Class #6

03. Designing Software architectures

Software Architectures

Master in Informatics Engineering

Cláudio Teixeira (claudio@ua.pt)

Agenda

- Software Quality Attributes

From previous class - discussion on results

DDD in practice - 2 define building blocks for system

https://docs.google.com/document/d/1hCNaT-oo67XSY93EPy6T1ShH9EU_biZyKUFzwSTCEqA/edit?usp=sharing

Software Quality Attributes

Software Quality Attributes

- Software Quality Attributes are the non-functional requirements that describe a system's behavior and characteristics.
  - Unlike functional requirements that outline what a system should do, quality attributes detail how a system performs its functions under various conditions.
- A Quality Attribute (QA) is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders beyond the basic function of the system.
  - You can think of a quality attribute as measuring the “utility” of a product along some dimension of interest to a stakeholder.
- Integrating quality attributes into the requirements phase ensures that the software is designed with these critical factors in mind.
- **Keep in mind the SDLC!**

Specifying Quality Attributes requirements: Scenarios

Scenarios are detailed narratives that describe how a system responds under specific conditions, focusing on quality attributes.

They help in identifying potential challenges and evaluating the system's behavior in real-world situations.

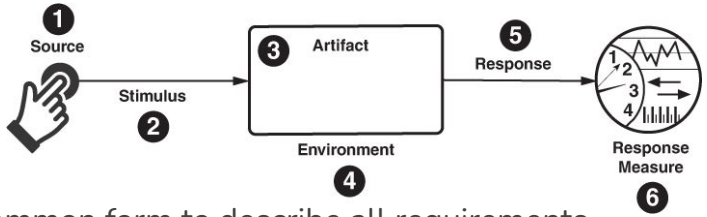
A quality attribute scenario is a short description of how the software system should respond to a particular stimulus.

Scenarios make quality attributes measurable and testable.

Examples

A performance scenario might detail how the system handles a high number of simultaneous requests;

A security scenario could describe the system's response to a breach attempt.



- Common form to describe all requirements
- Quality attribute scenarios have six parts:
  - Specification:
    - Stimulus source (1)
    - Stimulus (2)
    - Response (5)
    - Response measure (6)
  - Environment (4)
  - Artifact (3)

Not Examples

A quality attribute of performance or a requirement that states a particular function should be fast is not measurable or testable

Quality attribute scenarios

Prioritization

- How to settle on which scenario to address first?
- Rank scenarios:
  - Based on two criteria: business importance and technical risk.
  - Set a ranking scale for criteria (eg.: High (H), Medium (M), Low (L))
  - Stakeholders provide ranking for business importance
  - Software architect provides ranking for technical risk

Business importance/technical risk	L	M	H
L	6, 21	7, 13	15
M	3, 10, 11	14, 16, 17	1, 5
H	4, 18, 19, 20	2, 12	8, 9

Placement of hypothetical Quality Attribute Scenarios with in matrix.

Quality attribute scenarios

Stimulus

Stimulus: describes an event arriving at the system or the project.

It can be an event to the performance community, a user operation to the usability community, or an attack to the security community,

A stimulus for modifiability is a request for a modification; a stimulus for testability is the completion of a unit of development.

Quality attribute scenarios

Stimulus source

Stimulus source: the origin of the stimulus.

Some entity (a human, a computer system, or any other actor) must have generated the stimulus.

Quality attribute scenarios

Response

The response is the activity that occurs as the result of the arrival of the stimulus.

It consists of the responsibilities that the system (for runtime qualities) or the developers (for development-time qualities) should perform in response to the stimulus.

For example, in a performance scenario, an event arrives (the stimulus) and the system should process that event and generate a response.

In a modifiability scenario, a request for a modification arrives (the stimulus) and the developers should implement the modification—without side effects—and then test and deploy the modification.

Quality attribute scenarios

Response measure

When the response occurs, it should be measurable in some fashion so that the scenario can be tested — that is, so that we can determine if the architect achieved it.

Example:

- for performance, this could be a measure of latency or throughput;
- for modifiability, it could be the labor or wall clock time required to make, test, and deploy the modification.

Quality attribute scenarios

Environment

Environment is the set of circumstances in which the scenario takes place (the running state).

The system may be in an overload condition or in normal operation, or some other relevant state.

The environment can also refer to states in which the system is not running at all: when it is in development, or testing, or refreshing its data, or charging its battery between runs.

The environment sets the context for the rest of the scenario:

- A request for a modification that arrives after the code has been frozen for a release may be treated differently than one that arrives before the freeze.
- The fifth successive failure of a component may be treated differently than the first failure of that component.

Quality attribute scenarios

Artifact

The stimulus arrives at some target.

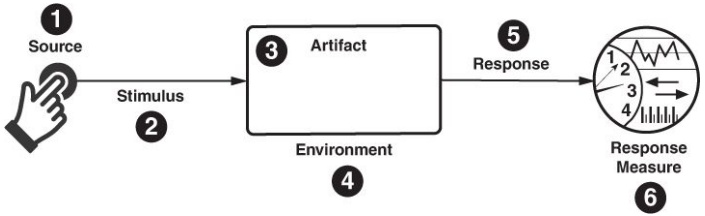
This is often captured as just the system or project itself, but it’s helpful to be more precise if possible.

The artifact may be a collection of systems, the whole system, or one or more pieces of the system.

A failure or a change request may affect just a small portion of the system.

A failure in a data store may be treated differently than a failure in the metadata store. Modifications to the user interface may have faster response times than modifications to the middleware.

Specifying Quality Attributes requirements: Scenarios



- Common form to describe all requirements
- Quality attribute scenarios have six parts:
  - Specification:
    - Stimulus source (1)
    - Stimulus (2)
    - Response (5)
    - Response measure (6)
  - Environment (4)
  - Artifact (3)

A general scenario for availability

Source	Stimulus	Artifact	Environment	Response	Response Measure
Internal/external: people, hardware, software, physical infrastructure, physical environment	Fault: omission, crash, incorrect timing, incorrect response	Processors, communication channels, storage, processes, affected artifacts in the system's environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation	Prevent the fault from becoming a failure Detect the fault Recover from the fault	Time or time interval when the system must be available Availability percentage Time to detect the fault

Achieving Quality Attributes:
Architectural Patterns vs Tactics

Architectural Patterns

Scope:

High-level, structural solutions for recurring design problems (e.g., Layered Architecture).

Focus:

Overall system organization, component roles, and interaction styles.

Example: Microservices improves scalability and maintainability by decomposing a system into autonomous services.

Tactics

Scope:

Targeted design decisions that directly influence specific quality attributes (e.g., performance, security).

Focus:

Fine-tuning or optimizing within any chosen architectural pattern.

Example: Caching to reduce response time, Encryption to protect sensitive data.

Patterns change the **macro-level structure** of an application

Tactics make **micro-level adjustments** to achieve or improve specific non-functional requirements

Performance

Performance is about time and the software system's ability to meet timing requirements.

It refers to how well a system responds to the demands of users in terms of speed, efficiency, concurrency, and resource utilization under specified conditions.

```
x = 1;
x++;
```

2 threads execute simultaneously. What will be the value of x? 2? 3?

Performance

Measuring & Improving

Measuring

Response Time: The time taken for the system to respond to a user's action.

Throughput: The number of transactions or operations a system can handle per unit of time.

Resource Utilization: The extent to which system resources (CPU, memory, disk I/O) are used under workload.

Scalability: The system's ability to maintain or improve performance as the workload increases.

Improving

Algorithm Optimization: Replacing inefficient algorithms with more efficient ones.

Resource Optimization: Reducing the memory footprint and optimizing resource allocation.

Concurrency, Parallelism and Replication: Leveraging multi-threading and distributed computing to process tasks in parallel.

Caching: Storing frequently accessed data in fast-access storage to reduce data retrieval times.

Performance
Patterns & Tactics

Load Balancing: Distributing workloads across multiple servers to optimize resource use and reduce response times.
Data Partitioning: Dividing data into smaller chunks that can be processed in parallel to improve scalability and throughput.
Asynchronous Processing: Performing long-running tasks in the background to keep the user interface responsive.
Control Resource Demand: Manage Work Requests - prevent overloading; **Limit Event Response** - queuing requests or discard excess.
Manage Resources: Increase Resources - horizontal or vertical scale**; Introduce Concurrency**; **Schedule resources** accordingly.

Building a Scalable Social Network Home Timeline

Group assignment
Part 1: 20 min
Part 2: 30 min
Part 3: 45 min

https://docs.google.com/document/d/1JD6gmk8oWPKhO5AsX-0HMm6fvmAxmLBmydPMqw8ac2c/edit?tab=t.0

More on Software Quality Attributes

(for your reference)

Availability

Measures the system's ability to remain accessible and functional when needed by its users

Availability (%) = (Total Operational Time - Downtime) / Total Operational Time * 100

Improvement Strategies: Implement redundancy, using failover systems, load balancing, etc., performing regular maintenance, and deploying robust security measures to prevent unauthorized access and attacks.

Patterns: Active redundancy (hot spare), Passive redundancy (warm spare), Spare (cold spare), Triple modular redundancy (TMR), ...

Calculations: ***MTBF/(MTBF + MTTR)***; where MTBF refers to the mean time between failures and MTTR refers to the mean time to repair.

Deployability

... why is this a software quality attribute?! Deployability refers to the ease with which a software system can be deployed, updated, or migrated to different environments.

Key metrics for quantifying deployability include: deployment frequency, success rate, average deployment time, and rollback rate.

A system with high deployability allows for frequent, reliable, and efficient updates with minimal downtime. Automated deployability (continuous deployment) is the goal.

Architectural choices affect deployability. For example, by employing the microservice architecture pattern, each team responsible for a microservice can make its own technology choices. (loose coupling, all over again)

Deployability

Architects are primarily concerned with the degree to which the architecture supports deployments that are:

- **Granular**. Deployments can be of the whole system or of elements within a system. If the architecture provides options for finer granularity of deployment, then certain risks can be reduced.
- **Controllable**. The architecture should provide the capability to deploy at varying levels of granularity, monitor the operation of the deployed units, and roll back unsuccessful deployments.
- **Efficient**. The architecture should support rapid deployment (and, if needed, rollback) with a reasonable level of effort.

Deployability

Deployability tactics

- Manage deployment pipelines
  - how to rollout?
    - to 1 server ?
    - to 1000 servers?
  - script and automate!
    - document, review, test and version control scripts
  - how to rollback?
- Manage deployed system
  - Service interactions
  - Dependencies
  - Toggle features
    - Canary and A/B test

Energy Efficiency

The estimated energy consumption of a Google search query is 0.0003 kWh (1.08 kJ). The estimated energy consumption of a ChatGPT-4 query is 0.001-0.01 kWh (3.6-36 kJ), depending on the model size and number of tokens processed. (unofficial “guestimates” 22/10/2023 -).

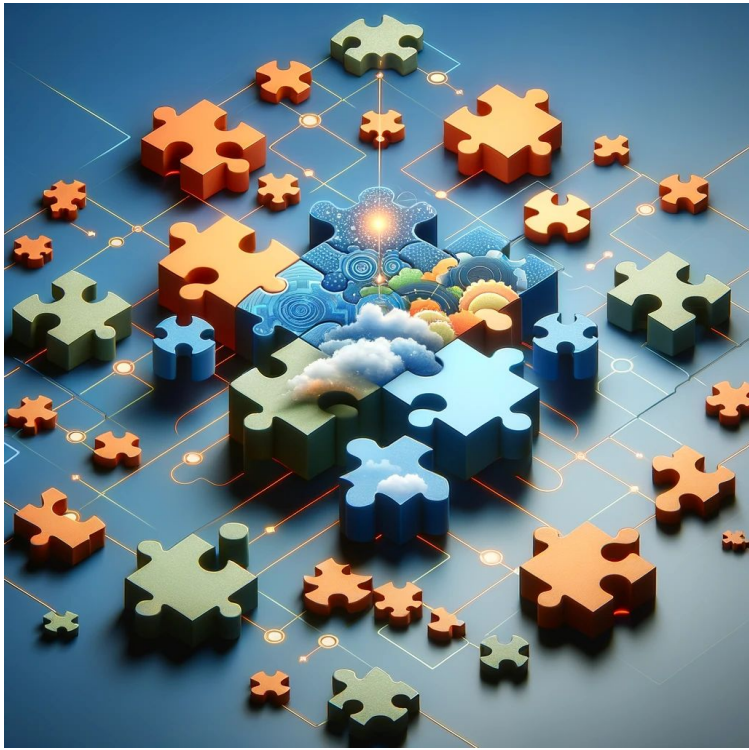
- Energy efficiency is even more important in the context of IoT or battery operated devices, where every watt matters
- Being energy efficient may mean also less running costs!
- How:
  - **Monitor resources:** You can’t manage what you can’t measure!
  - Handle resource allocation wisely: remove or deactivate unused resources whenever possible.
  - **Manage resource demand:** control event arrival, limit max response ratio, reduce computational overhead

Integrability

A software quality attribute that measures the ease with which a software component can be combined or added to other components and systems.

It focuses on the capability of the software to integrate seamlessly with existing and future components within its ecosystem.

This attribute is crucial for ensuring that software systems can evolve and adapt over time by incorporating new features, technologies, and services without significant rework or disruption.



Integrability

Integration difficulty—the costs and the technical risks—can be thought of as a function of the size of and the “distance” between the interfaces of {Ci} and S:

- Size is the number of potential dependencies between {Ci} and S.
- Distance is the difficulty of resolving differences at each of the dependencies
- **Syntactic distance:** integer argument in {Cix}, but floating point input in {Ciz}
- **Data semantic distance**: altitude in meters vs altitude in feet
- **Behavioral semantic distance**: {Cix} on startup interprets value differently from {Ciz}
- **Temporal distance**: cooperating elements must agree on assumptions about ***order*** of events
- **Resource distance**: agreement about resource sharing (10Gb ram for {Cix}, and 8 Gb ram for {Ciz}, but only 14 Gb in total available)

Identity these and you get a sense of magnitude of the integrability.

Integrability

Improvement Strategies

Standardized Interfaces: Adopting standardized interfaces and protocols ensures that components can communicate and work together without custom adaptation.

API Documentation and Versioning: Providing comprehensive and clear API documentation, along with careful API versioning, helps maintain integrability across system updates.

Loose Coupling: Designing software components with loose coupling enhances integrability by minimizing dependencies between components, allowing them to be updated or replaced with less impact on the overall system.

Adherence to Architectural Patterns: Utilizing architectural patterns such as microservices or service-oriented architecture (SOA) can facilitate integrability by organizing the system into well-defined, independently deployable components.

Integrability

Patterns and Tactics

Limit Dependencies - encapsulate, use intermediary components, use standards, restrict communication paths, abstract common services.

Facade Pattern: Implementing a facade pattern provides a unified interface to a set of interfaces in a subsystem, simplifying the integration process for external components.

Adapter Pattern: The adapter pattern can be used to convert the interface of a class into another interface clients expect, enabling incompatible classes to work together.

API Gateways: Using API gateways as a single entry point for managing API calls between clients and services enhances integrability by centralizing request routing, composition, and protocol translation.

Integrability

Patterns and Tactics

Limit Dependencies - encapsulate, use intermediary components, use standards, restrict communication paths, abstract common services.

Facade Pattern: Implementing a facade pattern provides a unified interface to a set of interfaces in a subsystem, simplifying the integration process for external components.

Adapter Pattern: The adapter pattern can be used to convert the interface of a class into another interface clients expect, enabling incompatible classes to work together.

API Gateways: Using API gateways as a single entry point for managing API calls between clients and services enhances integrability by centralizing request routing, composition, and protocol translation.

```
class DirectIntegration
{
    public void ProcessData()
    {
        var serviceA = new ServiceA();
        serviceA.ProcessA();
        var serviceB = new ServiceB();
        serviceB.ProcessB();
    }
}
```

```
// Facade providing a simplified interface
class IntegrationFacade
{
    private ServiceA serviceA = new ServiceA();
    private ServiceB serviceB = new ServiceB();

    public void ProcessData()
    {
        serviceA.ProcessA();
        serviceB.ProcessB();
    }
}

class EnhancedIntegration
{
    public void ProcessData()
    {
        var facade = new IntegrationFacade();
        facade.ProcessData();
    }
}
```

Modifiability

Most of the cost of the typical software system occurs after it has been initially released.

Add new features, alter or even retire old ones. Fix defects, tighten security, or improve performance. Enhance the user's experience. New technology, new platforms, new protocols, new standards. Make systems work together, even if they were never designed to do so.

Plan for: What can change? What is the likelihood of the change? When is the change made and who makes it? What is the cost of the change?

Consider types of costs: The cost of introducing the mechanism(s) to make the system more modifiable; The cost of making the modification using the mechanism(s).

Modifiability

Metrics for quantifying

Change Impact Analysis: Measuring the extent of modifications required for a particular change, often quantified by the number of components affected.

Time to Implement Changes: The average time it takes to implement and deploy changes, including development, testing, and deployment efforts.

Cyclomatic Complexity: A metric that measures the complexity of the software's structure, influencing how easily it can be modified.

Modifiability

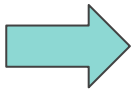
Tactics & Patterns

Increase Cohesion: split modules and redistribute responsibilities.

Reduce coupling

Modularise: client-server architectures, plug-in modules, layers patterns, pub-sub patterns

```
class DataProcessor
{
    public void ProcessData(string data)
    {
        // Hard-coded data processing logic
        Console.WriteLine($"Processing data:
{data} with default algorithm.");
    }
}
```



```
interface IProcessingStrategy{
    void Process(string data);
}

class DefaultProcessingStrategy : IProcessingStrategy{
    public void Process(string data)
    {
        Console.WriteLine($"Processing data: {data} with
default algorithm.");
    }
}

class CustomProcessingStrategy : IProcessingStrategy{
    public void Process(string data)
    {
        Console.WriteLine($"Processing data: {data} with custom
algorithm.");
    }}

class DataProcessor{
    private IProcessingStrategy _strategy;

    public DataProcessor(IProcessingStrategy strategy)
    {
        _strategy = strategy;
    }

    public void ProcessData(string data)
    {
        _strategy.Process(data);
    }
}
```

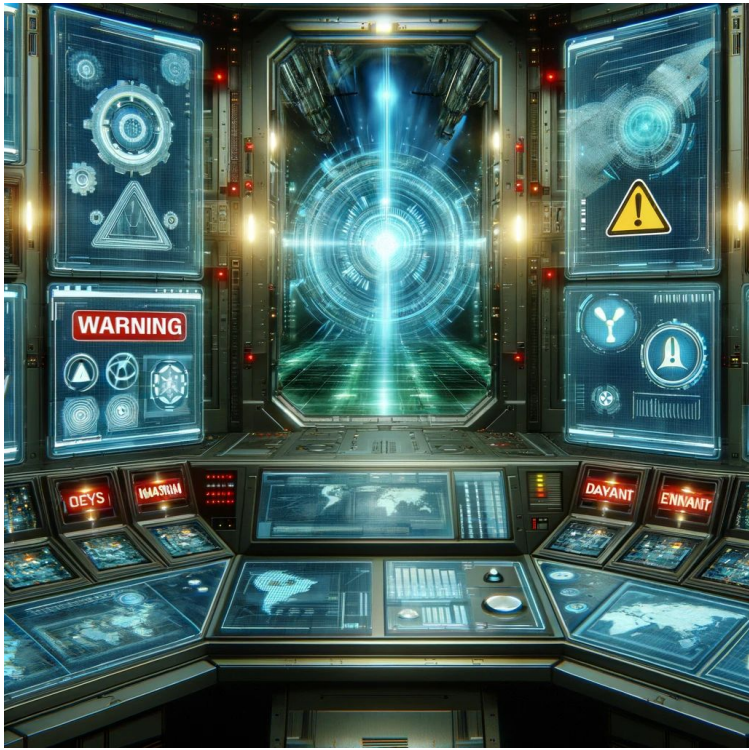
Safety

Safety is concerned with a system’s ability to avoid straying into states that cause or lead to damage, injury, or loss of life to actors in its environment.

Safety is also concerned with detecting and recovering from these unsafe states to prevent or at least minimize resulting harm.

It refers to the attribute that ensures the software operates without causing unacceptable risks of harm to people, environments, or processes.

It's particularly critical in systems where failure or malfunction could result in catastrophic outcomes, such as in medical devices, automotive systems, aerospace controls, and industrial automation systems.



Safety

Unsafe states can be caused by a variety of factors:

Omissions: the failure of an event to occur

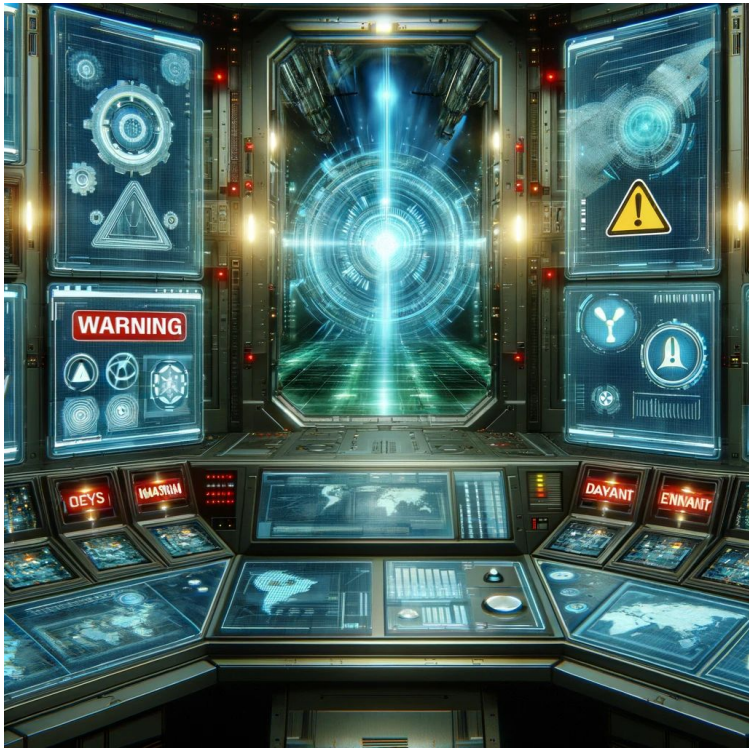
Commission: the spurious occurrence of an undesirable event. The event could be acceptable in some system states but undesirable in others.

Timing: early (the occurrence of an event before the time required) or late (the occurrence of an event after the time required) timing can both be potentially problematic.

Problems with system values: These come in two categories: Coarse incorrect values are incorrect but detectable, whereas subtle incorrect values are typically undetectable.

Sequence omission and commission: In a sequence of events, either an event is missing (omission) or an unexpected event is inserted (commission).

Out of sequence: a sequence of events arrive, but not in the prescribed order.



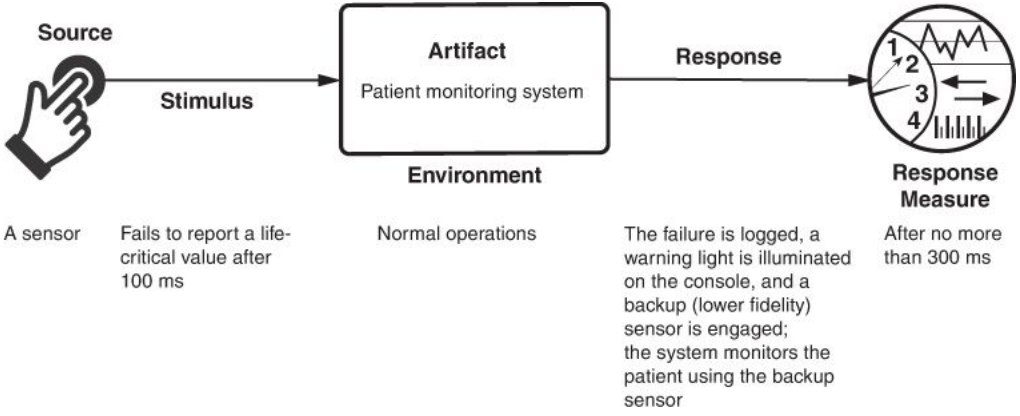
Architecting for safety

Identify the system's safety-critical functions: functions that could cause harm.

Use techniques such as failure mode and effects analysis (FMEA; also called hazard analysis) and fault tree analysis (FTA).

FTA is a top-down deductive approach to identify failures that could result in moving the system into an unsafe state.

Once the failures have been identified, the architect needs to design mechanisms to detect and mitigate the fault (and ultimately the hazard).



Safety
Tactics & Patterns

Unsafe State Avoidance: Substitution - Hardware dedicated devices instead of software versions; predictive analysis.

Unsafe State Detection: Timeout - detect late timings; **Timestamp** - detect incorrect sequences; **Condition Monitoring** - validate and assert data for sanity check and predictive models; **Sanity Checking** - checks the validity of date; **Comparison** - comparing the outputs produced with replicas.

Containment: **Redundancy** - enable operation to continue - Replication, functional redundancy (diversity), Analytic Redundancy (different implementations), **Limit consequences** - Abort (stop), Degradation (drops non essentially functionality), masking (replaces data from voting procedure in redundant systems); **Barrier** - Firewall; Interlock (ensuring sequence).

Recovery: Rollback - revert to a known good state**; Repair** - repairs from erroneous state**; Reconfigure** - remap the logic architecture onto the resources still functioning.

Patterns: **Redundant sensors; Monitor-actuator; Separated safety** (usually certified) - equipment certification, In avionics, the distinction is finer-grained, ...

Security

Security is a measure of the system’s ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized.

It refers to the protection of information and system resources from unauthorized access, use, disclosure, alteration, or destruction. It is a quality attribute that ensures the confidentiality, integrity, and availability of data and services, vital for maintaining trust and compliance in digital environments.

Security

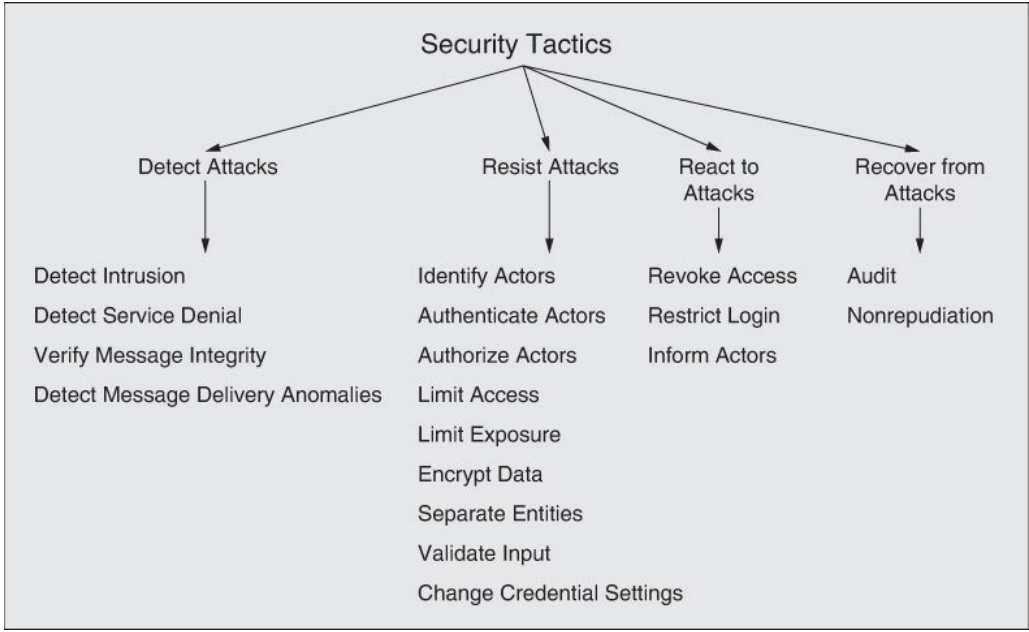
Focuses on three characteristics: confidentiality, integrity, and availability (CIA):

- **Confidentiality** is the property that data or services are protected from unauthorized access.
- **Integrity** is the property that data or services are not subject to unauthorized manipulation.
- **Availability** is the property that the system will be available for legitimate use.

... and ... Privacy!

- **Privacy** is about limiting access to information, which in turn is about which information should be access-limited and to whom access should be allowed.

Security Tactics & Patterns

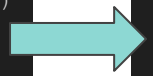


Implementing Encryption: Using strong encryption algorithms for data at rest and in transit to protect against eavesdropping and tampering.
Access Control: Enforcing strict access control policies and mechanisms to ensure that only authorized users can access or modify sensitive information.
Regular Security Audits and Updates: Conducting periodic security audits to identify and remediate vulnerabilities, and applying security patches and updates promptly.
Authentication and Authorization Frameworks: Implementing robust authentication mechanisms to verify user identities and authorization frameworks to control access to resources based on user roles and permissions.
Security Gateways: Deploying security gateways or firewalls to monitor and control incoming and outgoing network traffic based on predetermined security rules.
Intrusion Detection and Prevention Systems (IDPS): Utilizing IDPS to detect and prevent unauthorized access or attacks by monitoring network traffic and system activities for suspicious behavior.

Security

Access Control Demo

```
public class FileStorage
{
    public void SaveFile(string filePath, byte[] data)
    {
        // Save the file without checking user
permissions
        Console.WriteLine($"Saving file to
{filePath}.");
    }
}
```



```
public class FileStorage
{
    public bool CheckUserPermission(string userId, string filePath)
    {
        // Check if the user has permission to save the file
        // Simplified permission check logic
        return userId == "authorizedUser";
    }

    public void SaveFile(string userId, string filePath, byte[] data)
    {
        if (CheckUserPermission(userId, filePath))
        {
            Console.WriteLine($"User {userId} saving file to
{filePath}.");
        }
        else
        {
            Console.WriteLine($"Access denied for user {userId}.");
        }
    }
}
```

Testability

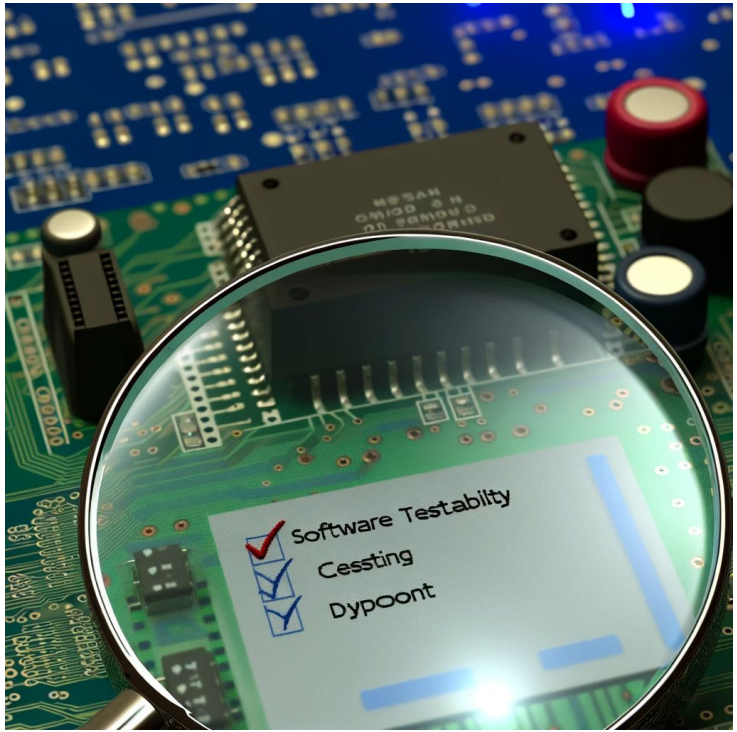
Imagine getting a flat tire. Even if you have a spare tire in your trunk, do you know if it is inflated? Do you have the tools to change it? And, most importantly, do you remember how to do it right? One way to make sure you can deal with a flat tire on the freeway, in the rain, in the middle of the night is to poke a hole in your tire once a week in your driveway on a Sunday afternoon and go through the drill of replacing it.

Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing.

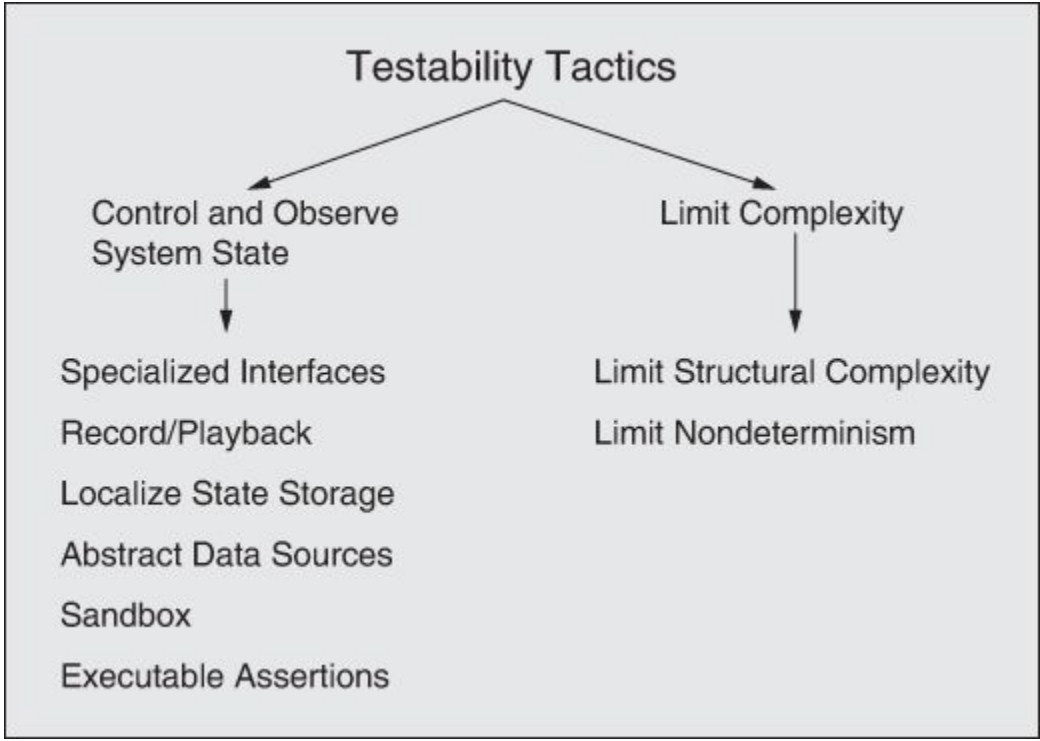
Specifically, testability refers to the probability, assuming that the software has at least one fault, that it will fail on its next test execution.

Intuitively, a system is testable if it “reveals” its faults easily. If a fault is present in a system, then we want it to fail during testing as quickly as possible.

For a system to be properly testable, it must be possible to control each component’s inputs (and possibly manipulate its internal state) and then to observe its outputs (and possibly its internal state, either after or on the way to computing the outputs)



Testability Patterns & tactics



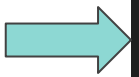
Observer Pattern: Allows for the decoupling of system components, making individual parts easier to test.

Builder Pattern: Facilitates the construction of complex objects that can be used in testing scenarios.

Test Double (e.g., Mocks, Stubs, and Fakes): Using these objects in place of real dependencies can isolate the system under test and simplify test setup.

Testability

```
public class ProductService
{
   private Database _database = new Database(); // Hard-coded dependency
   public Product FindProduct(int id)
   {
       // Lookup product in the database
       return _database.FindProductById(id);
   }
}
```



```
public class ProductService
{
   private IDatabase _database; // Dependency injected

   public ProductService(IDatabase database)
   {
       _database = database;
   }

   public Product FindProduct(int id)
   {
       // Lookup product using the injected database interface
       return _database.FindProductById(id);
   }
}
```

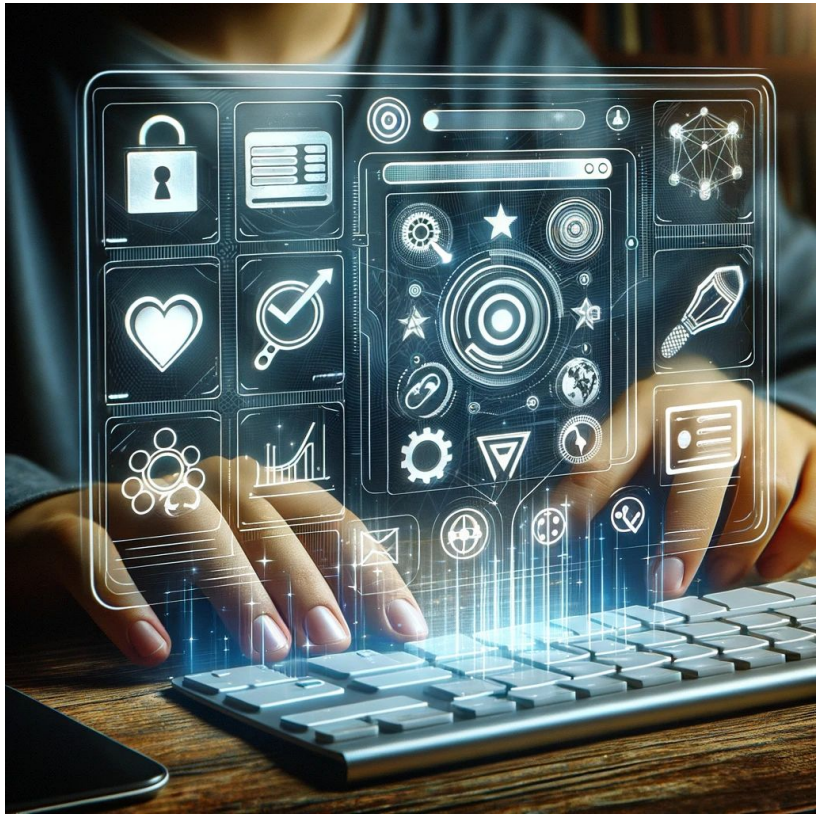
Introducing Dependency Injection for Testability

Usability

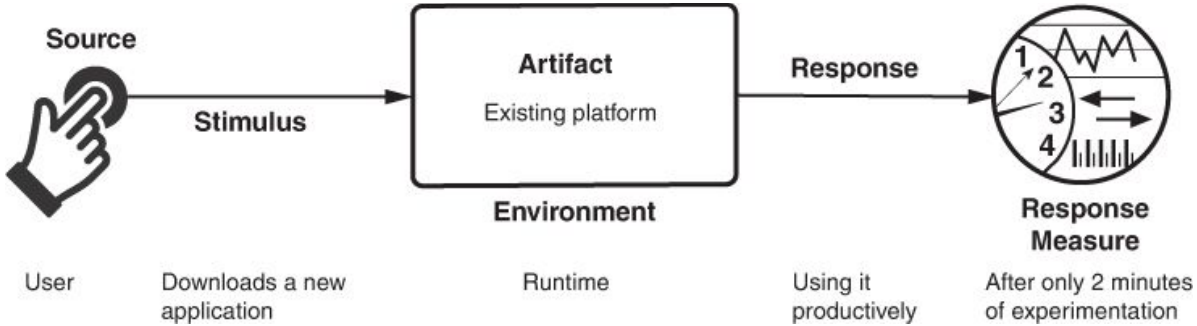
Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support that the system provides.

It measures how easily, efficiently, and satisfactorily users can achieve their goals within a software system. It encompasses the design of user interfaces, interaction flows, and the overall user experience (UX).

High usability impacts user satisfaction directly and can significantly influence the adoption and success of a software product.



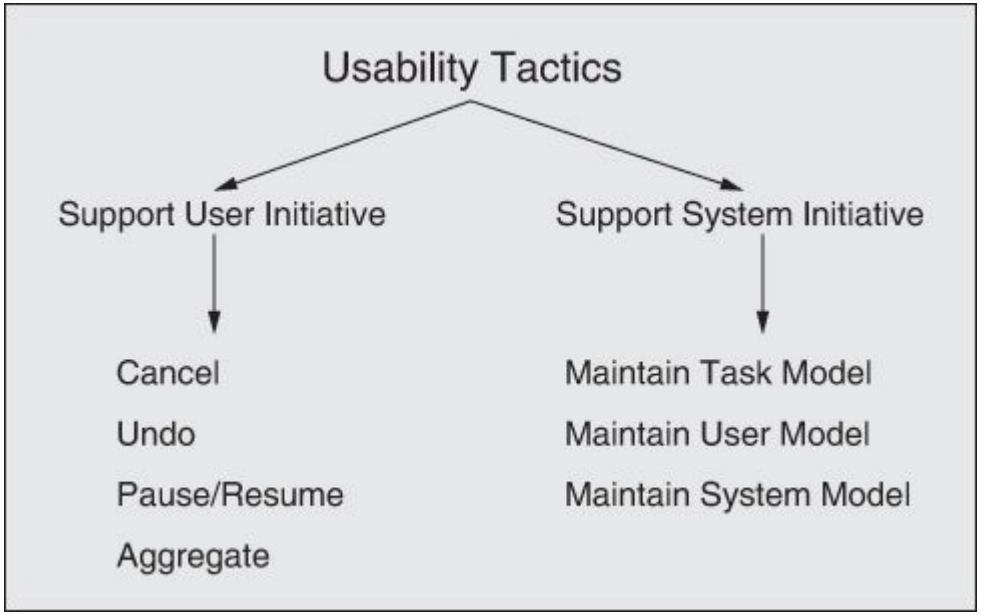
Usability



Usability comprises the following areas:

- **Learning system features:** Ensuring users can quickly understand and utilize the full range of functionalities offered by the software, reducing the learning curve and enhancing user proficiency.
- **Using a system efficiently:** Designing interfaces and workflows that allow users to achieve their objectives with minimal effort and time, optimizing the user's productivity and system performance.
- **Minimizing the impact of user errors:** Implementing safeguards and intuitive design choices that prevent errors before they happen or ensure they have minimal negative consequences, thereby fostering a forgiving and supportive user environment.
- **Adapting the system to user needs:** Allowing for customization and personalization of the software to meet the diverse preferences and requirements of its users, making the system more relevant and useful to individual users.
- **Increasing confidence and satisfaction:** Creating a user experience that not only meets users' expectations but also makes them feel confident in their use of the system, leading to higher overall satisfaction and loyalty.

Usability Tactics & Patterns



Model-View-Controller (MVC) and Model-View-ViewModel (MVVM): Architectural patterns that separate the UI logic from the business logic, facilitating iterative design and testing of user interfaces.

Progressive Disclosure: Presenting only the necessary or requested information to users at any given time to prevent overwhelm.

Feedback Loops: Providing immediate and meaningful feedback to users for their actions to guide and reassure them

Observer Pattern: links some functionality with one or more views.

Memento Pattern: a common way to implement the undo tactic (the originator, the caretaker, and the memento)

Usability

```
public class RegistrationForm
{
   public void RegisterUser(string username, string
password, string email, string address, string
phoneNumber)
    {
        // Complex and lengthy registration process
        Console.WriteLine("Registering user...");
    }
}
```

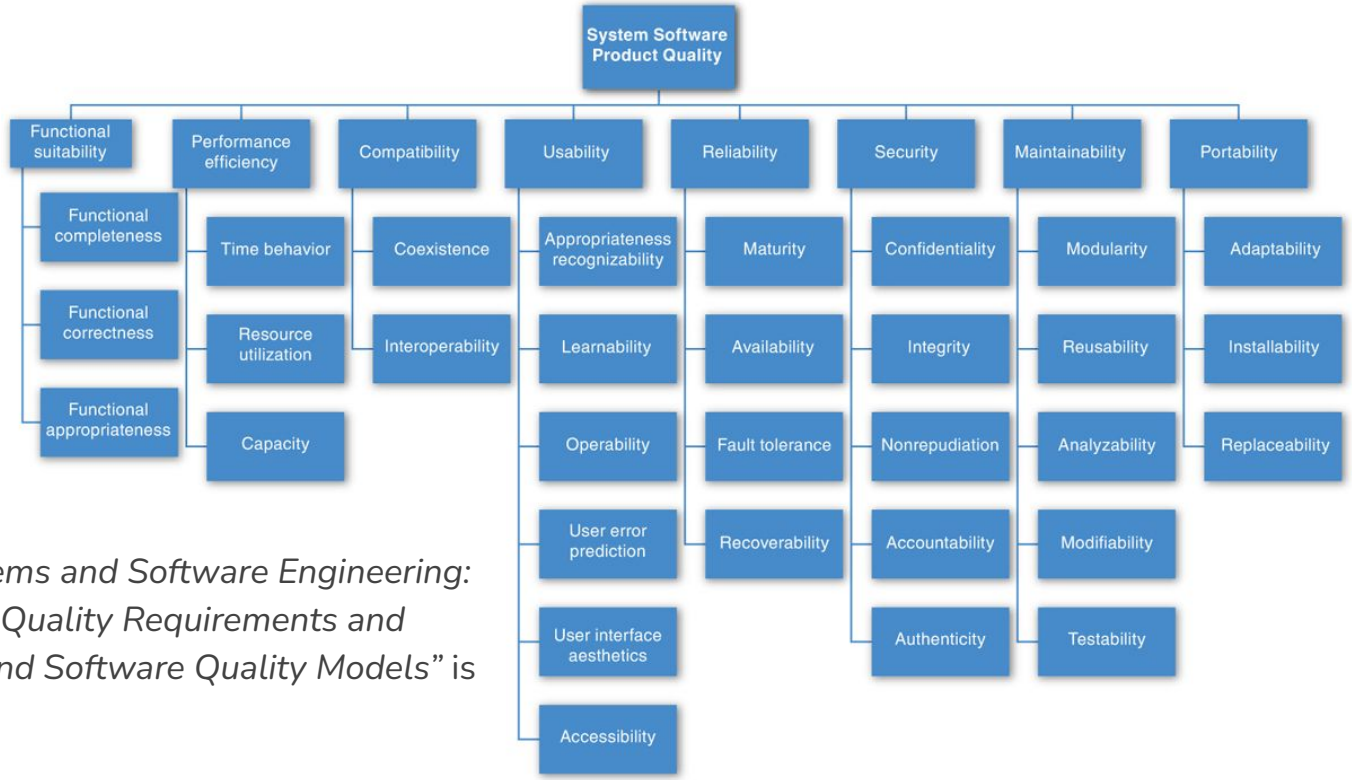
```
public class RegistrationForm
{
    public void RegisterBasicInfo(string username, string password)
    {
        // Basic registration
        Console.WriteLine("Registering basic user info...");
    }

    public void RegisterAdditionalInfo(string email, string address,
string phoneNumber)
    {
        // Optional additional information
        Console.WriteLine("Registering additional user info...");
    }
}
```

Streamlined User Interaction with Progressive Disclosure

Need more Quality Attributes?

Have no fear!

The “*ISO/IEC FCD 25010: Systems and Software Engineering: Systems and Software Product Quality Requirements and Evaluation (SQuaRE): System and Software Quality Models*” is here!



No “recipe” Attribute - X

When there is no compact body of knowledge on Quality Attribute:

1. Capture Scenarios for the New Quality Attribute
    a. Interview the stakeholders whose concerns have led to the need
2. Model the Quality Attribute
    a. an understanding of the set of parameters to which the QA is sensitive and the set of architectural characteristics that influence those parameters.
    b. Example: a model of modifiability defines modifiability as a function of how many places in a system have to be changed in response to a modification, and the interconnectedness of those places.
3. Assemble Design Approaches
    a. Enumerate parameters and architectural characteristics that affect the parameters

Ensuring Portability in a Multi-Cloud World

Group assignment
20 min

https://docs.google.com/document/d/1Y8wjI8dQ1SspBuy5Y38dln7xvO-dCuHrlOaamr7CZXE/edit?usp=sharing

Bibliography

- https://learning.oreilly.com/library/view/software-architects-handbook/9781788624060/9236a39f-8469-4632-b013-4ed9582aab77.xhtml
- https://learning.oreilly.com/library/view/software-architecture-in/9780136885979/part02.xhtml
- https://learning.oreilly.com/library/view/software-architects-handbook/9781788624060/0adb7344-e386-4f19-8493-55d6d02c2345.xhtml
- https://learning.oreilly.com/library/view/software-architecture-the/9781492086888/ch01.html