CSE 301
Software Design and Architecture

2. Design Concepts & Software Quality

Dr. Salisu Garba
Salisu.garba@slu.edu.ng

Quality Factors of SD

- Modularization
- Concurrency
- Cohesion
- Coupling
- Information hiding/Abstraction/Encapsulation

Modularization

- Modularization is a technique used to divide a software system into multiple, discrete and independent modules, which are expected to be capable of carrying out task(s) independently.
- Designers tend to design modules such that they can be executed and/or compiled separately and independently.
- Modular design unintentionally follows the rules of 'divide and conquer' problem-solving strategy.
- Smaller components are easier to maintain, easier to manage, easier to understand, easier fault isolation etc.

Concurrency

- Previously, all softwares were meant to be executed sequentially. Say, a software has multiple modules, then only one of all the modules can be found active at any time of execution.
- concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel.
- E.g. The spell check feature in word processor is a module of software, which runs alongside the word processor itself.

Coupling

- A measure of how strongly components are interconnected (level of inter-dependability among modules)
- Tightly coupled components share data (common coupling) or exchange control information (control coupling)
- Loose coupling is achieved by not having shared data, or at least restricting access (promotes separation of concerns)
- Object-oriented design promotes loose coupling, i.e. a class is coupled with its super-class

Cohesion

- Defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better the program designed.
- A measure of how functionally related the parts are
- For example, strong cohesion exists when all parts of a component contribute different aspects of related functions
- Strong cohesion promotes understanding and reasoning, and thus provides dividends with respect to maintenance

Information Hiding

- Information Hiding is a **goal**, abstraction is a **process**, and encapsulation is a **technique**.
- Design the modules in such a way that **information** (data & procedures) contained in one module is **inaccessible to other modules that have no need for such information**.
- A module having **high cohesion** and **low coupling** is said to be **functionally independent** of other modules.
- **Benefits:**
  - When modifications are required, it reduces the chances of propagating to other modules.
  - Error isolation
  - Scope of Re-use
  - Understandability

Introduction to SQ

- Conformance to explicitly-stated functional and non-functional requirements, explicitly-documented development standards, and implicit characteristics that are expected of all professionally developed software.
    - Lack of conformance to requirements will mean lack of quality.
    - Lack of conformance to development standards will mean lack of quality.
    - The implicit requirements, e.g. good maintainability, must still be followed. Else, software quality is suspect.

Standard Organizations

- ISO 9001 is the most general standard that applies to organizations concerned with the quality process to design, develop and maintain products
- ANSI (American National Standards Institute), NATO (North Atlantic Treaty Organization), IEEE (Institute of Electrical and Electronic Engineers)
- SON (Standard Organization of Nigeria)

Quality assurance teams that are developing standards for company should normally base their organizational standards on national and international standards

Software quality models

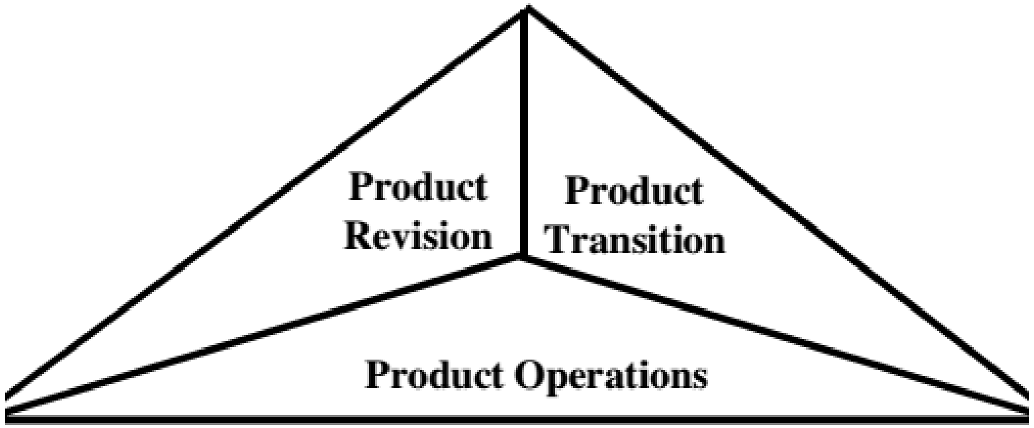
- Quality is the **excellence of the product or service**.
  - From a user's point of view, quality is '**fitness for purpose**'.
  - The **value-based** view of quality is concerned with the ability to provide **what the customer requires at a price that they can afford**.
  - From the **manufacturing** point of view, the quality of a product is the **conformance to specification**.
  - The product view sees the quality of a product as tied to **inherent characteristics** of the product.

Hierarchical models

- **McCall** divided software quality attributes into 3 groups

- Each group represents the quality with respect to one aspect of the software system while the attributes in the group contribute to that aspect.

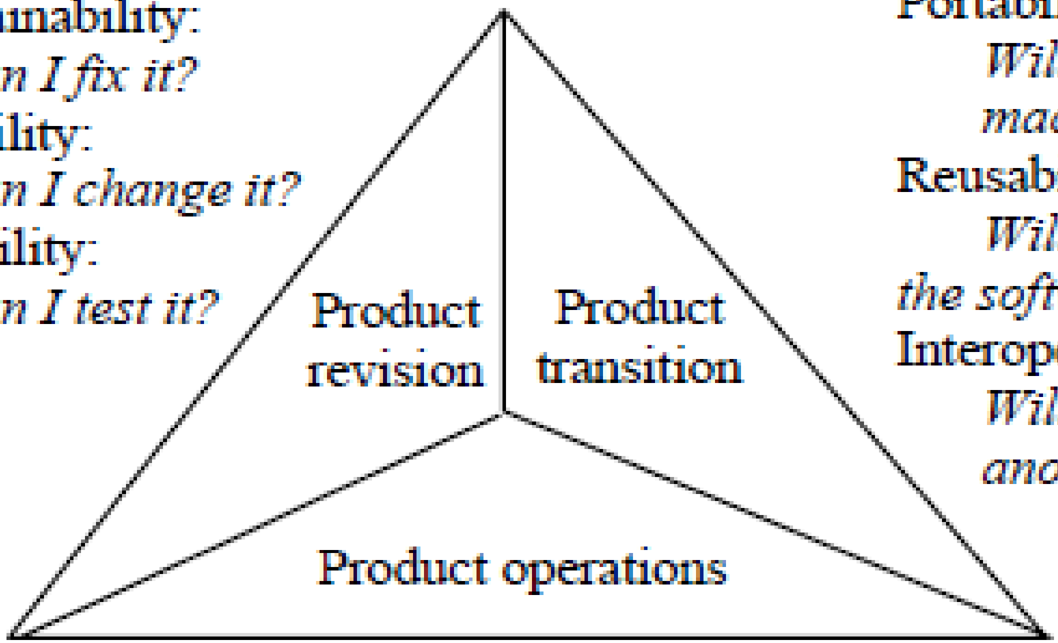
- Each quality attribute is defined by a question so that the quality of the software system can be assessed by answering the question.



McCall Model of Software Quality

Maintainability:
Can I fix it?
Flexibility:
Can I change it?
Testability:
Can I test it?

Portability:
Will I be able to use it on another machine?
Reusability:
Will I be able to reuse some of the software?
Interoperatability:
Will I be able to interface it with another machine/software?



Correctness:	*Does it do what I want?*
Reliability:	*Does it do it accurately all the time?*
Efficiency:	*Will it run on my machine as well as it can?*
Integrity:	*Is it secure?*
Usability:	*Can I run it?*

Relational models

- Perry's model contains three types of relationship between the quality attributes.
  - The direct relationship
  - The inverse relationship
  - The neutral relationship

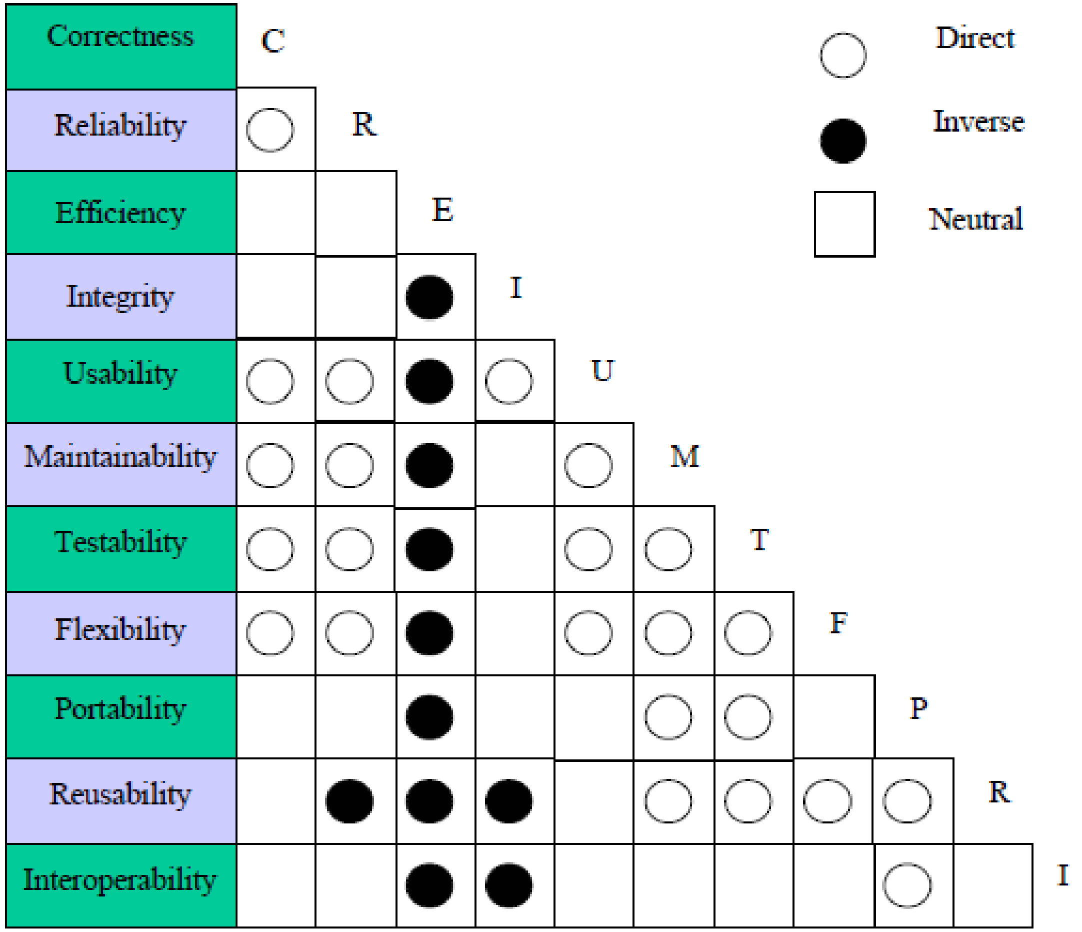


Figure 2.2 Perry's relational model of software quality

Quality Attributes – Design Objectives

User Needs	User Concerns with the Software System	Non-functional Categories
Operation How well does the system perform for daily use?	How well is it guarded against unauthorized access?	Access Security (ACS)
	How dependable is it during normal operation times?	Availability (AVL)
	How fast, how many, and how well does it respond?	Efficiency (EFC)
	How accurate and authentic is the data?	Integrity (INT)
	How immune is the system to failure?	Reliability (REL)
	How resilient is the system from failure?	Survivability (SRV)
	How easy is it to learn and operate the system?	Usability (USE)
Revision How easy is to correct errors and add on functions?	How easy is it to modify to work in different environments?	Flexibility (FLX)
	How easy is it to upkeep and repair?	Maintainability (MNT)
	How easy is it to expand or upgrade its capabilities?	Scalability (SCL)
	How easy is it to show it performs its functions?	Verifiability (VER)
Transition How easy is to adapt to changes in the technical environment?	How easy is it to interface with another system?	Interoperability (IOP)
	How easy is it to transport?	Portability (POR)
	How easy is it to convert for use in another system?	Reusability (REU)

Software Reliability

- Most hardware-related reliability models are predicated on failure due to **wear** rather than failure due to **design defects**.

- The opposite is true for software: in fact, all software failures can be traced to design or implementation problems; wear does not enter into the picture.

    - *Software Reliability* is the probability of failure-free operation of a computer software in a specified environment for a specified time.
    - *Software Availability* is the probability that a program is operating according to requirements at a given point in time.

Reliability Metrics

- A reliability metric is an indicator of how broken a software is.
- Metrics are best weighted by the severity of errors.
- A **minor error every hour** is better than a **catastrophe every month**.
- **MTBF = MTTF + MTTR**
- **Mean Time Between Failure (MTBF)** which measures how long a program is likely to run before it does something bad like crash. Where MTTF and MTTR are **mean time to failure** and **mean time to repair** respectively.
- **Reliability = MTTF/ (MTTF + MTTR) * 100%**

OR

- **Reliability = MTTF/ (MTBF) * 100%**

Example of Software Reliability

- A software program is said to fail on average once every twenty four 24 hours. Assuming that the program was implemented in a brand new machine @ 12:00pm on the 1st of January, calculate the reliability for each of the following situations. Show and explain all working where relevant:
    - After the first hour of implementation
    - After the fifth hour of implementation
    - After the twenty third hour of implementation
- **MTBF = MTTF + MTTR**
- **Reliability = MTTF/ (MTBF) * 100%**
- 1^{st} hr: =23/24 * 100% = 96%
- 5^{th} hr: =19/24 * 100% = 79%
- 23^{rd} hr: =1/24 * 100% = 4%

Software Design Methods

- In a Software development process, the Software Design Methodology (SDM) refers to specific **set of procedures used to manage and control the SDLC** (Software Development Life Cycle).

- The choice of the SDM primarily depends upon several factors, namely,
  - the ***type of the software*** (such as standalone or distributed and networked; Strategic or operational etc.)

  - the ***scope of the development project*** (such as **revamp of the existing system** or **new system**, the **number of modules** involved, underlying complexity of the coding, system testing and implementation etc),

  - the ***resources constraints*** (such as **time, money, expertise**)

Software Design Methods

- Systematic approaches to developing a software design.
    - **Structured Methods**
        - Process functions are identified
    - **Object-Oriented**
        - develop an object model of a system
    - **Data-Oriented**
        - Entities are determined for each sub-system, then entity inter-relationships are examined to develop the additional entities needed to support the relationships.
    - **Component-based**
        - Divide the system into components
    - **Formal Methods**
        - Requirements and programs are translated into mathematical notation

Which method to choose?

- **Data oriented design** is useful for **systems that process lots of data**, e.g. database and banking applications
- **Structured design** is useful for **process intensive systems** that will be programmed using a procedural language such as C. e.g. Crude **oil distillation systems**
- **OO methods** are useful for any system that will be programmed using an **object oriented language** such as C++. E.g. **mobile applications**
- **Component-based Methods** are used for the **large systems** that can be modularized. E.g. **ERP**
- **Formal methods** are considered to be an **alternative** to OO and classical design methods,
  - but their use is expensive and claims of reduced errors remain unproven.
  - However, the ability to formally **validate the correctness of a software** artifact is appealing and research on formal methods is ongoing.

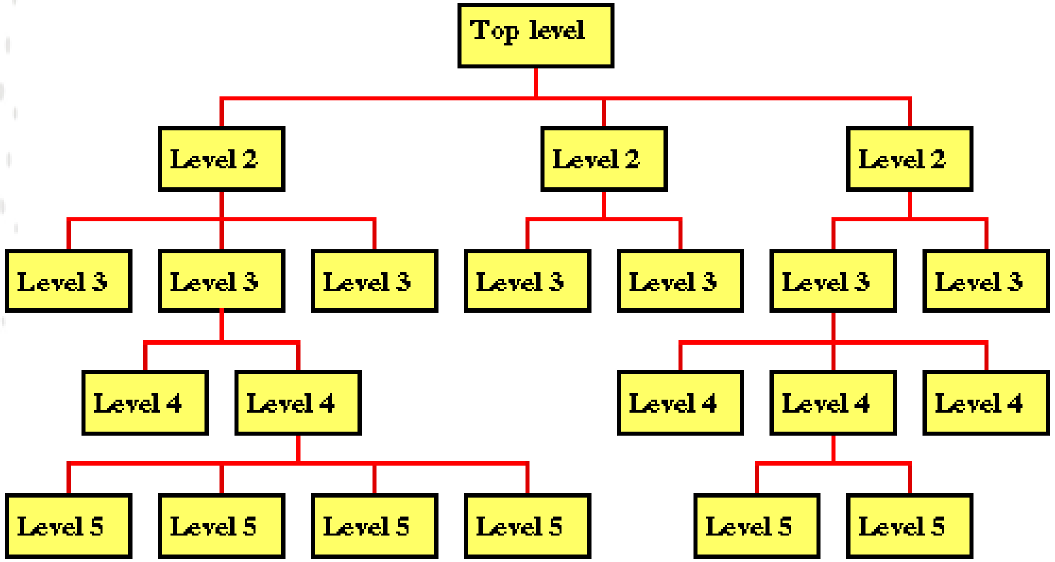
Software Design Paradigms

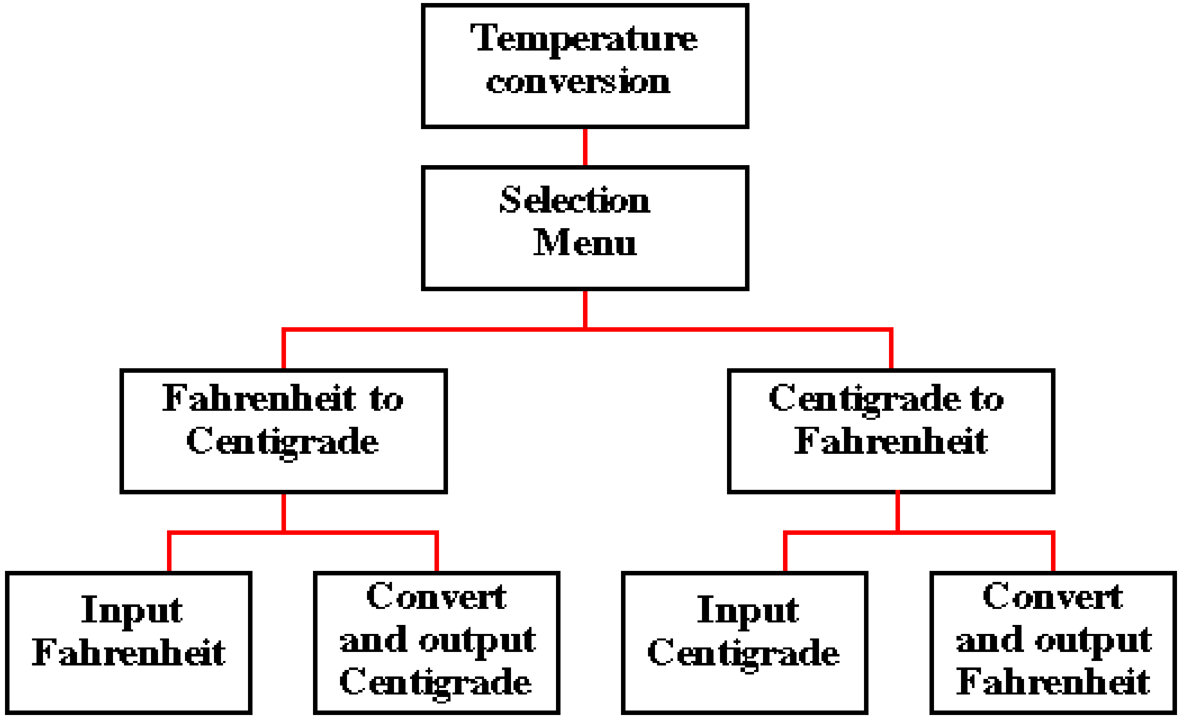
- Structured Design/Function Oriented Design
- Object-Oriented Design

COMPARISON FACTORS	FUNCTION ORIENTED DESIGN	OBJECT ORIENTED DESIGN
Abstraction	The basic abstractions are real world functions.	The basic abstractions are the data where the real-world entities are represented.
Function	Functions are grouped together by which a higher-level function is obtained.	Function are grouped together on the basis of the data they operate
Execute	Carried out using structured analysis and structured design i.e, data flow diagram	Carried out using UML
State information	In this approach the state information is often represented in a centralized shared memory.	In this approach the state information is represented in a distributed memory among the objects.
Approach	It is a top down approach.	It is a bottom up approach.
Begins basis	Begins by considering the use case diagrams and the scenarios.	Begins by identifying objects and classes.
Decompose	In function-oriented design we decompose in function/procedure level.	We decompose in class level.
Use	This approach is mainly used for computation sensitive application.	This approach is mainly used for evolving system which mimics a business or business case.

Structured/Procedural Paradigm

- **Focus on procedures and functions.**
- Design the system by **decomposing it based on the processes and functions.**
- **Top-down** algorithmic decomposition.
- This approach separates data from procedures.
- Drawback: Cannot be reused easily.



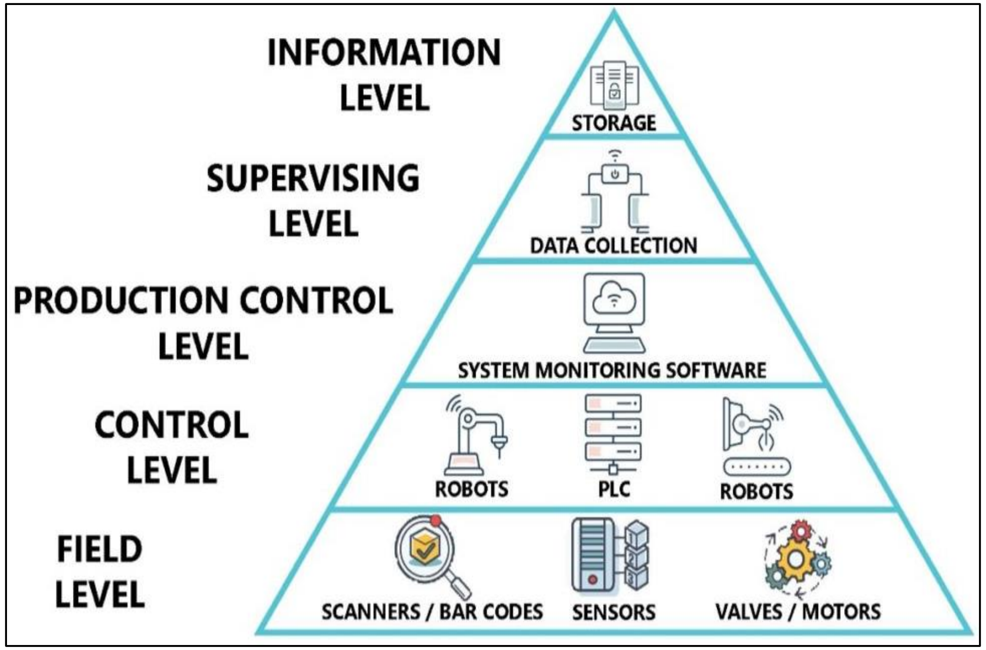


Object-Oriented Paradigm

- Describing the software solution in terms of **collaborating objects, with responsibilities.**
- **Bottom-up**
  - Encapsulate data and procedures in objects and classes.
  - Refinement (**movement from high level to low level design**) in classes lead to a composed larger system.

Benefits

- Enjoys all the benefits of Modular approach
- Dependencies can be handled **inheritance** and **polymorphism**.
- Naturalness because everything in real world is an object.
- Reusability by using the existing classes in future design.



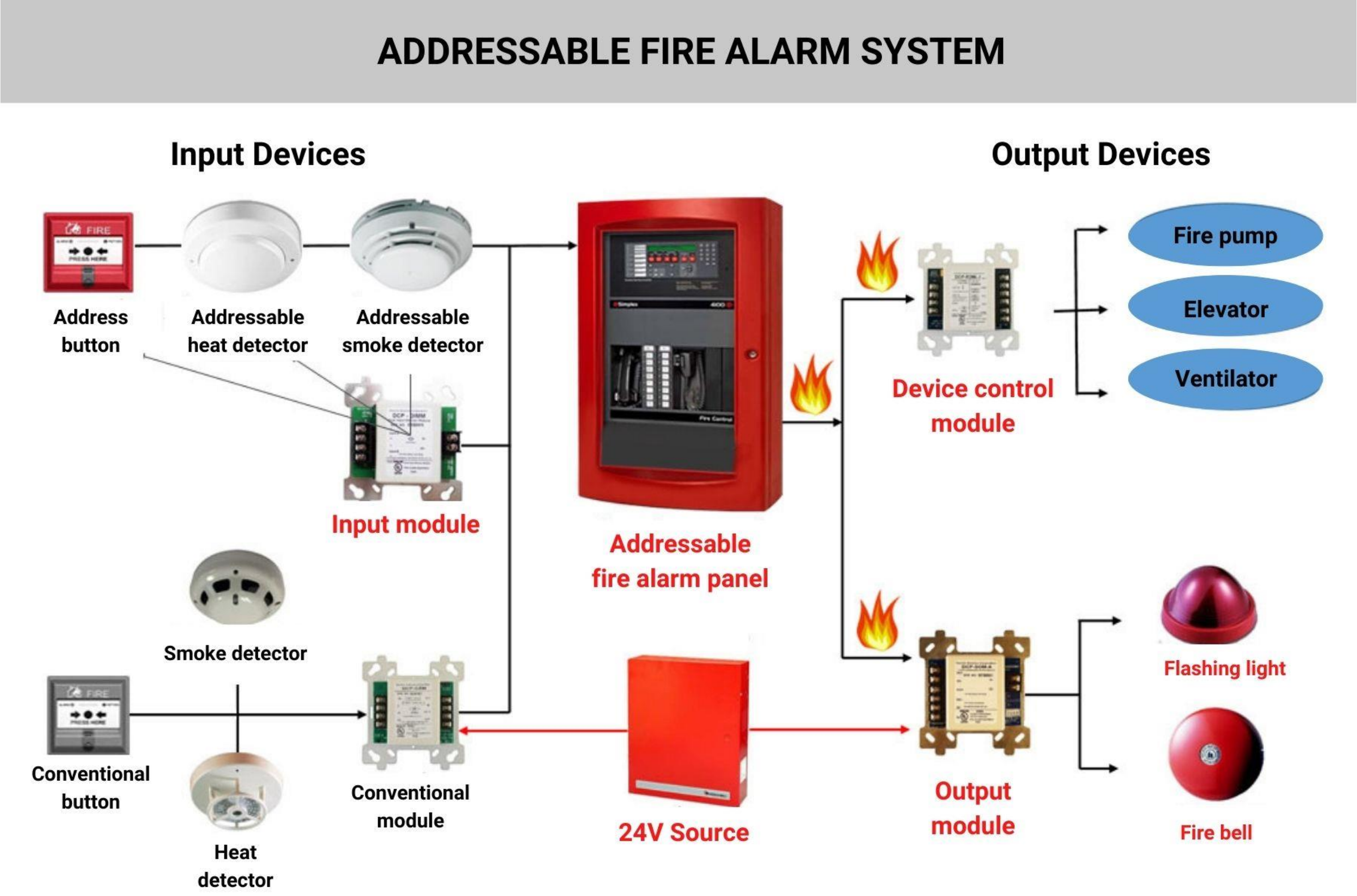
Case Study : Fire Alarm

- The owner of a large multi-stored building wants to have a computerized fire alarm system for his building.
- Smoke detectors and fire alarms would be placed in each room of the building.
- The fire alarm system would monitor the status of these smoke detectors.
- Whenever a fire condition is reported by any of the smoke detectors, the fire alarm system should determine the location at which the fire condition is reported by any of the smoke detectors
- The fire alarm system should determine the location at which the fire condition has occurred and then sound the alarms only in the neighbouring locations.
- The fire alarm system should also flash an alarm message on the computer console. Fire fighting personnel man the console round the clock.
- After a fire condition has been successfully handled, the fire alarm system should support resetting the alarms by the fire fighting personnel.

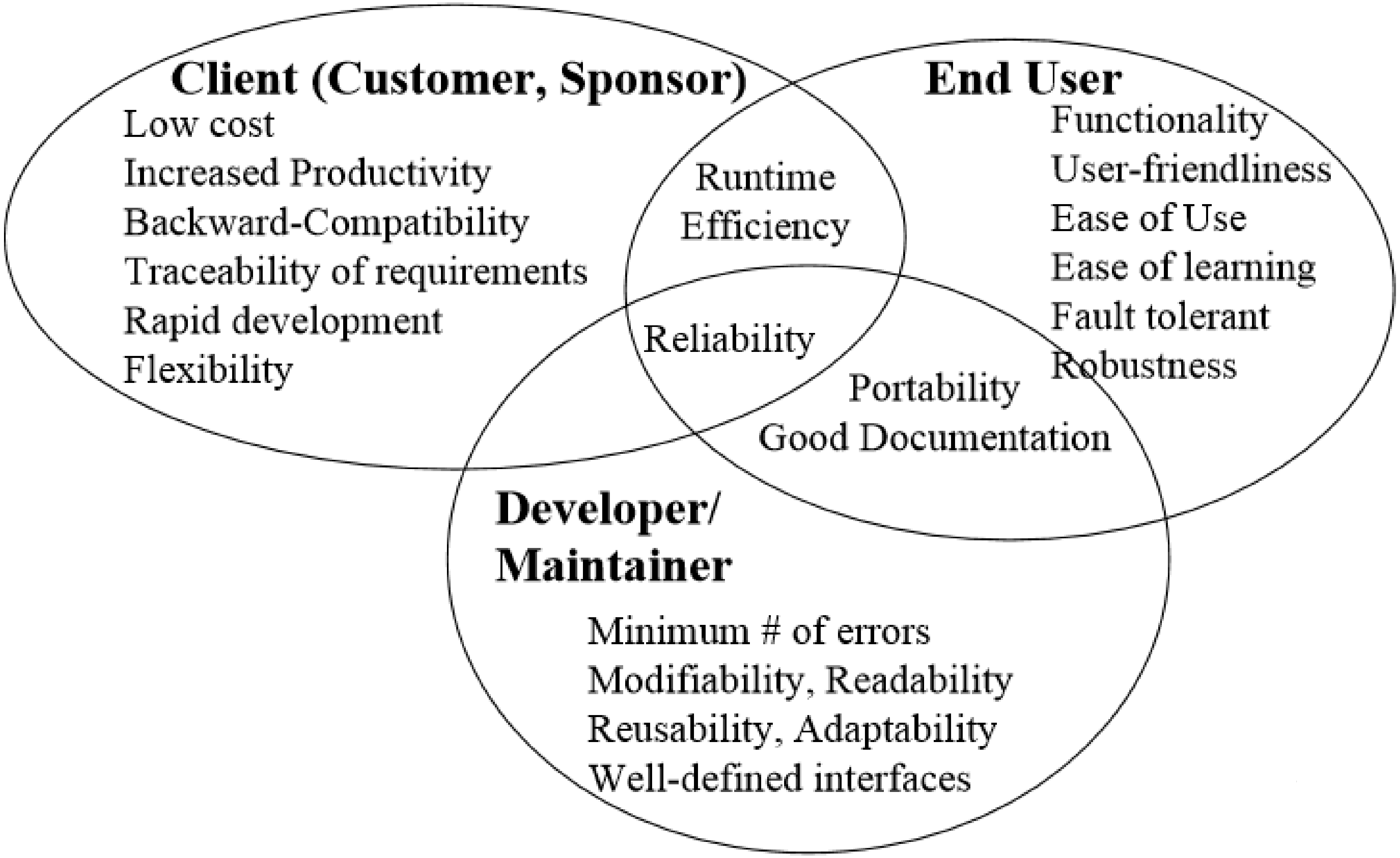
Function-Oriented Approach

The functions which operate on the system state are:

- interrogate_detectors();
- get_detector_location();
- report_fire_location();
- determine_neighbor();
- ring_alarm();
- reset_alarm();



Relationship Between Design Goals



Typical Design Trade-offs

The situation in which one attribute of a system or product is made less usable because another attribute has been given priority

- Functionality vs. Usability
- Cost vs. Robustness
- Efficiency vs. Portability
- Cost vs. Reusability

	Availability	Efficiency	Installability	Integrity	Interoperability	Modifiability	Performance	Portability	Reliability	Reusability	Robustness	Safety	Scalability	Security	Usability	Verifiability
Availability									+		+					
Efficiency	+				–	–	+	–			–		+		–	
Installability	+								+					+		
Integrity			–		–		–			–		+		+	–	–
Interoperability	+		–	–			–	+	+		+	–		–		
Modifiability	+		–				–		+	+			+			+
Performance		+			–	–		–			–		–		–	
Portability		–			+	–	–			+				–	–	+
Reliability	+	–		+		+	–				+	+		+	+	+
Reusability		–		–	+	+	–	+						–		+
Robustness	+	–	+	+	+		–		+			+	+	+	+	
Safety		–		+	+		–				+			+	–	–
Scalability	+	+		+			+	+	+		+					
Security	+			+	+		–	–	+		+	+			–	–
Usability		–	+				–	–	+		+	+				–
Verifiability	+		+	+		+			+	+	+	+		+	+	

Summary

- Good design
- Quality attributes of software design
- Software design principles
- A few concepts in design
- Software design methods
- Design Paradigms
- Typical Design Trade-offs

Q & A

Any Question?