CMPS411
Spring 2018

Lecture 10

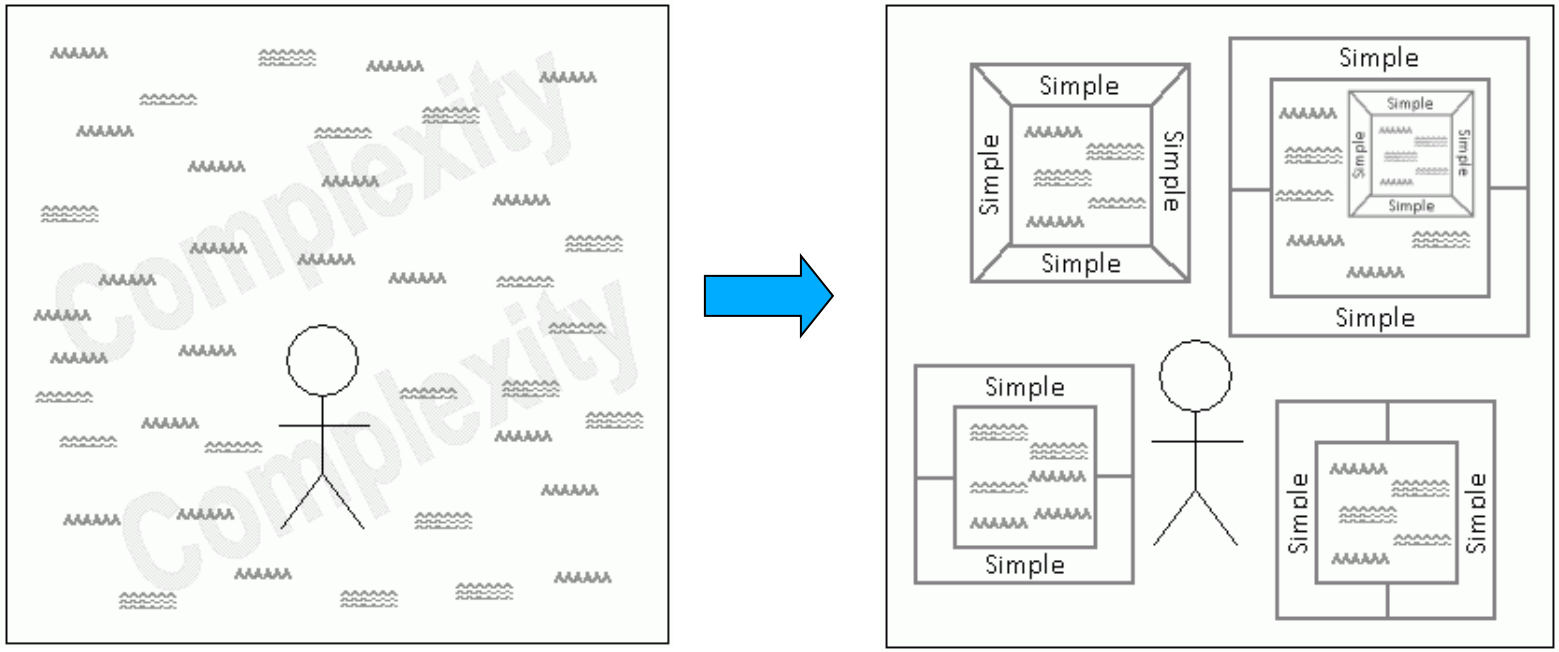
Software Quality

Fundamental Principles

- **Goal of software architecture** → satisfy the Non-Functional Requirements (NFR)/Quality
  - Architectural choices determines the non-Functional Requirements (NFRs) of a software
    - Such as the system's overall efficiency, reusability, security, reusability, maintainability, usability, portability, interoperability, etc.
- Basic fundamental principles guiding the design:
  - **Separation of concerns**: separation of components (computation) from connectors (communication)
  - **Abstraction**: hide the component's complexity behind simple interface
  - **Modularity**: divide the system into components
  - **High cohesion:** cohesive responsibility principle: the component's functions should be functionally related, and focused
  - **Low coupling:** reduce dependencies between components.

Design Techniques for Dealing with Software Complexity

- **Modularity** – subdivide the solution into smaller easier to manage components (Divide and Conquer).
- **Information Hiding** – hide details and complexity behind simple interfaces
- **Hierarchical Organization** – larger components may be composed of smaller components.



Making decisions

Choose architectural decisions based on their impact on NFRs:

- Choose a set of alternate design approaches
- Analyze the cost, quality, and feasibility of the alternatives
- Select the best solution that maximizes the achievement of NFRs/quality

Software Quality Attributes

Developer's view

- Maintainability
- Portability
- Reusability
- Testability

End user's view

- Performance
- Availability
- Usability
- Security

Business community view

- Time to market
- Cost and benefits
- Projected life time
- Targeted market
- Integration with legacy systems

Examples of Quality Attributes

- Often know as –ilities
  - Reliability
    - Fault tolerance, correctness
  - Availability
    - downtime/year
  - Scalability
    - Extendable
  - Performance (!)
    - Speed, Space, Interoperability
    - Can the software communicate with any type of other systems? Any restrictions?
  - Usability
    - How easy it is to use the system and to learn how to use it
  - Portability
    - Can an application be easily executed on a different software/hardware platform?
- Part of a system's NFRs
  - "how" well the system achieves its functional requirements
- Architects are often told: "My application must be fast/secure/scale"
  - Far too imprecise to be any use at all
- Quality attributes (NFRs) **must be made precise/measurable/testable** for a given system design

Examples - Quality Attribute Requirements

Quality Attribute	Architecture Requirement
Performance	Application performance must provide 15 seconds response time for 90% of requests.
Security	All communications must be authenticated and encrypted using certificates.
Resource Management	The server component must run on a low end server with 1GB memory. Idle servers must be utilized
Usability	The user interface component must run in an Internet browser to support remote users.
Availability	The system must run 24x7x365, with overall availability of 0.99.
Reliability	No message loss is allowed, and all message delivery outcomes must be known with 30 seconds Error can be recovered correctly
Scalability	The application must be able to handle a peak load of 500 concurrent users during the enrollment period.
Modifiability	• The architecture must support a phased migration from the current C++ version to a .NET solution. • More functions can be added later as customer increases

Introduction to Systems Constraints

- Almost all systems are limited (or constrained) in some way.
- Systems constraints are some factors that limit what the system can achieve.
- Some identifiable factor or component restricts the system's ability to perform.
- The limiting factor may be internal or external to the system.
- It may be a physical component, a condition, or an imposed policy of some kind.
- Whatever it is, however, it does frustrate efforts from within the system to achieve better performance.
- Constraints may affect your software quality and the development efforts, but are sometimes "out of your control".
- Constraints can be related to:
  - **Technology**
    - Use Prolog language, Unix O/S, O-O design
  - **Delivery schedule**
    - Deliver within 4 weeks
  - **Effort**
    - Budget is limited to QR. xxxxxx
    - Can recruit only 3 additional staff
  - **Quality/non-functional properties**
    - 40% components are reusable
    - The downtime software must not exceed 30 seconds average per day
    - The payment use case must complete processing in less than 5 seconds

Types of Constraints

- Some software/hardware requirements include constraints.
- Constraints impose restrictions on the software/hardware and are (almost always) non-negotiable.
- They limit the range of design choices an architect/ a software/hardware engineer can make.

Constraint	Examples
Business constraints	• The technology must run as a plug-in for MS BizTalk, as we want to sell this to Microsoft.
Development constraints	• The system must use structured systems development method • Intermediate design artifact must be approved before implementation
Schedule constraints	• The first version of this product must be delivered within one month.
Technical constraints	• The system must be modifiable if the customer size increases • The system components must support reusability • The core of the system must be protected from the direct use by the user • The system components should be highly cohesive and loosely coupled with other components • Use only Linux based C++
Resource constraints	• 2 programmers, one designer and one tester are allocated to this project. • No overtime should be paid • 2 dedicated servers are available. • Fast complier should be available.
Practical constraints	• Cost of the server must be less than QR 5000 • The salary of the programmer must be below QR 16000. • The privacy of the client must be respected

Examples: Typical SW/HW Technical Constraints

- Supply Voltage/Current/Power
  - 220VAC,1000W.
- Input/Output ranges
  - 0 to 1 mV, 12v DC
  - Integers, Characters
- Connectivity
  - USB connection to PC
  - Compiler/Interpreter, Web
- Usability
  - The software can be easily used without training
  - GUI is simple and easy to understand and use
- Accuracy
  - Minimum voltage change that can be detected is 1 mV
  - Estimated output is within ±#% of the actual.
- Data Storage
  - Maintains hourly log.
  - Requires minimum of 100 GB storage.
- Maintainability
  - The software can be easily modifiability
  - Module with a bug can be isolated
  - The software can be expanable.

Four Main Steps to Address Constraints in your Design

- **Identify constraint**
  - Only relevant constraints should be identified
- **Describe constraint**
  - Define the constraint in descriptive term.
  - Provide example if possible.
- **Find ways how the constraint can be addressed in your design**
  - A set of methods or steps that will be used to address the constraints
- **Technique used to test that the constraint is met.**
  - How to test or validate that the constraint is addressed

Example 1: Usability Constraints

- Usability is defined generically as the ease of use and learnability of a human-made object.
- In other words, how easy the system can be used by the target user.
- Usability constraints illustrate the suitability and simplicity of using the system by a particular type of users.
- A typical approach to meet a usability constraint is by getting a satisfactory feedback from the target user, for whom the system was originally designed.
- Examples of two "Usability constraints" and their descriptions:
  - **Example 1**: The user interface running on the application server must be easy to use by a normal physician, to simplify the browsing of the patient data through graphs, and charts that makes easy for the physician to identify adverse events resulting from the patient monitoring.
  - **Example 2**: The wireless sensors must be designed carefully to simplify the mounting of these sensors on the target patient.

Meeting Usability Constraints(Example 1)

How to meet the constraint in your design:

- Meeting these two constraints can be achieved by careful selection of equipment and design of the user interface on the application server.
- The careful selection of equipment includes buying of small form factor sensors, which are easy to mount on the patient, with possibly, straps/ belts to facilitate the mounting of these sensors on the target patient.
- The carful design of the user interface may include the chart/graph data representations, which simplifies the browsing of patient measurements by the physician.

How to verify that the constraints described n the last slide are met:

- The constraints can be verified using two joint approaches:
  - **Careful selection of equipment**: the careful selection of the sensors used so they meet the small form factor requirement with possibly belts/straps to make it easy for the patient to wear them conveniently.
  - **Testing using target user**: as part of system testing, we have a real elderly patient trying to wear the sensors continuously and getting satisfactory feedback on the ability to move around while the sensors are mounted.

Example 2: Speed Constraints

- **Description**
  - The robot must be fast enough to complete 2 laps in a 3 minutes period to be a winning robot, that is an average speed > 5.3 inches / sec
- **How to meet this constraint in your design:**
  - Meeting the above constraint can be achieved by the right selection of the robot components.
  - The robot must be able to maximize the opportunities for scoring by completing at least 2 laps.
  - The robot must achieve an average speed of 5.3 inches/second.
  - This figure assumes that the robot will take the worst-case route around the outside wall of the track twice, or 80' in 3 minutes.
  - A peak speed will be determined by strategy and number of implementation-specific factors including course-path, weight, motor, and wheel choice.

Testing Usability Constraint (Example 2)

- **How to test that the speed constraint is addressed**
  - The robot can be put through a series of tests that would evaluate each individual subsystem as well as the entire system as a whole.
  - These tests should be designed such that they would examine the correct functionality and accuracy of each respective subsystem.
  - The tests can be implemented by a differential drivetrain for robot propulsion.
  - The system requires two motors with wheels to drive the robot around the course.
  - Although the motors are a key component of the drive train, the wheel used, and particularly the size of the wheel, have a significant impact on performance.

Example 3: Accuracy Constraints

- **Description:**
  - Voltage must be measurable between 0 and 5 volts with a tolerance of ± 0.1 volts.
- **How to meet constraint in your design**
  - Meeting the above mentioned constraint since a typical Li-ion cell can only supply a maximum of 5 volts, being able to measure a voltage from zero volts to that maximum voltage is appropriate for the device.
  - Because Li-ion cells maintain a nearly flat voltage discharge rate for 80% of the capacity, the system must detect minimal voltage drops in order to preserve battery longevity.
  - A precision of ± 0.1 volts allows for minimal voltage drops to be detected.

Testing Accuracy Constraint (Example 3)

- **How to test that the speed constraint is addressed**
  - In testing the voltage sensing, the voltage levels of the individual cells were first measured as a reference.
  - The cells were then connected to the ISL9208 analog front end chip, and the PIC24 was connected to the analog front end.
  - Using the I2C interface, the voltages of the individual cells were measured and output to a terminal display.
  - The voltages sent from the analog front end and the initial cell voltage measurements are displayed in the table.

General Techniques to Deal Systems Constraints

- **Identify factors that influence the constraint**
  - Determine the factor that most limits the system's ability to perform.
  - This factor could be internal (a resource, knowledge or competence, financial condition, or policy).
  - It could be external (market demand, competitive environment issues, materials and suppliers, or government regulations and laws).
- **Decide how to meet the current constraint**
  - What action is required to address the constraints most efficiency and effective ways?
  - This action varies depending on what that limiting factor is.
- **Subordinate all other parts of the system to the exploitation of the current constraint.**
  - It requires all non-constraints components and aspects to subordinate (or sacrifice) their own efficiencies in the interest of maximizing the efficiency to compensate the constraints
- **Elevate the constraint.**
  - "Elevation" in this case means to increase capacity.
  - This may include purchasing more equipment, hiring more people, or expanding facilities.
- **Test the constraints**
  - How to test/validate that the constraints are met or not met

Performance

- Performance Metrics:
  - **Throughput** = Amount of work performed in unit time
    - Transactions per second
    - Messages per minute
    - e.g., System must process 1000 payment transactions per second
  - **Response time** = Measure of the latency an application exhibits in processing a request
    - Usually measured in (milli)seconds
    - e.g., Search catalog within 4 seconds for 95% of requests, and all within 10 seconds
  - **Deadline** that must be met: 'something must be completed before some specified time'
    - e.g., Payroll system must complete by 2am so that electronic transfers can be sent to bank

Scalability

- How well the application will work **when the size of the problem increases**.
- 2 common scalability issues in software systems:
  - Increased number of concurrent requests
  - Increased size of processed data
    - Constraint: Banner should be capable of handling a peak load of 1000 concurrent course registration requests from students during the 3rd week enrollment period
    - Solution: The Web server can be replicated on a three machines cluster to handle the increased request load.

Modifiability

- Modifiability measures how easy it **may** be to change an application to cater for new requirements.
  - Must estimate cost/effort
- Minimizing dependencies increases modifiability
  - Changes isolated to single components likely to be less expensive than those that cause ripple effects across the architecture.
  - Hide implementation behind well defined interfaces
- Modifiability Scenarios
  - Add a Mobile Interface to the Banking Application within 5 weeks with 3 developers.
  - If the speech recognition software vendor goes out of business then we should be able to replace this component within 4 weeks with 2 developers.

Availability

- Measured by the proportion of the required time the application is useable.
  - Key requirement for most IT applications
- Related to an application's reliability
  - Unreliable applications suffer poor availability
- Period of loss of availability determined by:
  - Time to detect failure
  - Time to correct failure
  - Time to restart application
- Example scenarios:
  - 100% available during business hours
  - No more than 2 hours scheduled downtime per week
  - 24x7x52 (100% availability)
- Key strategy for high availability is:
  - Eliminate single points of failure using Load-Balancing and failover

Quality Properties: More Examples

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K Bytes Number of RAM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Quality Trade Offs

- Conflicts between different non-functional requirements are common in complex systems
- Spacecraft system
  - To minimise weight, the number of chips in the system should be minimised
  - To minimise power consumption, lower power chips should be used
  - However, using low power chips may mean that more chips have to be used.

  Which is the most critical requirement?

Design as a 'Wicked' Problem

- Characteristics of software design are the 'wicked problems':
  - Software design may not be always analytical
  - Can have a number of acceptable alternative solutions (more of an art than science)
  - Providing design solutions for one aspect of a problem may put constraints on another aspect of the problem, or create a new problem
  - Conflicting design choices.

Conflicting Quality Properties

- Performance, Reliability, Security, Costs are relevant for a software architecture
- These properties may be conflicting
- Good performance may require more effort and costly
- Different stakeholders may have conflicting quality requirements
  - Performance is important for managers
  - Maintainability is relevant for developers
  - Costs and schedule matter for managers
- Architect and stakeholders must agree on certain quality properties that have to be achieved
- All quality properties are subject to design trade-off.

Design Trade-offs

- Quality attributes (NFRs) affect each other
  - Highly available application may trade-off lower performance for greater availability
  - High performance application may be tied to a given platform, and hence not be easily portable
  - Highly secure system may be difficult to integrate, or even use
  - Highly interoperable software may have weak security
- Architects must design solutions that make the right balance between competing non-functional properties (quality)
  - Not possible to fully satisfy all competing requirements
  - Understand competing requirements and trade-offs
  - Architect must decide which qualities are important for a given application
  - Must find the right balance between competing NFRs
  - Able to select right architecture and design choices
  - This is one of the difficult decisions the architect/software engineer to make
  - No silver bullet in software engineering…

How to Achieve Quality?

- Right choice of architecture can ensure some types of quality properties
  - Examples,
    - Layered architecture may support modularity, cohesion, modifiability, and portability
    - Pipe and filter ensures cohesion, portability, maintainability
- Design patterns are often used to achieve quality of the software
- An architectural style/pattern expresses a fundamental structural organization schema for software systems
  - Predefined subsystems and their responsibilities, rules and guidelines for organizing the subsystems
  - Relationships between subsystems.

Design Pattern

- A solution to a narrowly-scoped business/technical problem
  - ·A fragment of a solution, a partial solution, or a piece of the puzzle
  - ·A solution to a common design problem
  - Describes a common design problem
    Describes a proven solution to the problem A solution based on experience
  - ·Design patterns discuss the result and trade-offs of applying the pattern
  - ·Design patterns provide the capability to reuse successful designs
- A design pattern provides a scheme for refining the subsystems or components of a solution, or
- The relationships between them

Pattern Classification Categories

Scope	Purpose		
	Creational *(object creation)*	***Structural*** *(Composition of classes or objects)*	***Behavioral*** *(Object interaction, distribution of responsibility)*
Class **(relations between classes and subclasses; Static)**	• Factory	• Adapter	• Interpreter • Template method
Object **(Object relations, dynamic –run time)**	• Factory • Abstract Factory • Singleton • Prototype • Builder	• Proxy • Adapter • Façade • Decorator • Bridge • Composite	• Strategy • Observer • Chain of Responsibility • Command • Iterator • Mediator • Memento • Flyweight • State • Visitor

How Patterns Achieve Quality

- Helping in finding correct objects
- Determining object granularity
- Specifying object interfaces
- Specifying object implementation
  - Specifying an implementation (a class) vs. specifying an interface (a type)
  - Implementation inheritance vs. interface inheritance
  - Client programming to an interface, not an implementation
- Putting reuse mechanisms to work
  - Inheritance vs. composition
  - Delegation
  - Inheritance vs. parameterized types
- Relating compile-time and run-time structures
  - Class definition vs. collaborating objects
- Designing for change.

References

- Patrik Berander, et al., “Software Quality Attributes and Trade-Offs.” https://www.uio.no/studier/emner/matnat/ifi/INF5180/v10/undervisningsmateriale/reading-materials/p10/Software_quality_attributes.pdf
- Marlo Barbacci, et al. “Quality Attributes.” http://www.sei.cmu.edu/reports/95tr021.pdf