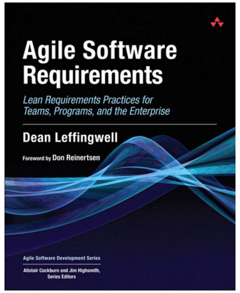
CE Department

Software Requirements Engineering

40688

These slides are designed to accompany Agile Software Requirements (2011) by Dean Leffingwell and support the university course Software Requirements Engineering, instructed by Mehran Rivadeh. Created and designed by Mahnaz Rasekhi.


Agile Software Requirements
Lean Requirements Practices for Teams, Programs, and the Enterprise
Dean Leffingwell
Foreword by Don Reinertsen
Agile Software Development Series
Alistair Cockburn and Jim Highsmith, Series Editors

Agile Software Requirements (2011)
Dean Leffingwell

NonFunctional Requirements

Chapter 17

Mehran Rivadeh
mrivadeh@sharif.edu
Software Requirements Engineering
October 2025 - Fall 1404 - SUT

Contents

Introduction

Introduction

User stories and **features** describe the **functional requirements** of the system.

- ➔ System behaviors whereby some combination of inputs (action) produces a meaningful output (result) for the user.
- ➔ We have learned how to discover, organize, and manage, in an agile manner, the requirements that we must understand in order to build the system functionality our users need to go about their business or pleasure.

Introduction

How to discover, understand, or deliver the other class of requirements, the **nonfunctional requirements**?

- ➔ The “ilities”: security, reliability, scalability
- ➔ System qualities that affect the overall usefulness and the actual viability of the solution.

Introduction

- ➔ Traditionally, one way to think about all the types of requirements is the acronym **FURPS**.
- ➔ We must build and manage the behavior of the system from a number of different perspectives.
  - **Functionality**: What the system does for the user.
  - **Usability**: How easy it is for a user to get the system to do it.
  - **Reliability**: How reliably the system does it.
  - **Performance**: How easy it is for us to maintain and extend the system that does it.
- ➔ we must consider **URP** types of requirements in our system design, even when we approach implementation in an agile manner.

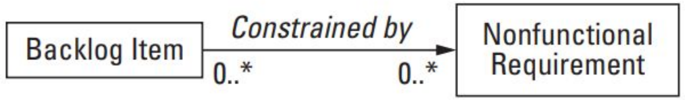
Introduction

Requirement Type	Description	Examples
Functional Requirements	Express how the system interacts with its users, its inputs, its outputs, and the functions and features it provides.	*Display a pop-up on the TV when the utility sends a brownout warning.*
NonFunctional Requirements	Criteria used to judge the operation or qualities of a system.	*The system must be available to its users at least 99.99% of the time.* *The systems should support 100 concurrent users with no degradation in performance.*
Design Constraints	Restrictions on the design of a system, or the process by which a system is developed, but that must be fulfilled to meet technical, business, or contractual obligations.	*Use Python for all client applications.* *Don't use any open source software that doesn't conform to the GNU General Public License.*

Introduction

Association Between Backlog Items And NFRS

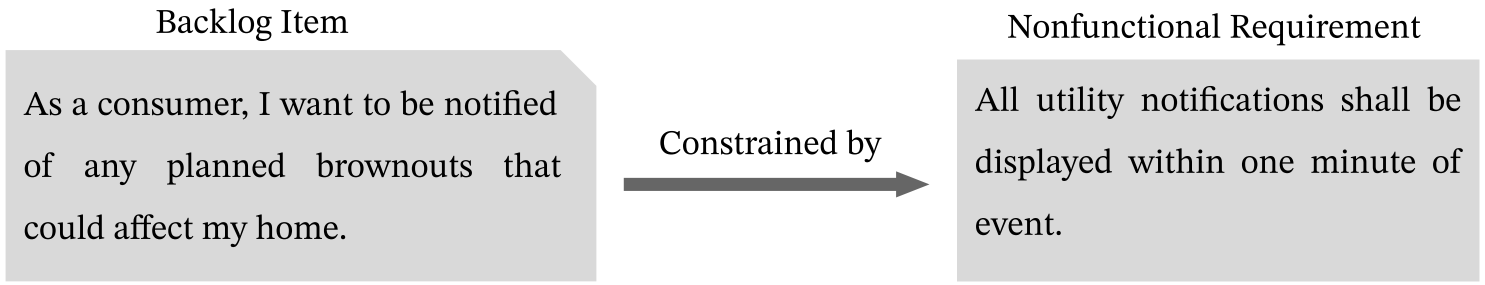
- ➔ A backlog item may be constrained by (zero, one, or more) nonfunctional requirements.
- ➔ Also, nonfunctional requirements apply to zero or more backlog items.
- ➔ Example:
    - ◆ Support 100 concurrent users might apply to zero, one, or many backlog items.



Introduction

Association Between Backlog Items And NFRS (Cont.)

Example:



Backlog Item

As a consumer, I want to be notified of any planned brownouts that could affect my home.

Constrained by

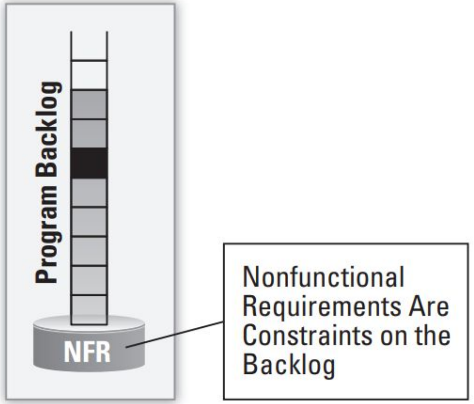
Nonfunctional Requirement

All utility notifications shall be displayed within one minute of event.

Introduction

- Once identified, relevant nonfunctional requirements must be captured and communicated to all teams who may be affected by the constraints.

- In agile, with its focus on the backlog, there is no obvious place to model them, so in the Big Picture, we just call them backlog constraints and represent them as shown in the Figure.



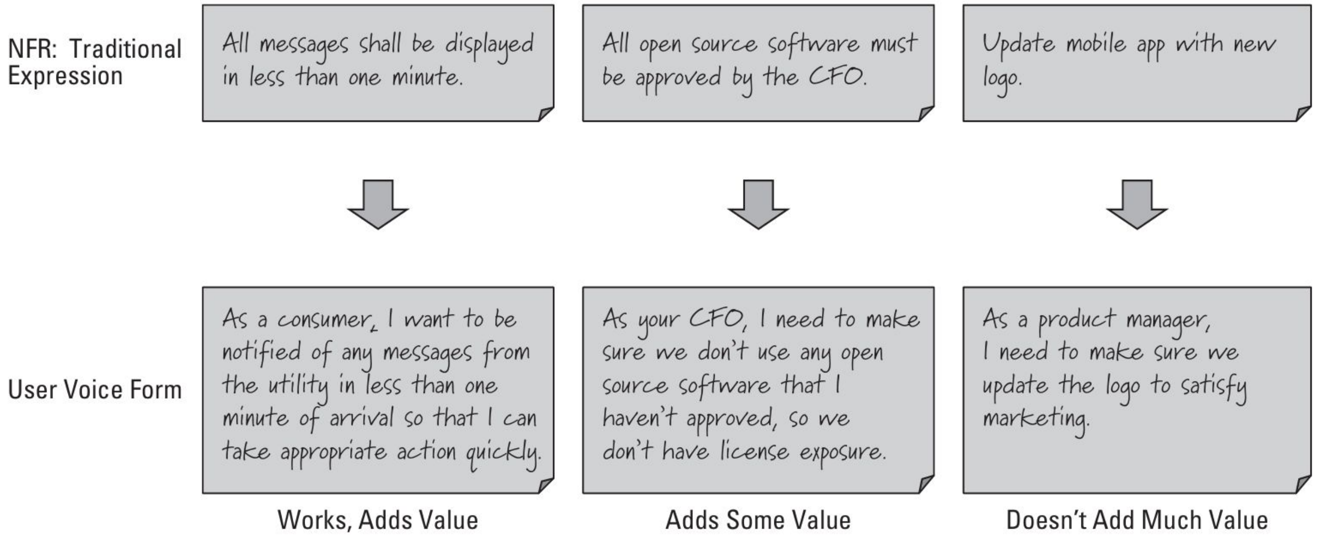
Expressing NFRs As User Stories

Given the predominance of user stories in agile and the more recent user voice form, **there may be value** in expressing nonfunctional requirements in user voice story form.

- ➔ It can **clarify** the **source** and the **business benefit to** the **user** or the **solution provider**.
- ➔ Sometimes it’s worth this little extra effort to communicate a nonfunctional requirement more clearly, but sometimes it isn’t.
- ➔ Using the user voice form **makes good sense** for **NFRs when it adds value**, and it makes equally good sense not to when it doesn’t.

Expressing NFRs As User Stories

Expressing nonfunctional requirements in user voice form


NFR: Traditional Expression
All messages shall be displayed in less than one minute.
All open source software must be approved by the CFO.
Update mobile app with new logo.
User Voice Form
As a consumer, I want to be notified of any messages from the utility in less than one minute of arrival so that I can take appropriate action quickly.
As your CFO, I need to make sure we don't use any open source software that I haven't approved, so we don't have license exposure.
As a product manager, I need to make sure we update the logo to satisfy marketing.
Works, Adds Value
Adds Some Value
Doesn't Add Much Value

Contents

2. Exploring NonFunctional Requirements

a. **U**seability

b. **R**eliability

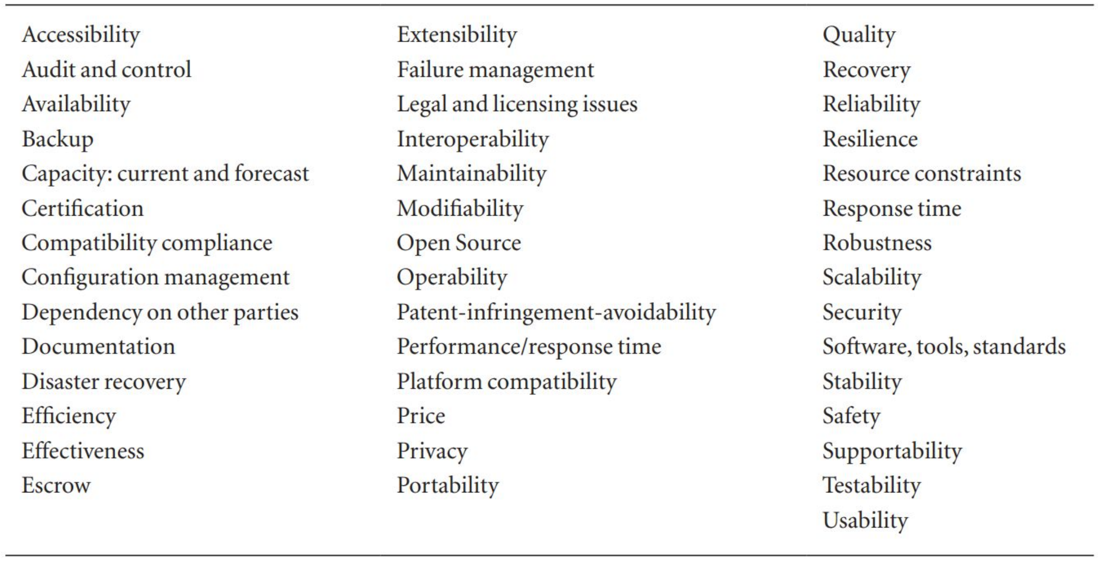
c. **P**erformance

d. **S**upportability (Maintainability)

e. Design Constraints

Exploring NFRs

- There’s a fairly long list of potential NFRs that may apply to a particular project context.
- The table shows a superset of these considerations.



Accessibility	Extensibility	Quality
Audit and control	Failure management	Recovery
Availability	Legal and licensing issues	Reliability
Backup	Interoperability	Resilience
Capacity: current and forecast	Maintainability	Resource constraints
Certification	Modifiability	Response time
Compatibility compliance	Open Source	Robustness
Configuration management	Operability	Scalability
Dependency on other parties	Patent-infringement-avoidability	Security
Documentation	Performance/response time	Software, tools, standards
Disaster recovery	Platform compatibility	Stability
Efficiency	Price	Safety
Effectiveness	Privacy	Supportability
Escrow	Portability	Testability
		Usability

Exploring NFRs

➔ To think about them in a more organized way, we'll return to our URPS acronym:

1. **U**sability
2. **R**eliability
3. **P**erformance
4. **S**upportability

Contents

2. Exploring NonFunctional Requirements

a. **Useability**
b. **R**eliability
c. **P**erformance
d. **S**upportability (Maintainability)
e. Design Constraints

Usability

In today's software products, ease of use ranks as one of the top criteria for commercial success and/or successful user adoption.

How To Specify Useability?

1. Specify the **training time objective** for a user to become minimally productive and operationally productive.
   a. **Minimally productive:** able to accomplish simple tasks
   b. **Operationally productive:** able to accomplish normal day-to-day tasks

Usability

How To Specify Useability? (Cont.)

2. Specify **measurable task times** for typical tasks or transactions that the end user will be carrying out.
    a. Although this could be affected by performance issues in the system (such as network speed, network capacity, memory, and so on), task performance times are also strongly affected by the usability of the system, and we should be able to specify that separately.
3. **Compare the user's experiences with other comparable systems** that the user community knows and likes.

Usability

How To Specify Useability? (Cont.)

4. Specify **any required user assistance features** such as online help, wizards, tool tips, context-sensitive help, user manuals, and other forms of documentation and assistance.
5. **Follow conventions** and **standards** that have been developed for the human-machine interface.
    a. Usability standards:
        i. IBM’s Common User Access (CUA) standards
        ii. Windows applications standards published by Microsoft

Contents

2. Exploring NonFunctional Requirements

a. **U**seability
b. **Reliability**
c. **P**erformance
d. **S**upportability (Maintainability)
e. Design Constraints

Reliability

- ➔ It is an absolute requirement for customer satisfaction.
- ➔ Under the category of Reliability, we might want to consider the following issues.
  1. Availability
  2. Mean time between failures (MTBF)
  3. Mean time to repair (MTTR)
  4. Accuracy
  5. Defects
  6. Security

Reliability

1. **Availability**

   ➔ The system must be available for operational use during a specified percentage of the time.

   ➔ In the extreme case, the requirement(s) might specify “nonstop” availability, that is, 24 hours a day, 365 days a year.

   ➔ It’s more common to see a stipulation of 99.9% availability or a stipulation of 99.99% availability between the hours of 8 a.m. and midnight.

Reliability

2. **Mean Time Between Failures (MTBF)**

   ➔ This is usually specified in hours, but it also could be specified in days, months, or years.

3. **Mean Time To Repair (MTTR)**

   ➔ How long is the system allowed to be out of operation after it has failed?

   ➔ A range of MTTR values may be appropriate; for example, the user might stipulate that 90% of all system failures must be repairable within five minutes and that 99.9% of all failures must be repairable within one hour.

Reliability

4. **Accuracy**

    ➔ What accuracy is required in systems that produce numerical outputs? Must the results in a financial system, for example, be accurate to the nearest penny or to the nearest dollar?

5. **Defects**

    ➔ Defects may be categorized in terms of minor, significant, and critical.

    ➔ Eliminating all critical defects is always the goal, but large numbers of lower-priority defects can also substantially reduce the usability and users' satisfaction with the system.

    ➔ Total defects by type are a pretty good indicator of a user's likely experience with the system.

Reliability

6. **Security**

➔ Some security issues can be addressed in functional requirements (require strong passwords).

➔ However, security can also be specified with nonfunctional requirements such as detect denial-of-service attacks, as well as certain design and coding principles such as verify controls for buffer overruns.

➔ Also, in mature markets, published security standards are likely to exist.

Contents

2. Exploring NonFunctional Requirements

a. **U**seability
b. **R**eliability
c. **Performance**
d. **S**upportability (Maintainability)
e. Design Constraints

Performance

- ➔ It specifies how responsive a system is to users or other systems.
- ➔ How a system is likely to degrade with increasing load.

Types of NFRs in the performance category might include the following:

1. **Response time :** Specify for transactions of a given type, average and worst case.
2. **Throughput :** Specify in transactions per second, latency, overhead, data transmission rates, and so on.
3. **Capacity :** Specify the number of customers, transactions, data, and so on, the system can accommodate.

Performance

4. **Scalability** : Specify the ability of the system to be extended to accommodate more interactions and/or users.

5. **Degradation Modes** : Define an acceptable behavior for when the system has been degraded.

   a. For example, it may be permissible for a system to become slower with load, or even deny users access to certain services, as opposed to a system crash.

6. **Resource Utilization :** If the new system has to share hardware resources with other systems or applications, it may also be necessary to stipulate the degree to which the implementation will make “civilized” use of such resources as the CPU, memory, channels, disk storage, and bandwidth.

Contents

2. Exploring NonFunctional Requirements

Supportability (Maintainability)

- The ability of the software to **be easily modified** to accommodate enhancements and repairs.
- For some application domains, the likely nature and even timing of future enhancements can be anticipated.
  - Like: protocol changes, annual tax rule changes, standards compliance response timelines, availability of new data sources, and so on.
- There may be a mandate for a team to be able to respond to these anticipated changes.
- Customers may also require service-level agreements for various types of defect fixes and system enhancements.

Contents

2. Exploring NonFunctional Requirements

a. **U**seability

b. **R**eliability

c. **P**erformance

d. **S**upportability (Maintainability)

e. Design Constraints

Design Constraints

- ➔ Design constraints are often treated as **another class of nonfunctional requirements.**
- ➔ Most typically, these are created to **enhance supportability**.
- ➔ Design constraints typically **originate** from **one** of three **sources**:
  - ◆ Some necessary restriction of design options
  - ◆ Conditions imposed on the development process itself
  - ◆ Regulations and imposed standards

Design Constraints

Restriction of Design Options

- ➔ Most stories allow for more than one design option.
- ➔ Whenever possible, we want to leave that choice to the developers or user experience experts rather than specifying it in the story, because they are in the best position to evaluate the technical and economic merits of each option.
- ➔ Whenever we do not allow a choice to be made (use Oracle DBMS), a degree of flexibility and development freedom has been lost.
- ➔ However, sometimes this is necessary to improve supportability.
- ➔ Imagine the challenges that an internal support team or customer database administrator would face if each product in a multiple product suite chose a separate database to persist customer data.

Design Constraints

Conditions Imposed on The Development Process

- ➔ Agile teams should usually follow some rules or conditions including adopting
    - Common programming languages
    - Unit testing tools
    - Agile project management tooling
    - Coding standards
    - Configuration management
    - Build environments, and so on.

Design Constraints

Conditions Imposed on The Development Process (Cont.)

- They do this:
  - To assure **systemic productivity** in the agile teams.
  - To support the principle of **collective ownership** (everyone shares responsibility for the code and project, instead of one person "owning" a piece).
  - To enhance **business agility (**the ability of team members to move from one project team to another).

Design Constraints

Conditions Imposed on The Development Process (Cont.)

- ➔ With respect to coding standards, for example, **agile teams take pride** in their code.
- ➔ This is a prudent development practice that **makes good economic sense**.
- ➔ In addition to improving inherent code quality, coding standards improve the ability to refactor and maintain the code, **improving** the **overall velocity** of the **team**.
- ➔ To achieve this, good **teams make** the **software** as **simple** as possible, easy to read (more time is spent reading code than writing code), and easy to refactor.

Design Constraints

Regulations and Imposed Standards

➔ **Some industries**, such as the medical device industry, **have** bodies of **regulations** and **standards** that govern development practices.

➔ Typically, these are **too lengthy** to incorporate directly and are therefore just **included by reference** (*application must fail safely per the provisions of TüV Software Standard, Sections 3.1–3.4*).

➔ Incorporation by reference has its **hazards**, however. For example, a reference of the form *The product must conform to ISO 601* effectively binds your product to all the standards in the entire document, so teams should be careful to incorporate specific and relevant references instead of more general references.

Design Constraints

Managing Design Constraints

- ➔ Design constraints are unique and may deserve a special treatment.
- ➔ You may want to include them in a special section of your backlog or perhaps even create a supplemental specification for that purpose.
- ➔ In addition, you may want to identify the source and rationale.
- ➔ This serves as a reminder of the derivation and motivation for the design constraint, so it can be appropriately applied in the individual team's context.
- ➔ That way, it can potentially be modified or eliminated if the business context changes.

Contents

3. Persisting NonFunctional Requirements

Persisting NFRs

- Nonfunctional requirements is typically **need to persist in** the **development life cycle**.
- User stories are lightweight and generally don't have to be maintained, which is one of the key benefits.
  - The details of a user story are captured in the acceptance test, which persist inside the team's automated or manual regression test environment.
  - That is why we can throw the user story away after implementation; because we have memorialized the important details in our test cases.

Persisting NFRs

That can work for some NFRs, too.

- ➔ For example, “a system must support 1000 concurrent users”
  - ◆ We could develop an automated test that simulated that load and build it in the regression test suite.
  - ◆ That would be an excellent practice because we could refactor the code at will, and if we accidentally created a performance bottleneck, it would be quickly discovered.
- ➔ In that case, we could forget about the NFR once we have seen it the first time, because the automated test remembers it for us.

Persisting NFRs

- ➔ There are **other types of NFRs**, however, that must be **treated** quite **differently**.
- ➔ Here are some examples:
  - ◆ Maintain PCI compliance (credit card industry user security standards) in all applications.
  - ◆ Localize the application in all then-current, supported languages prior to release in any language.
  - ◆ No open source without a CFO license review.
- ➔ We surely **can't forget these**, and we **can't write automated test cases** for them, either.

Persisting NFRs

- ➔ So, the **teams must have** an **organized way** to save them, find them, and review them when necessary.
- ➔ In practice, we've seen agile teams take **a number of approaches to persisting NFRs**:
  - Create a **separate backlog** in the agile project management tool.
  - Store and manage them in a **wiki**.
  - Maintain a **supplementary specification**.
  - Build the NFRs into the **definition of done**, and point to the special backlog, wiki, or supplemental specification that contains the details.

Persisting NFRs

- ➔ No matter the approach, it is mandatory that the teams do something to maintain and manage these specifications, because they could make the difference between success and failure.

Contents

4. Testing NonFunctional Requirements

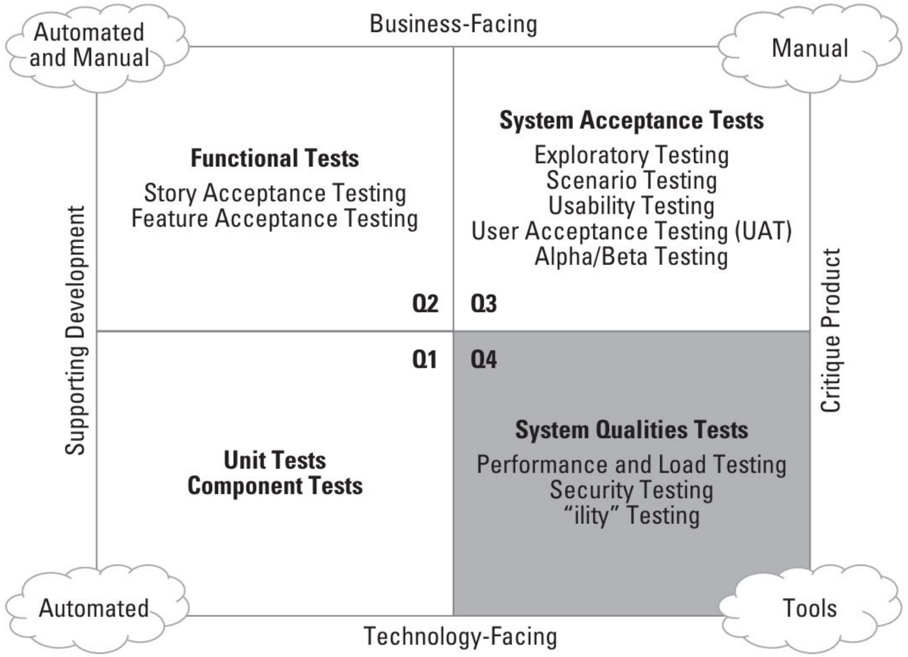
a. Useability Testing

b. Reliability Testing

c. Security Testing

d. Performance Testing

Testing NFRs

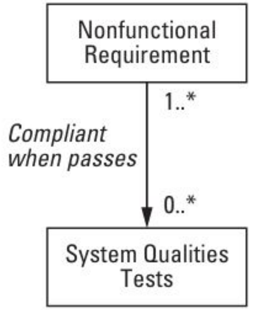


Testing NFRs

➔ Are nonfunctional requirements testable?

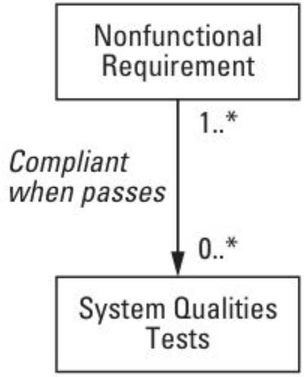
➔ The answer is assuredly yes, because most all of these constraints (performance, for example) can and should be objectively tested.

➔ Indeed, compliance to these requirements is just as critical as it is to meeting functional requirements, and we provide explicit support for this in the model, as illustrated in the Figure.



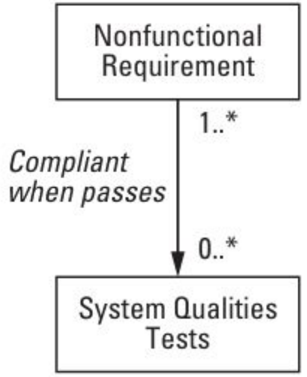
Testing NFRs

- ➔ How this set of tests help assure that the system is in continuous compliance with its nonfunctional quality requirements?
- ➔ The multiplicity (1..* and 0..*) further indicates the following:
  - **Not every NFR has a qualities test (0..).**
  - For example, some design constraints (program in Python) simply aren’t worth testing (other than perhaps by one-time inspection or acknowledgment).



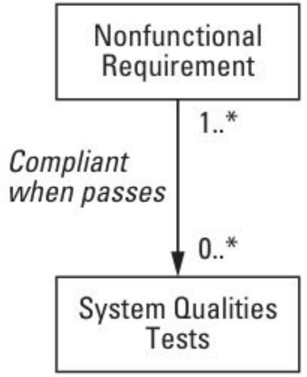
Testing NFRs

- How this set of tests help assure that the system is in continuous compliance with its nonfunctional quality requirements?
- The multiplicity (1..* and 0..*) further indicates the following:
    - **Most NFRs (..*) should have at least one objective test associated with them.**
    - Moreover, some NFRs (for example, runs on IE 8) might require an entire test suite to assure conformance.



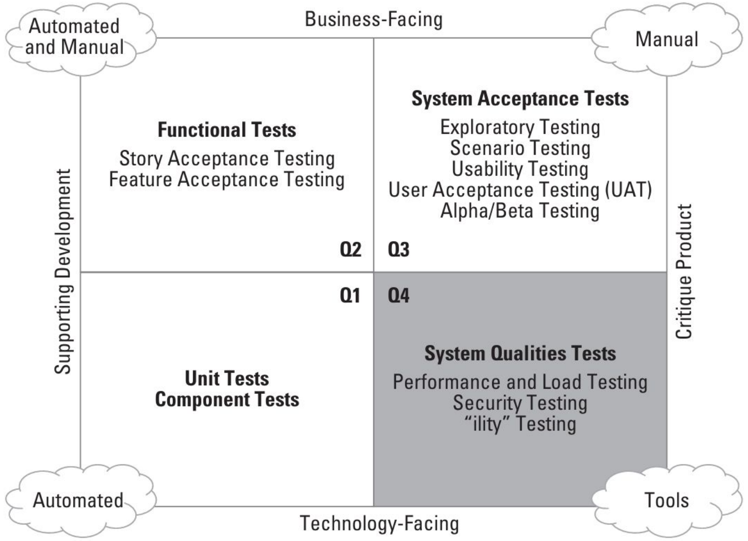
Testing NFRs

- How this set of tests help assure that the system is in continuous compliance with its nonfunctional quality requirements?
- The multiplicity (1..* and 0..*) further indicates the following:
  - **Every system qualities test should be associated with some NFR (1..*)**
  - Otherwise there would be no way to tell whether it passes!
  - And some system qualities tests could assure compliance with more than one NFR.



Testing NFRs

➔ Testing in quadrant 4 is quite different, because it doesn't tie directly to a functional implementation of a story or feature in a timebox.



Testing NFRs

- Some of the qualities tests will be more in the nature of a **one-shot test at** some **appropriate milestone**.
- Example:
  - Before launching a new mobile app, the team runs **a security scan** once to check if any personal data (like email addresses) is accidentally exposed.
  - Once this test is done and passed, it doesn’t need to be repeated every iteration _ it’s a one-shot quality check at that milestone.

Testing NFRs

- ➔ Other tests will be **recurring tests at critical release points**.
- ➔ Example: PCI Credit Card Handling Compliance
  - ◆ If your product processes credit card payments, it must comply with PCI DSS standards (security rules for handling card data).
  - ◆ Every time the product is prepared for a major release (e.g., version 1.0, 2.0, or a quarterly release), the team runs a recurring compliance test.
  - ◆ This test verifies:
    - Encryption of cardholder data is still intact.
    - No sensitive data is logged or exposed.
    - Security configurations meet PCI requirements.

Testing NFRs

- ➔ Regardless of the schedule, **these tests are vital to** the **success of** the **product** and must be factored into the team's thinking at iteration and release planning.
- ➔ The tests you run should match the specific nonfunctional requirement (NFR) you're checking

Contents

Usability Testing

- ➔ Usability testing is a **black-box testing** technique.
- ➔ It tests **how easy it is for users** to achieve their objectives with the system.
- ➔ Typically, usability testing involves **gathering** a **small number of users** (three to five) together and then having them **execute scripts** that use the system in predetermined ways.
- ➔ During this process, those administering the tests typically focus on measuring four aspects of the user's interaction with the system.

Usability Testing

- ➔ **Productivity** : How long does it take a user to perform a particular task?
- ➔ **Accuracy** : How many errors or missteps does the user experience along the way?
- ➔ **Recall** : How easily can a user recall how to use the system if they have been away from the system for a while?
- ➔ **Emotional Response** : How does the user react to the experience of using the system_drudgery, acceptable, interesting, or fun?

Contents

Reliability Testing

- Reliability is somewhat easier to test because the objectives may be clearer, especially if the team has stated any service-level requirements such as availability, mean time between failures, and so on.
- In addition, there are many language-specific profiling tools that assist developers in performing low-level tests such as testing for memory leaks, potential race, and other code conditions that have shown to be typical root causes of reliability problems.

Reliability Testing

However, there are still challenges:

- ➔ It may not be possible to test the product for the amount of time that would be required to collect such data.
- ➔ Instead, stress and load testing may be necessary to accelerate potential failure conditions.
- ➔ It can be difficult to simulate the user's real operating environment, because there could be many factors, such as other systems, environment, types of users, and so on, that affect the long-term results.

Contents

4. Testing NonFunctional Requirements

Security Testing

- Security is a special type of reliability testing.
- It can be approached from two perspectives,
  - White-box testing
  - Black-box testing

Security Testing

White-Box Testing

- ➔ In white-box testing, the testing regime **examines** the **actual code** to look for potential coding practices and paths through the code that can **allow security breaches**.
- ➔ However, it is not generally practical to examine all possible pathways and combinations.
- ➔ Instead, the number of potential combinations must be pruned to a manageable set of tests, usually by deciding on the **most likely** and **most important pathways** through the code.

Security Testing

White-Box Testing (Cont.)

- The **comprehensiveness** of the testing is determined by the **criticality** of the **product** and its **financial** and **legal impact**.
- Example : An online banking system that allows customers to transfer money.
  - If there’s a security flaw, attackers could steal funds or access private financial data.
  - Testing approach: Requires very comprehensive security testing such as penetration tests, encryption checks, and compliance audits.

Security Testing

Black-Box Testing

- ➔ Black-box testing **mimics** the way in which r**eal-life hackers** try to defeat a system.
- ➔ Using scripts and tools, the test regime can **inject various faulty inputs into** the **system** and try to "break the system."
- ➔ **Story-level black-box testing** can be performed **in** the course of the **iteration** because unit tests are comparatively easy to perform by the individual developer because of the limited scope of the required tests.
    - ◆ For example, it is fairly easy to inject faulty data into a low-level routine at the time the code is written, observe its behavior, and correct any resultant security flaws in the design at that early stage.

Security Testing

Black-Box Testing (Cont.)

- ➔ Things get more complicated as the system evolves into higher levels of functionality.
- ➔ As the level of sophistication evolves, the unit tests, component tests, scripts, and tools will necessarily have to evolve too.

Contents

Performance Testing

- ➔ Performance testing is usually done with the **assistance** of **specialized tools**.
- ➔ At the scope of system-level testing, user and other system load simulators, measuring, and monitoring tools are available.
- ➔ These are used to **simulate** a **heavy load on** a server, network, system component, or other object to test its resilience and to analyze overall performance under different load types.

Performance Testing

➔ Open source and commercial tools offer differing capabilities, and the teams need to select an appropriate set.

➔ Custom, purpose-built tools may also be required for complex systems.

➔ To assure that each increment of the new system works as intended, this type of testing should be started in early iterations, and the tests need to be added to the daily build, iteration, or release-level regression testing criteria.

End of Chapter 17

Contributions

- ➔ Author of Reference Book: **Dean Leffingwell**
- ➔ Course Instructor: **Mehran Rivadeh**
- ➔ Slide Creator: **Mahnaz Rasekhi**
    - ◆ These slides are primarily based on Agile Software Requirements by Dean Leffingwell, with occasional adaptations to enhance clarity and engagement.