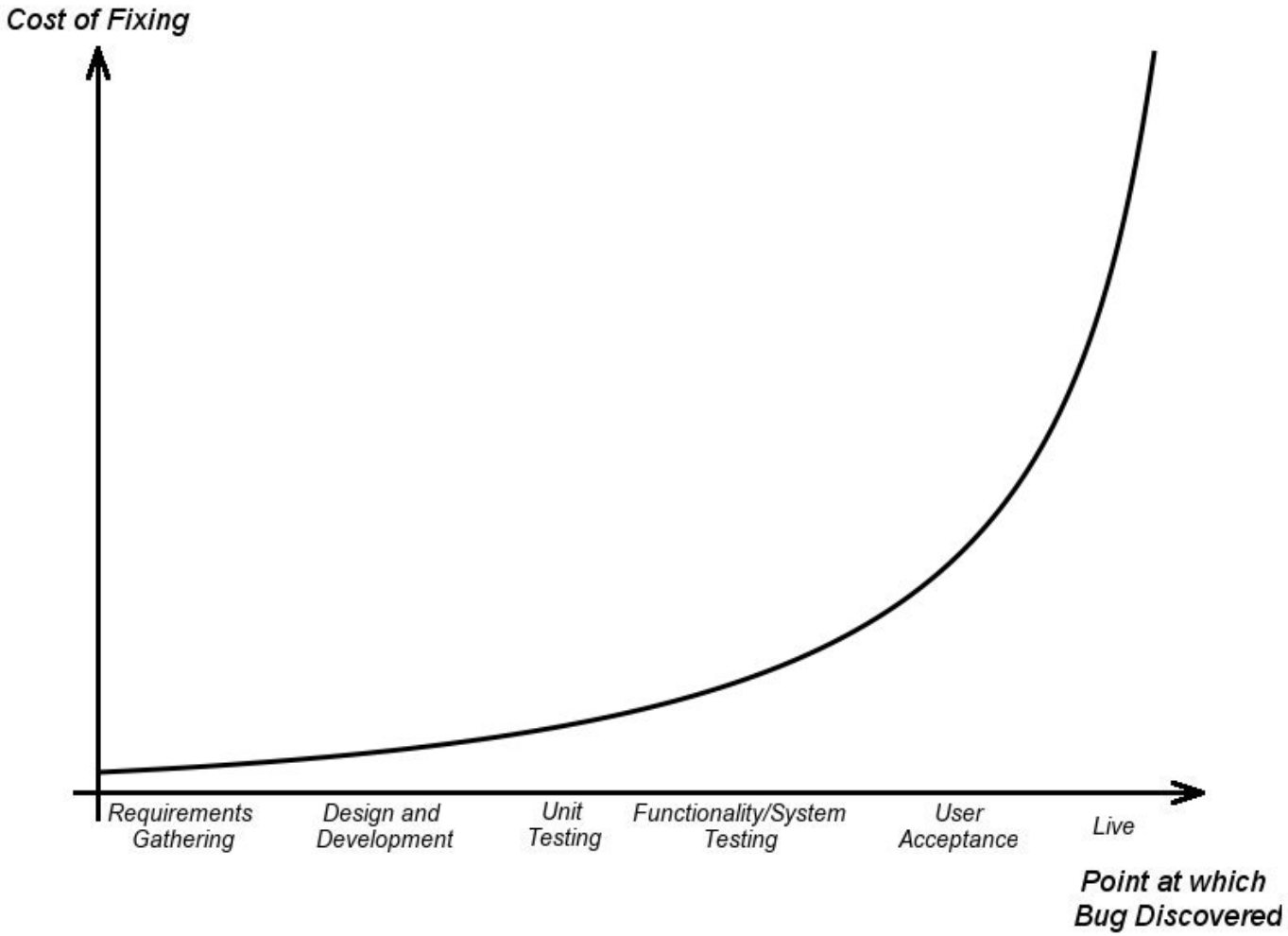
Preventing and discovering software defects

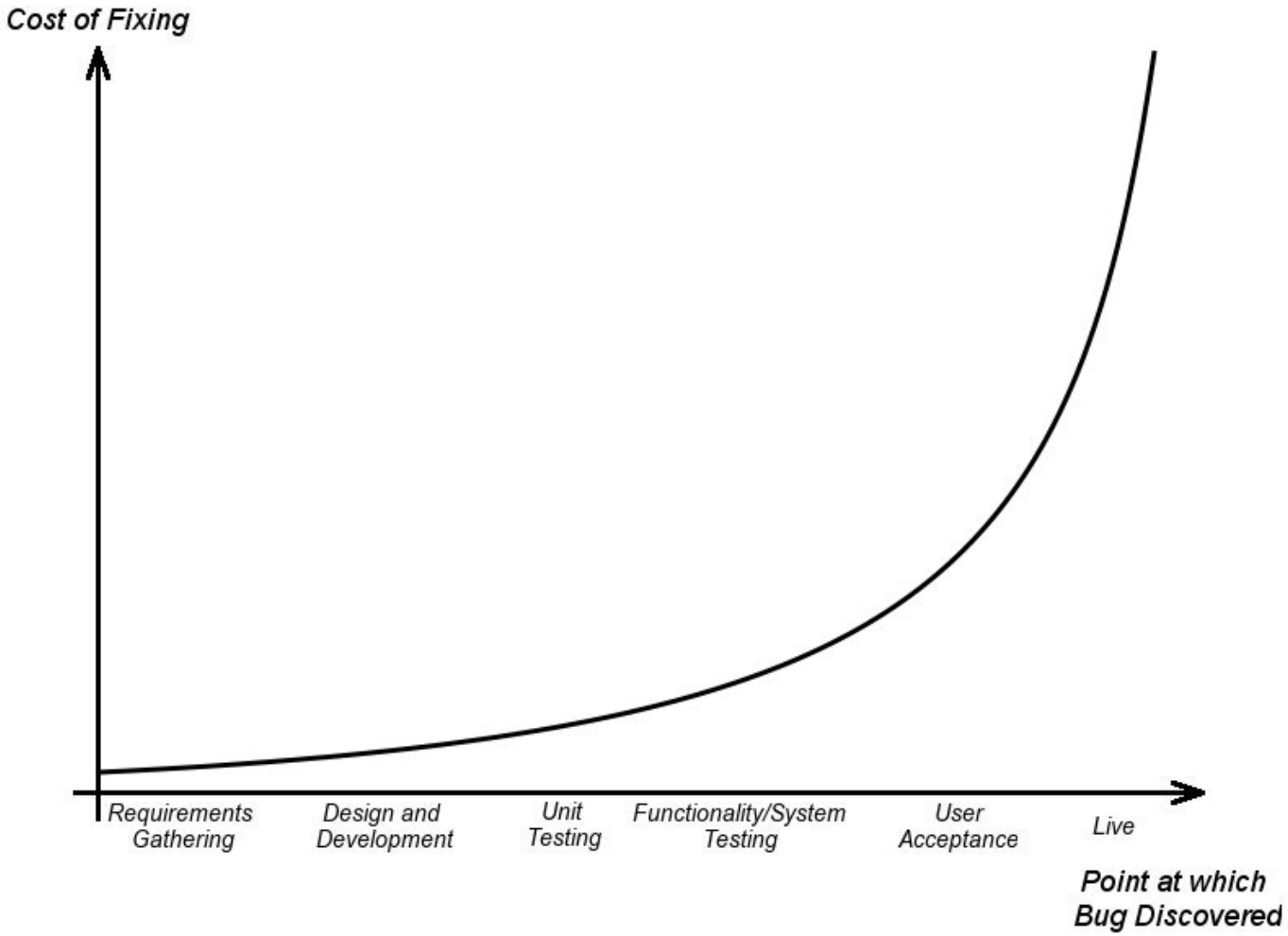
F. Giacomini

INFN-CNAF

ESC25 — Bertinoro, 29 September – 9 October 2025

https://agenda.infn.it/event/45177


Cost of Fixing
Requirements Gathering
Design and Development
Unit Testing
Functionality/System Testing
User Acceptance
Live
Point at which Bug Discovered



The aim should be to *shift left* the discovery of defects

During design and development

- Get familiar with the C++ Core Guidelines
- Design a class around its *class invariant*
  - A class invariant is a relation among the data members of a class that defines the valid values for the objects of that class
- Design a function around a *contract*
  - A contract is given by *pre-conditions* (constraints on the function arguments) and *post-conditions* (guarantees about the results)
- Waiting for proper support for contracts by the language, maybe in C++26, be generous with `asserts`

The compiler is your friend

- Enable as many warnings as reasonable and keep your compilation warning-free
- For gcc, for example:
  `-Wall -Wextra -Wpedantic -Wconversion -Wsign-conversion -Wshadow -Wimplicit-fallthrough -Wextra-semi -Wold-style-cast`
- But there are many many others

The compiler is your friend (cont.)

- Enable the assertions in the C++ standard library, to terminate the program in case of logical bugs
- For `gcc`, compile with `-D_GLIBCXX_ASSERTIONS`
- Keep these assertions also in production builds, the overhead should be negligible (but measure it)
- See `-fhardened`, but also Standard library hardening in C++26

The compiler is your friend (cont.)

- Profit from the sanitizers (address, undefined, thread, . . .)
- For example add `-fsanitize=address,undefined`, possibly combined with `-D_GLIBCXX_SANITIZE_VECTOR -D_GLIBCXX_SANITIZE_STD_ALLOCATOR`, to your debug builds
- Do not enable the sanitizers in production builds, due to their overhead

Unit testing

- Basic unit testing comes pretty easily with libraries like doctest, Catch, gtest, ...

```
TEST_CASE("Testing the factorial function"){
  CHECK(factorial(5) == 120);
  CHECK(factorial(0) == 1);
  CHECK(factorial(1) == 1);
  ...
}
```

- Be aware that testing can reveal the presence of bugs, not prove their absence
- Write tests with the purpose of breaking the code, not to confirm that it's correct
- Remember to enable the sanitizers
- You can even (ab)use `static_asserts` and run your tests directly at compile time

Before talking about debugging...

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

– Brian Kernighan

Using a debugger

- A debugger allows to execute a program step by step, printing variables, setting breakpoints and watchpoints, examining memory (incuding *core dumps* after a crash)
- Many exist. Let's consider gdb

```
$ gdb -version
GNU gdb (AlmaLinux) 14.2-4.1.el9_6
...
```

- For the basic commands see https://cht.sh/gdb
- When debugging, compile with `-g -Og -fno-omit-frame-pointer`
  - Recommended also when using sanitizers