testIDEA TRAINING

Objectives

At the end of this training, you will be able to

- Create unit tests that execute on-target using testIDEA
- Create tests for C and C++ applications
- Export code coverage and test reports for unit tests
- Export tests as Python scripts for test automation using Continuous Integration (CI) tools such as Jenkins

testIDEA

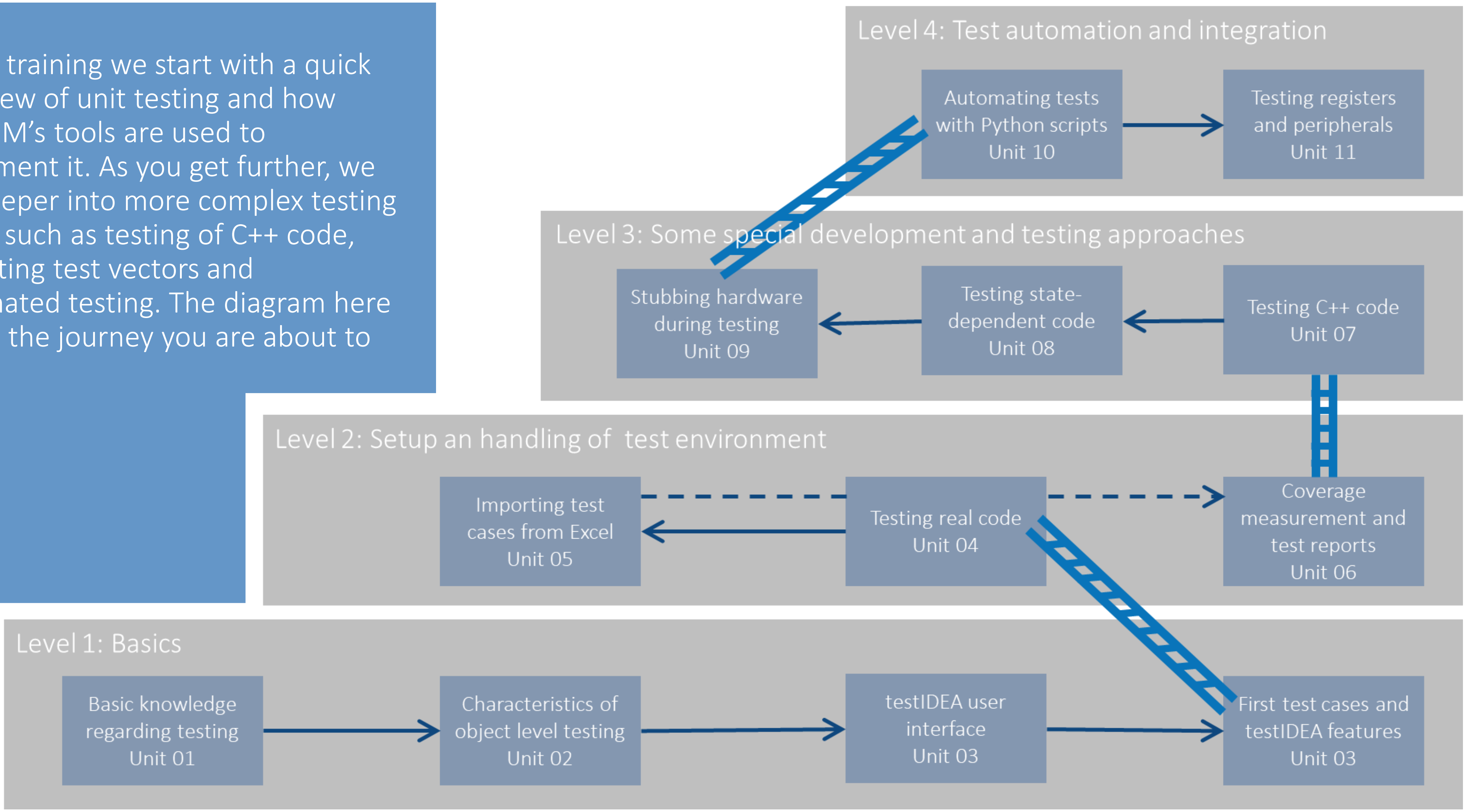
Contents

testIDEA TRAINING

testIDEA

testIDEA TRAINING - OVERVIEW

In this training we start with a quick overview of unit testing and how iSYSTEM's tools are used to implement it. As you get further, we get deeper into more complex testing topics such as testing of C++ code, importing test vectors and automated testing. The diagram here shows the journey you are about to take...



Level 4: Test automation and integration
Automating tests with Python scripts Unit 10
Testing registers and peripherals Unit 11
Level 3: Some special development and testing approaches
Stubbing hardware during testing Unit 09
Testing state-dependent code Unit 08
Testing C++ code Unit 07
Level 2: Setup an handling of test environment
Importing test cases from Excel Unit 05
Testing real code Unit 04
Coverage measurement and test reports Unit 06
Level 1: Basics
Basic knowledge regarding testing Unit 01
Characteristics of object level testing Unit 02
testIDEA user interface Unit 03
First test cases and testIDEA features Unit 03

Testing -

BASIC KNOWLEDGE

Objectives

At the end of this section, you will be able to

- Explain what unit testing is and how it differs from integration and system testing
- List two testing techniques for the generation of unit tests
- Explain how coding standards, design rules, documentation and abstraction facilitate testing

testIDEA

Contents

BASIC KNOWLEDGE

testIDEA

1 SOFTWARE TESTING - WHAT IS IT?

Increasingly, in all sectors of embedded development, we are becoming more reliant on software. Software-based systems, together with today's clever microcontroller peripherals, enable the implementation of applications that were previously considered too costly, complex or risky. However, the software associated with always-connected or uninterrupted usage, coupled with safety and quality demands, requires that our software is sufficiently tested if we want to avoid product recalls or, in the worst cases, life-threating injuries.

The shortened definition opposite (source: Wikipedia) sums up the core goals of software testing well.

Software testing is an **investigation** conducted to **provide stakeholders** with **information** about the **quality of the software** product. Software testing can also provide an **objective**, **independent** view of the software to allow the business to appreciate and understand the **risks of software implementation**.

Software testing involves the **execution** of a **software component** to evaluate the **extent to which** the component or system under test:

- **meets the requirements** that guided its design and development,
- **responds correctly** to all kinds of inputs,
- **performs its functions** within an acceptable time,
- is **sufficiently usable**, and
- **achieves the general result** its stakeholders desire.

1 SOFTWARE TESTING - PRINCIPLES

The testing and review of software are different from the analysis and development of it.

During software development we are working positively to solve the challenges posed during the development process to develop our product according to the user specification.

However, during testing or the review of our software, we are searching for the defects or failures in the software.

Thus, test development requires a different mindset from that required for the software's creation.

(ISTQB Exam Certification: http://istqbexamcertification.com/what-is-the-psychology-of-testing/)

1. Testing shows presence of defects
   Testing can show the defects are present, but cannot prove that there are no defects.
2. Exhaustive testing is impossible
   Testing everything including all combinations of inputs and preconditions is not possible. So, instead of undertaking exhaustive testing, we can use risks and priorities to focus our testing effort.
3. Early testing
   In the software development life cycle, testing activities should start as early as possible and should be focused on defined objectives.
4. Defect clustering
   A small number of modules contain most of the defects discovered during pre-release testing, or will show the most operational failures.

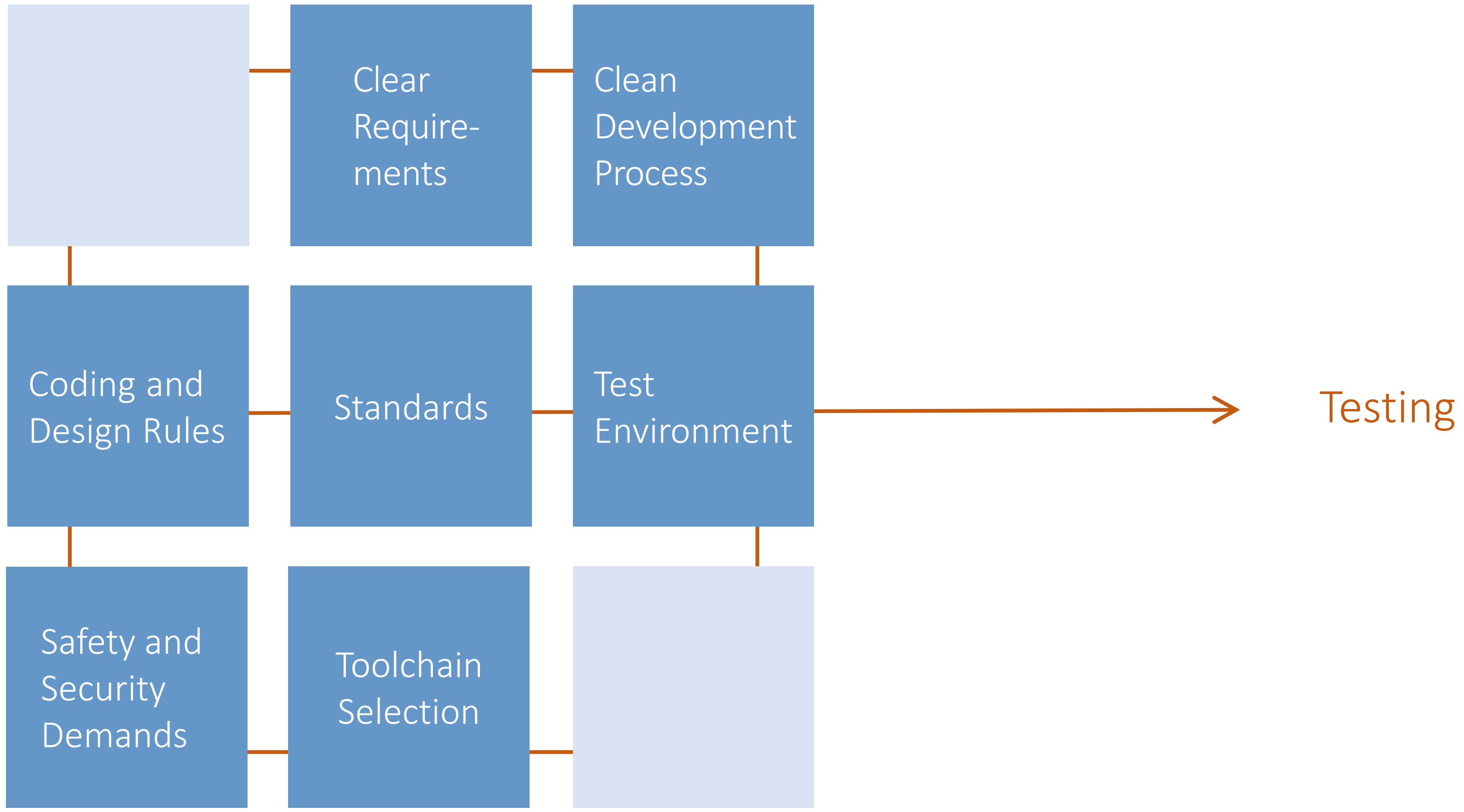
ISTQB Exam Certification:
http://istqbexamcertification.com/what-are-the-principles-of-testing/

1 SOFTWARE TESTING - INFLUENCING FACTORS

There are many factors that influence the software developed and how best to approach testing it. The diagram opposite highlights what could be considered to be the key factors that should always be included or considered to ensure that testing can be undertaken as easily and smoothly as possible.

Some elements will be demanded by the application itself (e.g. Safety Standards) whilst others simply make life easier when developing the testing strategy.

The following slides will look at these factors one-by-one and consider how they impact and simplify software testing.



1 SOFTWARE TESTING - REQUIREMENTS

Requirements help a development team to clarify what goal should be achieved with their software development and what will constitute “successful completion”. The requirements also form a critical element of the testing process as, without them, it is impossible to know if the software developed fulfils the demands made of it.

Regardless of the size of your organization, the importance of safety for your application, or even the size of the project, at a minimum a simple text document should declare what you intend to achieve.

It can be said that it is more important to document the requirements than concern yourself with which tool they are documented with.

There are many options for creating and managing requirements, including:

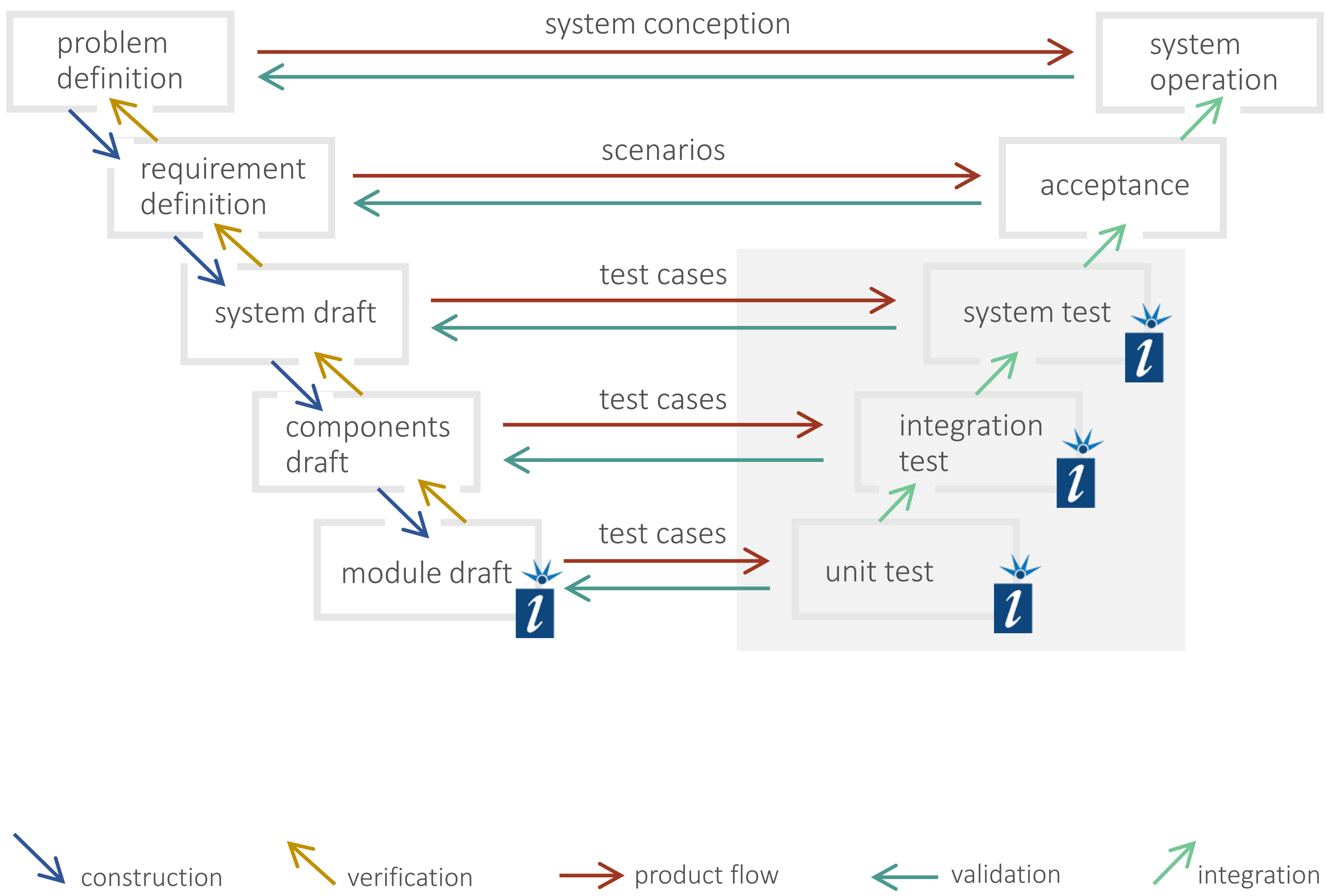
- Simple Text File
- Word or Excel Document
- Polarion (link)
- IBM Rational DOORS (link)
- Enterprise Architect (link)

1 SOFTWARE TESTING - POSITION IN THE DEVELOPMENT PROCESS

Testing fits in to the V-Model as shown opposite.

A **unit test** is the smallest testable part of an application, such as a function, a class's method. Unit testing is a method by which individual units of source code are tested to determine if they are fit for use.
ISTQB Exam Certification:
http://istqbexamcertification.com/what-is-unit-testing/

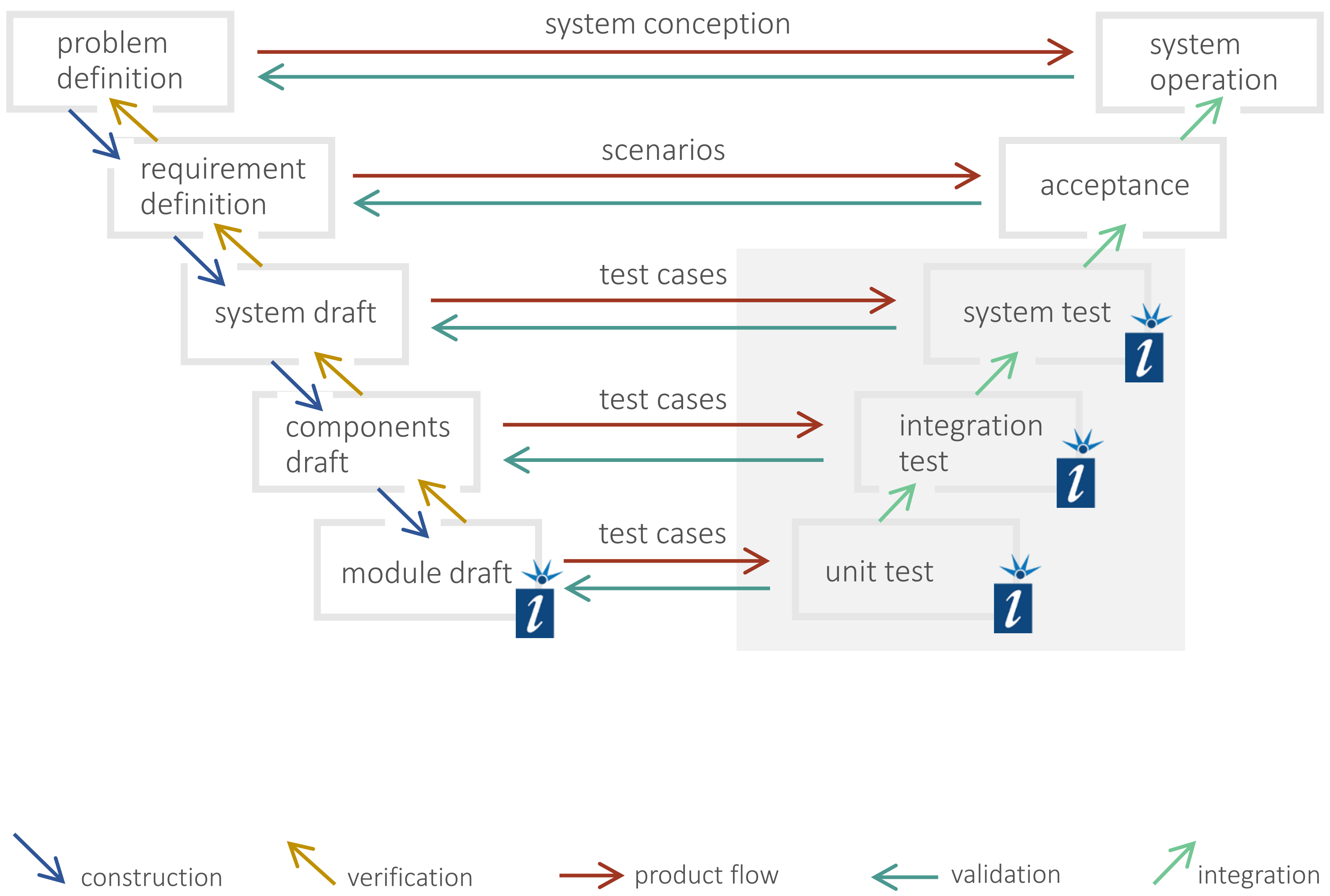
Integration testing tests the interfaces between components to prove the correct functionality of interaction between those components, e.g. between a file system and the hardware abstraction layers.
ISTQB Exam Certification:
http://istqbexamcertification.com/what-is-integration-testing/



1 SOFTWARE TESTING - POSITION IN THE DEVELOPMENT PROCESS

testIDEA is designed for the development and execution of unit tests. However, in the environment of embedded development, and the close overlap of these terms within the context of embedded development, testIDEA can often be used to implement some integration testing too.

For integration and system testing, iSYSTEM's other software tools are better placed. For example, we would recommend using the isystem.connect SDK together with our BlueBox™ hardware, potentially in conjunction with other 3rd party tools.

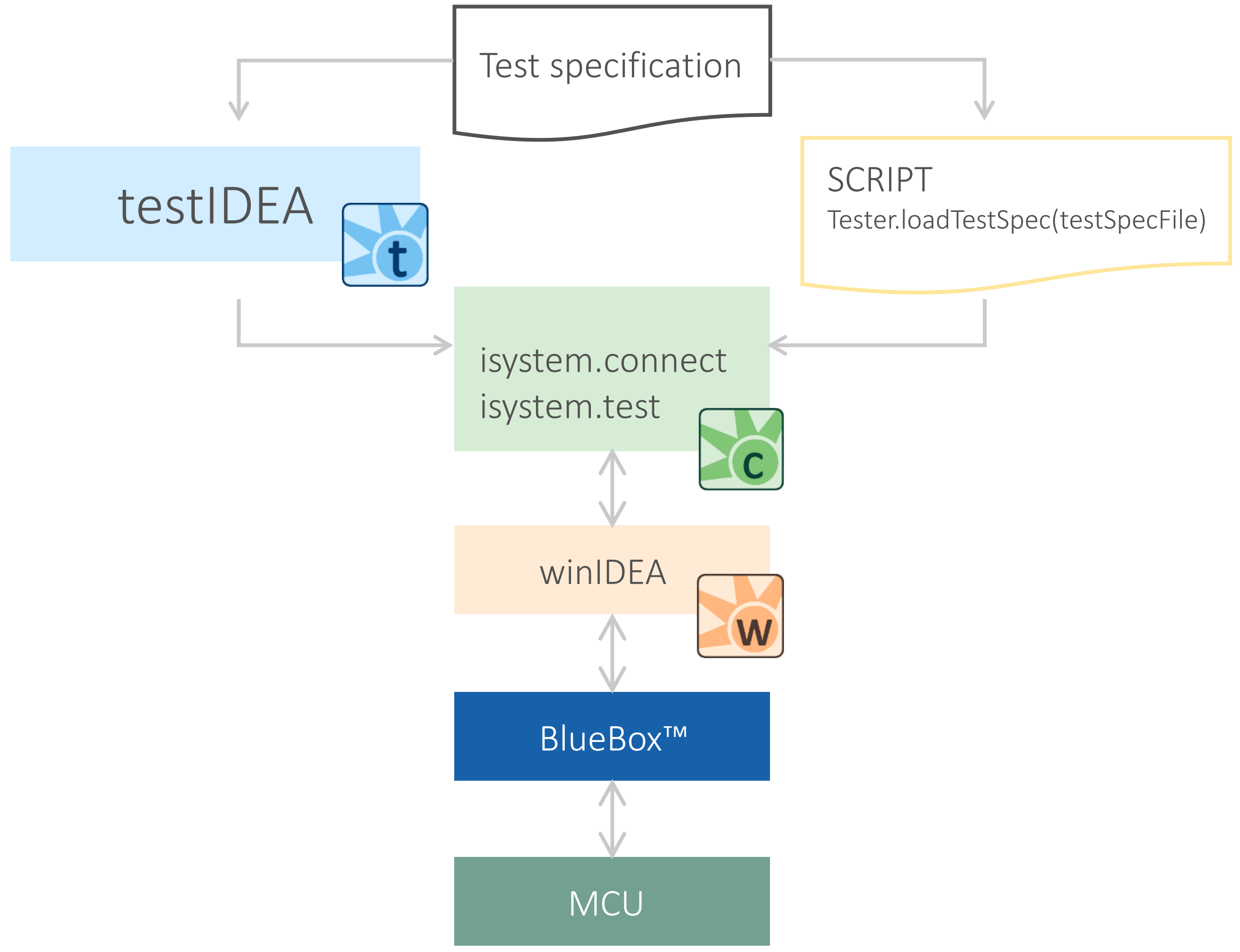


2 TESTING SOFTWARE WITH testIDEA

This training focuses on the use of the software tool **testIDEA** from iSYSTEM for the implementation and execution of tests on microcontroller-based embedded systems.

testIDEA leverages the microcontroller's debug interface, access to which is made available through winIDEA and the BlueBox™, to test software directly on the target MCU.

Despite enormous advances in technology, software testing still requires a significant amount of human endeavor from technically qualified personnel. This training will provide some background on how to approach the development of tests before showing how they can be implemented using testIDEA.



2 TESTING SOFTWARE WITH testIDEA

Prerequisites

The following software needs to be installed on your computer to be able to run and use iSYSTEM testIDEA:

- Java Runtime Environment (JRE) version 1.7 (automatically installed together with testIDEA)
- winIDEA version 9.12 or later. iSYSTEM's testIDEA is included when you install winIDEA. testIDEA can be used separately from winIDEA but there are quite some limitations without the link to the information winIDEA provides, e.g. the BlueBox™ and target in use and the symbols of your binary file's object code.

If you have installed a full winIDEA setup, testIDEA is already available on your system.

2 TESTING SOFTWARE WITH testIDEA

Download and install

If you need to update or install testIDEA separately from winIDEA, contact iSYSTEM support for the installation file. Then unzip the file to your hard disk. There is a **iSystem_testIDEA.exe** executable in the unzipped folder. Make a desktop shortcut for your convenience.

If you have installed a full winIDEA setup, testIDEA is already available on your system.

2 TESTING SOFTWARE WITH testIDEA

Professional and standard features of testIDEA

testIDEA has an advanced set of features that are available only in professional mode. This mode is activated if there is a testIDEA PRO license present in your BlueBox™ On-Chip Analyzer.

To activate these features, the BlueBox ™ must be connected to your computer and turned on. Then you should connect to winIDEA with command *iTools → Connect to winIDEA*. If you connect or turn on the emulator later, simply execute the command again and testIDEA will check the license.

Professional features:

- Report generation
- Wizard for test case generation
- Test templates
- I/O module & HIL support
- Code coverage
- Profiler/performance analysis
- Trace
- Python script generation
- Script execution before/after test
- Excel/CSV import/export
- Filtering of tests for execution
- Dry run
- Measurement of stack usage
- Function execution sequence verification
- Test points
- Renaming of functions and variables
- Native (real-time) stubs

Activities and functions that require a testIDEA PRO license will be marked with the logo shown on the right in these training slides.
Feel Fee free to contact iSYSTEM if you would like an evaluation license.

PRO

3 MAKING TESTING EASIER

Generally speaking, there are a few software development approaches that ensure that testing can be performed efficiently and successfully.

As previously stated, clear requirements and specifications are the starting point. This will help us to develop and test within a well-structured working process.

Furthermore, the use of coding standards, design rules and principles of abstraction are very important and help to ease testing later on.

Finally, documentation and commenting of code helps to ensure that critical information concerning coding decisions made doesn‘t get lost.

Some items that simplify testing process:

- Clear project requirements and specification
- Coding standards
- Development design rules
- Abstraction of *“functionality”*, “*algorithms*” and “*services*” from hardware
- Documentation

3 MAKING TESTING EASIER - CODING STANDARDS

Coding standards:

- defines how the software instructions are placed in the text file

Having a coding standard in place helps to ensure that the look and feel of code remains the same, regardless of who wrote the code. This makes it much easier to spot possible errors, since things like brackets are always in the same place, comments follow the same construction, and so on.

This also helps when developing unit tests, as the location of prototypes, and the way they are declared and documented, is easier to ascertain.

- Set of conventions
- Defines programming style
- Not limited to C/C++
- Simplifies code maintenance
- Invaluable in team environment
- Still relevant for individual development

Basic principles – taken from Micrium AN-2000

- Keep to the spirit of the standard
- Comply with ANSI C standards
- Keep the code simple
- Be explicit
- Be consistent
- Avoid complicated statements

3 MAKING TESTING EASIER - DESIGN RULES

Design rules:

- ensures software results in portable, re-entrant, safe and efficient code

It is slightly easier to explain what design rules are by showing how they differ from coding standards:

- If coding standards determine how the code is written and formatted on the page, it is design rules that define how the resulting code should function and interact with other software modules, RTOSs and functions.

This leads on to our next topic – Code Abstraction.

Design rules help to ensure code reusability, defining:

- Code re-entrancy
- How to use and support shared resources
  - Memory regions
  - Stack/Heap
  - Peripherals
- Keeping interrupt latency to a minimum
- Use of "*MCU specific*" features, such as:
  - Instructions for fast memory accesses
  - Cache and DMA usage
  - DSP or processor dependent instructions

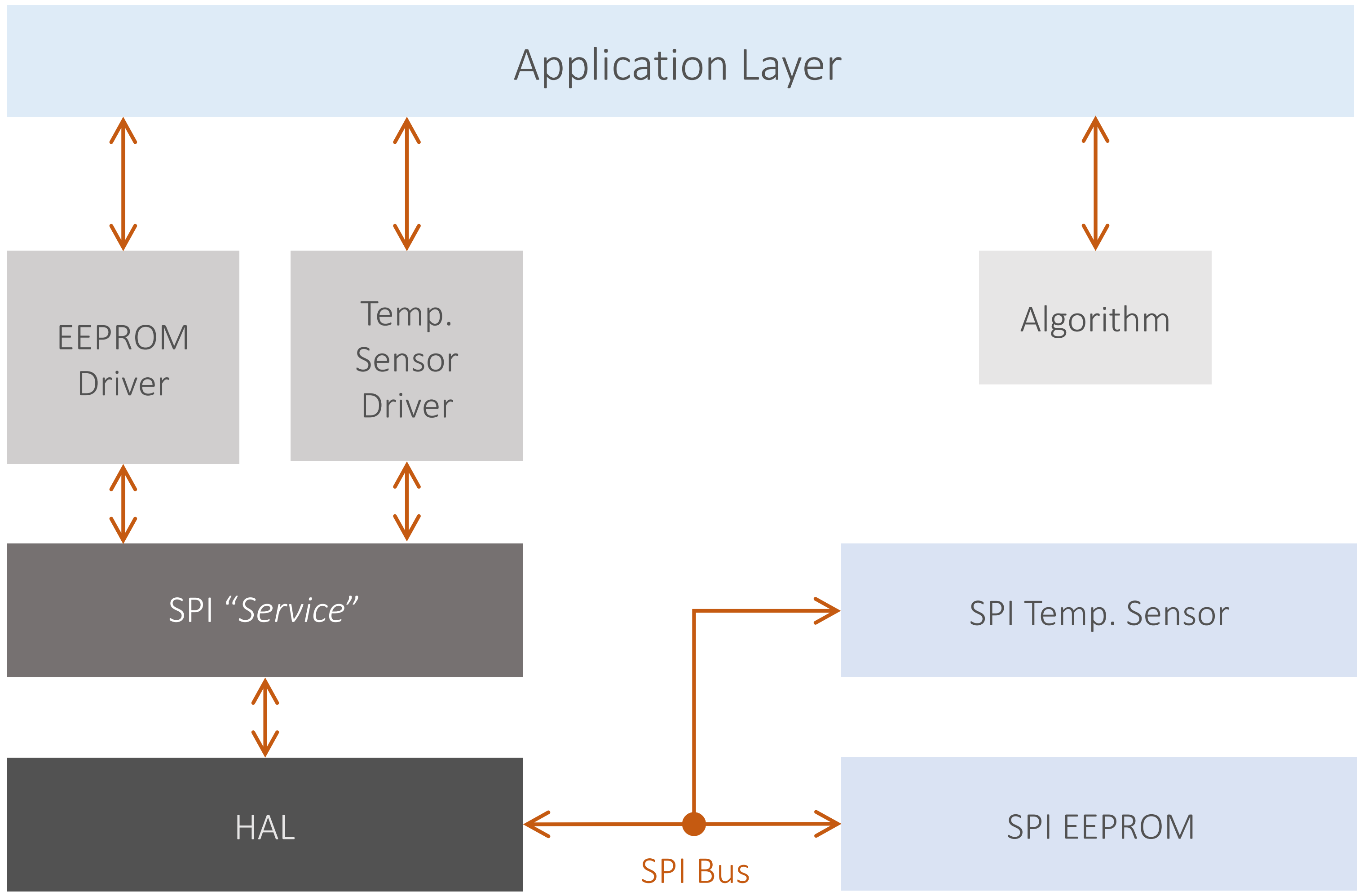
Texas Instruments provides a good example of design rules for developers wishing to create re-usable software modules for their DSP ecosystem (link)

3 MAKING TESTING EASIER - ABSTRACTION

Abstracting application code from software algorithms, services and hardware is also an important aspect of software development.

It enables reusability by making MCU specific capabilities and features irrelevant. In the example opposite, the EEPROM Driver and Temperature Sensor Driver both rely upon an “SPI Service” layer to use a single SPI interface to communicate with two different SPI-based off-chip peripherals.

If we were to change the MCU, all that should be required is the creation of a new “*HAL*” layer to make use the new MCU’s SPI interface. The remaining software modules should remain fully usable.

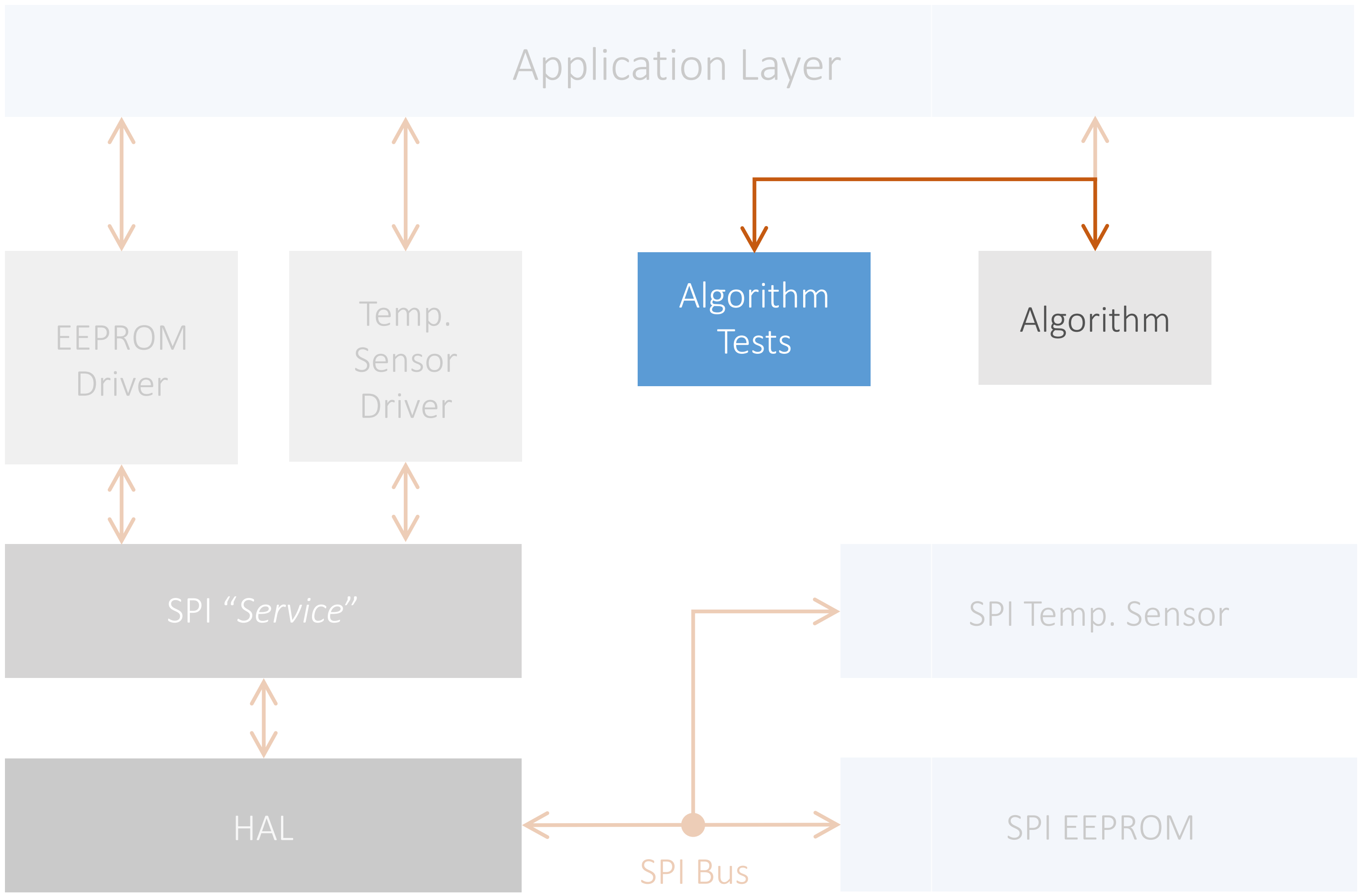


3 MAKING TESTING EASIER - ABSTRACTION

Further benefits arise later during testing. Each of the software modules included in the application can be tested in isolation of one-another.

Here the "Algorithm" module is being tested in isolation of the remainder of the application.

Assuming the "Algorithm" module passes all our tests but our full application delivers incorrect results, the error is very unlikely to be related to this software module. It is more likely that the "Algorithm" module is being fed with incorrect data from elsewhere in the application.



3 MAKING TESTING EASIER - DOCUMENTATION

Documentation is also an area where many of us can (and should) improve. There is no hard-and-fast rule regarding the expected code/comment ratio. However, the table on the right provides some advice on best practice in the industry.

Note that many open-source software projects, that rely upon resources across the world to work together and communicate well, deliver projects that are in the “good” to “excellent” range.

The opinion in the table opposite comes from this source.

Code/Comment ratio	
< 5 %	Horribly commented
> 5 %	Poorly commented
> 10 %	Average
> 15 %	good
> 25 %	excellent

4 STANDARDS - HOW MUCH TESTING IS DEMANDED?

Some industry sectors have stringent requirements for safety and have clear standards in place that must be fulfilled. The automotive industry uses ISO 26262 and acceptable approaches to proving software quality are made very clear, defined according to the stage in the V-Model that is currently being undertaken.

Such documents provide guidance on how to assess risk, approaches for requirements analysis and the documentation that needs to be delivered to prove testing was undertaken.

These standards are also helpful to answer questions of “how much testing is enough” and “which approach to take” for those simply trying to improve their software quality.

ISO 26262 Structural Coverage Metrics at Software Unit Level(*)

Method	ASIL			
	A	B	C	D
Statement Coverage	✓✓	✓✓	✓	✓
Branch Coverage	✓	✓✓	✓✓	✓✓
MC/DC (Modified Condition/Decision Coverage)	✓	✓	✓	✓✓

Chapter 9 of IS0 26262 offers suitable methods for software unit testing, method for deriving unit testing test case and, as shown here, coverage requirements for those tests that depend on the Automotive Safety Integrity Level that came out of the risk assessment.
A single tick is a “recommended” method, a double tick is “highly recommended”.

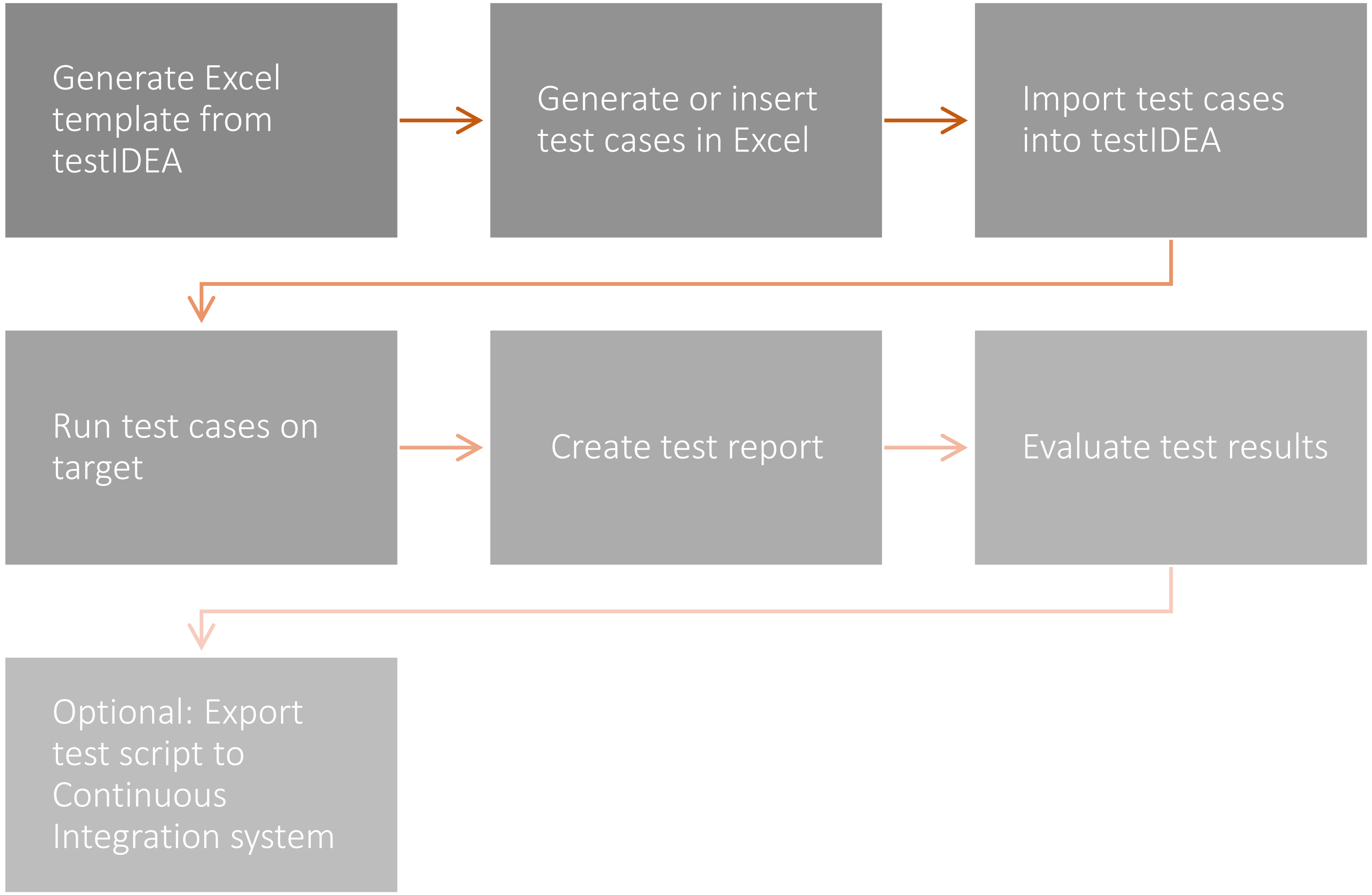
(*) Taken from ISO 26262-6, Table 12

5 EXAMPLE UNIT TEST DEVELOPMENT PROCESS

Opposite is just one simple example of how a testing process for the unit testing of a software module can be implemented.

This process provides a method that can handle a great number of test cases. It refers to an external tool (in this case, Microsoft Excel) and uses features of the chosen tool (automated test vector input and result calculation).

It is also possible (as will be shown in Unit 03) to create a number of tests solely within testIDEA.

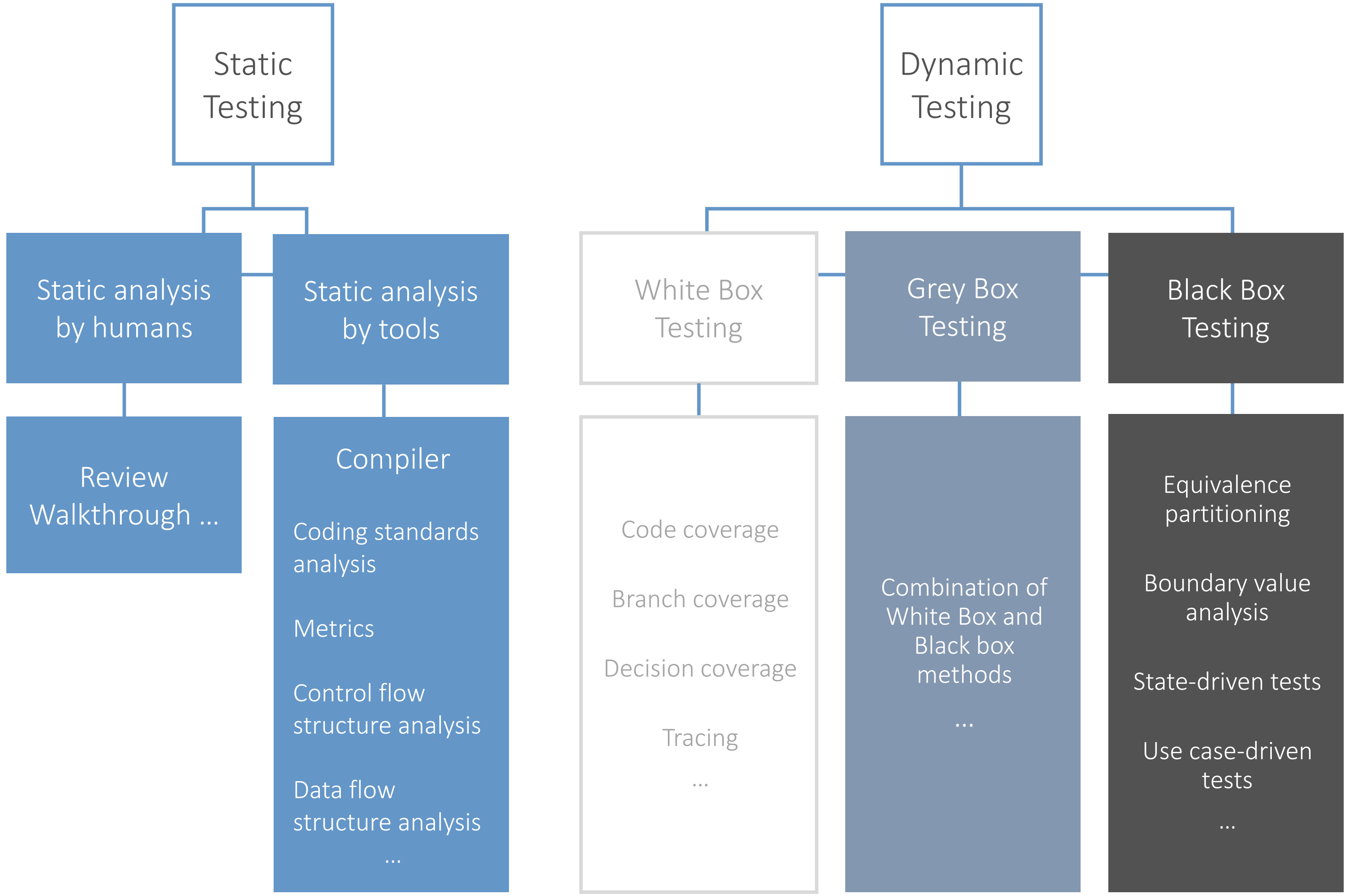

Generate Excel template from testIDEA
Generate or insert test cases in Excel
Import test cases into testIDEA
Run test cases on target
Create test report
Evaluate test results
Optional: Export test script to Continuous Integration system

6 METHODS OF TESTING

Software testing, and the tools that are on offer, can be broadly categorized for use in Static or Dynamic software testing.

Static Testing approaches analyze the source code without consideration for other factors, such as the microcontroller being used, interaction of hardware with one another, or the tool chain being used for compilation.

Since embedded systems depend on the interaction of hardware with software, Dynamic Testing approaches are typically of more use. These range from simply testing the functionality of the completed software-based product (Black Box) through to testing the source code on the microcontroller, often together with other hardware (White Box).



6 METHODS OF TESTING - WHITE BOX - GREY BOX - BLACK BOX

The differences between these testing methods depend largely on how much access the tester has to the source or binary code used in the application.

If a tester has full access to the contents of the code and the architecture of the system, they will likely undertake white box testing.

If there is limited or no access to the source code, they will have consider testing the application as a black box.

These differences also require a different mindset to approach the testing of the application.

Techniques for the generation of “just enough” tests are covered in the next slides.

White Box

White Box = Structure-based testing
→ Tester requires knowledge of how the software is implemented, how it works along with access to the source code

Grey Box

Grey Box = Structure- and Specification-based testing
→ Tester has as little knowledge of the code implementation as possible, and as much insight as required

Black Box Testing

Black Box = Specification-based testing
→ Tester likely has no knowledge of how the system or its components are structured within the product

6 METHODS OF TESTING - EQUIVALENCE PARTITIONING

Equivalence partitioning (EP) is a specification-based or black-box testing technique.

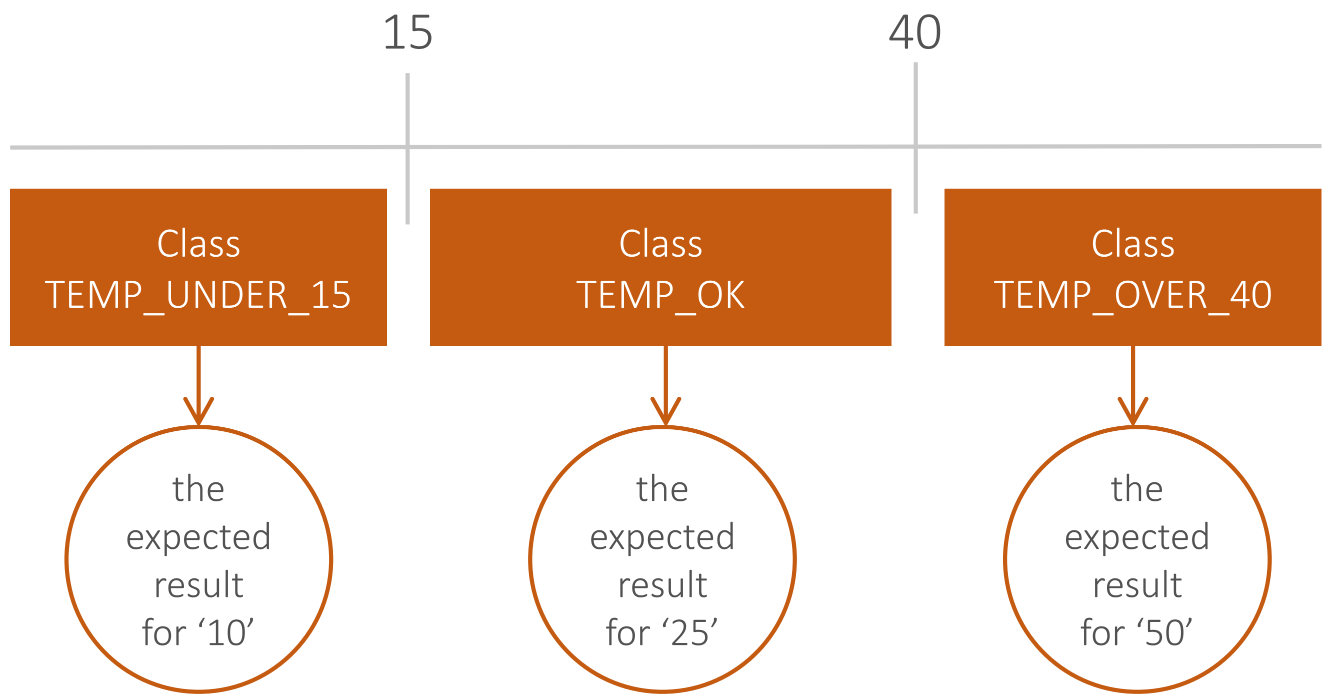
It can be applied at any level of testing and is often a good technique to use first.

The idea behind this technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence "equivalence partitioning".

Equivalence partitions are also known as equivalence classes - the two terms mean exactly the same thing.

ISTQB Exam Certification
http://istqbexamcertification.com/what-is-equivalence-partitioning-in-software-testing/

```
if (temperature < 15) {
    returnValue = TEMP_UNDER_15;
} else if (temperature <= 40) {
    returnValue = TEMP_OK;
} else {
    returnValue = TEMP_OVER_40;
}
return returnValue;
```


15
40
Class TEMP_UNDER_15
Class TEMP_OK
Class TEMP_OVER_40
the expected result for '10'
the expected result for '25'
the expected result for '50'

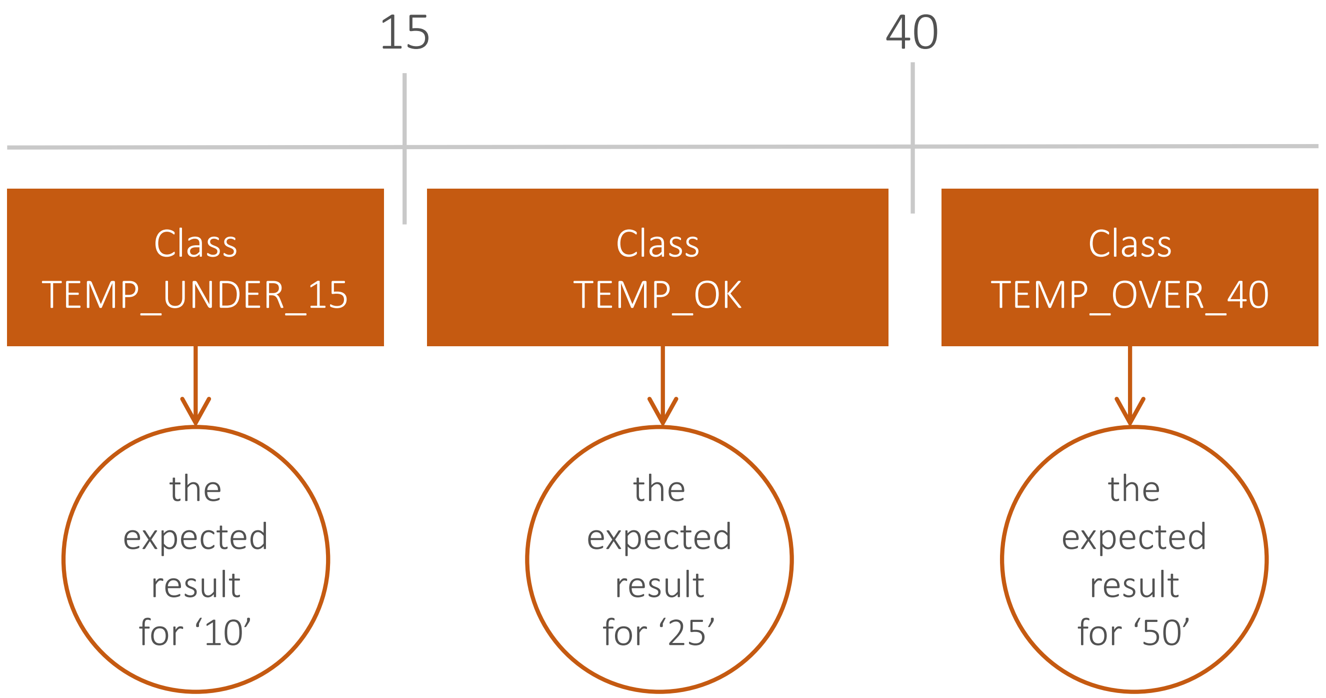
6 METHODS OF TESTING - EQUIVALENCE PARTITIONING

With the equivalence-partitioning technique we only need to test one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, it is assumed all of the conditions in that partition will work, and so there is little point in testing the other conditions.

Similarly, if one of the conditions tested in a partition were to fail, then we would assume that none of the conditions in that partition will work. Thus there would be no point in testing any more within that partition.

ISTQB Exam Certification
http://istqbexamcertification.com/what-is-equivalence-partitioning-in-software-testing/

```
if (temperature < 15) {
    returnValue = TEMP_UNDER_15;
} else if (temperature <= 40) {
    returnValue = TEMP_OK;
} else {
    returnValue = TEMP_OVER_40;
}
return returnValue;
```


15
40
Class TEMP_UNDER_15
Class TEMP_OK
Class TEMP_OVER_40
the expected result for '10'
the expected result for '25'
the expected result for '50'

6 METHODS OF TESTING - BOUNDARY VALUE ANALYSIS

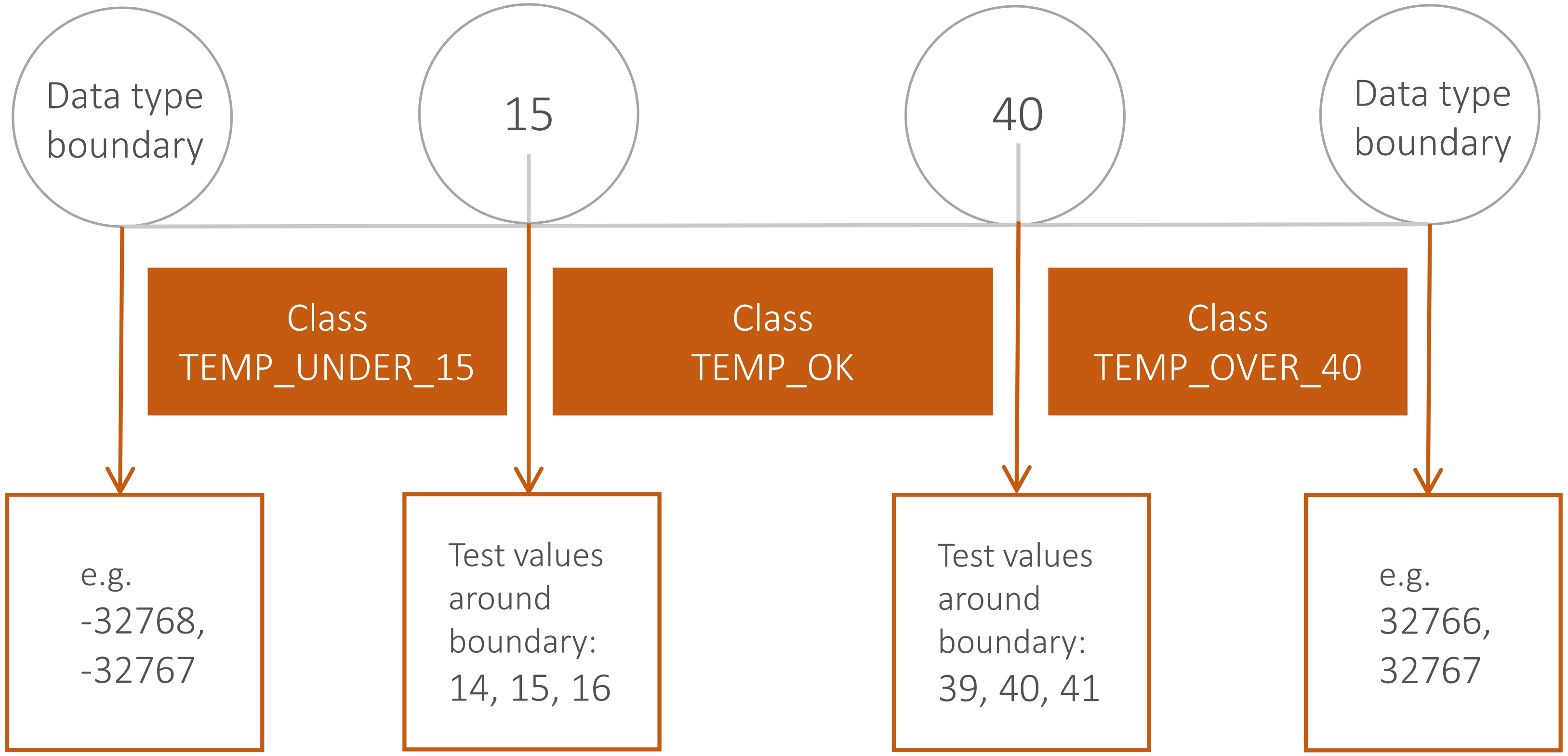
Boundary value analysis (BVA) is based on testing at the boundaries between partitions. This can be seen here where test vectors are generated around the boundaries to ensure the `if...else` logic functions as designed.

Programming languages, such as C, create further boundaries with their datatypes. Thus it may also be considered prudent to add these boundaries to the list of test vectors, especially for reusable embedded software modules.

ISTQB Exam Certification
http://istqbexamcertification.com/what-is-boundary-value-analysis-in-software-testing/

```
if (temperature < 15) {
    returnValue = TEMP_UNDER_15;
} else if (temperature <= 40) {
    returnValue = TEMP_OK;
...
```

```
...
} else {
    returnValue = TEMP_OVER_40;
}
return returnValue;
```



SUMMARY

testIDEA

7 SUMMARY

- testIDEA can be used for unit and, in some cases, integration testing as part of a Dynamic Testing approach
- testIDEA can be used to create and manage software tests, executing them on the target hardware
- The development of tests is easier when the software is clearly specified, well documented and suitably abstracted
- The tests themselves must still be considered, defined and implemented. Equivalence Partitioning and Boundary Value Analysis are examples of test creation techniques.