Embedded Systems Development - 7. Constructors, destructors, overriding and unit tests.

Electrical Engineering / Embedded Systems
School of Engineering and Automotive

Johan.Korten@han.nl

HAN_UNIVERSITY
OF APPLIED SCIENCES

Schedule (exact info see #00 and roster at insite.han.nl)

		C++	UML
Step 1	Single responsibility	Scope, namespaces, string	Use cases / Class diagram
Step 2	Open-Closed Principle	Constructors, iterators, lambdas	Inheritance / Generalization
Step 3	Liskov Substitution Principle	lists, inline functions, default params	Activities
Step 4	Interface Segregation Principle	interfaces and abstract classes	Depencencies
Step 5	Dependency Inversion Principle	threads, callback	Sequence diagrams
Step 6	Coupling and cohesion	polymorphism	Composition, packages
Step 7	**Embedded SOLID**	**constructors, destructors, const**	

Next period

Work on your project(s)

Peer reviewing of code

Feedback on diagrams

Feedback on Code

Please fill in the Excel I shared…

Please use https://github.com/HANEmbedded

Exam preparation (deprecated), use this for final project…

On top of the topics from the previous slide (some items might be on both slides):

C++
- **constructor and destructor (RAII)**
- access (visibility: private / public / protected)
- inheritance: base class, derived class, virtual functions, **override**
- class / struct
- abstract / interface, Abstract Base Class, abstract member function = 0
- getter / setter
- **const-correct**
- composite and aggregate
- range based for-loop (foreach) for container classes
- std:: name space, std::vector

UML and concepts
- class diagram, composite and aggregate, inheritance
- stereotype / object / actor
- abstract class / interface
- package diagram
- sequence diagram
- polymorphism, polymorphic arrays and vectors
- use cases
- state diagrams, events and states (prior knowledge)

Note: topics from previous C/C++ courses are considered as existing prior knowledge.

Note: the five *SOLID* principles will not be explicitly asked during the exam but can be helpful during the exam and are considered standard practices for good software engineering.

C++ Language: Lifecycle / lifetime of objects

Typically a variable or object exists as long as it is "in scope"

```cpp
#include <iostream>

class IamAlife {

public:
    IamAlife() {
        std::cout << "I was created ex nihilo!" << std::endl;
    }

    ~IamAlife() {
        std::cout << "My lifecycle ended: IamAlife is no more :(" << std::endl;
    }

    void shoutAloud() {
        std::cout << "Whoohoo, IamAlife!" << std::endl;
    }
};

int main() {
    std::cout << "There we go..." << std::endl;
    IamAlife toBeOrNotToBe;
    toBeOrNotToBe.shoutAloud();
    std::cout << "Main almost out of scope..." << std::endl;
}
```

C++ Language: Lifecycle / lifetime of objects

So default lifecycle:

- object is created in a certain scope…
- see the example if you don’t or want to check if you understand scope

Note: lifetime is a *runtime* property of objects.

C++ Language: Lifecycle / lifetime of objects

So default lifecycle:

- object is created in a certain scope…
- see the example if you don’t or want to check if you understand scope

But what if you create an object using new…

C++ Language: Lifecycle / lifetime of objects

Typically a variable or object exists as long as it is "in scope": <u>in case you use new this is not handled automatically anymore…</u>

```cpp
#include <iostream>
#include <vector>

class IamAlife {

public:
    IamAlife() {
        std::cout << "I was created ex nihilo!" << std::endl;
    }

    ~IamAlife() {
        std::cout << "My lifecycle ended: IamAlife is no more :(" << std::endl;
    }

    void shoutAloud() {
        std::cout << "Whoohoo, IamAlife!" << std::endl;
    }
};

std::vector<IamAlife *> _hamlets;

int main() {
    std::cout << "There we go..." << std::endl;

    IamAlife *toBeOrNotToBe = new IamAlife(); // requires delete!!!
    toBeOrNotToBe->shoutAloud();

    std::cout << std::endl;
    std::cout << "Main goes out of scope after this..." << std::endl;
    std::cout << "Whoops IamAlife is becoming a zombie..." << std::endl;
    delete toBeOrNotToBe; // phew... just in time...
}
```

C++ Language: constructor and destructor (RAII)

Resource Acquisition Is Initialization (RAII):

- binds the *life cycle of a resource* to the *lifetime of an object*

- that must be acquired before use

- use cases: e.g. allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection

This helps to prevent resource leaks and makes code exception-safe.

Three Principles of RAII

- Encapsulate each resource into a class
- Instance ownership
- No manual release

Three Principles of RAII

- Encapsulate each resource into a class

The constructor of the class acquires the resource, and the destructor releases it.

- Instance ownership
- No manual release

Three Principles of RAII

- Encapsulate each resource into a class
- Instance ownership

Objects are usually instantiated on the stack or through smart pointers, ensuring deterministic destruction.

- No manual release

Three Principles of RAII

- Encapsulate each resource into a class
- Instance ownership
- No manual release

Avoid explicitly releasing resources.

Instead, rely on the destructor to do it automatically when the object goes out of scope.

Three Principles of RAII

- Encapsulate each resource into a class
- Instance ownership
- No manual release

Benefits:

- Memory Management: Helps avoid memory leaks.
- Exception Safety: Ensures that resources are properly cleaned up if an exception is thrown.
- Simpler Code: Reduces the need for manual resource management.

C++ Language: Lifecycle / lifetime of objects

Strategies to avoid memory leaks:

- *garbage collection (e.g. Java)* see: https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
- *reference counting (C#, Swift, etc)*

In C++ one option is to use *smart pointers*…

C++ Language: constructor and destructor (RAII)

C++ does not perform reference-counting by default, fulfilling its philosophy of not adding functionality that might incur overheads where the user has not explicitly requested it. Objects that are shared but not owned can be accessed via a reference, raw pointer, or iterator (a conceptual generalisation of pointers).

However, by the same token, C++ provides native ways for users to opt-into such functionality: C++11 provides reference counted smart pointers, via the `std::shared_ptr` class, enabling automatic shared memory-management of dynamically allocated objects.

Programmers can use this in conjunction with weak pointers (via `std::weak_ptr`) to break cyclic dependencies. Objects that are dynamically allocated but not intended to be shared can have their lifetime automatically managed using a `std::unique_ptr`.

In addition, C++11's move semantics further reduce the extent to which reference counts need to be modified by removing the deep copy normally used when a function returns an object, as it allows for a simple copy of the pointer of said object.

https://en.wikipedia.org/wiki/Reference_counting

C++ Language: Lifecycle / lifetime of objects

Using smart pointers: https://www.youtube.com/watch?v=UOB7-B2MfwA

```cpp
#include <iostream>
#include <vector>
#include <memory>

class IamAlife {

public:
    IamAlife() {
        std::cout << "I was created ex nihilo!" << std::endl;
    }

    ~IamAlife() {
        std::cout << "My lifecycle ended: IamAlife is no more :(" << std::endl;
    }

    void shoutAloud() {
        std::cout << "Whoohoo, IamAlife!" << std::endl;
    }
};

std::vector<IamAlife *> _hamlets;

int main() {
    std::cout << "There we go..." << std::endl;
    std::cout << std::endl;

    { // local sub scope
        std::unique_ptr<IamAlife> toBeOrNotToBe(new IamAlife());
        toBeOrNotToBe->shoutAloud();
    }

    std::cout << std::endl;
    std::cout << "Main goes out of scope after this..." << std::endl;
}
```

C++ Language: Lifecycle / lifetime of objects

Using smart pointers: https://www.youtube.com/watch?v=UOB7-B2MfwA

```cpp
// A. unique pointer:
std::unique_ptr<IamAlife> toBeOrNotToBe = std::make_unique<IamAlife>();
toBeOrNotToBe->shoutAloud();

// is preferred over this B.
std::unique_ptr<IamAlife> toBeOrNotToBe(new IamAlife());
toBeOrNotToBe->shoutAloud();
```

Reason to prefer option A over B is for exception purposes (A is safer / more robust agains exceptions).

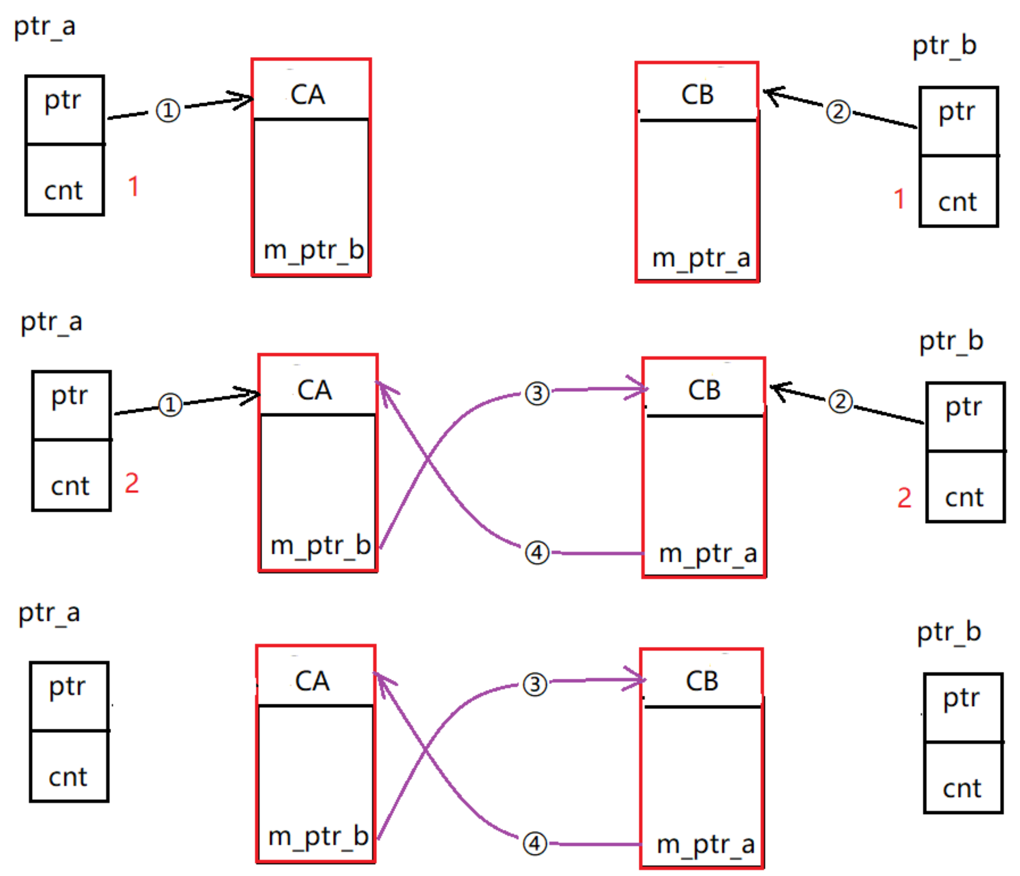
C++ Language: Lifecycle / lifetime of objects

Using smart pointers: https://www.youtube.com/watch?v=UOB7-B2MfwA

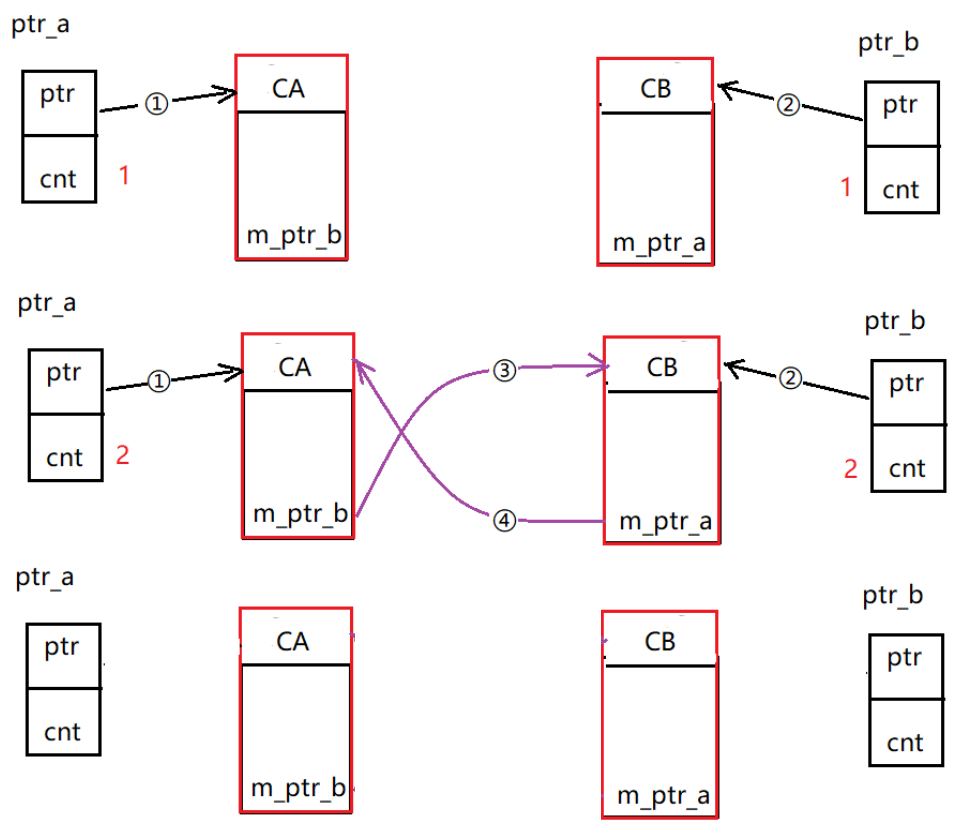
```cpp
// Shared pointer:
std::unique_ptr<IamAlife> toBeOrNotToBe = std::make_shared<IamAlife>();
toBeOrNotToBe->shoutAloud();

// Weak pointer:
std::weak_ptr<IamAlife> weakPointer = toBeOrNotToBe;
weakPointer->shoutAloud();
```

Weak helps reference counting to work properly and avoid memory leaks:

from:



to:

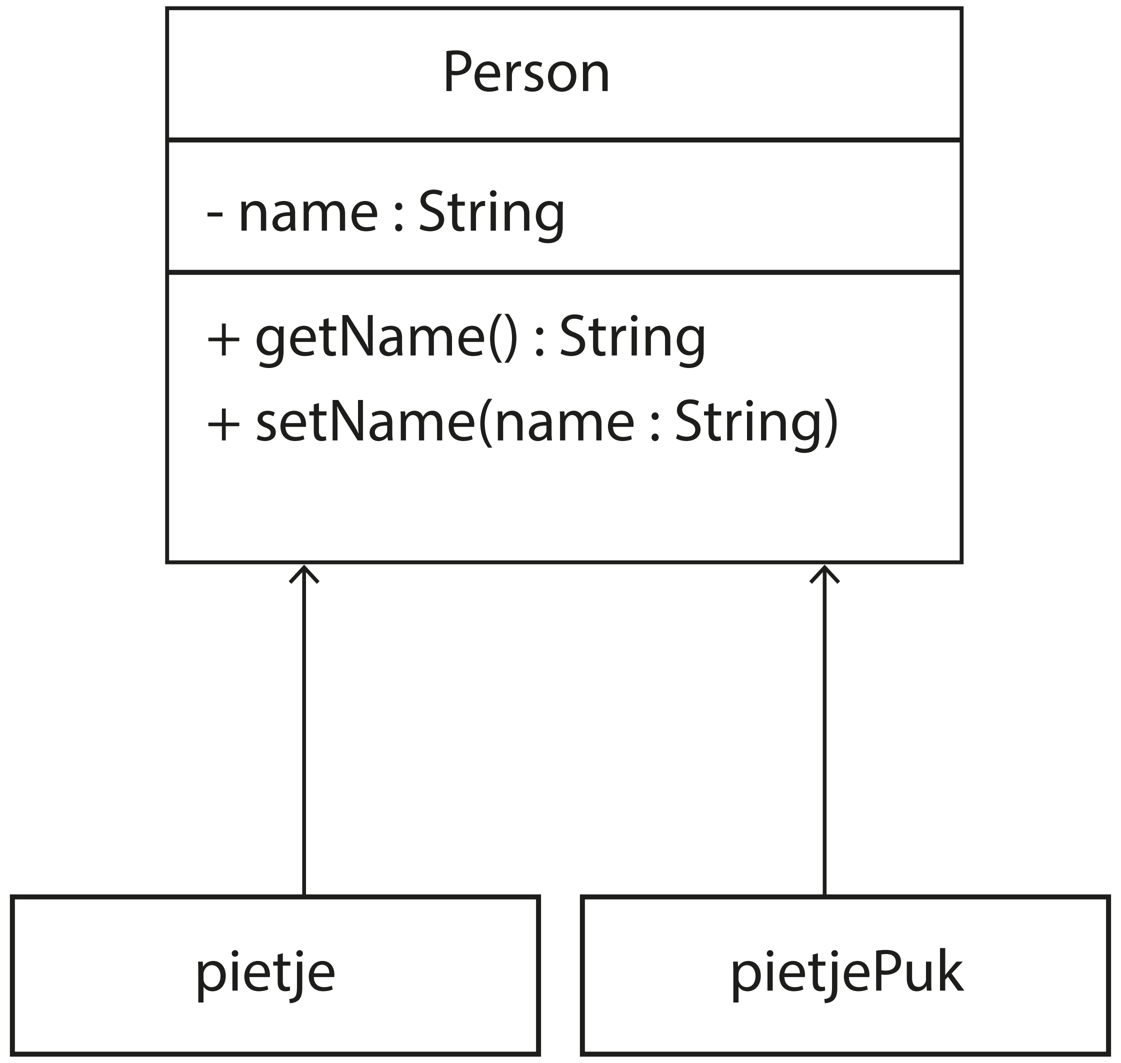


https://www.programmersought.com/article/7943189022/

C++ Language: Lifecycle / lifetime of objects

Note: the following is of course very powerful (useful cases would include doubly linked lists).

This will not work when using unique pointers!




```
Person pietje = new Person("Pietje");
Person pietjePuk = pietje;
```

Let's have a look at the example code…

simpleLifecycle.cpp

smartPointersLifecycle.cpp

moreComplexLifecycle.cpp

betterSmartPointersLifecycle.cpp

C++ Language: const-correctness (simple variables)

Const-correctness means *using the keyword const* to prevent const objects from getting mutated (in other words: making them *immutable*).

- the const keyword in front of a property
- const int counter = 10;
- remember: use a constructor to write only once to a const class attribute

C++ Language: const-correctness (arguments for methods)

Const-correctness means *using the keyword const* to prevent const objects from getting mutated.

Arguments (aka parameters) in methods:

```
- void f1(const std::string& s);      // Pass by reference-to-const
- void f2(const std::string* sptr);   // Pass by pointer-to-const
- void f3(std::string s);             // Pass by value
```

Recommended reading: https://isocpp.org/wiki/faq/const-correctness

C++ Language: const-correctness (const methods)

Const-correctness means *using the keyword const* to prevent const objects from getting mutated.

E.g. if you want to make a calculation using some function (= a method intended to give back a result):

```
int Loan::calcInterest() const
{
        return loan_value * interest_rate;
}
```

C++ Language: override

Override will allow to use the same method (including arguments and return) and override it's behavior in the same and/or subclasses.

```
struct A
{
    virtual void foo();
    void bar();
};

struct B : A
{
    void foo() override; // OK: B::foo overrides A::foo
};
```

C++ Language: override / shadowing

Override will allow to use the same method (including arguments and return) and override it's behavior in the same and/or subclasses.

```cpp
struct Base
{
    virtual void foo();
    void bar();
    Base(); // constructor of Base
};

struct Derived : Base
{
    using Base::foo; // now foo method(s) from Base will become visible in Derived
    using Base::Base; // now we can use the constructor from Base in Derived
};
```

C++ Language: warning: shadowing/masking

You could get into trouble if you are not careful with overloading (inadvertently typecasting):

```cpp
class Base {
 public:
  int F(int i) { return i; };
};

class Derived : public Base {
 public:
  //using Base::F;
  double F(double d) { return d; };
};

int main() {
    // Note if you forget using using Base::F, this will still work:
    Derived d = Derived();
    std::cout << std::to_string(d.F(10)) << std::endl;
}
```

First output with using Base::F; enabled:

Second output with code as shown above:

```
jakorten@mbp-van-ja-2221 Constructors % ./constructors
10
jakorten@mbp-van-ja-2221 Constructors % c++ constructors.cpp -o constructors
jakorten@mbp-van-ja-2221 Constructors % ./constructors
10.000000
jakorten@mbp-van-ja-2221 Constructors %
```

C++ Language: override

Making override explicit by adding *override (C++11 and up)* will allow the compiler to check the base class to see if there is a virtual function with this exact signature. And if there is not, the compiler will show an error.

```cpp
struct A
{
    virtual void foo();
    void bar();
};

struct B : A
{
    void foo() const override; // Error: B::foo does not override A::foo
                               // (signature mismatch)
    void foo() override; // OK: B::foo overrides A::foo
    void bar() override; // Error: A::bar is not virtual
};
```

SOLID in an Embedded Context

SOLID principles in embedded systems may have (additional) challenges.

However it can significantly enhance the *design*, *scalability*, and *maintainability* of embedded software, leading to more **robust** and **reliable** embedded systems.

SOLID in an Embedded Context

SOLID principles in embedded systems may have (additional) challenges.

However it can significantly enhance the *design*, *scalability*, and *maintainability* of embedded software, leading to more **robust** and **reliable** embedded systems.

SOLID in an Embedded Context: Single Responsibility Principle (SRP)

Embedded Context:

Modularize your code:

- Write functions and modules with a single responsibility. For example, separate the code that handles hardware interfacing from the business logic.

Use clear and concise naming conventions:

- This makes it easy to understand what each module or function is responsible for, enhancing maintainability.

SOLID in an Embedded Context: Open/Closed Principle (OCP)

Embedded Context:

Leverage polymorphism for hardware abstraction:

- You can define a base class (or interface) for a hardware device (HAL), and then extend it for specific device implementations.
- This way, your code remains open for extension but closed for modification.

Use function pointers or callbacks:

- This allows you to change behavior without modifying existing code.

(e.g. use patterns including Depency Injection)

SOLID in an Embedded Context: Liskov Substitution Principle (LSP)

Embedded Context:

Ensure compatibility of derived classes:

- If using object-oriented programming, make sure that derived classes can be used in place of their base classes without causing unexpected behavior.

Carefully manage hardware dependencies:

- Ensure that derived classes do not introduce hardware dependencies that could break the substitutability principle.

SOLID in an Embedded Context: Interface Segregation Principle (ISP)

Embedded Context:

Provide device-specific interfaces:

- For hardware drivers and peripherals, offer interfaces tailored to the specific functionalities each component uses, rather than a monolithic interface.

- Minimize dependencies: Ensure that each module or component only includes the minimal interface it needs, reducing the footprint and complexity.

SOLI**D** in an Embedded Context: Dependency Inversion Principle (DIP)

Embedded Context:

Use abstract interfaces for hardware components:

- This allows high-level modules to depend on abstractions rather than concrete implementations, facilitating testing and system modifications.

Employ dependency injection:

- Pass dependencies (e.g., hardware interfaces) as parameters to functions or objects, allowing you to easily substitute different implementations for testing or system configuration.

(e.g. use patterns including Depency Injection)

SOLID in an Embedded Context: Considerations

- Memory Constraints
- Performance Constraints
- Testing and Simulation
- Documentation and Code Comments

SOLID in an Embedded Context: Considerations

- **Memory Constraints**

  Be mindful of the memory overhead introduced by abstractions and object-oriented features. Optimize data structures and minimize dynamic memory allocation.

- Performance Constraints
- Testing and Simulation
- Documentation and Code Comments

SOLID in an Embedded Context: Considerations

- Memory Constraints

- Performance Constraints

Evaluate the performance impact of abstractions and interface layers, ensuring that the system meets its real-time requirements.

- Testing and Simulation

- Documentation and Code Comments

SOLID in an Embedded Context: Considerations

- Memory Constraints

- Performance Constraints

- Testing and Simulation

Use the abstraction layers to facilitate unit testing and hardware-in-the-loop simulation by substituting real hardware interfaces with mock or simulated implementations.

- Documentation and Code Comments

SOLID in an Embedded Context: Considerations

- Memory Constraints
- Performance Constraints
- Testing and Simulation
- **Documentation and Code Comments**

  Maintain thorough documentation and inline comments to help developers understand the design principles and interfaces, contributing to easier maintenance and enhancements.

  (May I suggest you to use Doxygen?!)

Introduction to testing.

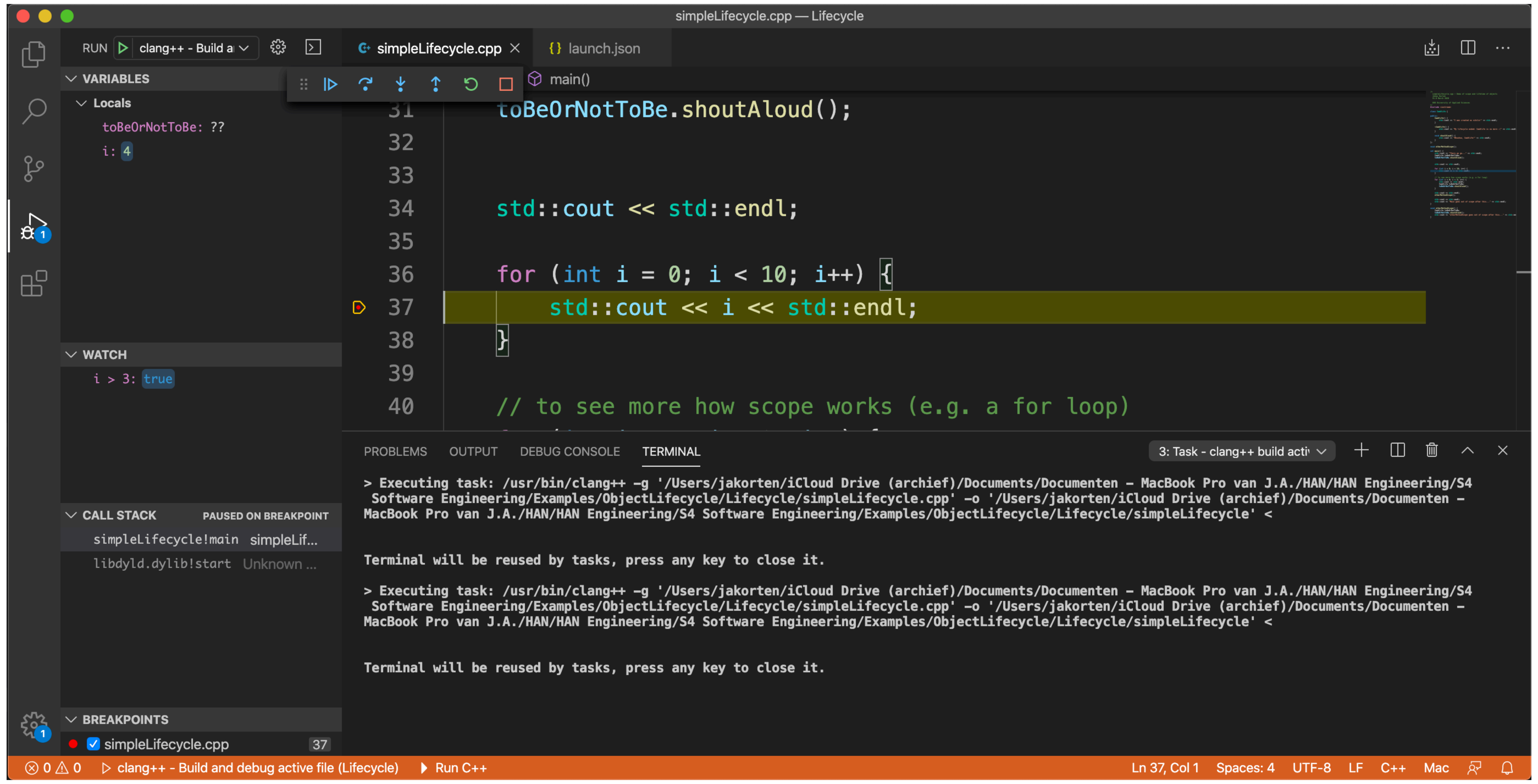
First of all: What is the difference between debugging and testing?

Introduction to testing in VS Code

Debugger in VS Code:

- https://code.visualstudio.com/docs/cpp/launch-json-reference
- https://code.visualstudio.com/docs/editor/debugging

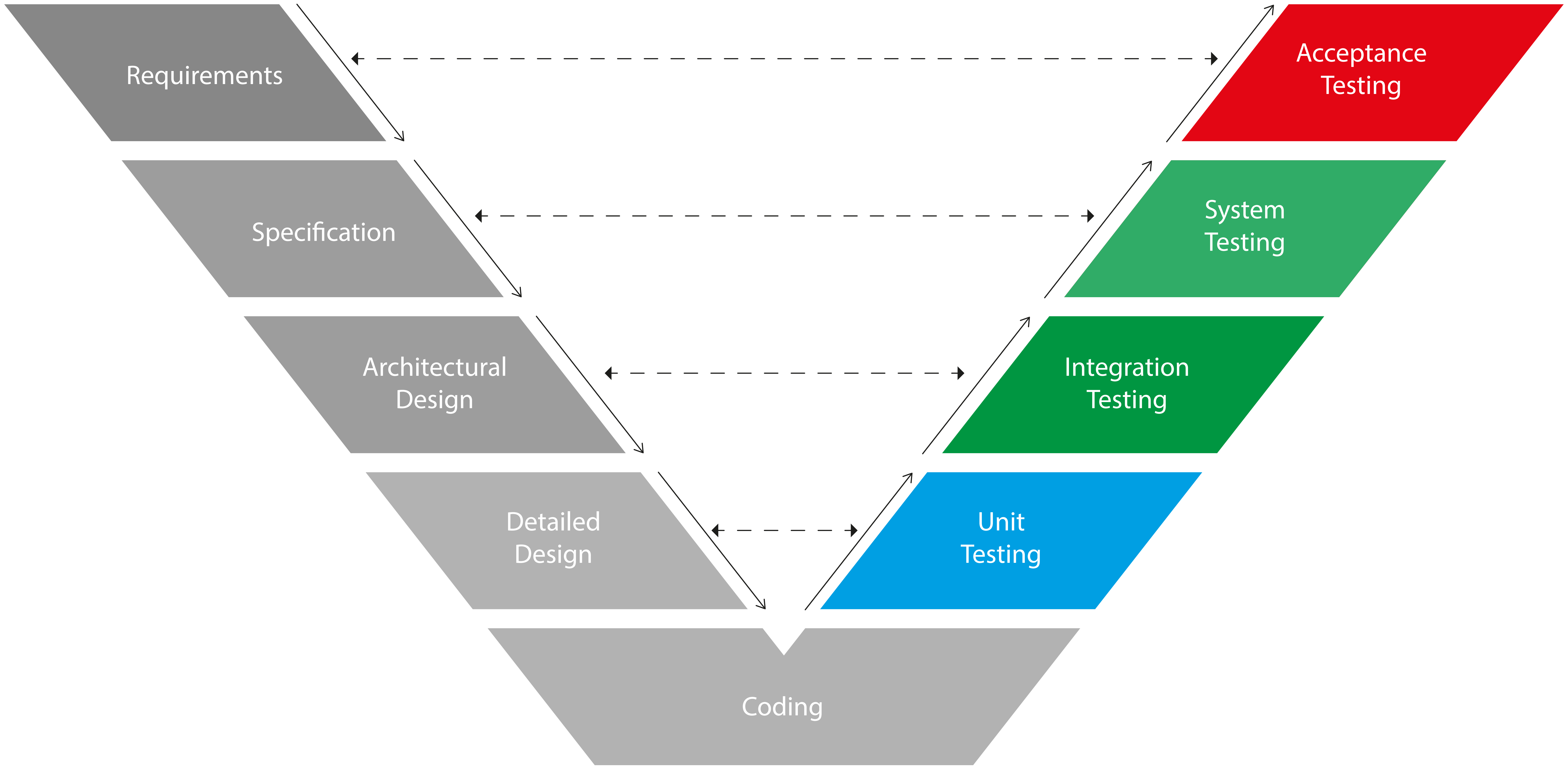
Breakpoints, locals and watches:



Note: (at least on macOS) it seems to require clang++ and llvm

HAN_UNIVERSITY OF APPLIED SCIENCES

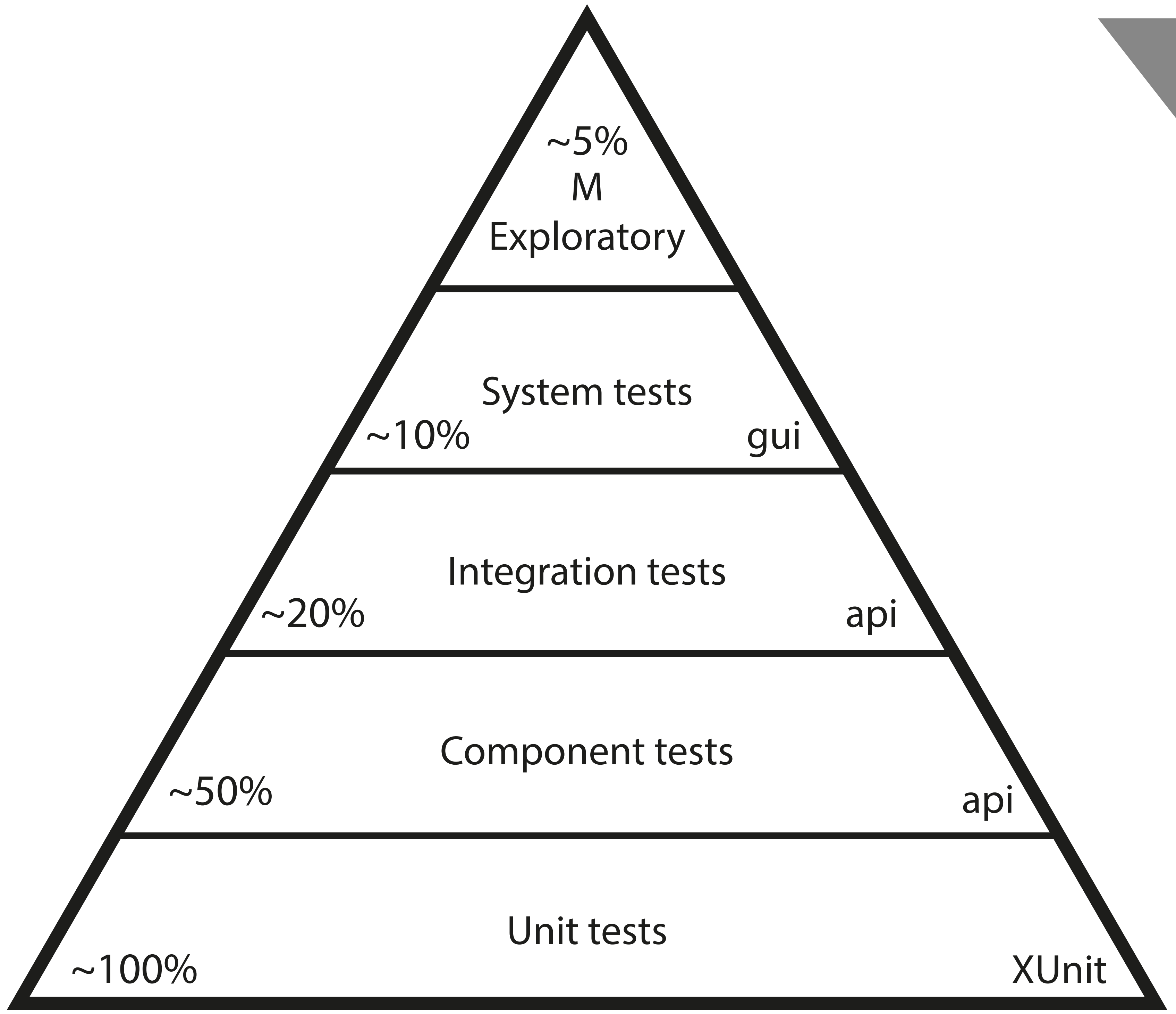
Testing in an Embedded Systems Software Engineering context

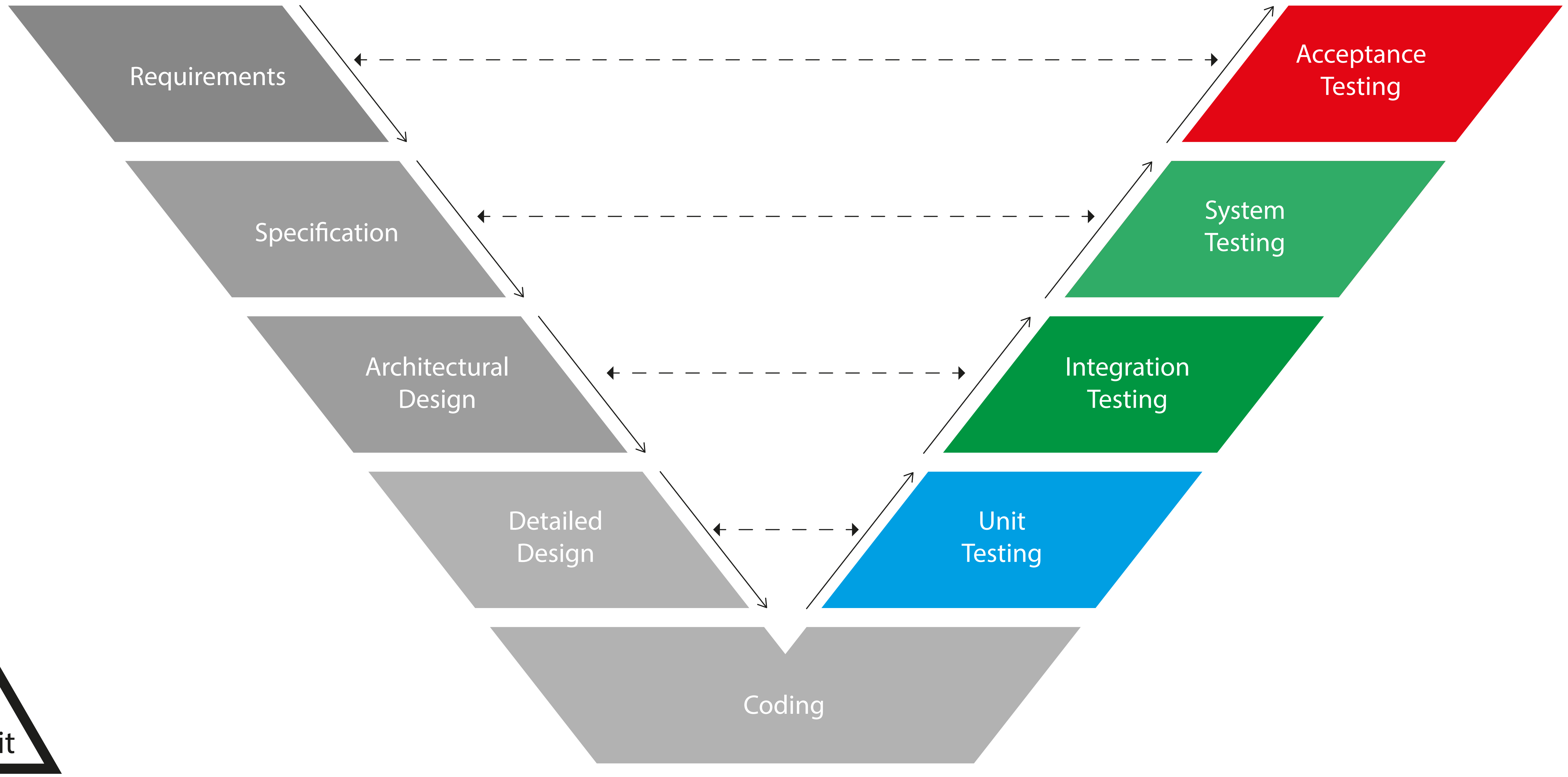


Testing

Martin (2011, p.115)
provides us with

the “Test Automation Pyramid”:





Testing: Basic Unit testing in C++

For this period we will only introduce Unit testing.

Some basic rules for Unit tests:

- Testing uses exactly one assert per test.
- Avoid if-statements.
- Unit tests only “new()” the unit under test.
- Unit tests do not contain hard-coded values unless they have a specific meaning.
- Unit tests are stateless.
- Unit test should be fallible (read: are only useful if they can fail in the first place).

Testing: Basic Unit testing in C++

https://code.visualstudio.com/api/working-with-extensions/testing-extension

https://marketplace.visualstudio.com/items?itemName=drleq.vscode-cpputf-test-adapter

Note: For some reason I need to add #include <cmath> to CppUnitTestFramework.hpp

Testing: Basic Unit testing in C++

```cpp
#define GENERATE_UNIT_TEST_MAIN
#include "CppUnitTestFramework.hpp"

#include <iostream>

class TestMe {

public:
    const int age = 10;

    TestMe() {
        std::cout << "I was created ex nihilo!" << std::endl;
    }

    ~TestMe() {
        std::cout << "My lifecycle ended: TestMe is no more :(" << std::endl;
    }
};

TestMe iLoveTesting;

TEST_CASE(TestMe, Test1) {

        CHECK_EQUAL(age, 9);

}
```

One more thing:

Remember to use doxygen for your documentation.

"

Any questions?