AU™ Certified Practitioner in Agile Testing (CPAT 2.0)

Released Version

Copyright Notice

Thank you to the authors

- Bas Kruip, Bram van den Berg, Ewald Wassink, Jonathan Huizenga, Rob van Steenbergen

Thank you to the review committee

- Arian de Jong, Emilie Potin-Suau, Kristof van Kriekingen, Kyle Siemens, Guillaume Cousin

Revision History

Version	Date	Remarks
0.1	26-8-2025	First version, input from whitepaper / slides
0.2	16-9 2025	Rework due to new concept (“Less is more”)
0.9	14-10-2025	Formatting, chapters, last review comments. Ready for Brightest review
0.95	11-11-2025	Review Brightest included
0.96	01-12-2025	Highlighted items to be solved (BK)
0.97	19-12-2025	Updates cross check slideshow
1.0	10-02-2026	Last updates. Final version
1.1	26-2-2026	2nd review Brightest (Guillaume Cousin) included
2.0	09-03-2026	Public launch of AU-CPAT V2.0

Table of Content

0 Certified Practitioner in Agile Testing (AU-CPAT)

0.1 Business Outcomes

Business outcomes (BOs) are a brief statement of what you are expected to have learned after the training.

BO-1	Understand the role of a tester in an Agile environment.
BO-2	Understand the Agile Test Model as to understand the overall context of software development and the test activities as part of that development.
BO-3	Be able to analyze and build a detailed understanding of the client context, the product to be build, and the product already realized.
BO-4	Facilitate the identification of all quality attributes as defined in ISO/IEC 25010 to be addressed in the client context.
BO-5	Be able to execute a successful SWOT (Strengths, Weaknesses, Opportunities, Threats) analysis.
BO-6	Add value to the development process by analyzing client needs, choosing different approaches, like Behavior Driven Development, horror plots, checking user stories with the INVEST rules and looking for omissions, contradictions and uncertainties in the requirements.
BO-7	Execute a 'concerns and damages'-session to map risks as a guideline to implement extra test activities and to fix existing production bugs.
BO-8	Understand what verifying the software product means and what techniques can be used.
BO-9	Understand the value of automation and its limits.
BO-10	Understand what validating the software product means and which techniques can be used to do validation.
BO-11	Apply exploratory testing as a validation technique.
BO-12	Be able to do effective reporting to management and other stakeholders.
BO-13	Be able to do effective questioning.

0.2 Learning Objectives/Cognitive Levels of Knowledge

Learning objectives (LOs) are brief statements that describe what you are expected to know after studying each chapter. The LOs are defined based on Bloom's modified taxonomy as follows:

Definitions	**K1 Remembering**	**K2 Understanding**	**K3 Applying**
Bloom´s definition	Exhibit memory of previously learned material by recalling facts, terms, basic concepts, and answers.	Demonstrate understanding of facts and ideas by organizing, comparing, translating, interpreting, giving descriptions and stating main ideas.	Solve problems to new situations by applying acquired knowledge, facts, techniques and rules in a different way.

Verbs (examples)	Remember Recall Choose Define Find Match Relate Identify Recognize	Summarize Generalize Classify Compare Contrast Differentiate Explain Interpret Rephrase	Implement Execute Use Apply Plan Select Prepare

For more details of Bloom's taxonomy please, refer to **[BT1]** and **[BT2]** in References.

0.3 Hands-on Objectives

Hands-on Objectives (HO) are brief statements that describe what you are expected to perform or execute to understand the practical aspect of learning. The HOs are defined as follow:

- HO-0: Live view of an exercise or recorded video.
- HO-1: Guided exercise. The trainees follow the sequence of steps performed by the trainer.
- HO-2: Exercise with hints. Exercise to be solved by the trainee, utilizing hints provided by the trainer.
- HO-3: Unguided exercises without hints.

0.4 Prerequisites

Mandatory

- None

Recommended

- Some Agile or Scrum certificate like PSM or CSM or ASF. At least read the Scrum guide.
- Basic knowledge of testing in general.
- Having at least read the latest version of the Scrum guide available.
- Having at least one year or more working experience in Agile and in testing.

0.5 Extra information concerning this syllabus

- This syllabus describes the business outcomes and learning objectives for agile testing.
- Any person with a +1 year of experience in software testing that is interested in how to function in an agile environment is the target audience for this syllabus (and training).
- Please be aware that studying this syllabus alone will not guarantee you will pass the exam. Additional training may be necessary.
- The objects described in this syllabus follow the agile principles: they should only be used if they help your team to make the product better (client centered).
- The word 'tester' is used throughout this syllabus. What we mean with this term is any member in your agile team involved in testing / quality.

1 Agile Test Model (ATM)

2.1 Why do we test software?

LO-1.1	K1	Remember the ISO 9000 definition of (software) quality
LO-1.2	K1	Remember the definition of software testing
LO-1.3	K1	Recall the structure the Agile Test Model (ATM)
LO-1.4	K2	Explain the model of David Hestenes
LO-1.5	K2	Explain how to translate the model of David Hestenes to the Agile Test Model (ATM)
LO-1.6	K3	Apply the Agile Test Model (ATM) to place tester activities within the model
HO-1.1	HO-2	Practice and discuss daily activities in relation to the ATM

Consider buying a used car. Or imagine yourself being a carpenter who just installed a door. Or a medieval king who employs a taster.
Testing is about experiencing whether a product (or service) meets our expectations *before* starting to use it. In other words, we want to have ***confidence*** in the product (or service).

A used car might look beautiful on a website, a door might look neatly installed, or dinner might smell delicious, but we want to be sure that disappointment (in the form of a rattling car, a stuck door, or unwanted sick leave) will be avoided.

When we talk about confidence, we talk about quality. To ensure that we do not trigger a big discussion on the definition of the concept of "quality," the ISO 9000 standard **[ISO-1**] is chosen: "Quality is the degree to which a set of inherent characteristics of an object meets the requirements."

This standard is specified for software quality as: "The degree to which a set of inherent features of a software product meets the requirements". And this leads ultimately to the definition of software testing:

"To pursue the highest possible degree to which a set of inherent characteristics of an object fulfills requirements."

1.1 David Hestenes and his modelling theory

To make sure that a software tester can strive for the highest quality, (s)he needs to be able to realize that quality control on every step of the software product development process. To be able to give a clear sight in that process, we developed the Agile Test Model (ATM). This ATM is primarily based on the works of David Hestenes **[DHE-1]**, an American theoretical physicist and educator who developed a model that encompasses any product development. His model consists of three worlds and the relationships between them, which we'll refer to as "bridges" for convenience.

Three Worlds

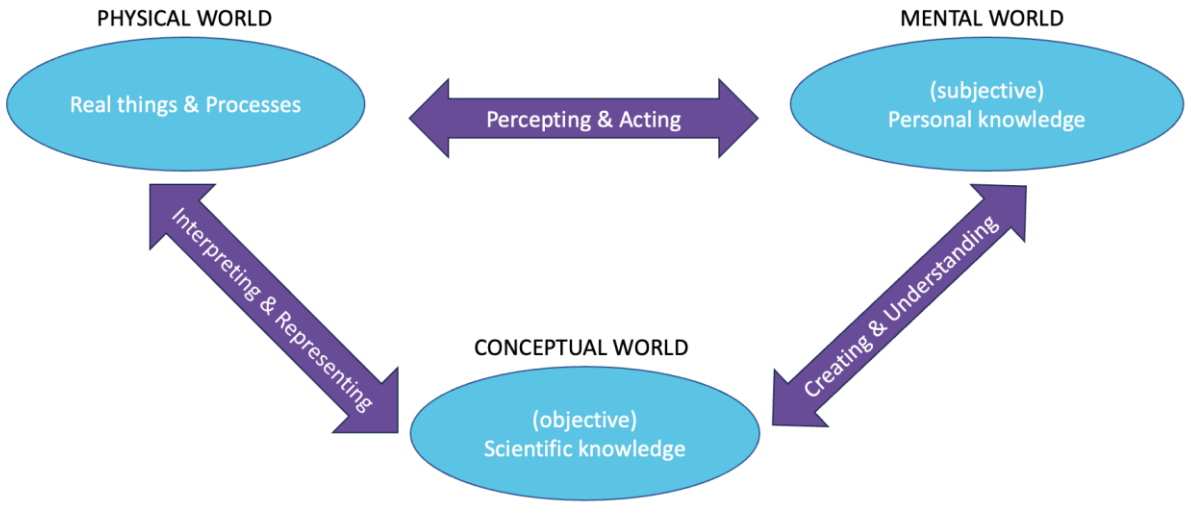
1. The first world is the personal *mental* world, from which needs arise.
2. The second world is the interpersonal *conceptual* world, where the needs of one or more personal mental worlds are brought together and formed into a shared concept that is understandable to all involved.
3. The third world is the *physical* world, which is a concrete interpretation of the conceptual world. This physical world forms a source for formulating (new) needs.

Three Bridges and its activities

The three worlds defined above are connected by three bridges. These symbolize the activities that ensure the three worlds interact, so that the needs from the personal mental world can ultimately be realized in the physical world. The activities that take place on the three bridges are as follows:

1. There are activities that take place between the personal mental world and the conceptual world. By communicating the needs arising from our own mental model, mutual understanding arises regarding a solution to be found. We understand each other, and based on that mutual understanding, we can imagine (create) a shared (concept) solution.
2. Once that solution is sufficiently unambiguous and clear for all involved (in other words: a shared conceptual world has been realized), we can then make this solution physical, with that physical realization being an interpretation of the concept. The previously created concept then, in turn, becomes a representation of the physical solution.
3. Once this solution has become reality, the new physical world (which has changed due to the realization of the product) provides input for the personal mental world through observation and interaction (percepting) to define new needs (acting).

This is summarized in the following figure:



(Figure 1)

1.2 Agile Test Model (ATM)

We can apply Hestenes' model to software testing.

Three worlds

1. First, we transform the mental world into the client's world. This makes sense, as we saw in the definition of testing that the client's needs are the starting point for any testing activity.
2. Second, we replace the conceptual world with the actual concept, which serves as a blueprint for realizing the client's complex software needs. These can be wireframes or prototype versions of software, user stories, system drawings, etc.
3. Third, in this model, the physical world consists of the actual software product. This is either a completely new, standalone product (for example the release of a new app in an app store), or a partial product that forms part of a new system in development.

Three bridges and the actions taken when testing software

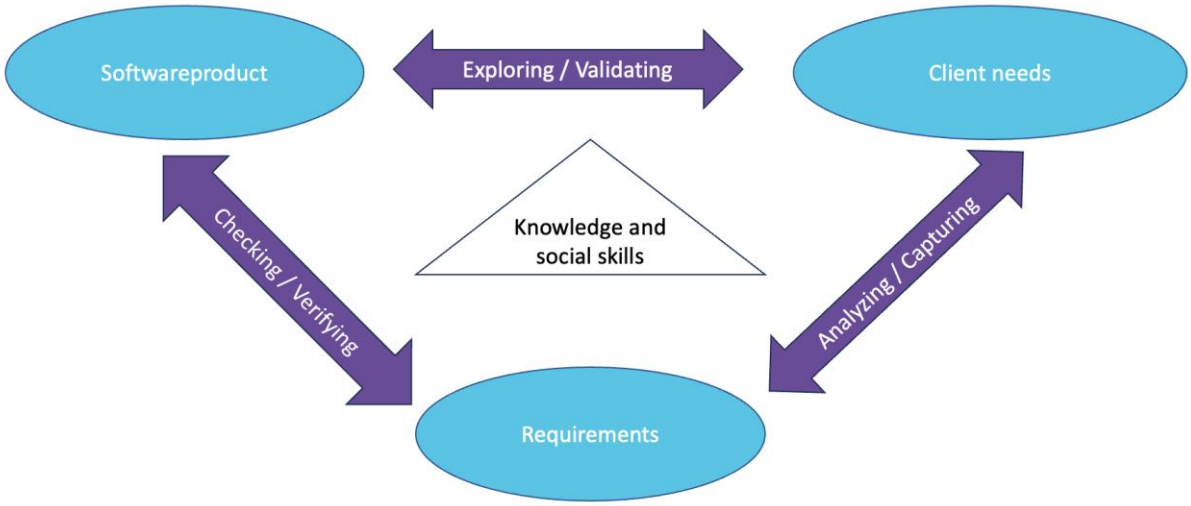
All testing activities take place on the three bridges between these worlds:

1. The bridge between client needs and concept can be summarized by the terms "capture" (client-focused) or "analysis" (test-focused). Client requirements must be captured and translated into clear, objective concepts. All testing activities that help the team analyze client needs and concerns and reduce them to clear and unambiguous requirements apply to this bridge. Typical testing activities on this bridge include 'three amigos'-sessions, risk analyses, and reviewing the concept documentation (focusing on omissions, ambiguities, and contradictions).
2. On the bridge between the concept and the software product, the software tester "verifies" (client-focused) or "checks" (test-focused) the software product based on the agreed-upon requirements. Typical testing activities here include all verification testing techniques, as well as the execution of test scripts, whether automated or not.
3. The third bridge focuses primarily on how the realized software product relates to the real-world context within which it must function. On the one hand, only now can we search for hazards (exploration: identifying causes of potential product value depreciation) that were not anticipated during the analysis. On the other hand, new insights emerge from the client's real-life experience with the actual product (validation) and new client needs emerge based on these insights. Typical testing activities on this bridge include user acceptance testing, exploratory testing, bug hunts, and penetration testing.

Knowledge and social skills

What Hestenes' model doesn't cover are the general consultancy skills and knowledge the software tester needs to achieve the objective within the socially dynamic context of any agile work environment. These social skills include communication skills, such as reporting and questioning. These skills must be deployed across all bridges.

If we plot the entire transition on Hestenes' model and consider this our Agile Test Model, it looks like this:



(Figure 2)

Summary
Software testing is to pursue the highest possible degree to which a set of inherent characteristics of an object fulfils requirements. We do this by:

1. Analyzing / Capturing the client needs,
2. Checking / Verifying the requirements and
3. Exploring / Validating the software product.

1.3 On 'pursuing'

The "reason for being" of a software tester is to pursue quality. And this entails quite a few things:

1. First, the software tester must be aware of the client's needs.
2. Furthermore, the software tester must be aware of the (proposed) product features that meet these needs and, by extension, be able to identify and anticipate associated risks.
3. The software tester must then be able to deploy resources, knowledge, skills, and experience to compare the requirements with the product, and manage any discrepancies found between them. This means either exerting influence to adjust the client's needs, and/or persuading the development team to adjust the concept and/or product. This also implies the use of communication skills such as asking questions and reporting.
4. Finally, the software tester must be able to extensively validate the completed software product in an environment that mimics reality as closely as possible, whereby any issues found lead to refactor work or to the creating of new requirements.

1.4 Next level tester

Compared to the "waterfall" tester, this makes the agile tester a whole next level in software testing. This 'next level' agile tester:

- Helps the client in the process of reviewing and creating the product.

- Helps the team actively to constantly improve the product.
- Develops the quality awareness of the team.
- Performs next to the verifying tests validations tests like exploratory testing.

2 Preparing the Agile Test Approach (ATA)

The Agile Test Model presented in the previous chapter gives us an easy way in to understand what information we need (that is: the information confined in the three separate worlds) to be able to do our work as a software tester (that is: executing all our activities on the three separate bridges).

2.1 World outlines

Let's look at these worlds first: an easy way to get the information held by these three worlds, is by creating outlines. An outline is a sketch, like chalk lines on concrete. The outlines we need are the client outline, the concept outline, and the product outline.

On creating the outlines

The outline is developed and enhanced by collecting information from the organization itself. Preparing for testing involves actively reaching out and asking questions to build a well-rounded understanding of every relevant aspect.
The term 'outline' implies a freedom of creation. The most important part of an outline is that all relevant information is disclosed. How you do that, or how you organize that information is of less importance. The categories that we propose are suggestions only, not strict rules.

2.1.1 Client outline

HO-2.1	HO-3	Create a client outline and present it to everybody
LO-2.1	K2	Explain the definition of client outline
LO-2.2	K3	Apply the client outline to a given project context
LO-2.3	K2	Explain the relation between a client outline and testing
LO-2.4	K1	Recall the mnemonic "CLIENT" related to the client outline

The client outline gives the software tester a picture of the client context. The client determines the needs of the software product. The components below form categories to structure this needed information.

Campaign

The very first question to ask is why?
What is the client's vision and mission? And how does the proposed software solution fit in (purpose)? The answers to these questions are more important than you might initially think. Every desired detail should ultimately be traceable to the big picture. If this is not the case, waste and loss of quality quickly arise.

Liaisons

This encompasses all the relevant players. It primarily involves everyone who (co-)determines the client's needs or at least influences them. It also encompasses the people who create the product.

Impact
What does the client (and beware: the client seldom consists of only one person) expect of the software tester? Think about metrics and reporting, but also automation, or how to fill the role of quality ambassador, etcetera.

Equipment
This category is about everything the software tester needs to fulfil their task. Which tools and access to systems does the tester have?

Narrative
How agile development is implemented in the organization: is it scrum, of kanban? Do team rules apply and if so, which ones?

Time
Finally, when it comes to time: consider both overall planning and key performance indicators (KPIs). Another useful approach is to focus testing efforts on "deal breakers" (the minimum quality criteria that must be met for a product to be released) to help determine whether the planned schedule can be achieved.

2.1.1.1 CLIENT mnemonic

The mnemonic "CLIENT" helps you remember our categories of a client outline:

Campaign	Mission and Vision
Liaisons	Relevant relationships, client, PO, managers, key users, team(s)
Impact	Client's needs concerning tester performance, test deliverables
Equipment	Tools and access for tester, test environments, etc.
Narrative	Software Development Process
Time	Deadlines, or ongoing process

2.1.2 Concept outline

LO-2.5	K2	Explain what a concept outline is
LO-2.6	K3	Use the concept outline to research information sources
LO-2.7	K3	Apply the concept outline to a project case to create an inventory of information sources
HO-2.2	HO-3	Practice with a team to fill the CONCEPT outline with usable sources
LO-2.8	K1	Recall the mnemonic CONCEPT

Testers need access to a variety of sources to understand what needs to be built and the level of quality that is expected. These sources could include internal documentation, agile artifacts, design and technical documentation, external sources. By organizing these into categories in a concept outline, testers can systematically determine where to look for information.
The concept outline is meant as a complete reference to the collective ideas for building the software product.

Compliance
External and internal constraints that influence how we build, test, and release software. These constraints are non-negotiable and must be demonstrably met.

- Organization: Test / quality policy, security standards, user interface standards, organization test strategy
- Legal and regulatory: WCAG (Accessibility), AVG / GDPR (Privacy), EU AI Act
- Industry specific regulations

Orchestration
Organization, tracking, and coordination of the work being done. Orchestration ensures that everyone knows what to do, what's next, and where to find things.

- Product backlog
- Project boards: scrum, kanban
- Work items: user stories, issues, tasks

Notation
How we express, visualize, and specify what the system should do and how it is designed. Notation artifacts make thinking visible.

- Functional: use cases, feature specifications, process models
- Design: mockups / prototypes, UI/UX designs, UI style guides, prototype
- Infrastructural: data models / database schemas, configuration guides, deployment guides, system diagrams / architecture diagrams, API specifications

Coverage
The information sources in this outline may reach beyond the current sprint, feature, timeframe, or project. Coverage describes the scope and boundaries of the feature, change, or current insight: what is included in the present context, what is excluded, and where testing should focus.

- Traceability: risks, features, code / components
- Audit checklists: internal QA, DoR, DoD
- In scope
- Non-scope

External sources
Information from outside the organization that helps us improve product quality, prioritization, and competitiveness.

- Competitor documentation: feature descriptions, help pages, release notes
- Comparable products: demos, trial accounts, product reviews
- User feedback: customer reviews, support tickets, interviews

Product roadmap
How we create, communicate, and maintain a shared understanding of where the product is going, why, and how work connects across teams and timelines.

- Product increment overview: past, current, next
- Cross-team planning: dependencies, integration timelines
- Marketing planning: release campaigns, feature messaging
- Progress dashboard: burn up, burn down, KPI tracking
- User journey maps: end to end user flows

Technical
How are the code repositories & version history organized? Is there any CI/CD pipeline documentation? Where is the test data & metadata? Where to find and analyze deployment logs & changelogs? The technical outputs and environments that enable development, testing, releasing and operating the product.

- Code repository: GitHub, Bitbucket
- CI/CD pipelines: deployment, automated tests
- Test data: anonymized data, test sets
- Logs: application, runtime logs
- Change & release notes: release history, deployment notes, known issues list

Using the categories
Go through these categories during refinement or preparation for new features or iterations. A concept outline should address the following questions:

- What sources are available to understand the feature(s)?
- Where are the quality expectations documented (if at all)?
- Are there external factors (e.g., regulations, market trends) that we must consider?
- Are there gaps in the available information? If so, who can close it?

This approach encourages thinking beyond the user story, ensuring that test ideas and strategies are based on a broader and more comprehensive understanding of quality.

2.1.2.1 CONCEPT Mnemonic
The categories you would find in a concept outline are (mnemonic CONCEPT):

Compliance	External and internal constraints that influences our work
Orchestration	How we organize, track, and coordinate the work
Notation	Our specifications on what the system should be
Coverage	What is included and excluded in the present context
External Sources	Information from outside the organization
Product roadmap	A shared understanding of the direction of the product
Technical	Technical output enabling dev/ops, testing, release

2.1.3 Product outline

HO-2.3	HO-3	Create a product outline by exploring the product (Case software/hardware)
LO-2.9	K2	Explain the definition of a product outline

LO-2.10	K3	Apply the product outline to a given project context
LO-2.11	K2	Explain the relation between a product outline and testing
LO-2.12	K1	Recall the mnemonic PRODUCT

A product outline focuses on the integrative experience with the “bad” outside world, connecting the new construction of the software to the existing context within which the product must operate. It is possible, of course, that the isolated new construction works perfectly, but causes disasters after connecting to the bigger picture.

To understand what we are exploring, we need an overview of the product itself. A tester needs to create a map to help them understand the product. If we do not understand the product, we cannot explore it properly.

The product outline is the sketch of the product, which gives a structured visualization of the realized, delivered software.

Platform
This category is about anything the product relies on: on which interfaces (laptops, mobile devices, and which browsers) should the product function? Are there any constraints related to the hardware (laptops, mobile devices), middleware (webservers, databases) or software (browsers, operating systems)?

Relations
This category covers everything that interacts with the product. What external connections does the product have? This includes data delivery systems, output screens and printers. in short, any point where the new system interacts with the existing environment or the real world.

Operate
What does this product do? What about the application – any function that defines the product? Does it perform calculations or include any mathematical functionality? Does it incorporate security measures (user rights, data security, encryption, front and back-end security)?

Data
What does the product process? What happens to the data during input, throughput, and output as the new software integrates into the existing production environment?
How does the product handle data? Think for example about Input / output, persistency, invalidity, the CRUD (Create, Read, Update, Delete) lifecycle.

Users
Who is/will be using the product? More than personas, this category is to introduce all potential system users to the system for testing. Power users? Hackers? Competitors trying to shut down the system (within the product’s functional scope)? These are just a few examples from a wide variety of actual users.

Construction
This concerns all tangible elements of the total product that have been realized as well as the characteristics of the finished product. Think for example about all non-executables (all files other than multimedia or programs, such as sample data, help files) or collateral matter, like packaging, licenses, etc.

Time
What about any time-consuming actions? When integrating the new build into the existing situation, it is important to test whether, within this larger and realistic context, any challenges are observed regarding processing speeds, time-out functionality, time zones, etc. Consider input output delays and intervals, fast or slow input, spikes, bursts, hangs, bottlenecks, interruptions; concurrency: multi-user, time sharing, threads, shared data. Or time-out settings, periodicals, time zones, company holidays, guarantee periods, chrono functions.

2.1.3.1 PRODUCT mnemonic

Platform	Anything the product relies on
Relations	Anything that interacts with the product
Operate	Anything the product does
Data	Anything the product processes
Users	Who are / will be using the product
Construction	All the tangible elements of the product
Time	Any time-consuming actions

2.2 Quality Attribute Address List (QAAL)

HO-2.4	HO-3	Create a QAAL
LO-2.13	K1	Remember the 9 categories of the ISO25010:2023
LO-2.14	K3	Apply the ISO25010:2023 list to structure the QAAL

The outlines are all results of a so-called **inside-out** approach: the specific client-situation is used as a starting point to gather information, which then serves as the foundation for achieving the highest possible product quality.
An **outside-in** approach complements this: based on a generic set of quality attributes, it is determined whether all possible quality aspects of the desired product are covered, and by whom.
Experience shows that the created concept documentation (in the form of, for example, user stories) often does not cover all the quality attributes of the desired product. As a result, client needs are, often unintentionally, only partially mapped. Using a QAAL (Quality Attributes Address List) helps prevent such omissions.

The ISO 25010:2023 standardization
In 2023, the ISO (International Organization for Standardization) last updated the ISO 25010 standard (the standard for software product development). This standard consists of nine quality categories, which include a total of 40 quality attributes **[See Appendix]**.

Method
With sufficient business and development representatives from the organization (e.g., product owners, the business architect, and business analysts), the quality attributes are clarified and systematically mapped one by one. The QAAL should ultimately provide a comprehensive overview of all quality attributes, along with the corresponding business and development stakeholders responsible for them.
This approach ensures that all quality attributes are consciously and completely integrated across the entire development organization.

2.3 SWOT

HO-2.5	HO-3	Create a SWOT
LO-2.15	K2	Explain the value of a SWOT analysis
LO-2.16	K1	Remember what SWOT stands for

Once all information has been gathered, the software tester can determine to what extent they are able to meet all expectations, how to maximize their own contribution, and initiate any additional measures.

An easy way to approach this is by using the well-known SWOT framework (Strengths, Weaknesses, Opportunities, Threats). Ask yourself: what strengths and weaknesses do I have in the context of this client? What opportunities and threats are related to the organization of this client? And finally: what is my strategy on how to deal with this (tailormade) SWOT?

3 Executing the Agile Test Approach

With the preparation complete (including the outlines, QAAL, and SWOT analysis), the agile software tester's playing field has been defined. What now? What will the tester do during the operational phase? In other words: what activities will take place across the three bridges that connect the three worlds?
To explore this further, we use Scrum in this course as the agile framework within which the test activities are placed; for a full understanding of Scrum, we recommend taking a dedicated Scrum course.

3.1 The first bridge: capturing / analyzing

On this first bridge, the software tester's activities can be broadly divided into two main tasks:

1. First, the tester needs to analyze that what the client requires in detail, so that they have a thorough understanding of what is going to be realized.
2. The other activity is to get a firm grip on the risks involving these requirements, so that damage at an early stage can be avoided or mitigated.

3.1.1 Analyzing the requirements

Requirements in Scrum are analyzed throughout the entire sprint and discussed during so-called 'refinement sessions". The goal of refinement is to capture the client's needs as clearly and concisely as possible. This capture forms the foundation for development and testing.
Nevertheless, requirements can be unclear, incomplete, too big, untestable, too dependent on other requirements or have other shortcomings that can make them a potential source of issues. There are several ways to improve the quality of these requirements.

3.1.2 Behavior Driven Development

LO-3.1	K1	Recall the three stages of Behavior Driven Development (BDD)
LO-3.2	K2	Explain how Specification by Example (SBE) relates to BDD

First, there is a software development method called 'Behavior Driven Development' (BDD) **[BDD-1]**. This method encourages collaboration among developers, testers and business analysts, especially when creating requirements. Collaboration among the three parties should ensure the creation of a shared understanding, using clear examples to illustrate how business rules should function. This process establishes a so-called "single point of truth."

BDD consists of three phases which finally lead to working software:

1. Discovery: in this phase, a user story will be explored in a structured manner: concrete examples will be used to create a clear picture of the required business outcomes of some function that is represented by the story.
2. Formulation: the examples will be rephrased into a structured language what turns them into executable specifications and
3. Automation: finally, these specifications will be turned into automated acceptance tests. These tests work as an OK/Not OK reference for the software to be delivered.

3.1.3 User stories

LO-3.3	K1	Recall the organization of user stories (feature, epic)
LO-3.4	K1	Recall the structure of a user story
LO-3.5	K2	Explain the value of user stories

User stories are the most used standard building block of documenting specifications and serve as small, independent pieces of functionality, like “adding something to a shopping basket.” Normally, several user stories are organized under a ‘feature’ (in the example “The shopping basket”) which on their turn are organized under an ‘epic’ (for example: “the buying client”).

3.1.3.1 INVEST rules

LO-3.6	K1	Recall the INVEST rules
LO-3.1	HO-2	Apply the INVEST rules to a requirement

When analyzing a user story, one can check the story using the INVEST rules.

Independent Is the story independent (of other stories)?
Negotiable Is the story negotiable (can it be changed when not on the sprint backlog)?
Valuable Is the story valuable (does it deliver value to the stakeholder)?
Estimable Can the story be estimated (complete and concrete enough)?
Small Is the story as small as possible? (can it be completed within an iteration?)
Testable Can the story be tested? (are all criteria clear enough verification?)

3.1.3.2 Horror plots

LO-3.7	K1	Explain the value of a horror plot
HO-3.2	HO-3	Create horror plots

Horror plots are scenarios based on user stories that focus on a single question: what can go wrong? When creating a list of possible “horrors” that might occur when implementing the story, extra acceptation criteria or even new requirements can be found. For example: if a user story’s main function is to reserve a room, a horror story can be that after reservation, no confirmation (or error message) is sent, resulting in a nightmare at check-in, because the hotel has never received the reservation.

3.1.4 Omissions, contradictions, uncertainties

LO-3.8	K2	Explain the difference between omissions, contradictions and uncertainties
LO-3.9	K1	Remember the causes of omissions, contradictions and uncertainties
HO-3.3	HO-2	Be able to spot omissions, contradictions and uncertainties in requirements.

LO-3.10	K1	Remember preventative measures concerning omissions, contradictions and uncertainties.

As a software tester, you search for omissions, contradictions or ambiguities within any requirement.

Any of those problems will lead to multiple interpretations among the development team, which will inevitably cause issues during the software's implementation.

3.1.5 Concerns and damages

At the start of a sprint, the sprint goal (the various requirements that together form a specific goal to be achieved during the sprint) is presented, and the sprint is planned. It is recommended that the software tester also hold a "concerns and damages" session. Based on the information obtained during this session, the test activities on bridges two and three can be better estimated and prioritized (i.e., the depth of the tests and the associated timeframe can be better determined).

To gain an understanding of the pain points and concerns for a sprint, two different sources of information are used. The first is concerns, i.e., the concerns the customer and the development team have regarding the product that has yet to be developed (i.e., risks). The second is damages, i.e. the existing bugs in the current system in production that are already affecting the sprint goal.

3.1.5.1 Concerns

LO-3.11	K2	Explain how product importance is assessed in relation to risks
LO-3.12	K2	Explain why a concern is only valuable when it is sufficiently specific
LO-3.13	K2	Explain the challenges involved in defining risks
LO-3.14	K2	Expalin when concerns should be discussed during testing activities

The concerns listed above are those that the customer and/or development team have about the software still to be built, or about the process of building it. Asking stakeholders within (and outside) the team to share any concerns encourages them to think about quality. This is an elegant way to raise quality awareness.
To map these concerns, the following actions can be taken:

- Interviews: ask (key) users, developers, and business analysts what they are most concerned about regarding the software to be developed in the sprint.
- Brainstorming session: hold a session with the development team, business analysts, and/or other stakeholders. This can be either informal or formal. In the latter case, a serious game such as The Nightmare Headline game or Risk Mining can be used. The goal of this session is to gather the following information:
    - the existing concerns.
    - a relative prioritization of the concerns.
    - a brainstormed list of possible causes.

  Based on the causes, charters (input for exploratory testing) can be built.

3.1.5.2 The Nightmare Headline Game

LO-3.15	K1	Remember the steps in a risk headline game
HO-3.4	HO-2	Manage a risk headline game

A simple way to perform a risk analysis is by using the Nightmare Headline Game [**NHG-1**]. This method is based on defining risks inviting the team to imagine what they, or the organization, wouldn't want to see in the newspapers the morning after releasing the software.

It consists of the following steps:
1. Set the stage
2. Gather the headlines
3. Choose a big risk to work on
4. Brainstorm contributing causes
5. Refine causes into charters

3.1.5.3 Damage management

LO-3.16	K2	Explain damage management and its approaches
LO-3.17	K3	Manage a damage management session

It is quite possible that existing bugs in production can be addressed during the upcoming sprint. During the sprint planning, guide the team through the list of current production bugs and decide together which bugs can be solved in the current sprint-stories.

To gain a good overview of known errors, several approaches are possible:
- Past bug analysis: review bug reports to identify problem patterns or problem areas in the existing software.
- Feedback analysis: interview (key) users, developers, and business analysts about known or recurring problems related to the sprint goal.

3.2 The second bridge: verifying / checking

3.2.1 On verifying / checking

LO-3.18	K2	Explain the purpose of activities performed during verification
LO-3.19	K1	Recall static and dynamic validation techniques
LO-3.20	K2	Explain the difference between syntactic, semantic, and construction quality
LO-3.21	K2	Explain the difference between static and dynamic testing

The absolute foundation of verification test techniques involves three independent approaches:

- Syntactic: Is the syntax of the data correct throughout its entire lifecycle? Is invalid data introduced, does data become corrupted during operations, or does the output of one system cause problems for a connecting system because it cannot process the output?
- Semantic: Assuming the syntax is correct, semantic tests verify whether the data is processed in accordance with technical and business specifications. The scope of semantic testing can (naturally) range from unit testing to testing the entire system.
- Construction: While syntactic and semantic tests focus on the properties of the product being tested, construction focuses on passive characteristics: for example, whether the product is intuitive enough, the color schemes are appropriate, there are no language errors, and whether a website meets accessibility standards.

These three approaches should answer the following main question when dealing with verification / checking: "Are we building the product right?" The main goal is to verify that the requirements are built as intended.

Verification techniques can be categorized into static and dynamic test techniques

Examples of static testing techniques

- Code review / Peer review: developers review each other's code for logic errors, style, and potential bugs.
    - Pair programming (informal)
    - Pull request reviews (formalized)
- Static code analysis: Automated analysis of source code for quality issues, security vulnerabilities, or coding standard violations.
- Walkthrough: The author of a document or code leads a team through it to gather feedback.
- Linting: Automatically checking source code for stylistic or programming errors

Examples of dynamic testing techniques: checking when executing code

- Functional Testing: verifying individual functions.
- Automated Testing: verifying the system behaves as expected with automation tooling.
- Integration Testing: Testing combined components or systems for interface defects
- API Testing: Verifying webservices independently of the UI as specified.
- Regression Testing: Re-running existing tests to confirm new changes haven't broken old features.
- Mutation Testing: Intentionally introducing code changes (mutants) to check if tests detect them.

3.2.2 Test automation & tooling

LO-3.22	K2	Explain the distinction between *checking* and *testing* in the context of the Agile Test Model.
LO-3.23	K2	Explain the definition of test automation
LO-3.24	K2	Explain the value of test automation
LO-3.25	K2	Explain the limitations of test automation

LO-3.26	K3	Apply the test automation pyramid
LO-3.27	K2	Explain the difference between the test automation pyramid and the test automation ice cone anti-pattern
LO-3.28	K2	Explain the value and need of automation in an agile context

Test Automation

Checking ≠ testing - "a means to execute test cases automatically, including the checks on the results. (= automatic verifications)."

Test automation can be defined as: an approach for automatically re-executing test cases, including performing the checks against the expected (pre-defined) results through automated validations.

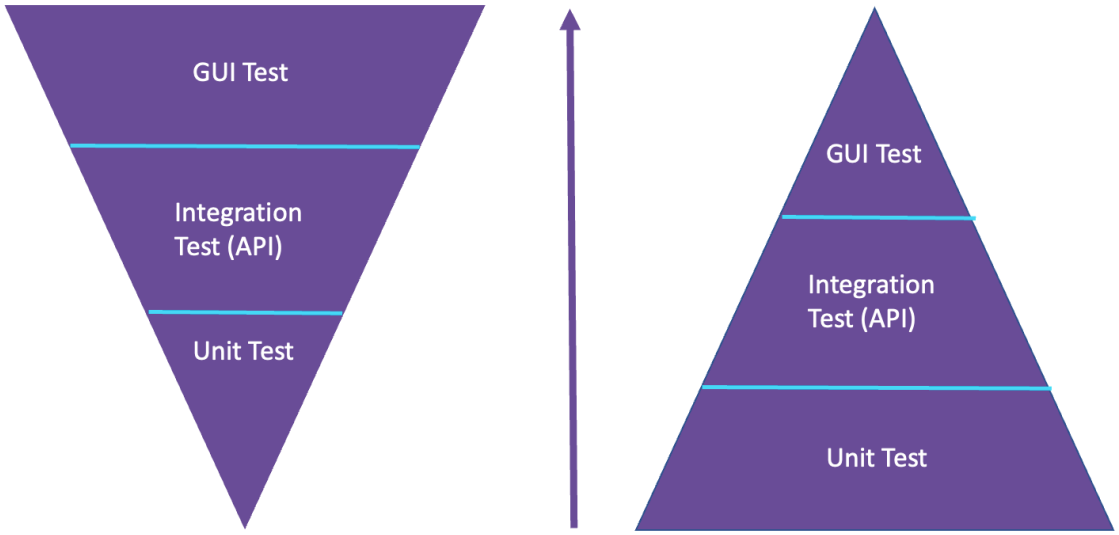
This also highlights the limitations of test automation:

- We need pre-defined results that do not change
- We need to create test scripts upfront, which requires us to already know how the system is expected to behave
- We only cover the facts (checks), Automation mainly covers factual checks, not intuition or exploration.
- We should always question whether repeatedly executing the same automated checks still provides value
- Return on Investment (ROI) on test automation should be considered

The test automation pyramid introduced by Mike Cohn **[MCO-1]** introduces 3 layers of test automation:

- UI level: this level changes frequently, automated tests are flaky and hard to maintain, and tools are expensive to buy.
- API or service level tests are valuable as well as an API has very clear pre-defined content which makes it suitable for automated checking.
- Unit level: an automated unit test delivers in general the highest value as the feedback loop is very short and many checks can be easily covered.

Unfortunately, as the GUI level is the most visible level, a lot of organizations choose to mainly create GUI-level automated tests. This is what is called the "ice cone" anti-pattern of test automation. Having this information gives the tester the opportunity to explain and show the benefit of a good test automation strategy.



Test automation ´ice cream cone´ anti-pattern

Test automation pyramid

(figure 3)

Other test tooling

Most people are familiar with testing tools; however, they often mainly think of them as *test automation* tools. In reality, there are way more tools that can be very helpful at times: to help testers generate test data, set up environments, document tests, such as screen recorders, Word, a wiki or Confluence; some other tools can help find potential bugs more easily or run some first basic security checks, check for first potential performance issues or check on potential usability issues.

3.3 The third bridge: Exploring / Validation

The main goal

While the key question during verification on the second bridge was “Are we building the product right?”, the focus on the third bridge shifts to “Are we building the right product?” Here, software testers aim to confirm that the final product serves its intended purpose and truly meets the client’s needs.

3.3.1 Examples of validation techniques

LO-3.29	K1	Remember validation techniques
LO-3.30	K2	Explain the difference between validation and verification techniques

- Ensemble testing: Testing with the team as if you were one tester.
- User Acceptance Tests: only exploratory when no scripts (or charters) are involved
- Bug hunts: a gamification technique for teams with the business together
- Demo: show the client the new software and ask for feedback
- Exploratory testing in sessions: structured way of testing and learning
- Performance testing: always exploratory if you are searching for boundaries
- Usability testing: the ultimate “ask the client” test

3.3.2 Exploratory Testing

LO-3.31	K1	Remember the definition of exploratory testing according to Cem Kaner
LO-3.32	K3	Execute an exploratory test
LO-3.33	K2	Explain the different sources and possibilities for setting up exploratory testing
LO-3.34	K1	Select appropriate sources to execute an exploratory test
LO-3.35	K2	Explain what a test charter is
LO-3.36	K2	Explain how to set up an exploratory test session
LO-3.37	K2	Explain how to document your exploratory test sessions
LO-3.38	K2	Explain session-based testing
LO-3.39	K1	Recall the definition of BRIEF
LO-3.40	K2	Apply BRIEF to evaluate testing activities
HO-3.5	HO-3	Practice creating charters, using those for testing and debrief with BRIEF

Definition and explanation
Exploratory testing is explained according to the definition of Cem Kaner as follows:
"Exploratory software testing is a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the value of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project."
[CKA-1]

The product is explored with a focus on the risks that the client may face. During this process, testers often encounter areas of the application that were previously unknown or undocumented simply because they had not been discovered yet. In the absence of a clearly defined test basis, testers rely on alternative oracles to identify potential issues.

Test charter and notes
The test charter provides structure and direction for exploratory tests, while the session sheet is used to it to write down the results of our tests. A test charter typically includes:

- Explore where? – the target of your test
- With what recourses? – What you need to explore, data techniques, tools, etc.
- To discover what? – the type of information you aim to discover (security, risks, consistency, etc.)

During exploratory testing, you can use a session sheet to take notes. The BRIEF part of the sheet is to evaluate your session, alone or with a team.

- A session sheet includes:
  - Time box
  - Notes (about what is tested, the results, questions raised, issues found, etc..).
- Debriefing information (BRIEF)
  - **B**ehavior: how did the software behave during our session?
  - **R**esults: what did we achieve?
  - **I**mpediments**:** What obstacles or challenges did we face?

- **E**xpectations: what still needs to be done?
- **F**eelings: how do I feel about the product?

3.3.3 Communication skills

3.3.3.1 Issues and issue management

LO-3.41	K2	Explain what an oracle is
LO-3.42	K2	Explain the value of oracles in testing
LO-3.43	K1	Recall the information to be recorded when registering a bug (defect)

The main goal of testing is not to find as many bugs as possible. The main goal of testing is to give insights on the quality of the product and to help make the product better. Simply documenting bugs will not make the product better. A tester needs to test as much as possible to get valuable insights. As soon as a bug is found, a tester should talk to the developer and/or product owner about the risk posed by the bug found and, as a team, decide together what to do with the bug. If it must be solved, then the team solves it immediately and does not document it other than by mentioning it in a test log. If it cannot be solved immediately, the team can decide to document the bug for later investigation and ask the PO to add a bug fixing item to the product backlog.

3.3.3.2 Reporting

LO-3.44	K2	Explain which essential information is required to create an overview
HO-3.6	HO-3	Create a dashboard for an overview of the quality of the software

Reporting should provide clear visibility into what has been done regarding quality and testing. What activities were performed and with what results? Reporting needs to be lightweight, reflecting the Agile principle of valuing working software over comprehensive documentation. Reporting is usually also necessary for governance, risk and compliance (GRC) purposes. What type of reporting is required highly depends on the context of the organization, the specific project and the product. One rule always applies: reporting needs to fulfil a need. The scope and format of reporting should always be determined by the specific need it aims to address.

Typical information to be reported about:

- What was tested and what was not tested (yet) and why not.
- What were the results of the tests, and the insights gained.
- Who performed the tests and when.
- Which areas require further investigation.
- In what way regulations are incorporated (if applicable)

By following this approach, the team establishes a way of working that truly aligns with Agile principles. Preventing issues over finding issues and solving issues over documenting issues. Testers can also use retrospectives to reinforce this mindset across the team.

3.3.3.3 Questions

HO-3.7	HO-3	Ask questions about a specific subject
LO-3.45	K2	Explain the importance of asking questions to understand contexts, needs, and where to start testing effectively
LO-3.46	K2	Explain the value of questions
LO-3.47	K2	Explain the value of questions in relation to testing

Before testing something, testers should gather as much relevant information as possible. To do so, asking questions is the standard way to go. Questions are also asked to check your own biases, other misconceptions and to promote a shared understanding of the software the team you are working with is building.

4 References

Reference	Source
[BT1]	A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives - By L. Anderson, P. W. Airasian, and D. R. Krathwohl (Allyn & Bacon 2001)
[BT2]	https://www.apu.edu/live_data/files/333/blooms_taxonomy_action_verbs.pdf
[ISO-1]	https://www.iso.org/quality-management
[DHE-1]	https://davidhestenes.net/geocalc/pdf/NotesonModelingTheory.pdf
[BDD-1]	https://dannorth.net/blog/introducing-bdd/
[NHG-1]	https://explorist.in/all-about-the-nightmare-headlines-game/
[MCO-1]	https://www.researchgate.net/publication/234803335_Succeeding_with_Agile_Software_Development_Using_Scrum
[CKA-1]	https://kaner.com/?p=46
Source	ISO25010:2023 ISO/IEC 25010:2023 — Systems and software engineering — Systems and software Quality Requirements and Evaluation - https://iso25000.com/en/iso-25000-standards/iso-25010

5 Appendix: ISO 25010:2023 Complete List

Functional Suitability	**capability of a product to provide functions that meet stated and implied needs of intended users when it is used under specified conditions**
Functional completeness	capability of a product to provide a set of functions that covers all the specified tasks and intended users' objectives
Functional correctness	capability of a product to provide accurate results when used by intended users
Functional appropriateness	capability of a product to provide functions that facilitate the accomplishment of specified tasks and objectives
Performance efficiency	**capability of a product to perform its functions within specified time and throughput parameters and be efficient in the use of resources under specified conditions**
Time behavior	capability of a product to perform its specified function under specified conditions so that the response time and throughput rates meet the requirements
Resource utilization	capability of a product to use no more than the specified amount of resources to perform its function under specified conditions
Capacity	capability of a product to meet requirements for the maximum limits of a product parameter
Compatibility	**capability of a product to exchange information with other products, and/or to perform its required functions while sharing the same common environment and resources**
Co-existence	capability of a product to perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product
Interoperability	capability of a product to exchange information with other products and mutually use the information that has been exchanged
Interaction capability	**capability of a product to be interacted with by specified users to exchange information between a user and a system via the user interface to complete the intended task**
Appropriateness recognizability	capability of a product to be recognized by users as appropriate for their needs
Learnability	capability of a product to have specified users learn to use specified product functions within a specified amount of time
Operability	capability of a product to have functions and attributes that make it easy to operate and control
User error protection	capability of a product to prevent operation errors
User engagement	capability of a product to present functions and information in an inviting and motivating manner encouraging continued interaction
Inclusivity	capability of a product to be utilized by people of various backgrounds
User Assistance	capability of a product to be used by people with the widest range of characteristics and capabilities to achieve specified goals in a specified context of use
Self descriptiveness	capability of a product to present appropriate information, where needed by the user, to make its capabilities and use immediately obvious to the user without excessive interactions with a product or other resources
Reliability	**capability of a product to perform specified functions under specified conditions for a specified period of time without interruptions and failures**
Availability	capability of a product to be operational and accessible when required for use
Fault tolerance	capability of a product to operate as intended despite the presence of hardware or software faults
Recoverability	capability of a product in the event of an interruption or a failure to recover the data directly affected and re-establish the desired state of the system
Faultlessness	capability of a product to perform specified functions without fault under normal operation

Security	**capability of a product to protect information and data so that persons or other products have the degree of data access appropriate to their types and levels of authorization, and to defend against attack patterns by malicious actors**
Confidentiality	capability of a product to ensure that data are accessible only to those authorized to have access
Integrity	capability of a product to ensure that the state of its system and data are protected from unauthorized modification or deletion either by malicious action or computer error
Non-repudiation	capability of a product to prove that actions or events have taken place, so that the events or actions cannot be repudiated later
Accountability	capability of a product to enable actions of an entity to be traced uniquely to the entity
Authenticity	capability of a product to prove that the identity of a subject or resource is the one claimed
Resistance	capability of a product to sustain operations while under attack from a malicious actor
Maintainability	**capability of a product to be modified by the intended maintainers with effectiveness and efficiency**
Modularity	capability of a product to limit changes to one component from affecting other components
Reusability	capability of a product to be used as assets in more than one system, or in building other assets
Analysability	capability of a product to be effectively and efficiently assessed regarding the impact of an intended change to one or more of its parts, to diagnose it for deficiencies or causes of failures, or to identify parts to be modified
Modifiability	capability of a product to be effectively and efficiently modified without introducing defects or degrading existing product quality
Testability	capability of a product to enable an objective and feasible test to be designed and performed to determine whether a requirement is met
Flexibility	**capability of a product to be adapted to changes in its requirements, contexts of use, or system environment**
Adaptability	capability of a product to be effectively and efficiently adapted for or transferred to different hardware, software or other operational or usage environments
Scalability	capability of a product to handle growing or shrinking workloads or to adapt its capacity to handle variability
Installability	capability of a product to be effectively and efficiently installed successfully and/or uninstalled in a specified environment
Replaceability	capability of a product to replace another specified product for the same purpose in the same environment
Safety	**capability of a product under defined conditions to avoid a state in which human life, health, property, or the environment is endangered**
Operational constraint	capability of a product to constrain its operation to within safe parameters or states when encountering operational hazard
Risk identification	capability of a product to identify a course of events or operations that can expose life, property or environment to unacceptable risk
Fail safe	capability of a product to automatically place itself in a safe operating mode, or to revert to a safe condition in the event of a failure
Hazard warning	capability of a product to provide warnings of unacceptable risks to operations or internal controls so that they can react in sufficient time to sustain safe operations
Safe integration	capability of a product to maintain safety (3.9) during and after integration with one or more components