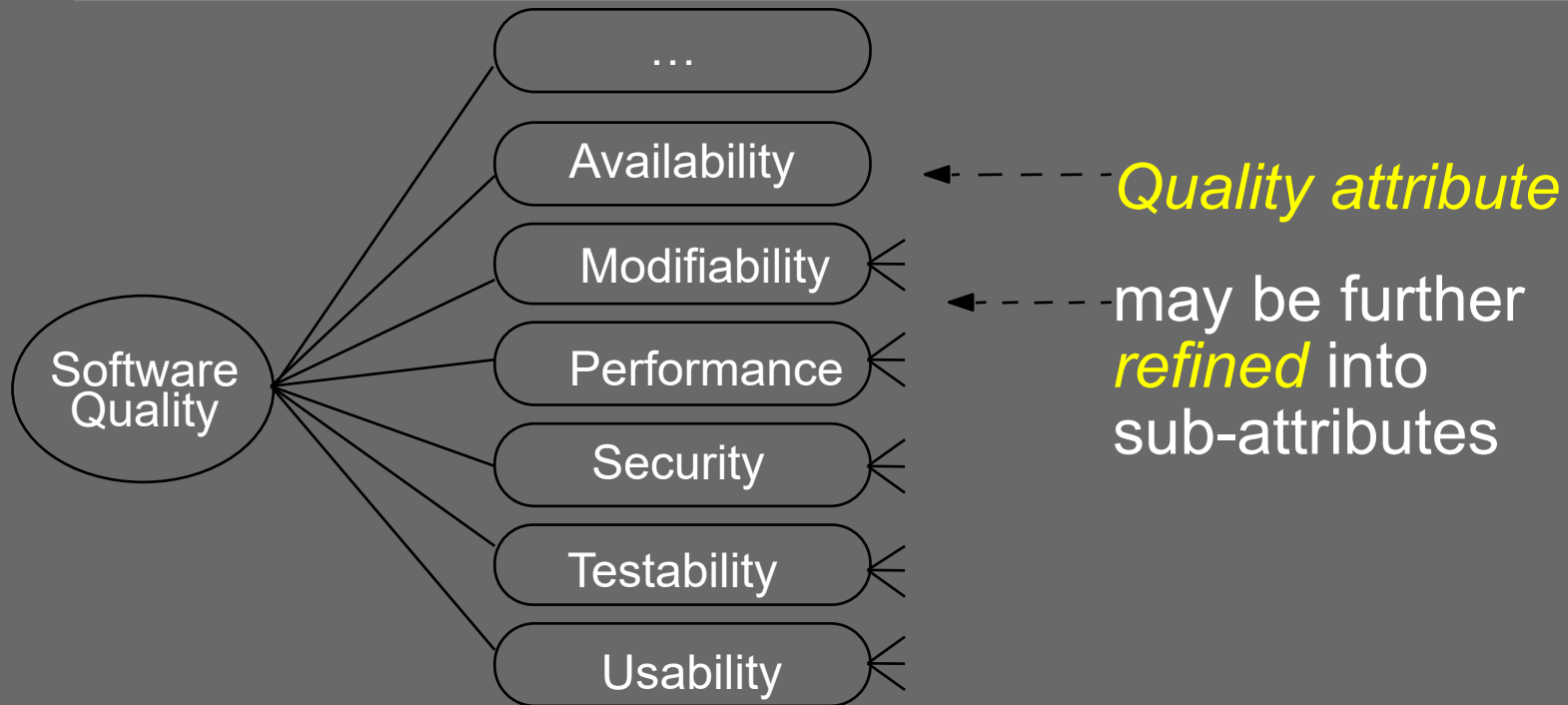
Focus on the problem domain ...

- Soren Lauesen, *Software Requirements: Styles and Techniques*, Addison-Wesley, 2002, Chapters 1 to 3.
- Soren Lauesen, *Task Descriptions as Functional Requirements*, IEEE Software, March 2003. (on Canvas under week 5)

Non-Functional / Quality Requirements

Hierarchical Quality Model







Define software quality via a *hierarchical quality model*, i.e. a number of *quality attributes* (aka quality factors, quality aspects, etc.)



Hierarchical Quality Model (cont.)

- **Availability:** is the system providing the service it is supposed to when it is expected to?
- **Modifiability:** can reasonable changes be made at reasonable cost?
- **Performance:** does the system respond quickly enough to user input?
- **Security:** can authorized users access the services of the system while unauthorized people cannot?
- **Testability:** can the system be exercised to a degree that ensures confidence in correctness?
- **Usability:** how easy is it for the user to do what is desired?

What about other Qualities?

- *Portability*: the system must be changed to run on different hardware/operating system  **modifiability**
- *Maintainability* (“fixing bugs”): the system must be changed to remove invalid/wrong behavior  **modifiability**
- *Scalability*: the system must be changed to have more capacity (can handle more requests than currently, store more information, etc.)  **modifiability**
- *Operability*: the system must be “easy” to use, being “easy” to learn etc.  **usability**
- *Reliability*: the system must not fail “too often”  **availability**
- *Integrity*: data of the system cannot be changed in an unauthorized manner  **security**

Quality Models/Frameworks of McCall and Matsumoto

- McCall and Matsumoto

- ISO 9126 Revision

Either is fine as a reference model / framework

A simple ways to go ...

- Based on a quality model/framework ...
 - Identify the top 3 to 5 quality attributes/factors of the domain
 - ☞ Assume that qualities of similar applications will also be of relevance for the application under consideration
 - *Systematically* go through all identified user tasks/cases
 - Identify these tasks that make explicit mention of the identified quality attributes
 - Note: “critical” aspects of user tasks generally imply some quality requirements!
 - Spell out the resulting non-functional requirements in a *verifiable* form.
- ☞ *Maybe not be 100% perfect, but a good start...*

Quality of Requirements

Quality Criteria for a Requirement

- *Correct*: each requirement reflects a need
- *Traceable*: to goals/purpose (also to design/implementation...)
- *Complete*: all necessary requirements are included
- *Consistent*: all parts of SRS match; there are no inconsistencies
- **Verifiable**: possible to see whether it can be met/tested
- *Unambiguous*: all stakeholders agree on meaning
- *Ranked for importance/stability*: priorities; expected changes
- *Modifiable*: easy to change; maintain consistency
- *Understandable by customer and developers*

Requirements Verifiability

Requirements must be written so that they can be *validated* and **objectively verified**.

Imprecise:

- *“The system should be easy to use by experienced controllers and should be organized in such a way that user errors are minimized.”*

Terms like “easy to use” and “errors shall be minimized” are *useless* as specifications as they are too vague.

Verifiable:

- *“Experienced controllers should be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users should not exceed two per day.”*

Verification and Validation

Starts after functional specifications have been documented and initial model development has been completed.

Verification and validation is an iterative process that takes place throughout the development of a model.

Validation:

- Are we building the *right product*?
 - does it meet stakeholders' *expectations*?
- Ways to do this: ... Going through Scenarios

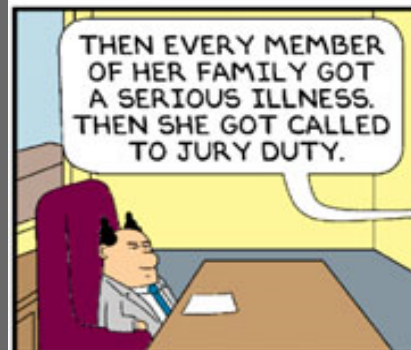
Verification:

- Are we *building the product right*?
 - does it conform to the specification(s)?



DILBERT[®]

BY
SCOTT ADAMS

© Scott Adams, Inc./Dist. by UFS, Inc.

Software Design



- Software design
- Architecture Design
- Detailed Design

... Solution Domain

Software Design



“Design is the creative process of transforming the problem into a solution. The description of the solution is also called the design.”

Activity & Artefact — Shari Lawrence Pfleeger, 1998

“The design of a system determines a set of components and inter-component interfaces that satisfy a specified set of requirements.”

— DeMarco, 1982

Artefact

Levels of Software Design



- Architecture design – high level
- Detailed design, eg, OO design, functional design
- ...



Architecture Design

Software Architecture – one view

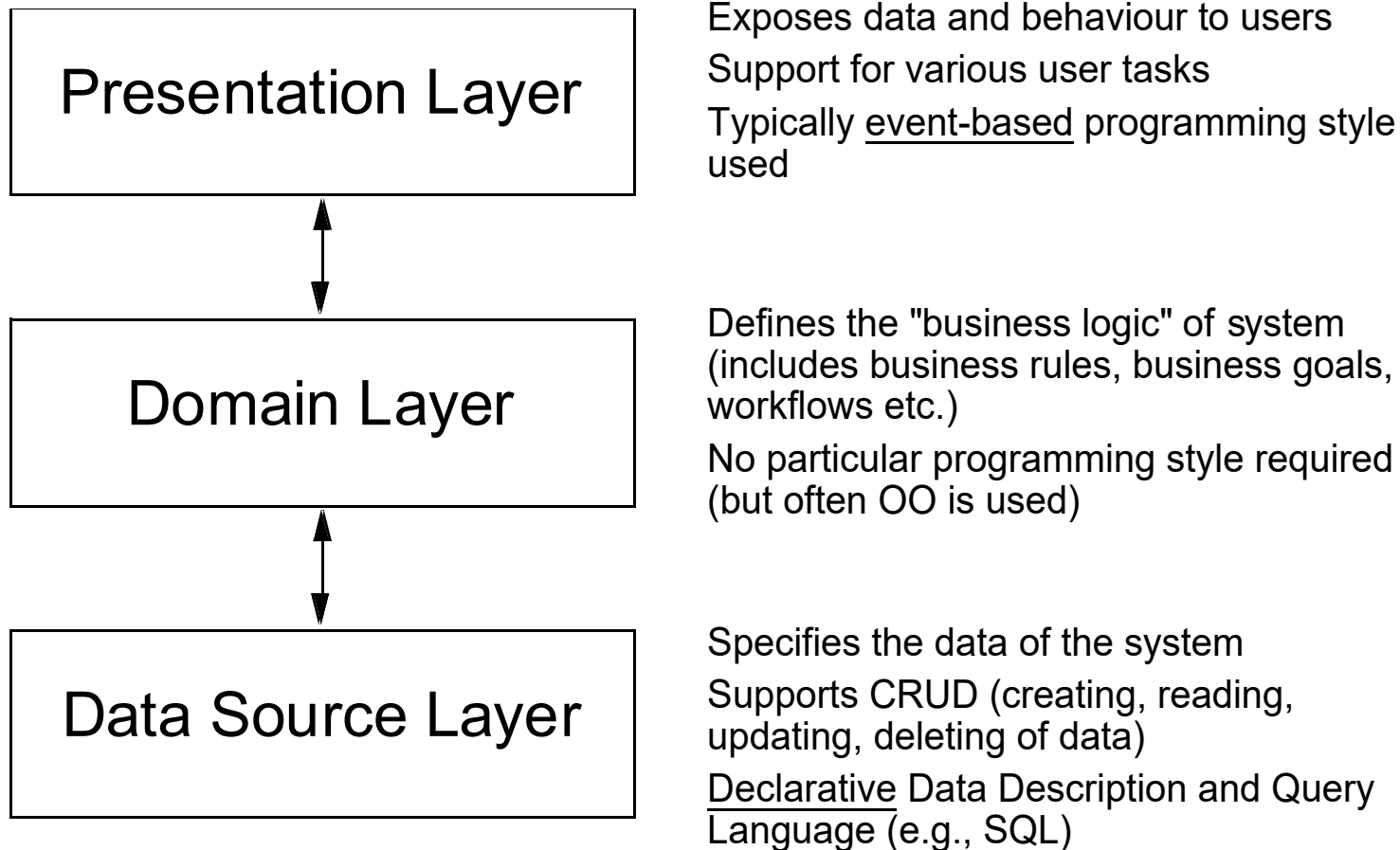


*“In most successful software projects, the expert developers working on that project have a **shared understanding** of the system to be implemented. This shared understanding is often called ‘**architecture**’.*

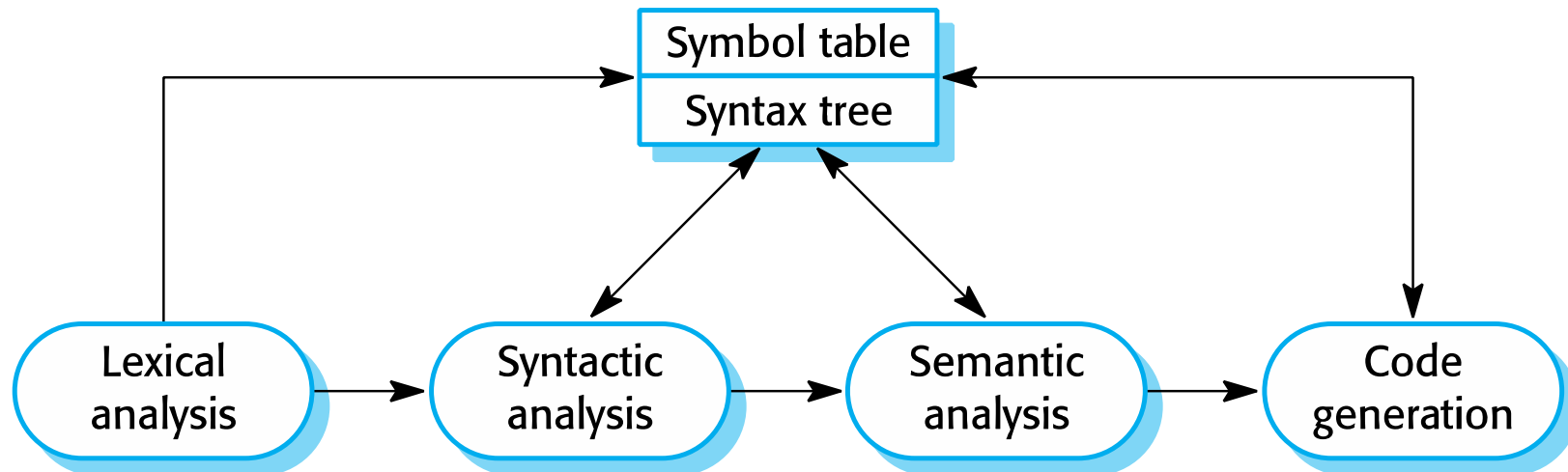
*It includes how the system is divided into **processing elements** (i.e. components) and their **interaction** through interfaces. These components are usually **composed** of smaller components, but the architecture only includes the ones that are understood by all developers.”*

— Ralph Johnson, 2003

Example – Enterprise Systems



Example – Compiler



Software Architecture (cont.)



The architecture of a software system is described as:

- the *structure(s) of its high-level processing elements*,
- the *externally visible properties* of the processing elements, and
- the *relationships and constraints* between them.

in other words:

The set of *design decisions* about any system (or subsystem) that keeps its implementors and maintainers from exercising “*needless creativity*”.

Software Architecture (cont.)



*“A software architecture is a set of architectural (design) elements that have a particular form. Properties **constrain** the choice of architectural elements whereas **rationale** captures the motivation for the choice of elements and form.”*

— Dewayne Perry and Alexander Wolf, 1992

Rationale



Rationale explains why a decision was made and what the implications are if changing it! Use rationale to explain:

- implications of *system-wide* design choices on meeting requirements,
- effects on the architecture in the context of changing/adding requirements,
- design alternatives that were rejected (and why!),
- ...



Architecture Styles

Architectural Styles



- Like building architecture, software architecture also has styles ...

*“An **architectural style** defines a **family of software systems** in terms of their structural organization. An architectural style expresses components and the relationships between them, with the constraints of their application, and the associated **composition and design rules** for their construction.”*

— Dewayne Perry and Alexander Wolf, 1992



Popular Architectural Styles

- Data-flow architectures:
 - ☐ Batch sequential, *Pipes-and-Filter*
- Call-and-Return architectures:
 - ☐ Main program and subroutine, Object-oriented
 - ☐ *Layered*
 - ☐ *Client-Server*
- Data-centred architectures:
 - ☐ *Repository*, Blackboard
- Independent component architectures:
 - ☐ *Peer to Peer*
 - ☐ Communicating processes
 - ☐ Event systems: implicit invocation, explicit invocation
- Virtual machine architectures:
 - ☐ Interpreter, rule-based systems
- Service oriented architecture:
 - ☐ Software services, WS*(SOAP) Web services, REST services, Microservices

Documenting Architecture - Purpose



Communication: communicating ideas about possible software solutions to stakeholders for feedback etc.

- ☐ Who is the audience? What do they want/need to know?
- ☐ What means of communication do they understand?

☞ *Focus on the big-picture ideas!*

Specification: blue-print for implementation

- ☐ Purpose: a detailed design specification for implementers.
- ☐ Needs to avoid any form of ambiguity.

☞ *Focus on specific details!*

Documenting Architecture - Views



*“A view is a representation of a **coherent set of architectural elements**, as written by and read by stakeholders. It consists of a representation of a set of [architectural] elements and the relationships amongst them.”*

— Len Bass et al., Software Architecture in Practice (2nd Edition), 2003.

- 👉 Views focus on specific aspects of a software system by using appropriate **abstractions**!

Categories of Views



■ Component and Connector:

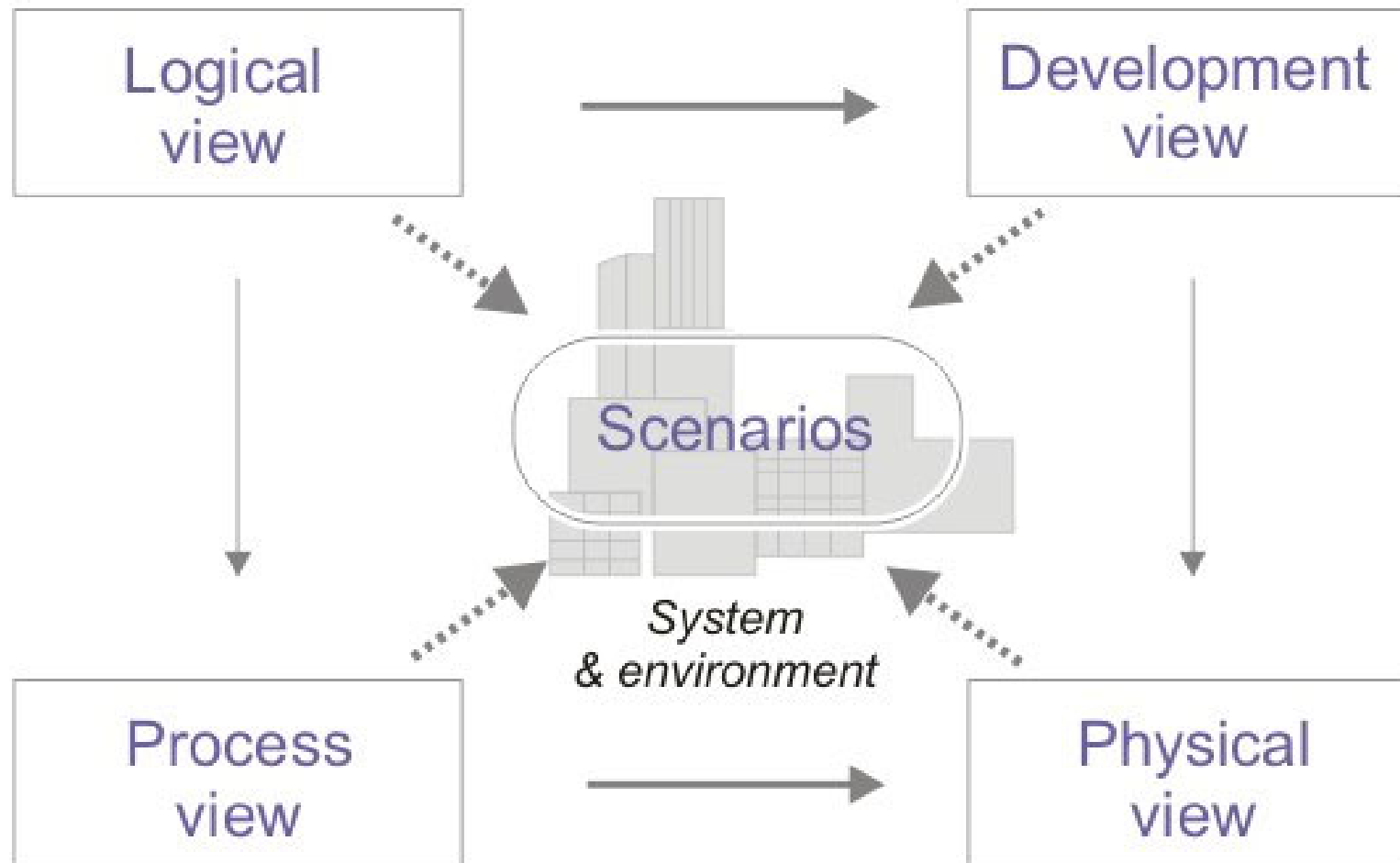
- ☐ Focus on (run-time) *computational entities* (i.e. components, processing elements) and their means of *communication* (i.e. connectors).
- ☐ *What are the main executing components and how do they interact?*

■ Deployment / Allocation:

- ☐ Focus on computational entities and their *(physical) environment*.
- ☐ *What processing node(s) does each component execute on?*

■ Other views ...

Kruchten's 4+1 View



Kruchten's 4+1 View (cont.)



- *Logical View* (Module View):
 - concerned with the *functionality* that a system provides to end-users
- *Process View* (Component & Connector View):
 - focus on system processes (i.e. components) and how they communicate; addresses concurrency, distribution, performance, scalability etc.
- *Development View* (Allocation View):
 - illustrates a system from a developers perspective; concerned with software management
- *Physical View* (Allocation View):
 - concerned with mapping of components onto processing and communication nodes

Kruchten's 4+1 View (cont.)



■ Scenarios:

- ☐ illustration of architectural solution using selected use cases,
- ☐ describe sequences of interactions between system processes (i.e. processing elements),
- ☐ used to illustrate and verify the architectural design,
- ☐ serve as a starting point for tests and architectural prototypes.

☞ *In many situations, Kruchten's 4+1 view model is a good starting point for documenting architectural designs.*

UML support: Package Diagram



Decompose
system into
packages
(containing any
other UML
element, incl.
packages)

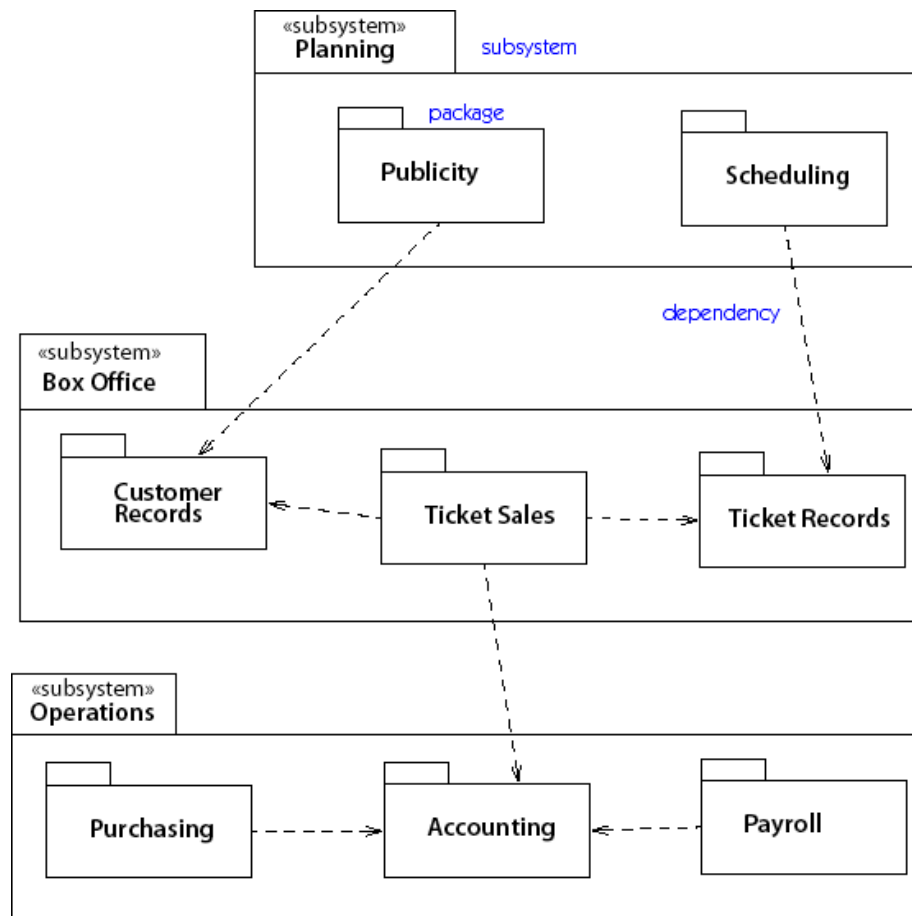


Figure 3-10. Packages

UML support: Deployment Diagram



Physical layout of run-time components on hardware nodes.

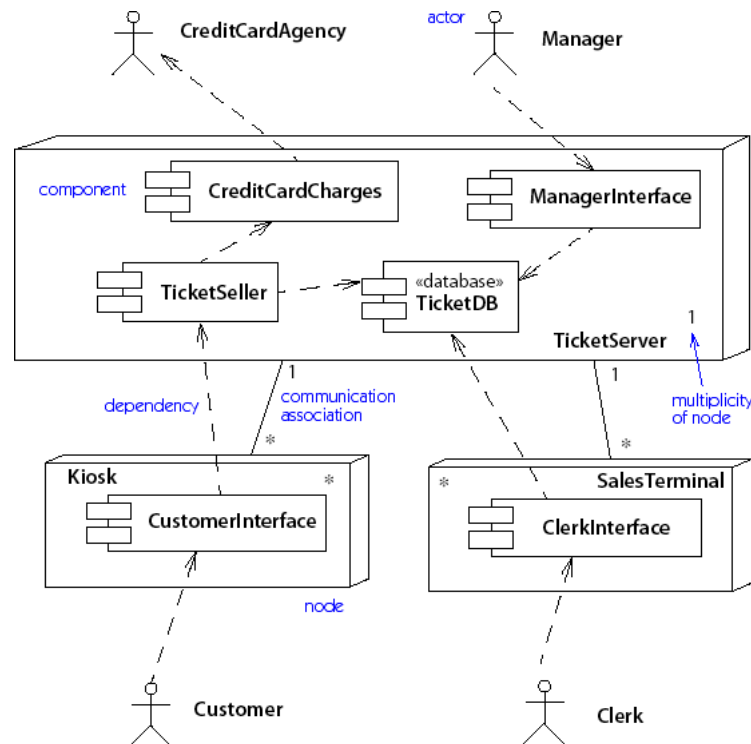


Figure 3-8. Deployment diagram (descriptor level)

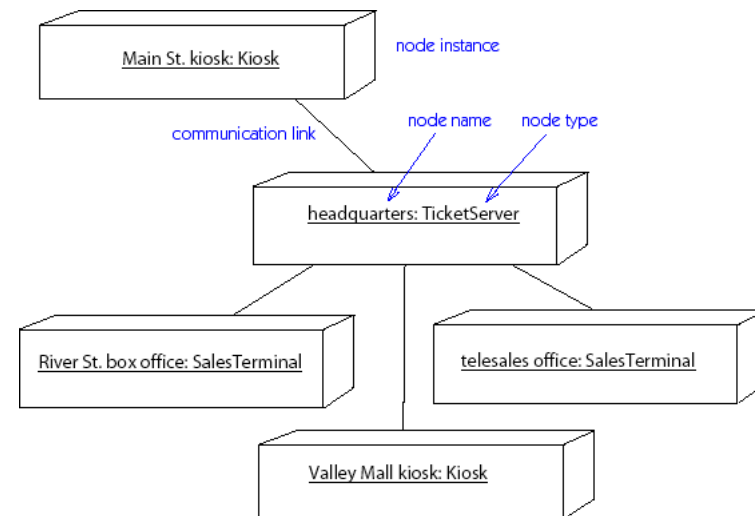


Figure 3-9. Deployment diagram (instance level)

Architecture Design



- There is no single correct design ...
 - Design for purpose ... (what the client wants to achieve)
 - Quality requirements: the determinator
- Consider two (2) or more design alternatives
 - Compare, debate (rationale), and deliberation ...
- Choose the one that most satisfies the client's (quality) priorities



Detailed Design

- As needed in the SDLC
- Approaches
 - OO Design
 - Function Driven Design
 - ...

Responsibility-Driven OO Design

– an example approach



- Finding Classes (not attributes/methods yet)
- **CRC Cards**
- Identifying Responsibilities
- Finding Collaborations

Principal References



- Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice* (3rd Edition), Addison-Wesley, 2013.
- Philippe B. Kruchten, *The 4+1 View Model of Architecture*, IEEE Software, 12(6): 42—50, November 1995.
- Ian Sommerville, *Software Engineering* (8th Edition), Addison-Wesley, 2007, Chapters 11 to 13.
- Desmond F. D'Souza and Alan Cameron Wills, *Objects, Components and Frameworks with UML*, Addison-Wesley, 1999, Chapter 12.
- Rebecca Wirfs-Brock and Alan McKean, *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley, 2003.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, Addison-Wesley, 1995.
- Paul Clements, et.al. *Documenting Software Architectures: Views and Beyond*, Addison Wesley, 2002.
- Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

Peer Reviews

- Peer Reviews are due:
 - End of Week 6
 - End of Week 12
 - Peer Review Form available on Canvas under **Modules/Unit Resources**
- Please make arrangements with your Project Supervisor on how to submit
- One-on-one meeting with supervisor – week 7

