Developing a Solution's IT Architecture

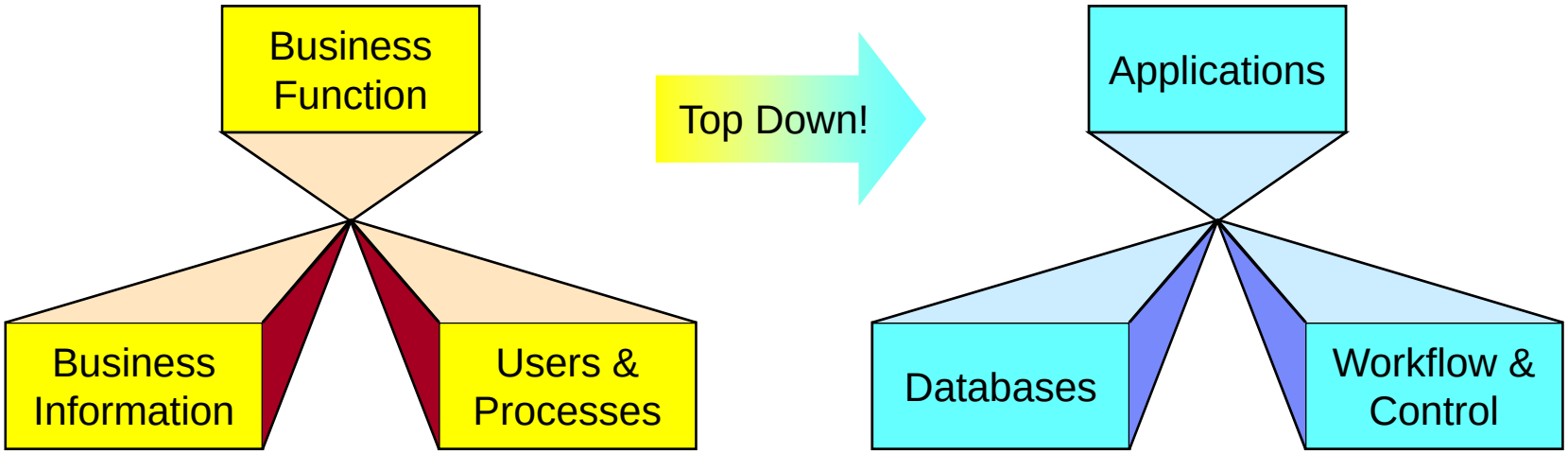
Using a "top down", requirements driven approach
Separating concerns: organizing the requirements and design into distinct parts
Incrementally developing business requirements and their IT solution
Complexity

Dr. Marcel Schlatter
IBM Distinguished Engineer
Member of the IBM Academy of Technology
marcel.schlatter@ch.ibm.com

The design of an IT solution to a business requirement must be based on business requirements

1) "top down" requirements driven

- In commercial systems, the business requirements are most easily described in terms of *who* is doing *what* to *which* information, *when* and *where.*



- The IT Architect, working with the Business Analyst, decides which parts of the business's requirements are to be supported by IT
- But there's more to it than "just" delivering the functional requirements…

The design of an IT solution to a business requirement must be based on business requirements ***including the non-functional requirements as well!***

Availability
- Scheduled service hours
- Outage costs
- Speed of service recovery
- Disaster recovery

Skills
- Development
- Users
- Operations

Scalability

Systems Management
- Event and Log Management
- Configuration Management
- Security Management
- Performance Management
- Scheduling
- Backup and Recovery

Process & Data Integrity

Cost
- Development
- Education & Rollout
- Operations

Performance
- Response time
- Throughput
- Capacity

Security
- Access to system/data
- Threats
- Controls

Timescales

Data Currency

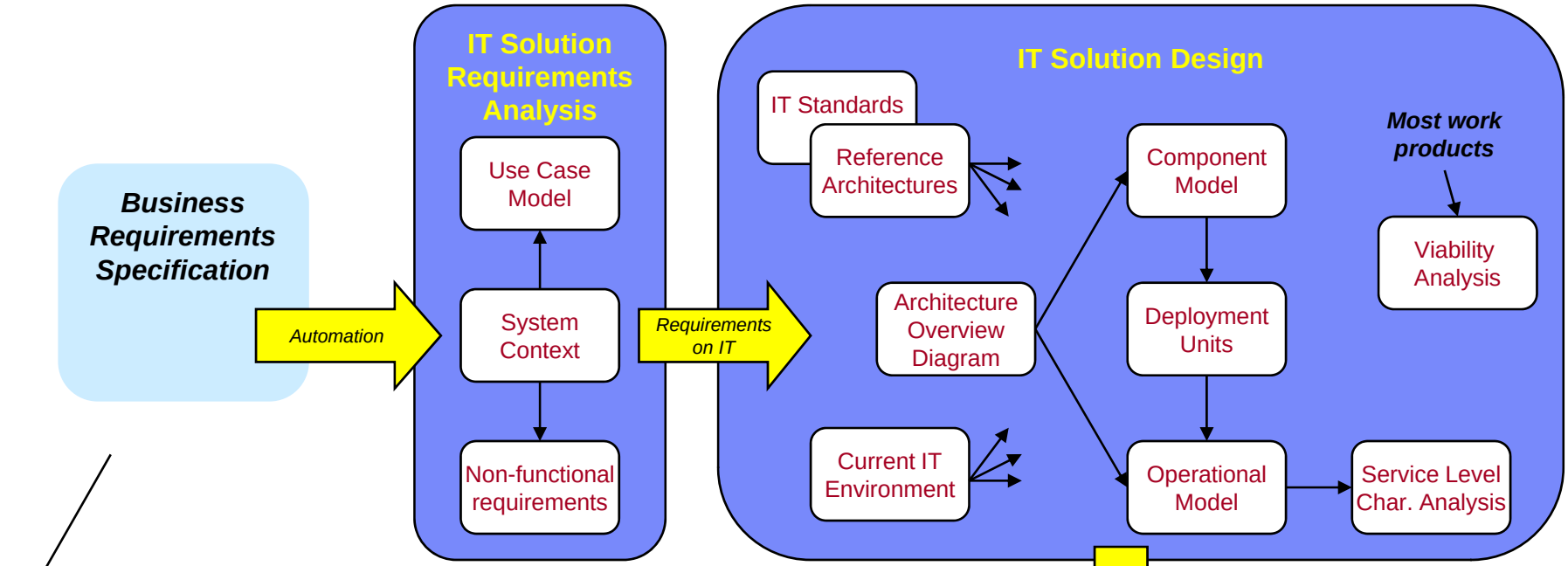
And don't forget the external constraints!

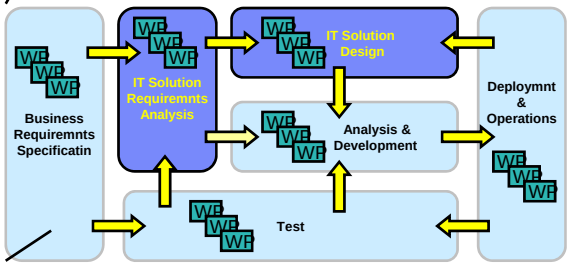
Key: "Service Level Characteristic (SLC)"

"Other Qualities"

Defining and documenting the various aspects of the IT solution's requirements and design is achieved by using a set of **IT Architecture work products**, each focused on a specific view of the IT system

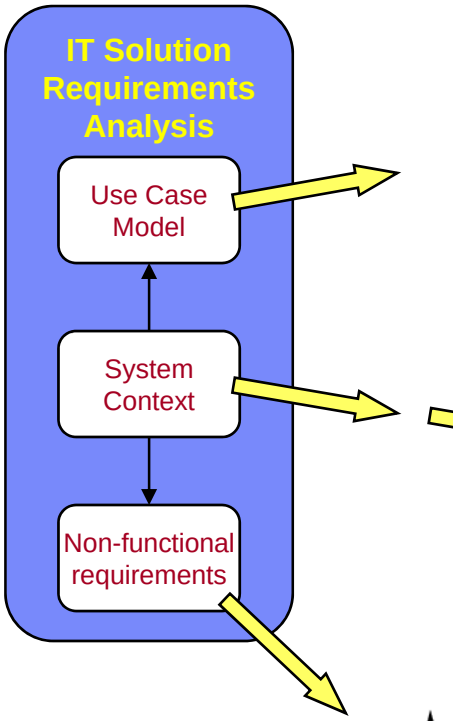
2) Separating concerns

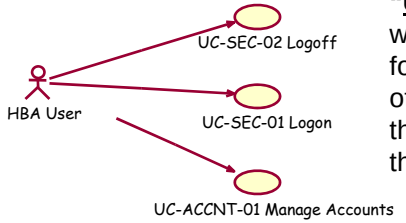




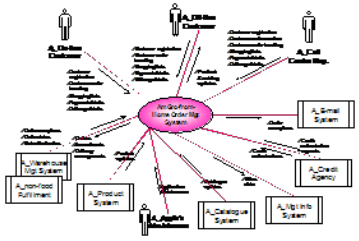
The IT architect uses three core work products to document the business requirements their IT System will support…

2) Separating concerns





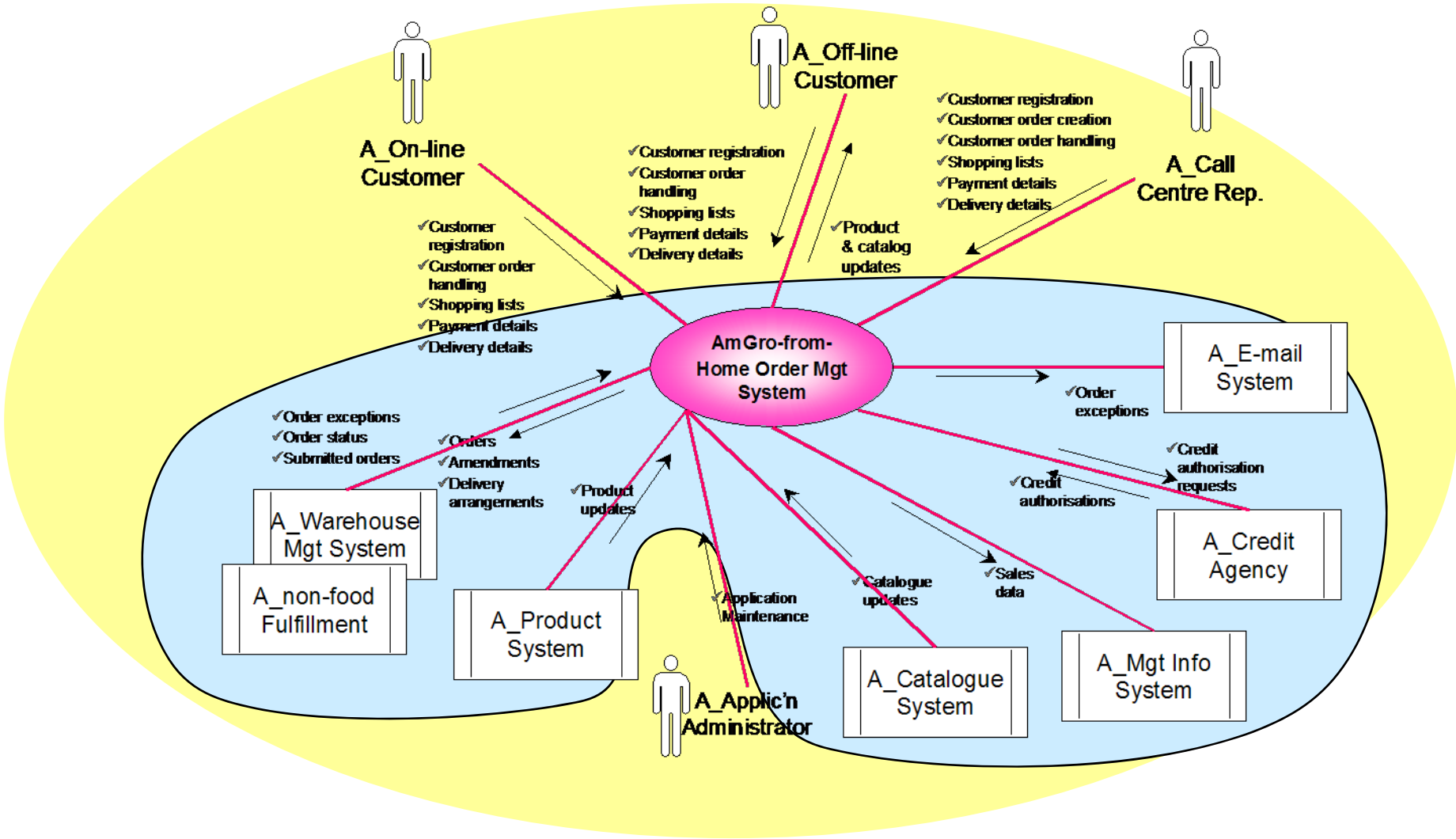
"<u>Use Case Model</u>" defines what the IT system must do for all its business (and other) users – it describes the system behaviour at the "automation boundary"



"<u>System Context</u>" defines the boundaries of the to be designed system, documenting the external users and other IT systems with which this system must interact

"<u>Non functional requirements</u>" describe the characteristics of the IT systems functionality, such as transaction response times and workload volumes

System Context

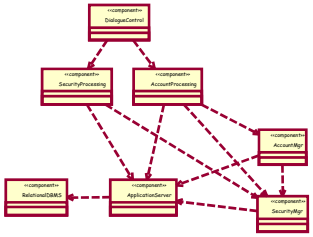


…four to document and communicate their IT system's design…

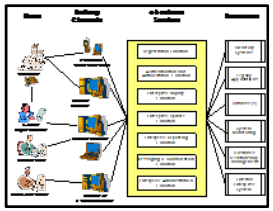
2) Separating concerns

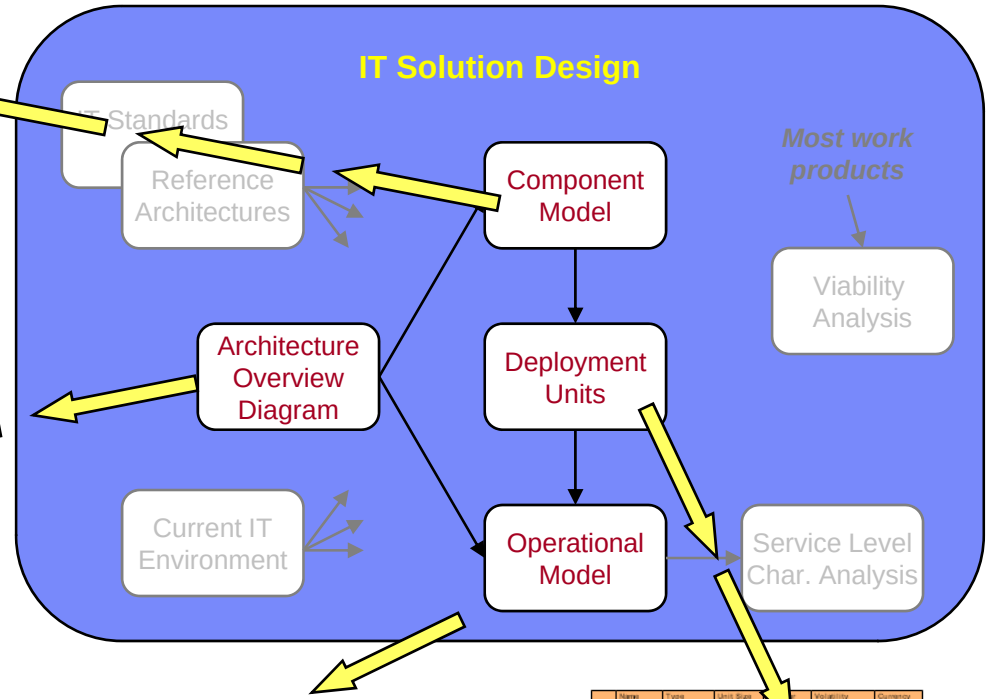
Software Engineering

"Component Model" describes the structure of an IT System in terms of its software components with their responsibilities, interfaces and relationships, and the way they collaborate to deliver the required functionality.



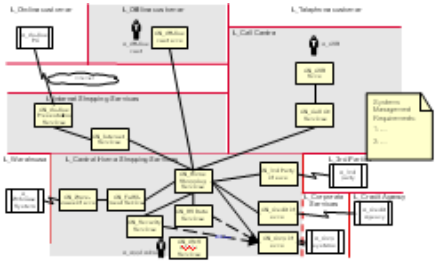
"Architecture Overview Diagram" provides a picture (not a model) of the whole IT system "on a page" as a means of communicating the salient points of the design. AODs are audience specific

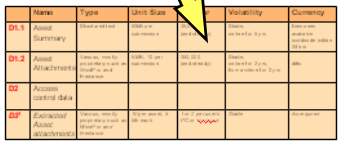




Systems Engineering

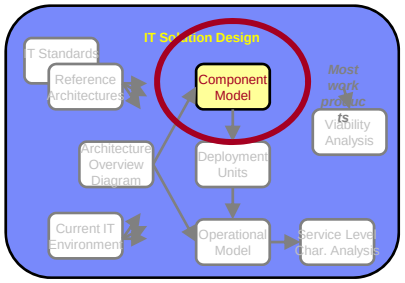
"Operational Model" defines the organisation of the IT system across locations, documenting the placement of the solution's components onto nodes connected across the organisation, in order to achieve the solution's operational NFRs

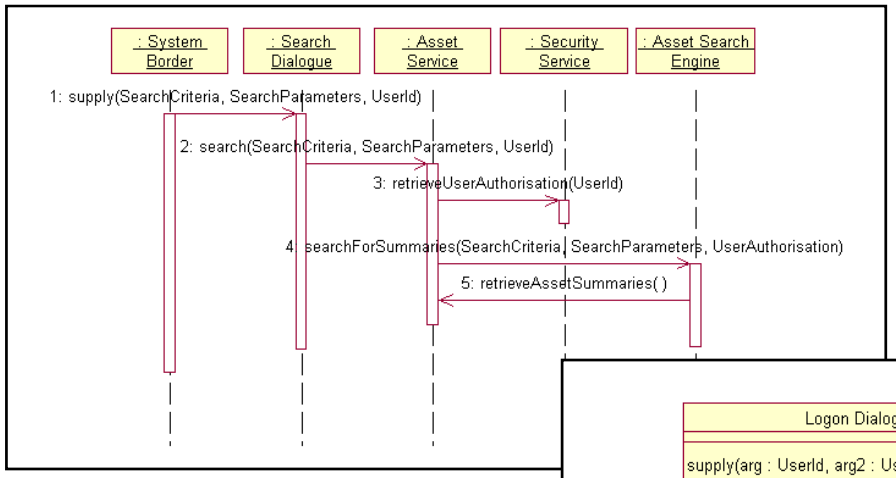




"Deployment Units" represent various aspects of components, as a convenient means of documenting their non functional requirements, as well as their placement across the Operational Model

two of which - the Component Model and Operational Model, deserve particular focus

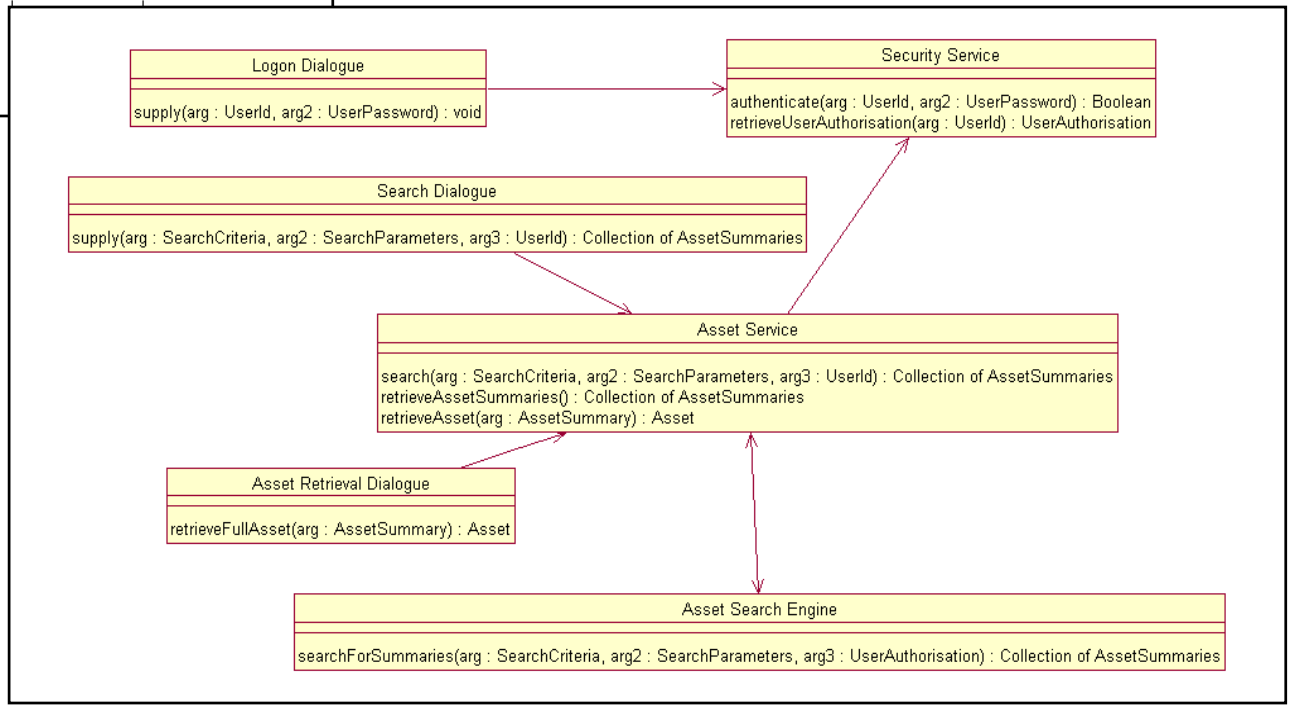




interaction diagram

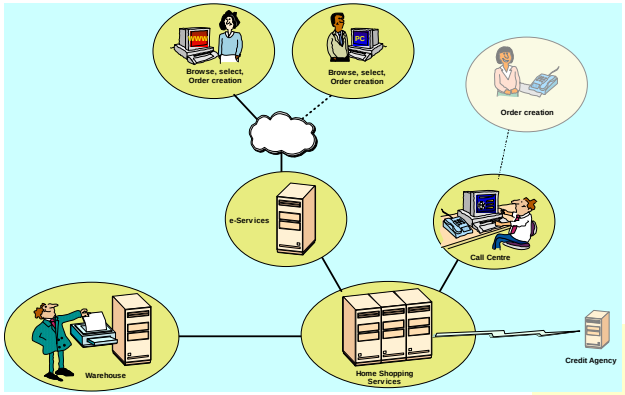
Component Model

Software Engineering

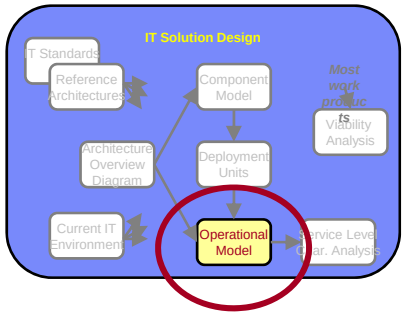


relationship diagram

two of which - the Component Model and Operational Model, deserve particular focus

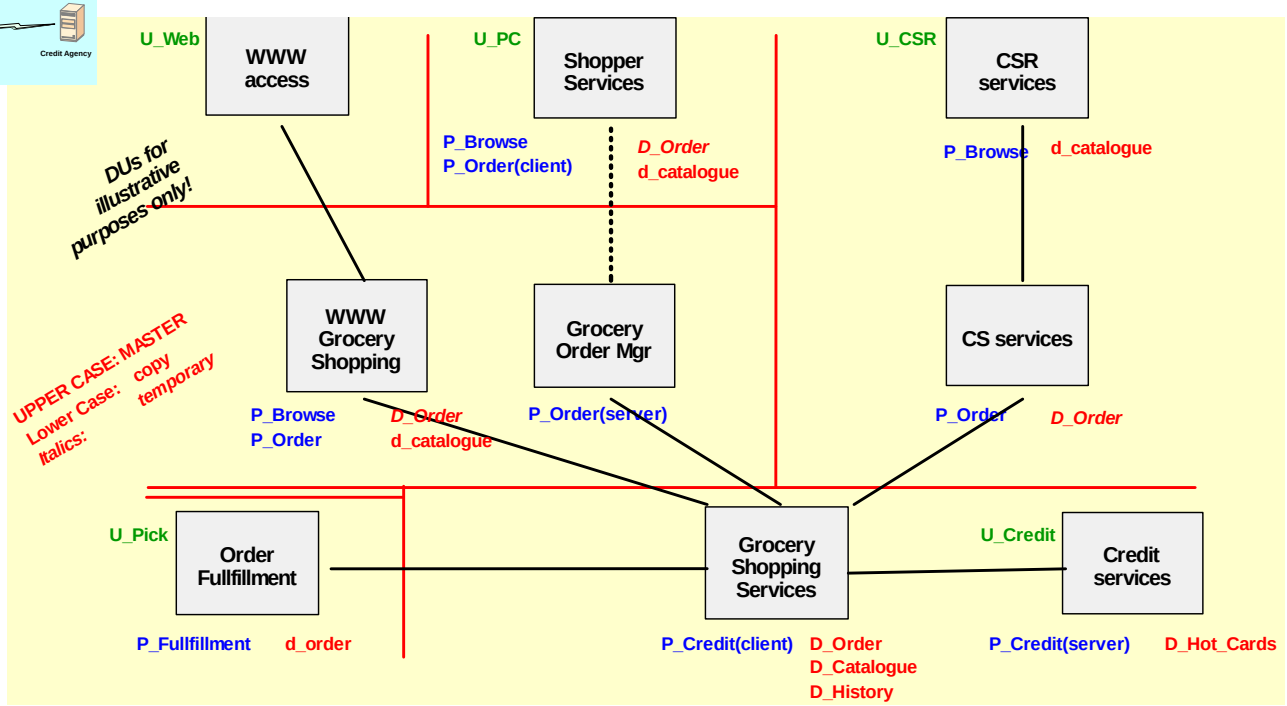


Geographic background



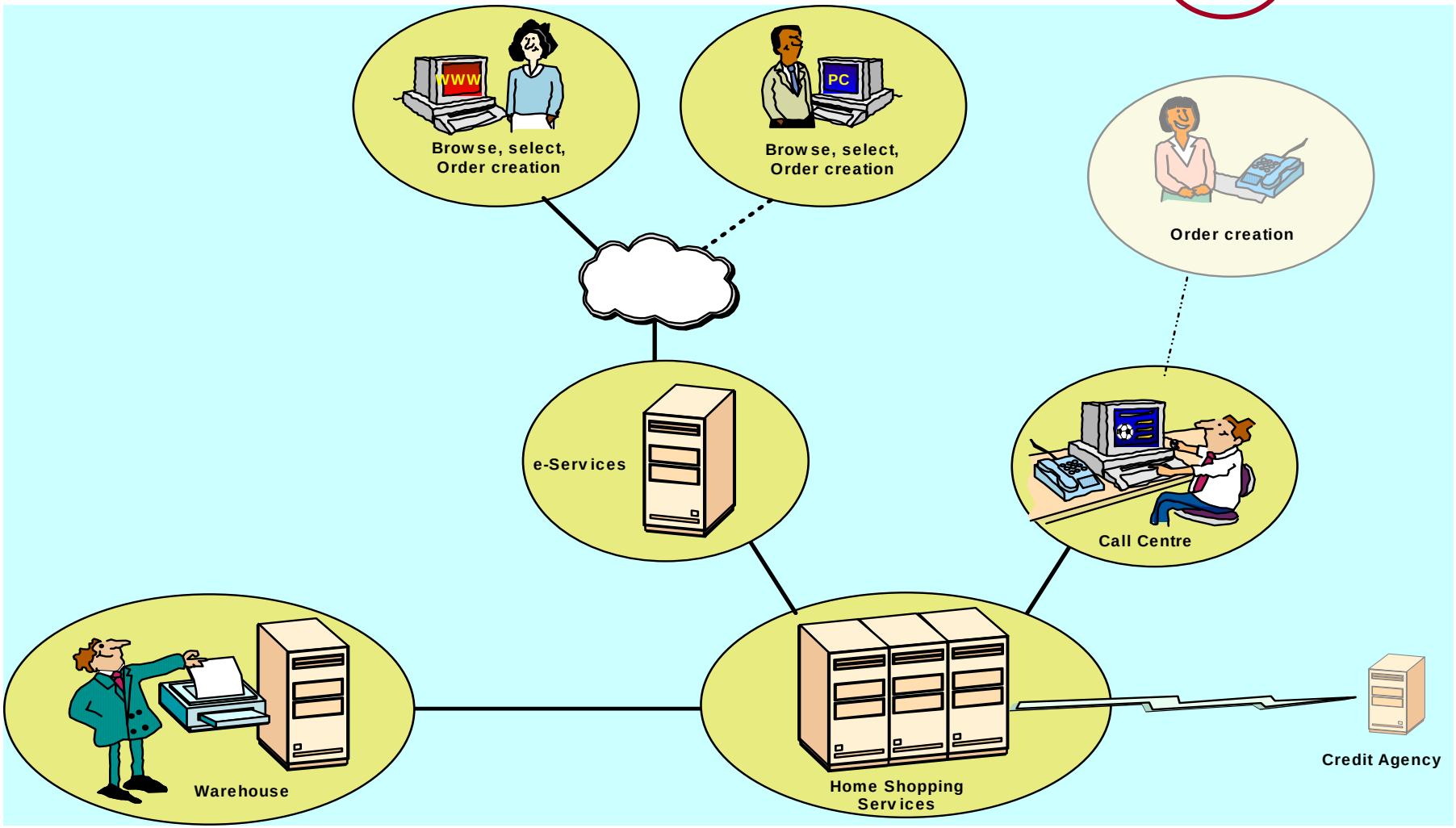
Operational Model

Systems Engineering

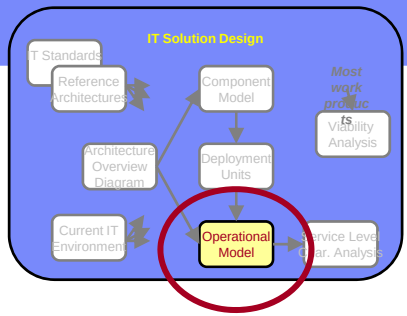


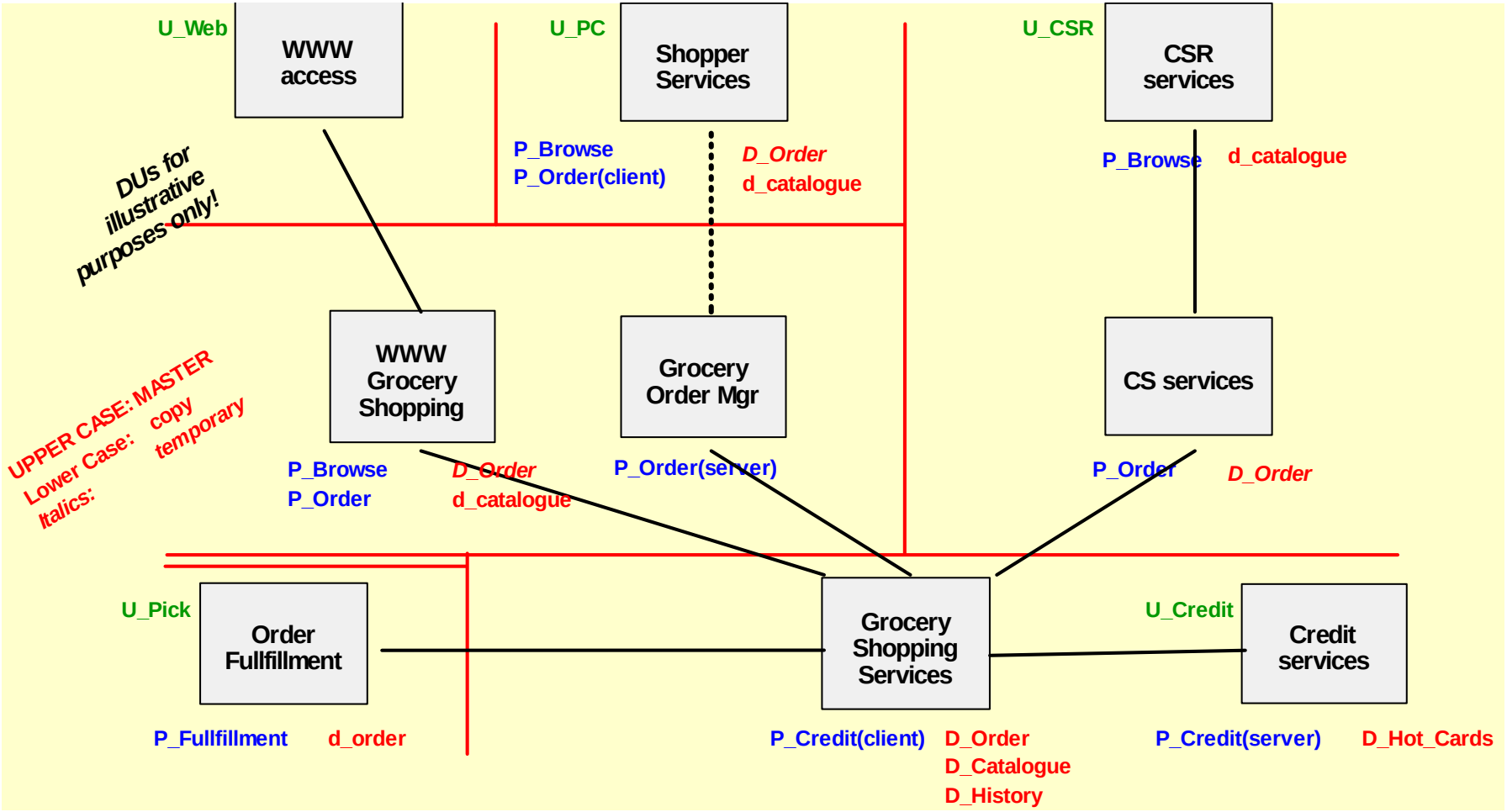
relationship diagram

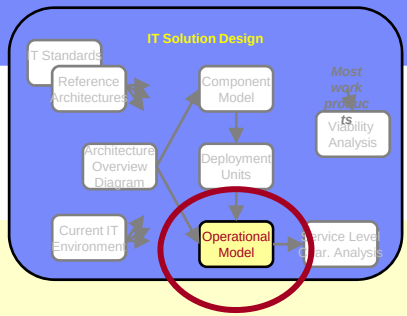
Operational Model -- Geographic Background



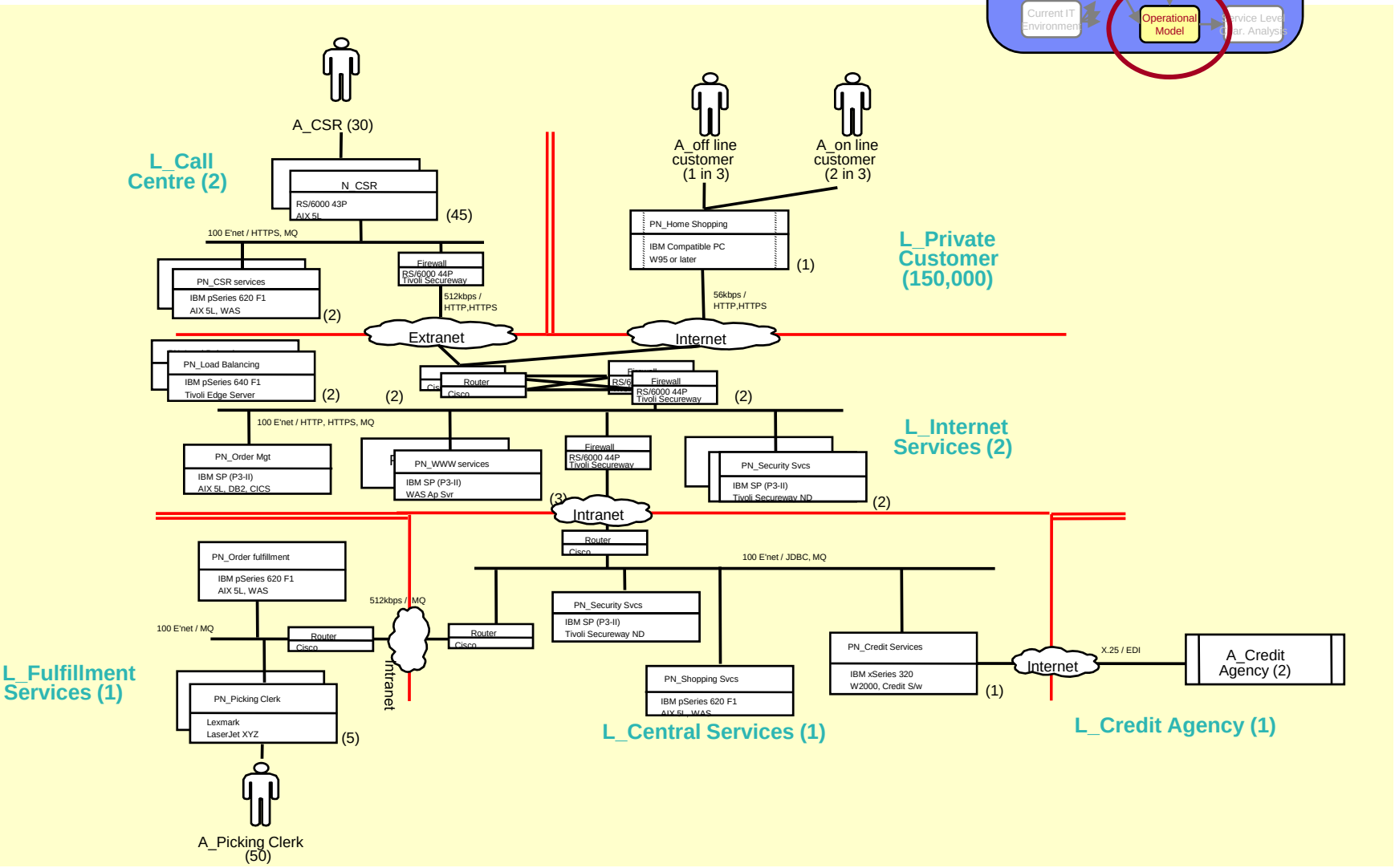
Specified Operational Model – Relationship Diagram





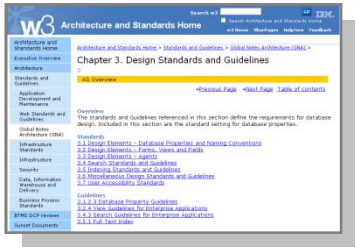


Physical Operational Model

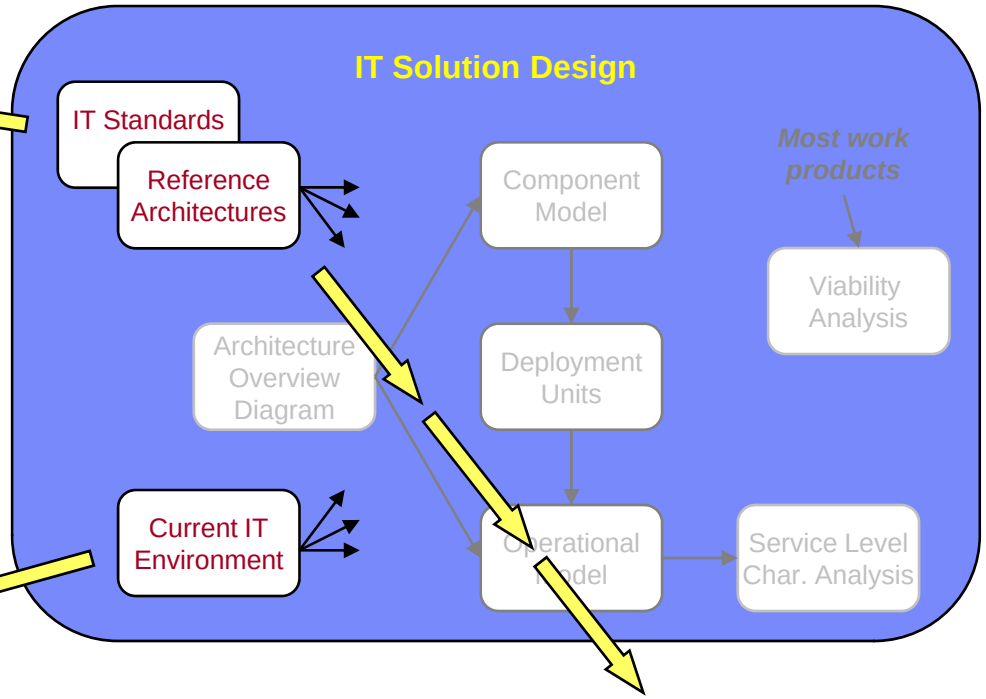


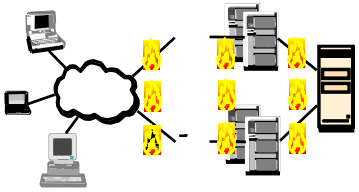
…three as a means of understanding the wider IT constraints placed on the solution by the enterprise or project…

2) Separating concerns



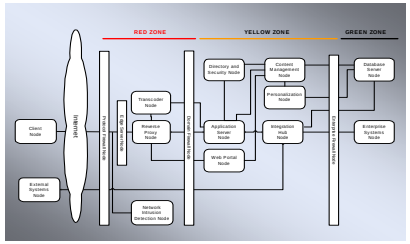
"IT Standards" document the project's IT design, development and deployment standards, and therefore normally include standards that range across all aspects of the IT Architect's work. They can be derived from external, enterprise level sources such as an Enterprise Architecture, or they may be agreed on from within the project.



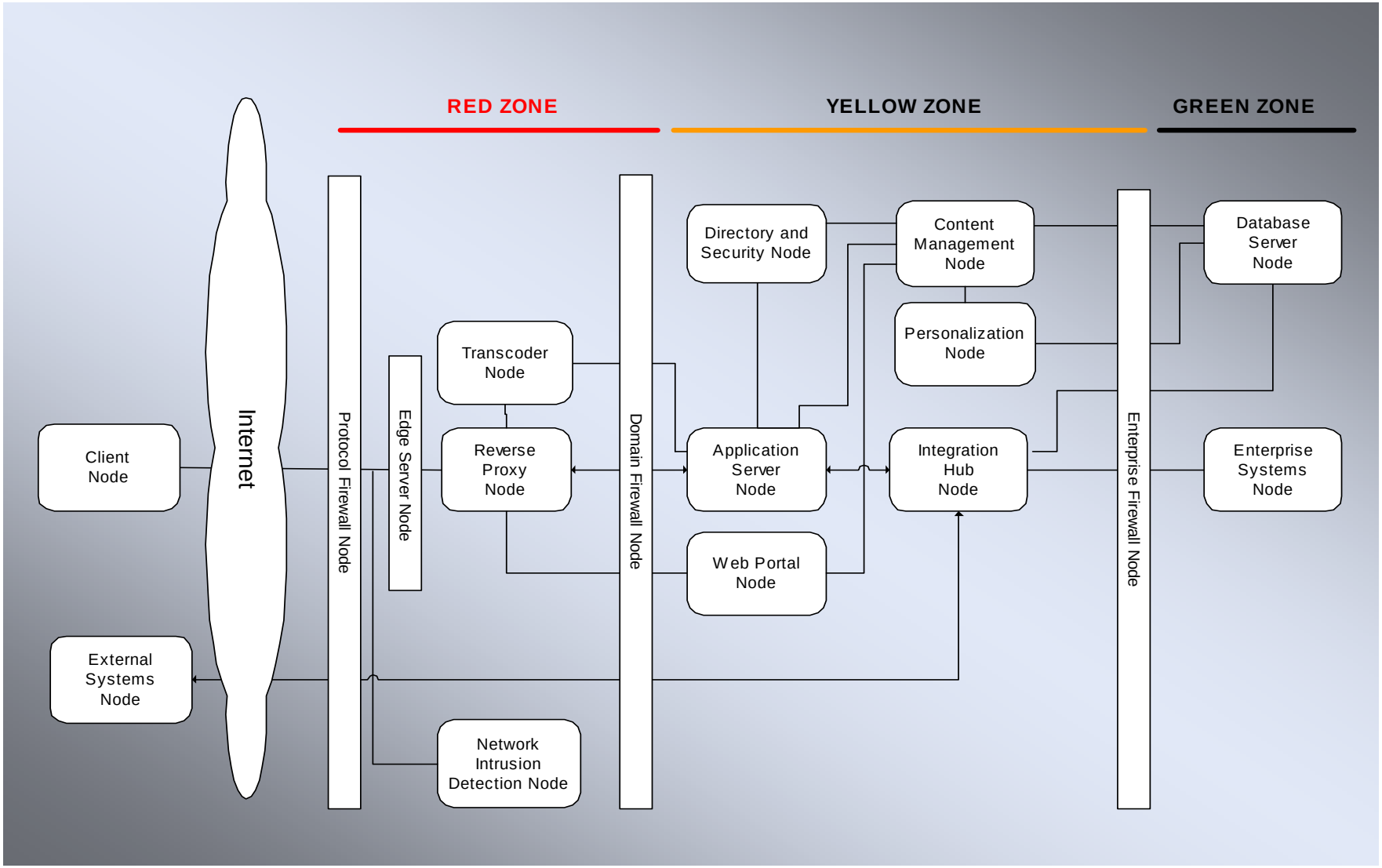


"Current IT" documents the environment into which the IT system will be deployed. (It is normal for a design to require modifications to an existing system)

"Reference Architectures" describe best practice patterns on the manner in which well understood requirements can be solved. RAs are often provided as part of an enterprise architecture, in which case they document the enterprise's preferred approaches to IT systems design, usually exploiting a standard set of IT system "building blocks"



An example of a reference architecture

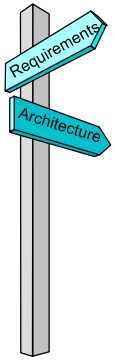


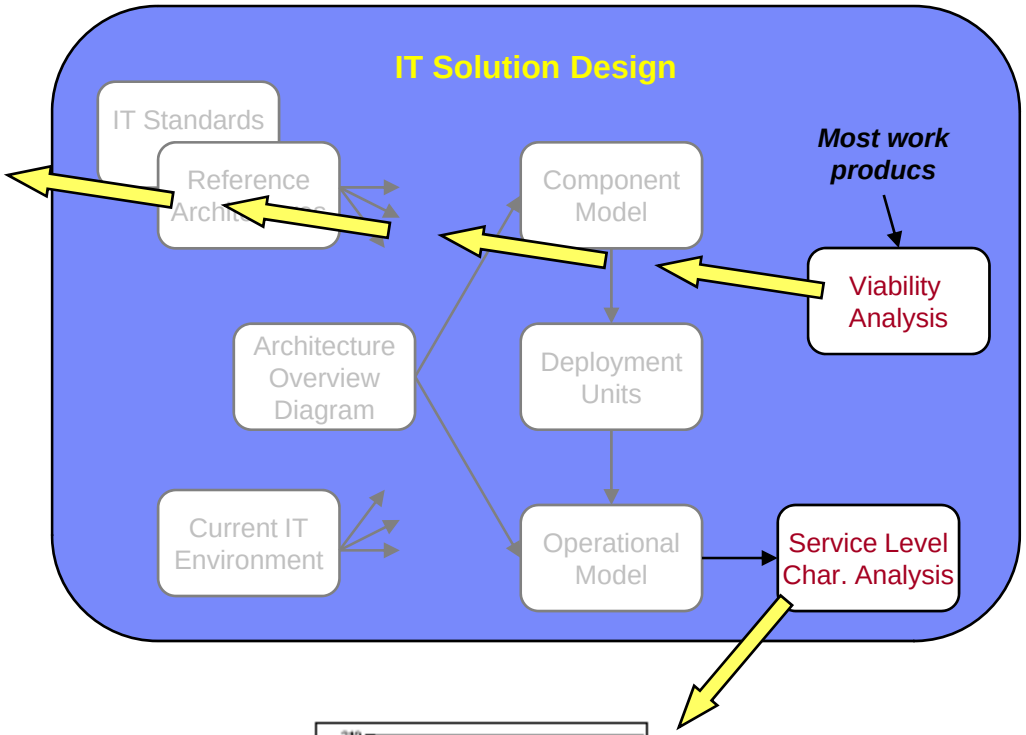
…and two to ensure the overall IT Architecture is viable

2) Separating concerns

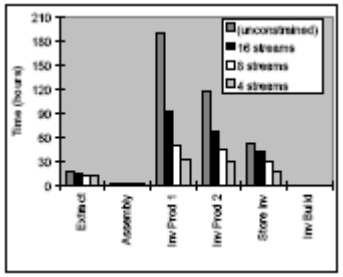
…can we do it within cost & time budget, with acceptable risk ?

"Viability Analysis" - a systematic approach, reviewing the IT system "from all angles" to help ensure it will actually work and meet the needs of the business. Includes an assessment of the IT system's Requirements (and constraints), functional and operational architecture, and analysis of it's ability to deliver the required performance, availability, security and all other non functional characteristics.





"Service Level Characteristics Analysis"
Focuses specifically on assessing the IT system's ability to achieve the solution's non functional requirements (NFRs), often comparing design options (e.g. centralised v distributed) and NFR tradeoffs (e.g. performance, or availability?)



Work products are not, generally, developed sequentially ("waterfall"). IT architects discuss "the art of the possible" throughout the project lifecycle with business analysts, developers and others, enabling the IT consequences of the business's requirements to be properly thought through…

3) Incremental Development

Waterfall…

Phase Deliverable	Phase A	Phase B	Phase C
Business Requirements Specification	**Comple**	*Ch control*	*Change control*
IT Solution Requirements Analysis		**mplete**	*Change control*
IT Solution Design			**Complete**

Phase Deliverable	Solution Outline	Macro Design	Micro Design
Business Requirements Specification	High Level, qualitative	Complete	*Change control*
IT Solution Requirements Analysis	Outline System Requirements	IT s pe fully de d, key NFRs	Complete
IT Solution Design	Key Elemen identified	utline solution defined	Complete

iterative…

…so that the project team (business and all parts of IT) can work more closely together, as well as helping ensure the project deals with difficulties very early in the lifecycle.

3) Incremental Development

Phase Deliverable	Phase A	Phase B	Phase C
Business Requirements Specification	**Complete**	*Change control*	*Change control*
IT Solution Requirements Analysis		**Complete**	*Change control*
IT Solution Design			**Complete**

Bang!

Phase Deliverable	Solution Outline	Macro Design	Micro Design
Business Requirements Specification	High Level, qualitative	Complete	*Change control*
IT Solution Requirements Analysis	Outline System Functionality	IT scope fully defined, key NFRs	Complete
IT Solution Design	Key Components Identified	Outline solution defined	Complete

Bang!

Catch “show-stopping problems” early in the project, enabling (if necessary) the project to be terminated at much less cost

Complexity

Some considerations in the context of developing a solution‘s IT Architecture

Murphy's Law "If anything can go wrong, it will."

- **Simplify, simplify, simplify.**
- **The first line of defense against complexity is simplicity of design.**
- **Simplify, combine, and eliminate.**
- **The most reliable part … is the one that isn't there – because it isn't needed.**

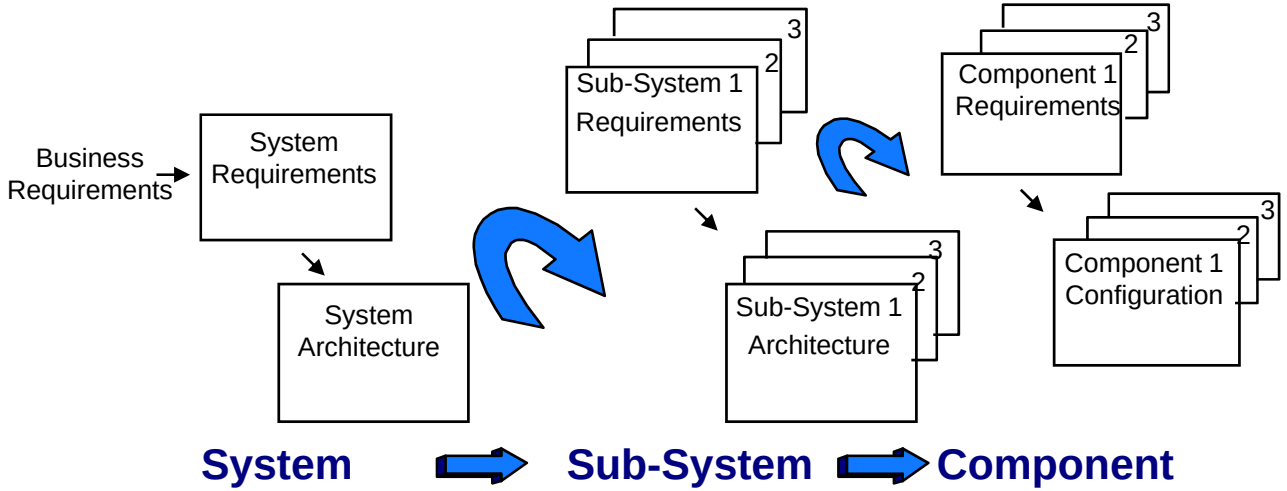
Maier, Mark W. and Rechtin, Eberhardt, The Art of Systems Architecting, CRC Press, 2002.

System Architecting Hall of Shame Candidates

- **OS/2**
  - IBM solved the **wrong problem** — needed to produce an inexpensive product with easy to use features, not a well tested, complicated consumer product
  - **Result:** Relegated to IT oblivion
- **Beta**
  - Sony solved the **wrong problem** — needed an inexpensive, flexible product with widespread licensing
  - **Result:** Loss of entire market category
- **Napoleon's Grande Armée**
  - Napoleon solved the **wrong problem** and equipped his army to fight the Russians, not typhus
  - **Result:** As we might say, the rest is history. System engineering principles don't just apply to IT systems
- **Denver International Airport**
  - Solution that was **too complicated**, not maintainable and not implementable, and had no backup solution
  - **Result:** Nearly 2 year delay in opening a $5 billion airport

Fundamentals of System Complexity: Introduction to partitioning, aggregating and layering

- **The objective is to split a complex system into manageable chunks such as sub-systems and components and in doing so**
  - **Allocate** requirements specified for the system down to the component to provide meaningful input to design
  - **Distribute** or apportion quantitative and qualitative requirements
  - Make sure there are no requirements **gaps** in the allocation
  - Maintain two way **traceability** between allocated and business requirements

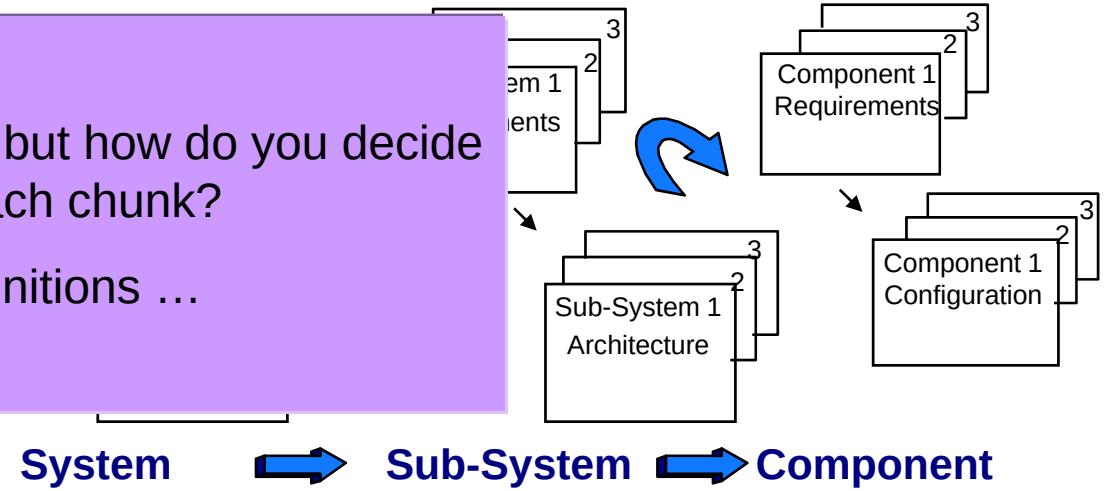


Fundamentals of System Complexity: Introduction to partitioning, aggregating and layering

- **The objective is to split a complex system into manageable chunks such as sub-systems and components and in doing so**
  - **Allocate** requirements specified for the system down to the component to provide meaningful input to design
  - **Distribute** or apportion quantitative and qualitative requirements
  - Make sure there are no requirements **gaps** in the allocation
  - Maintain two way **traceability** between allocated and business requirements

This seems simple enough, but how do you decide what goes in each chunk?

First some definitions …



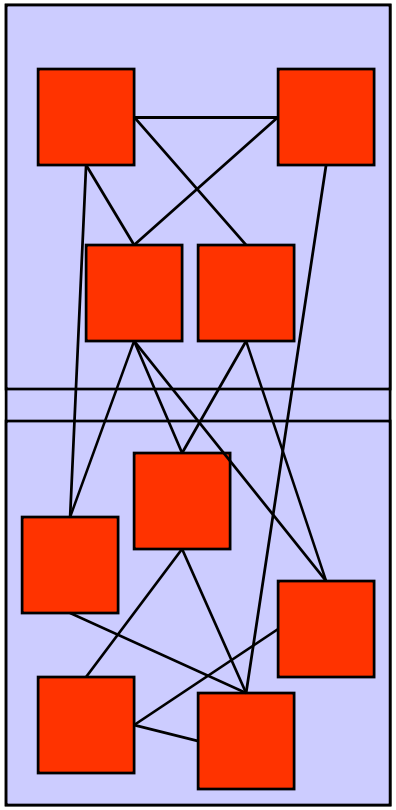
Important Terms in Simplifying Solution Complexity

- **Partitioning divides the system into several logically independent components**
- **Aggregation combines highly *coupled* components into a highly *cohesive* component**
  - **Cohesion** is the manner and degree to which the actions performed by those components are related to one another
    - Goal is to have high cohesion within components
  - **Coupling** between components is the manner and degree to which the two components are interdependent
    - Goal is to have low coupling between components
- **Layering is the decomposition of a system according to different levels of *abstractions***
  - **Abstraction** is the representation of a layer or interface into a higher altitude view of functionality

Aggregation and Partitioning

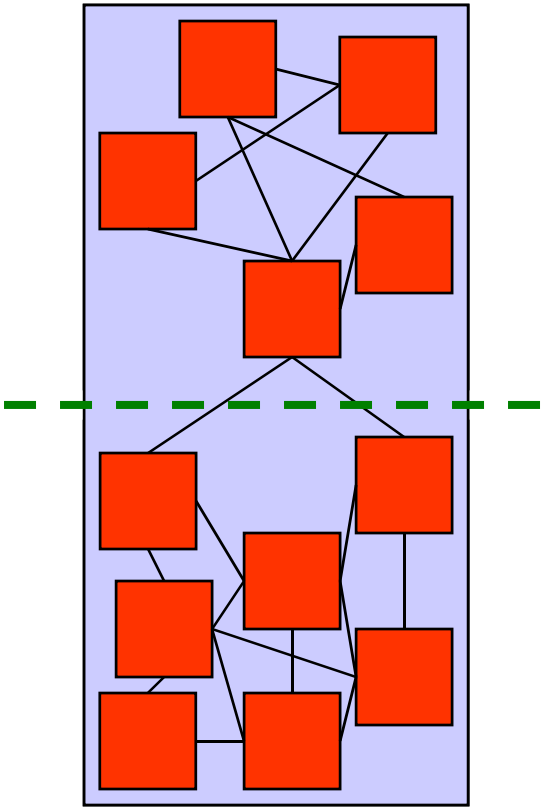
Consider Aggregation

Inter-node connectivity for many nodes is as high as or higher than intra-node connectivity



Consider Partitioning

Connectivity between the two sets is limited to only 2 links



Classic Heuristics for Simplification: Partitioning

- **Do not slice through regions where high rates of information exchange are required.**
- **The greatest leverage in architecting is at the interfaces.**
  - Guidelines for a good quality interface specification: they must be simple, unambiguous, complete, concise, and focus on substance.
  - The efficient architect, using contextual sense, continually looks for likely misfits and redesigns the architecture so as to eliminate or minimize them.
  - **It is inadequate to architect up to the boundaries or interfaces of a system; one must architect across them.**

Maier, Mark W. and Rechtin, Eberhardt, The Art of Systems Architecting, CRC Press, 2002.

Classic Heuristics for Simplification: Aggregation

- **Group elements that are strongly related to each other, separate elements that are unrelated.**
- **Subsystem interfaces should be drawn so that each subsystem can be implemented independently of the specific implementation of the subsystems to which it interfaces**
- **Choose a configuration with minimal communications between the subsystems.**
  - Choose the elements so that they are as independent as possible; that is, elements with low external complexity (low coupling) and high internal complexity (high cohesion).
  - Choose a configuration in which local activity is high speed and global activity is slow change.
- **Poor aggregation results in gray boundaries**
  - Aggregate around “testable” subunits of the product; partition around logical subassemblies.
  - Iterate the partition / aggregation procedure until a model consisting of 7 ± 2 chunks emerge.
  - The optimum number of architectural elements is the amount that leads to distinct action, not general planning.
- **System structure should resemble functional structure.**
  - Except for good and sufficient reasons, functional and physical structure should match.
  - The architecture of a support element must fit that of the system which it supports. It is easier to match a support system to the human it supports than the reverse.

What are some others that you use?

Maier, Mark W. and Rechtin, Eberhardt, The Art of Systems Architecting, CRC Press, 2002.

Partitioning and Aggregating: Developing and Integrating the System(s)….

Developing

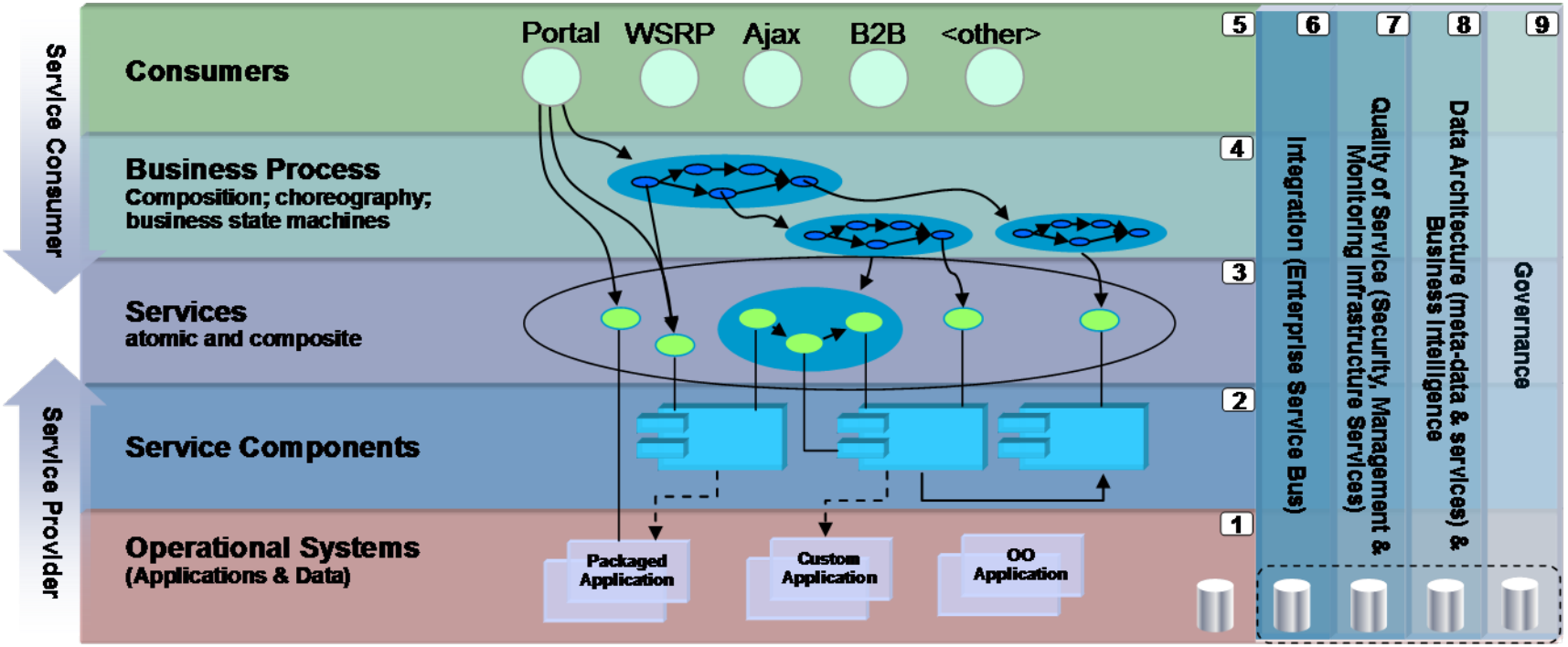
- **Consider who will be developing each piece of the system**
- **Design the partitions such that each developing organization is supplying a whole, separate, testable piece**
- **Minimize the number of "hops" any piece of the system takes before it reaches you**
  - Any piece of software should go through only one or two other organizations before it reaches final integration
- **As you refine the requirements for each piece, note how each should be tested for compliance and acceptance**
  - Is special test data or equipment needed by that development organization?
  - Who will provide it?

Reassembling

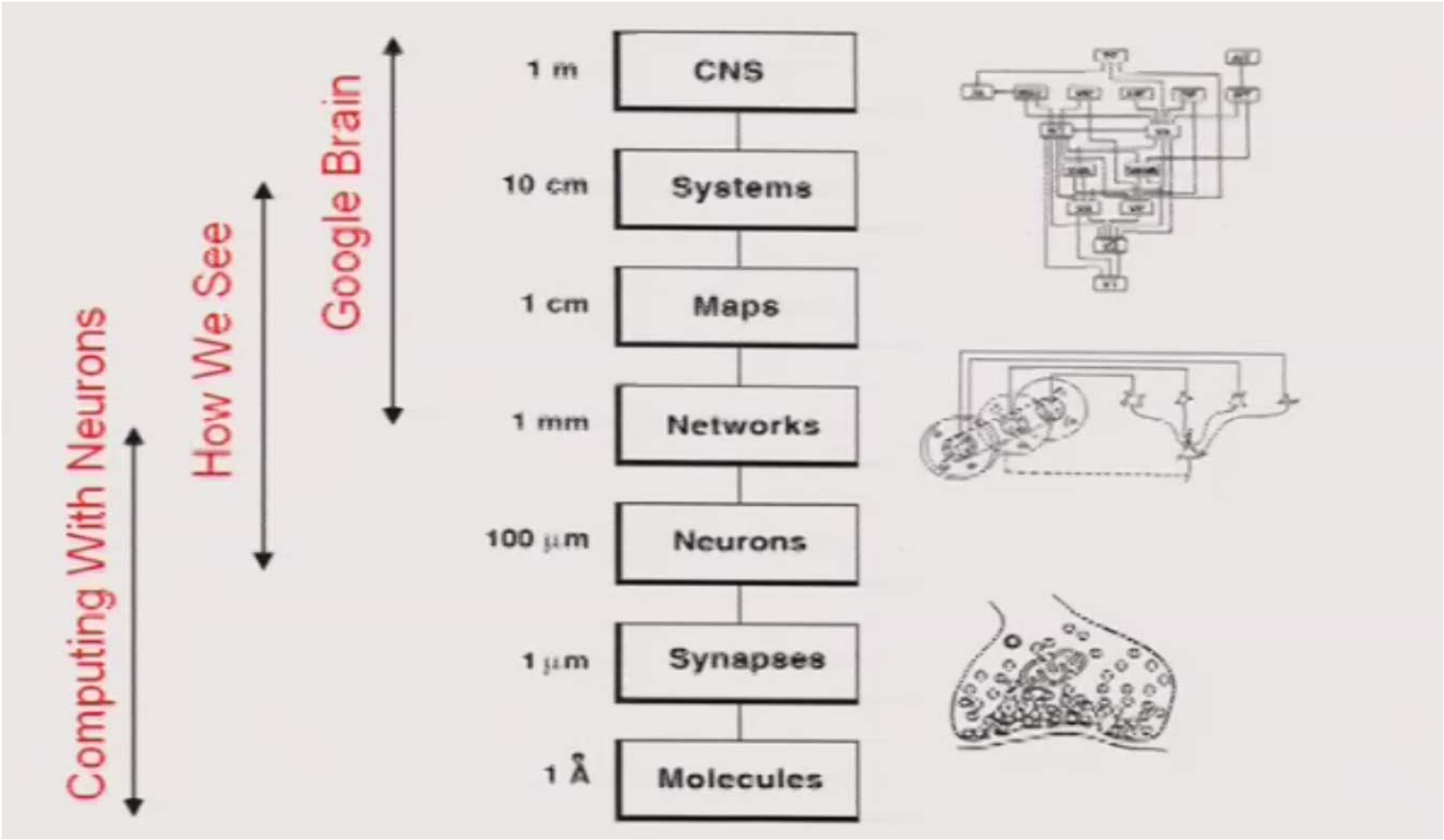
- **Consider how the components and elements of the system will be integrated**
  - Staged integration
  - Big Bang
- **What requirements will that integration approach drive?**
  - Test hooks/modes in the software
  - Integration environments
  - Test environments
  - Special test equipment or test data
  - Interface drivers
- **How will the system be deployed and activated?**
  - Any requirements driven by deployment plans?
  - If replacing legacy system, do you need a special mode for parallel operations?

Layering

- Layering can be used to manage complexity by separating and / or controlling dependencies between systems and between layers… as in Service Oriented Architecture



Layering in the architecture of natural things (Pauli Lectures 2008, ETH, Prof. Terrence J. Sejnowski)



The first line of defense against complexity is simplicity

- **Simplify. Simplify. Simplify.**
- **Simplify, combine, and eliminate.**
- The most reliable part on an airplane is the one that isn't — because it isn't needed. [DC-9 Chief Engineer, 1989]
- If you can't explain it in five minutes, either you don't understand it or it doesn't work.

There is a class of problems better avoided than solved

Prevention is better than cure, in particular if the illness is unmastered complexity, for which no cure exists.

E. W. Dijkstra, „The Tide, not the Waves", in „Beyond Calculation", P.J. Denning, R.M. Metcalfe (ed.), Springer, 1997.

Exercises on Heuristics

- **Choose a system, software product, or software development process with which you are familiar and assess it using heuristics**
  - What was the result?
  - Which heuristics are or were particularly applicable?
  - What further heuristics were suggested by the system chosen?
  - Were any of the heuristics clearly incorrect for this system?
  - If so, why?

- **Try to spot heuristics and insights in the technical literature.**
  - Some are easy; they are often listed as principles or rules.
  - The more difficult ones are buried in the text but contain the essence of the article or state something of far broader application

- **Try to create a heuristic of your own — a guide to action, decision making, or to instruction of others.**