Informatik / Computer Science

C++ Test-driven Development

Unit Testing, Code Assistance and Refactoring

IFS INSTITUTE FOR SOFTWARE

Prof. Peter Sommerlad

Director IFS Institute for Software

September 2014

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL

FHO Fachhochschule Ostschweiz

Peter Sommerlad
peter.sommerlad@hsr.ch

Work Areas

- Refactoring Tools (C++, Scala, ...) for Eclipse
- Decremental Development (make SW 10% its size!)
- Modern Agile Software Engineering
- ISO C++ standardization committee

Pattern Books (co-author)

- Pattern-oriented Software Architecture Vol. 1
- Security Patterns

Background

- Diplom-Informatiker / MSc CS (Univ. Frankfurt/M)
- Siemens Corporate Research - Munich
- itopia corporate information technology, Zurich
- Professor for Software Engineering, HSR Rapperswil, Director IFS Institute for Software

Credo:

People create Software

- communication
- feedback
- courage

Experience through Practice

- programming is a trade
- Patterns encapsulate practical experience

Pragmatic Programming

- test-driven development
- automated development
- Simplicity: fight complexity

Software Quality

Software Quality

Software Quality
"An Elephant in the Room"

- **classic approach:**
  - manual testing after creation

But:
Small Cute Things

Peter Sommerlad

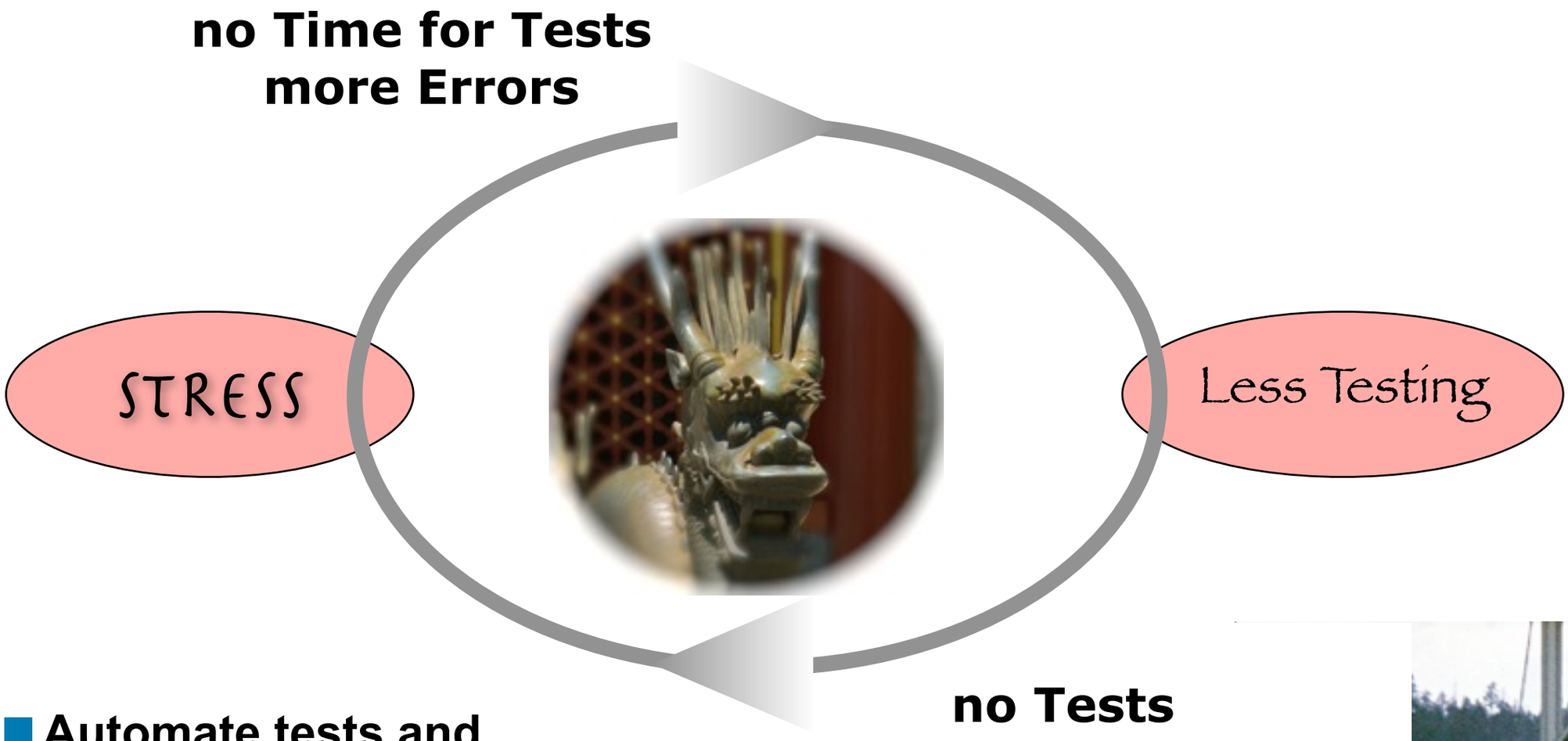
Grow to become larger Problems!

Is that Testing?

- **“it compiles!”**
  - no syntax error detected by compiler
- **“it runs!”**
  - program can be started
- **“it doesn’t crash!”**
  - …immediately with useful input
- **“it runs even with random input!”**
  - the cat jumped on the keyboard
- **“it creates a correct result!”**
  - a single use case is working with a single reasonable input

- **Automated (unit) testing gives you much more!**

Vicious Circle: Testing - Stress



- **Automate tests and run them often!**

C++ hello world (as generated by Eclipse CDT)

- **What is wrong here - and how can we test it?**

```
//============================================================================
// Name        : helloworld.cpp
// Author      :
// Version     :
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//============================================================================

#include <iostream>
using namespace std;

int main() {
    cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
    return 0;
}
```

belongs into version management system

bad practice, very bad in global scope

ridiculous comment

redundant

inefficient, redundant

using global variable! really bad if not in main() :-(

A better (hello) world? How would it be?

- **Unit testable code mustn't use global (non-const) variables**
  - separate functionality from main() into a separate compilation unit or library
  - write unit tests against the library
  - make main() so simple, it cannot be wrong
- **using namespace pollutes namespace of compilation unit**
  - therefore never ever put "using namespace" in global scope within a header
  - prefer using declarations, like "using std::cout;" to name what you are actually using
    - functions and operators are automatically found when called, because of argument dependent lookup
- **ostream std::cout will flush automatically when program ends anyway**
  - use of std::endl most of the time a waste, because of flushing (except asking for input)

How can we test Hello World?

- **extract function with the functionality to be tested**
- **rid the function of dependencies you can not control (to std::cout)**
  - parameterize with std::ostream&
- **move function to a library that can be tested with unit tests to separate it from main**
- **write tests against the function**
- **run the tests**

A tested/testable hello world

Test:

```
#include "cute.h"
#include "ide_listener.h"
#include "xml_listener.h"
#include "cute_runner.h"
#include <sstream>
#include "sayHello.h"
void testSayHelloSaysHelloWorld() {
   std::ostringstream out;
   sayHello(out);
   ASSERT_EQUAL("Hello, world!\n",out.str());
}
void runAllTests(int argc, char const *argv[]){
    cute::suite s;
    s.push_back(CUTE(testSayHelloSaysHelloWorld));
    cute::xml_file_opener xmlfile(argc,argv);
    cute::xml_listener<cute::ide_listener<>> lis(xmlfile.out);
    cute::makeRunner(lis,argc,argv)(s, "AllTests");
}
int main(int argc, char const *argv[]){
    runAllTests(argc,argv);
    return 0;
}
```

Library:

```
#ifndef SAYHELLO_H_
#define SAYHELLO_H_

#include <iosfwd>
void sayHello(std::ostream &out);

#endif /* SAYHELLO_H_ */
```

```
#include "sayhello.h"
#include <ostream>
void sayHello(std::ostream &out){
   out << "Hello, world!\n";
}
```

Executable:

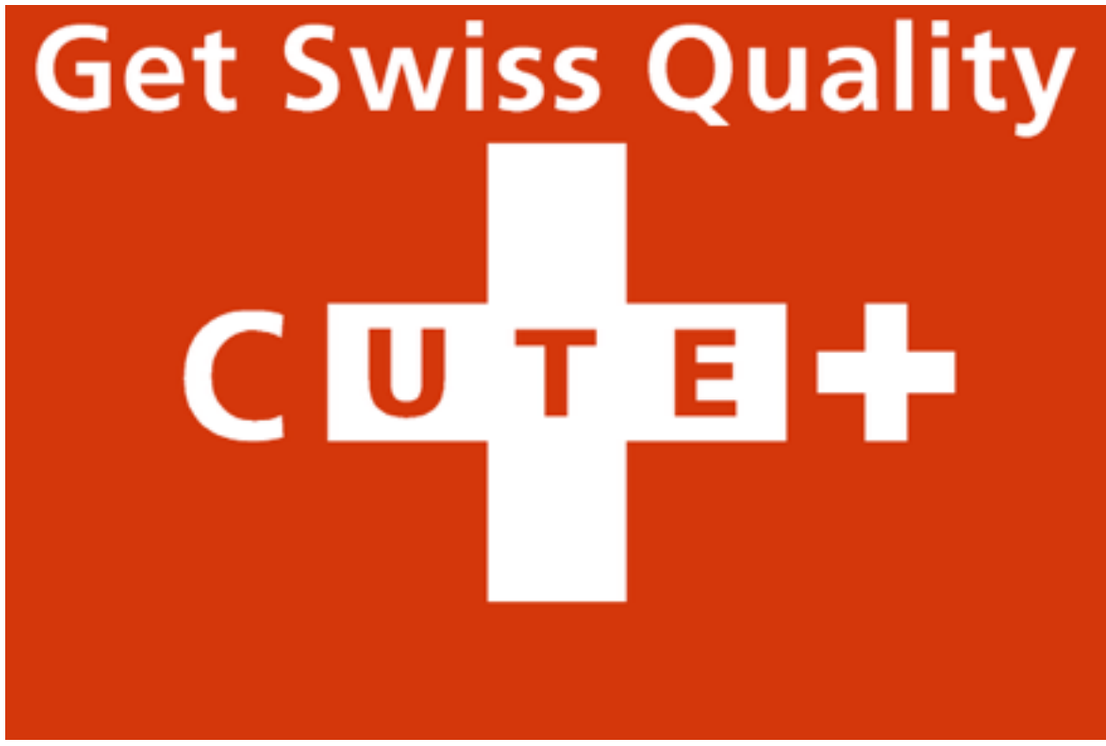
```
#include "sayhello.h"
#include <iostream>
int main(){
   sayHello(std::cout);
}
```

CUTE - A modern C++ Unit Testing Framework

C++ Unit Testing with CUTE in Eclipse CDT

TDD Test-Driven Development and Refactoring

- **CUTE http://cute-test.com - free!!!**
- **simple to use - test is a function**
  - understandable also for C programmers
- **designed to be used with IDE support**
  - can be used without, but a slightly higher effort
- **deliberate minimization of #define macro usage**
  - macros make life harder for C/C++ IDEs



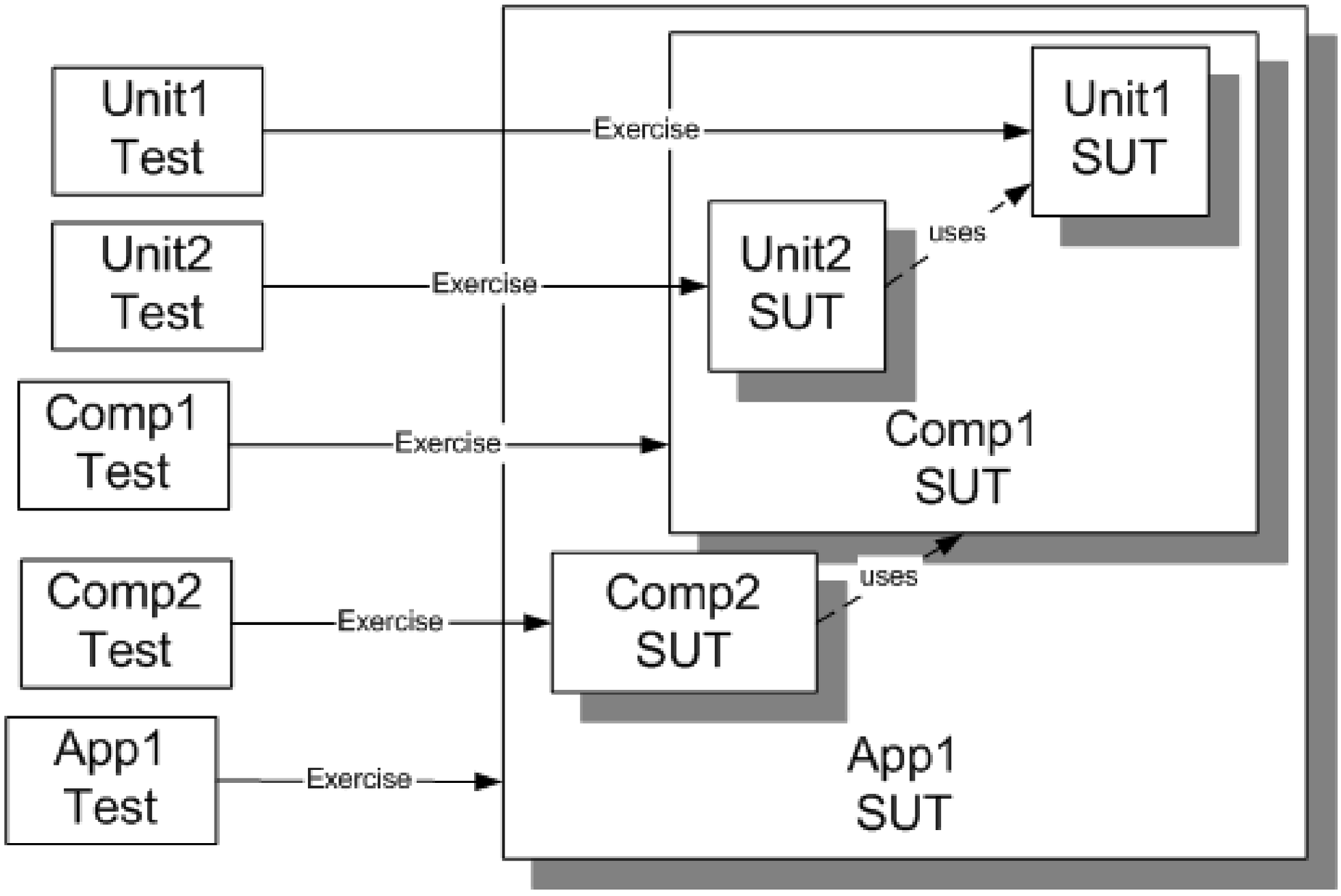
How do I write good Unit Tests?

- **If the code is correct, how would I know?**
- **How can I test this?**
- **What else could go wrong?**
- **Could a similar problem happen elsewhere?**

- **Writing good automated tests is hard.**
- **Beginners are often satisfied with “happy-path” tests**
  - error conditions and reactions aren’t defined by the tests
- **Code depending on external stuff (DB, IO, etc) is hard to test. How can you test it?**
- **Will good tests provide better class design?**
- **How can tests be designed well?**

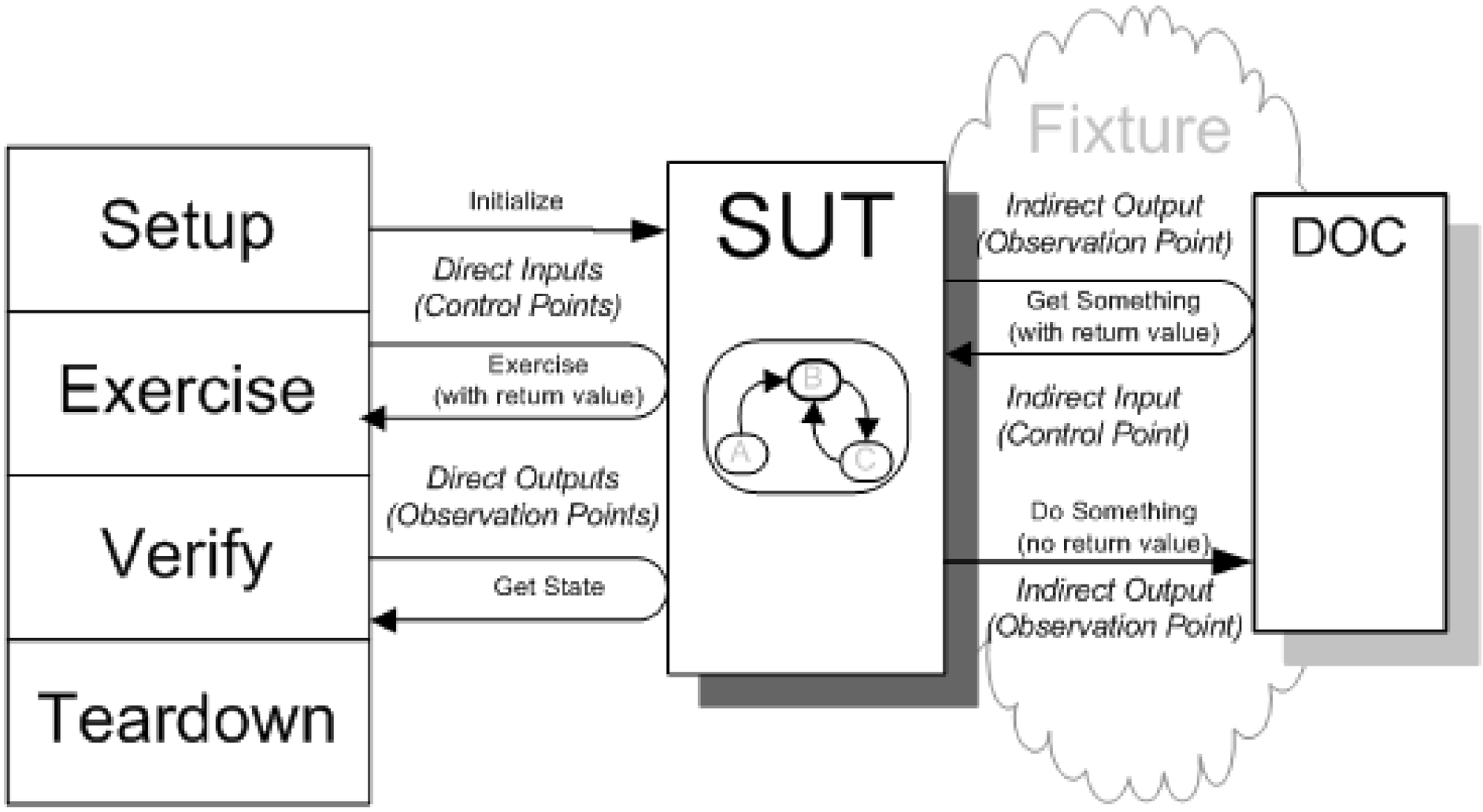
Some Terminology (source: xunitpatterns.com)

- **SUT System Under Test**



- **source: xunitpatterns.com - Gerard Meszaros**

Four Phase Test

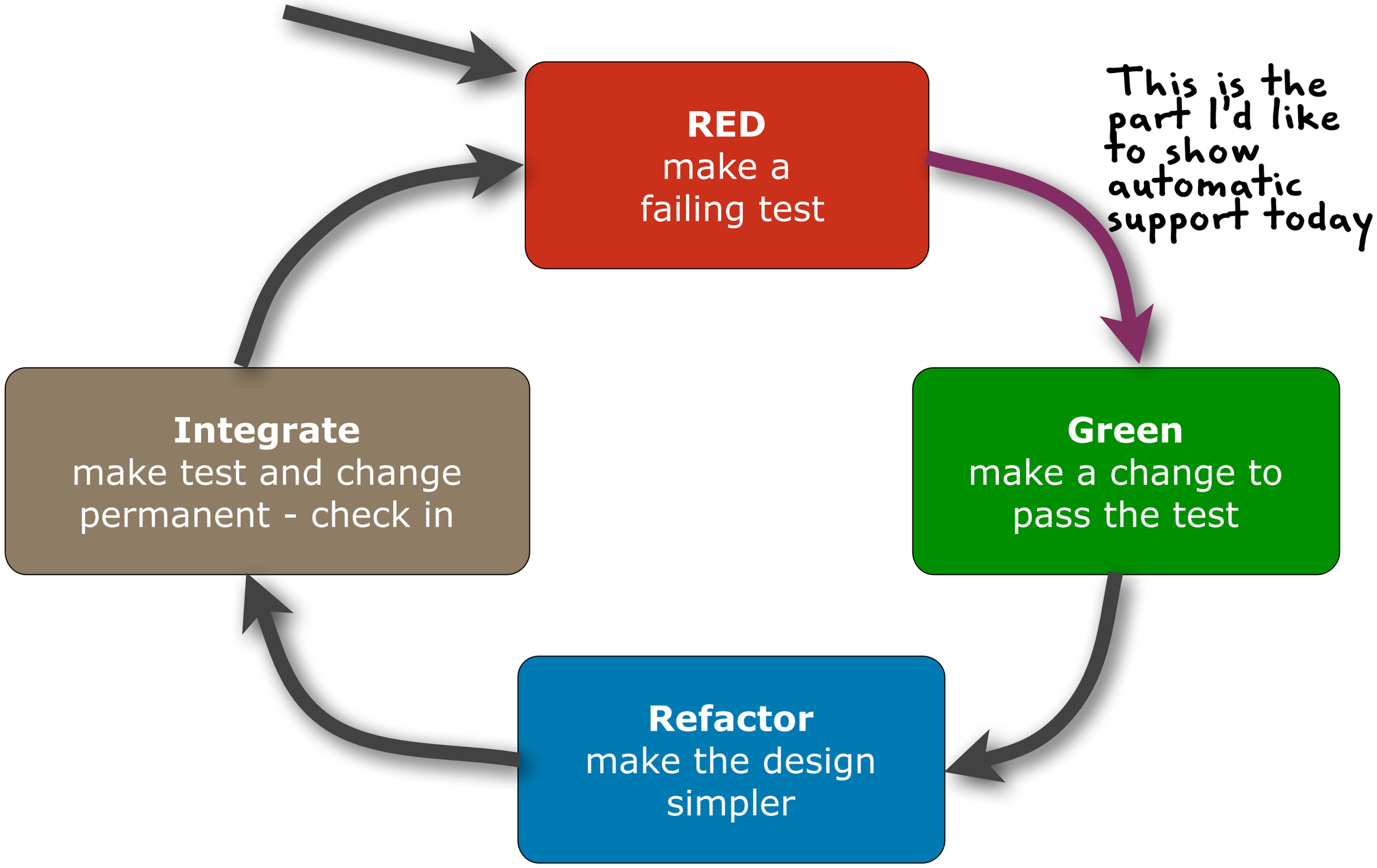


- **Setup: create locals needed for test**
- **Exercise: call function(s) which's behavior you want to check**
- **Verify: assert that results are as expected**
- **Teardown: usually trivial in C++, could mean to release external resources**

What are GUTs (Good Unit Tests)? (A. Cockburn)

- **are GOOD, DRY and Simple:**
  - no control structures, tests run linear
  - have the test assertion(s) in the end
- **test one thing at a time**
  - not a test per function/method, but a test per function call
  - a test per equivalence class of input values
- **have no (order) dependency**
  - leave no traces for others to depend on
- **all run successfully if you deliver**
- **have a good coverage of production code**
- **are often created Test-First —> Test-Driven Development**

TDD Cycle



Eclipse CDT TDD Support for CUTE and CUTE-plug-in features

- **get the cute plug-in from http://cute-test.com/updatesite or download Cevelop**

Download IDE at:
www.cevelop.com

- **create CUTE test project**
- **create test function**
- **create function definition from call (in test case)**
- **create type definition from usage (in test case)**
- **create variable definition from usage (in test case)**
- **move type definition into new header file (from test case file)**
- **toggle function definition between header and implementation (part of CDT)**

TDD Demo
Simulation of a Switch

- **Simulation of a Switch (class)**
  - alternative possible, if this seems too simple and time permits

THE LIST FOR CLASS SWITCH

1. CREATE,GETSTATE → OFF
2. TURNON,GETSTATE → ON
3. TURNOFF,GETSTATE → OFF

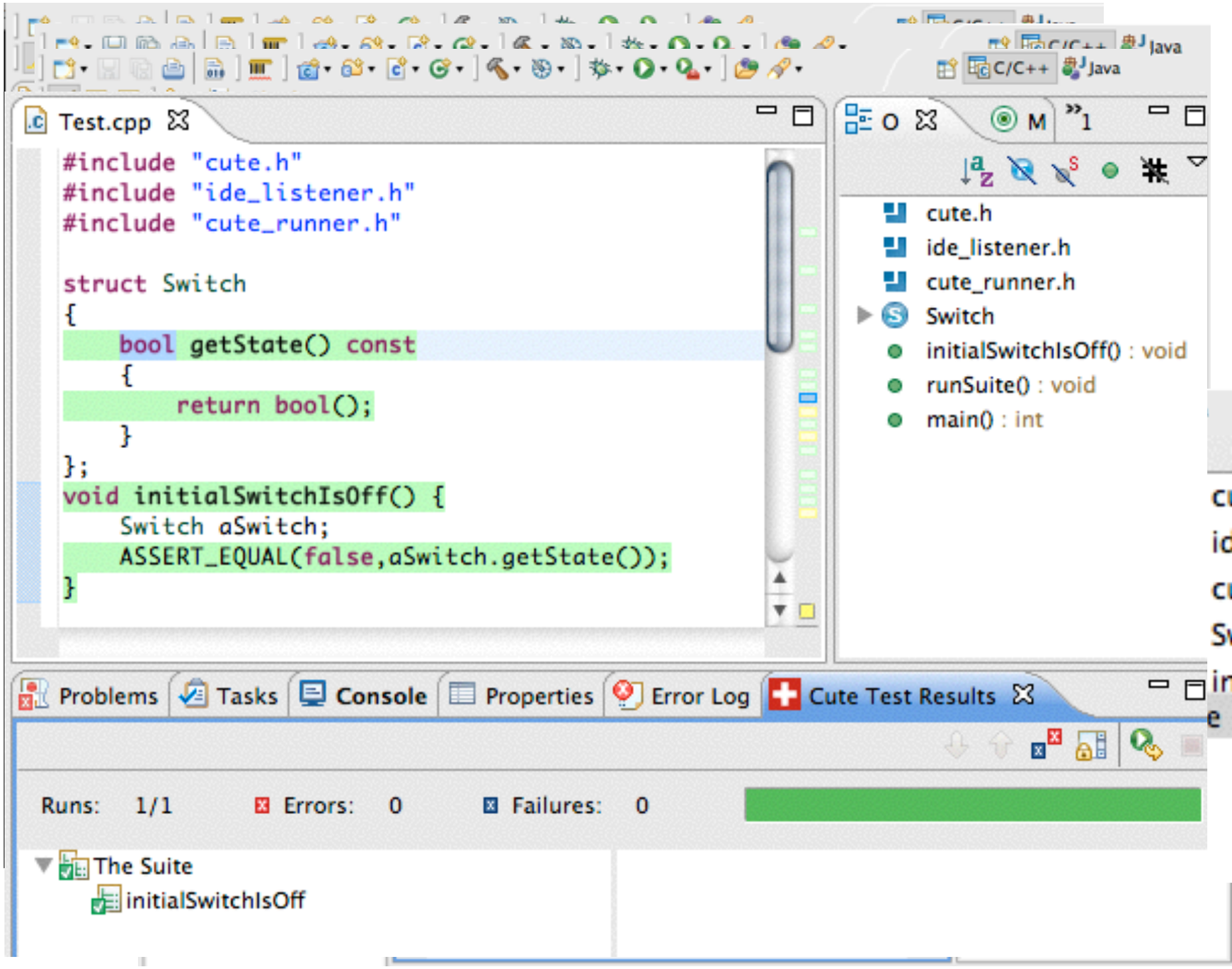
Demo 

TDD habits/patterns - Writing Tests

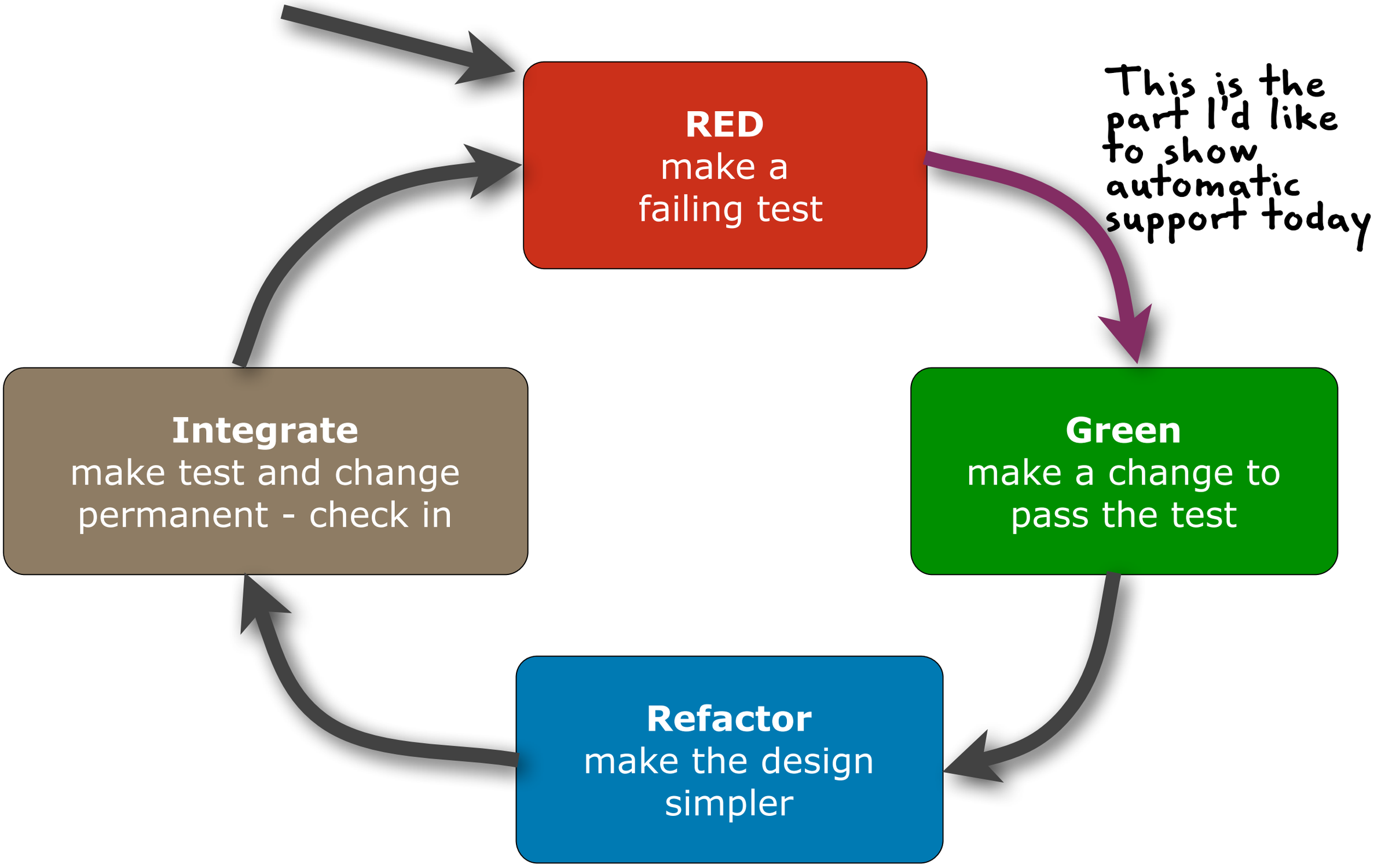
- **Isolated Tests**
  - write tests that are independent of other tests
- **Test List**
  - use a list of to-be-written tests as a reminder
  - only implement one failing test at a time
- **Test First**
  - write your tests before your production code
- **Assert First**
  - start writing a test with the assertion
  - only add the acting and arrangement code when you know what you actually assert

Demo

- **New C++ Project**
  - CUTE Test Project
- **Run as CUTE Test**
- **Fix Failing Test**
  - write first "real" test
    - rename test case function
    - write test code
    - make it compile through TDD quick-fixes (2x)
      - create type
      - create function
- **Run Tests**
  - green bar
- **iterate...**



TDD Cycle



What is Refactoring?

- **Ongoing cleaning**
  - “Clean Code”
- **Assure long-term quality**
  - code is always used longer than expected
- **Find better design**
  - understandability - habitable code
- **Remove duplication**
  - maintainability

TDD habits/patterns - “red bar” patterns

- **How do I find the tests I need to write?**

- **One Step Test**
  - solve a development task test-by-test
    - no backlog of test code, only on your test list
    - select the simplest/easiest problem next
- **Starter Test**
  - start small, e.g., test for an empty list
  - refactor while growing your code
- **Explanation Test**
  - discuss design through writing a test for it
- **Learning Test**
  - understand existing code/APIs through writing tests exercising it
  - Important, when you have to deal with legacy code base without GUTs

TDD habits/patterns: Green Bar patterns (how to make tests work)

- **Fake It ('Til You Make It)**
  - It is OK to “hack” to make your test succeed.
  - BUT: Refactor towards the real solution ASAP
- **Triangulate**
  - How can you select a good abstraction?
  - try to code two examples, and then refactor to the “right” solution
- **Obvious Implementation**
  - Nevertheless, when it’s easy, just do it.
- **One to Many (or zero, one, many)**
  - Implement functions with many elements first for one element (or none) correctly
- **Regression Test**
  - For every bug report write tests showing the bug
- **Break (Enough breaks are important! e.g. drink water while coding)**
- **Do Over (delete code when you are stuck and restart)**

TDD Demo
(3+4)*6 → 42

- **Expression Evaluator for simple Arithmetic**
- **Test-First Development with CUTE**
- **Incremental Requirements Discovery**

THE LIST FOR EVAL (V0)

"" → ERROR

"0" → 0

"2" → 2

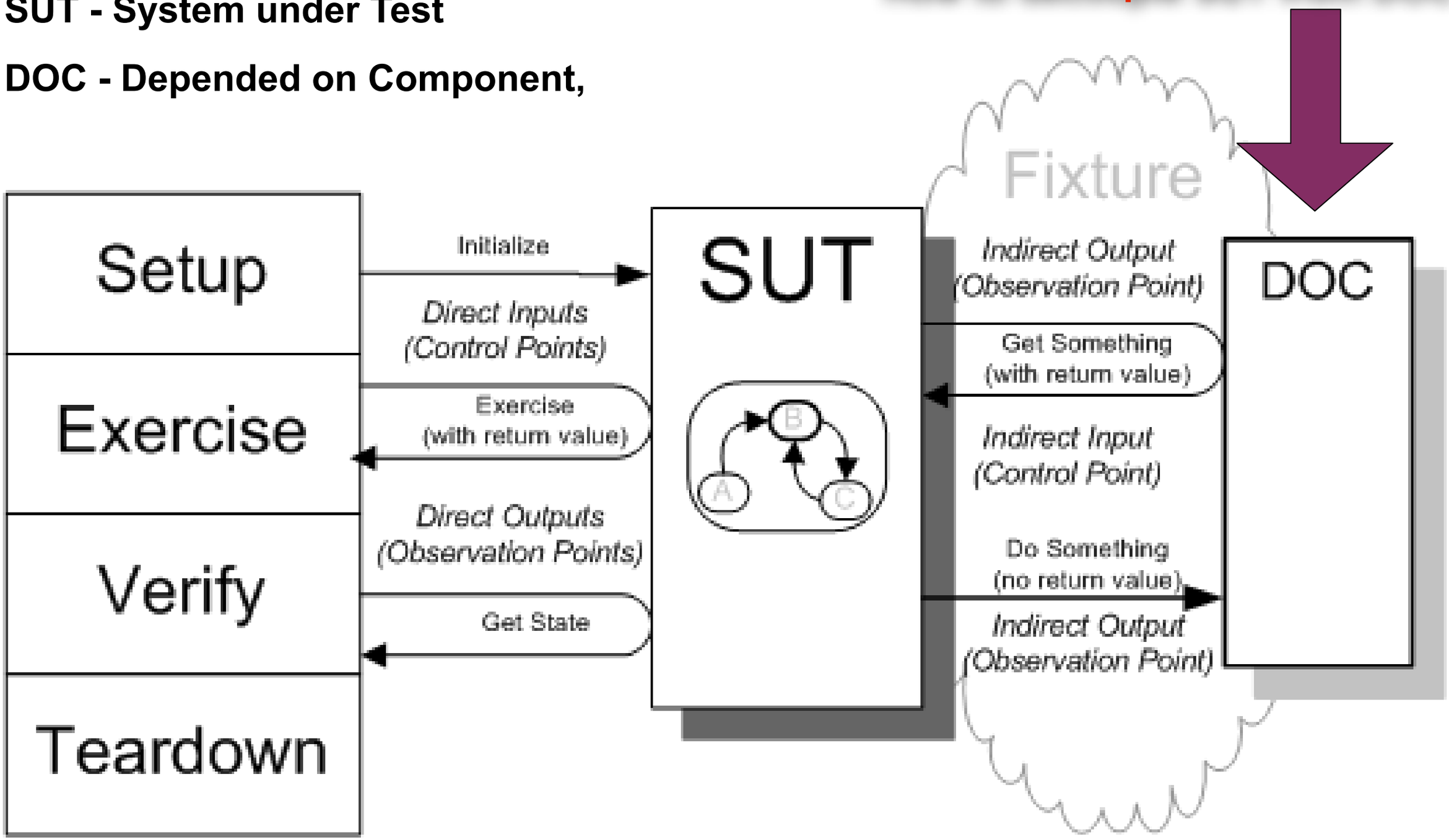
"1+1" → 2

Demo 

Four Phase Test Case
Setup, Exercise, Verify, Teardown

- **SUT - System under Test**
- **DOC - Depended on Component,**

How to decouple SUT from DOC?



- **source: xunitpatterns.com - Gerard Meszaros**

Mockator

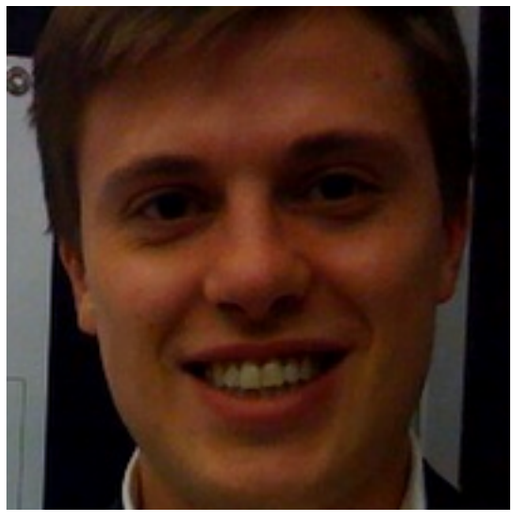
C++ Legacy Code Refactoring enabling Unit Testing

Seam Introduction
- Object Seam
- Compile Seam
- Link Seam
- Preprocessor Seam

Test Double generation

Mock Object generation

Function Tracer generation



Master Thesis by Michael Rüegg

inspired and supervised by Prof. Peter Sommerlad
soon to be released on http://mockator.com
beta at http://sinv–56033.edu.hsr.ch/mockator/repo

Principle of Mock objects

- **A unit/system under test (SUT) depends on another component (DOC) that we want to separate out from our test.**
- **Reasons**
  - real DOC might not exist yet
  - real DOC contains uncontrollable behavior
  - want to test exceptional behavior by DOC that is hard to trigger
  - using the real DOC is too expensive or takes to long
  - need to locate problems within SUT not DOC
  - want to test usage of DOC by SUT is correct

Why the need for Mock Objects?

- **Simpler Tests and Design**
  - especially for external dependencies
  - promote interface-oriented design
- **Independent Testing of single Units**
  - isolation of unit under testing
  - or for not-yet-existing units
- **Speed of Tests**
  - no external communication (e.g., DB, network)
- **Check usage of third component**
  - is complex API used correctly
- **Test exceptional behavior**
  - especially when such behavior is hard to trigger

Types of Mock Objects [Dave Astels]

- **There exist different categories of Mock objects and different categorizers.**

- **Stubs**
  - substitutes for “expensive” or non-deterministic classes with fixed, hard-coded return values
- **Fakes**
  - substitutes for not yet implemented classes
- **Mocks**
  - substitutes with additional functionality to record function calls, and the potential to deliver different values for different calls

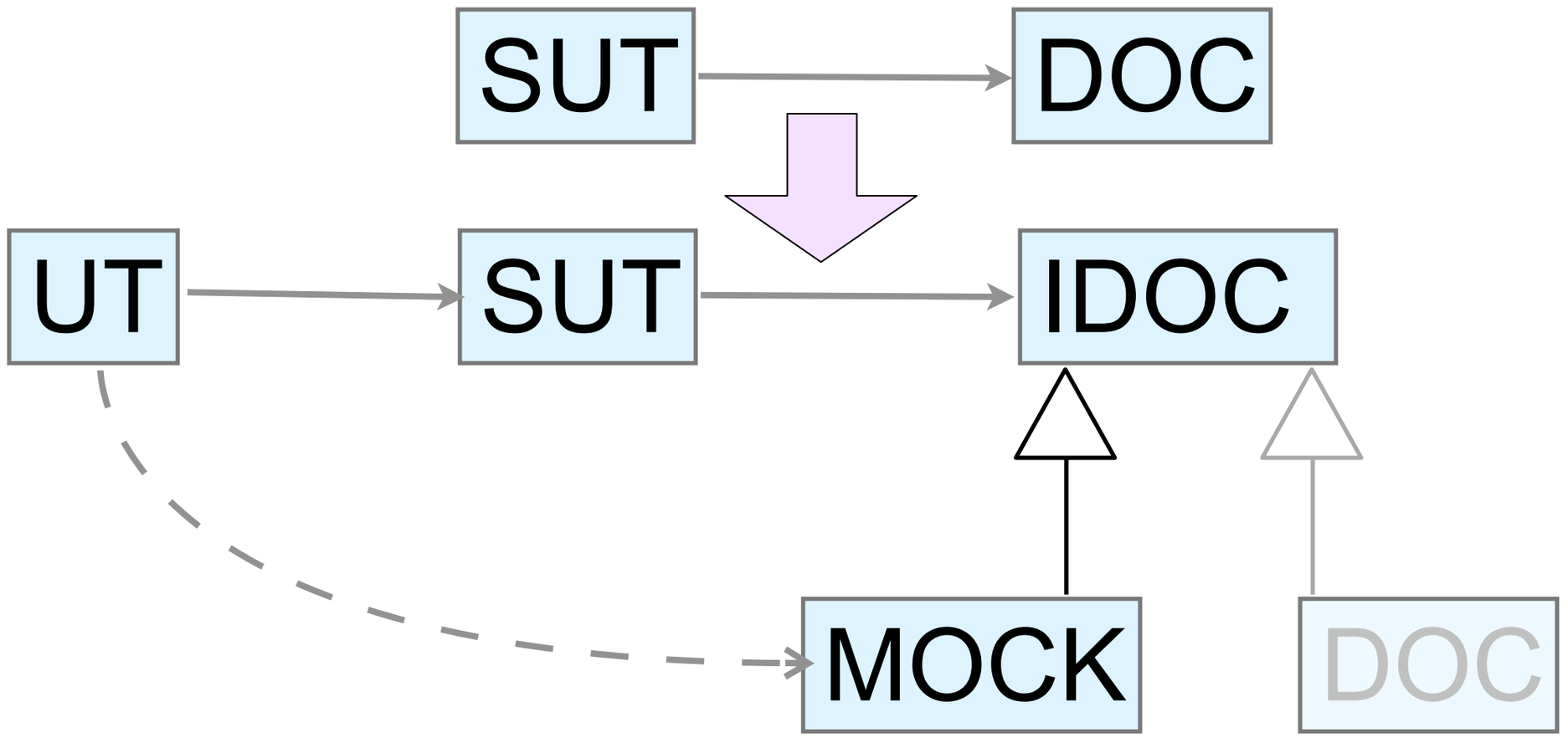
Seams!

How to decouple SUT from DOC?

- **Introduce a Seam:**
  - makes DOC exchangeable!
  - C++ provides different mechanisms
- **Object Seam (classic OO seam)**
  - Introduce Interface - Change SUT to use Interface instead of DOC directly
    - introduces virtual function lookup overhead
  - Pass DOC as a (constructor) Argument
  - Pass Test Double as Argument for Tests
- **Compile Seam (use template parameter)**
  - Make DOC a default template Argument

Introduce an Object Seam

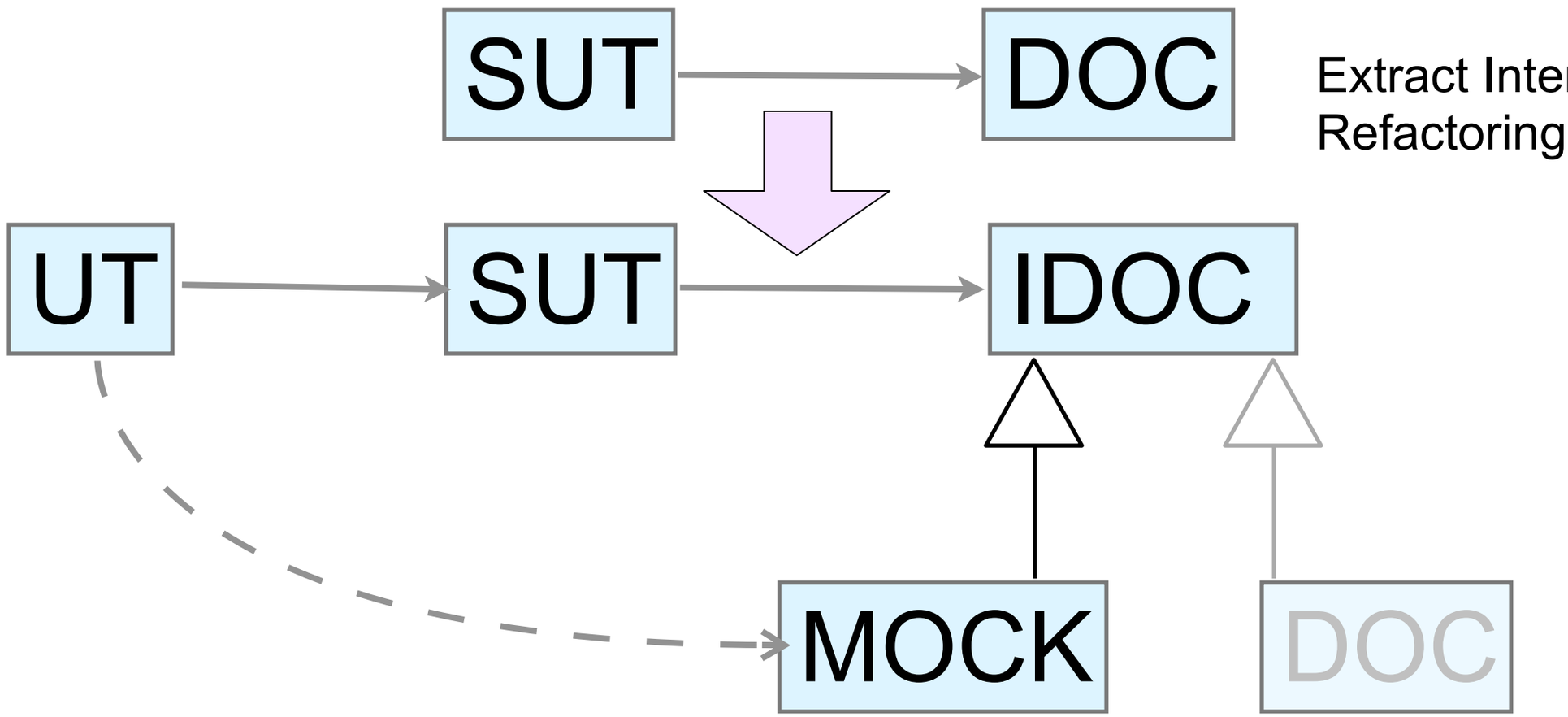
- **classic inheritance based mocking**
  - extract interface for DOC -> IDOC
  - make SUT use IDOC
  - create MOCK implementing IDOC and use it in UT



  - in C++ this means overhead for DOC (virtual functions)!

Introduce an Object Seam

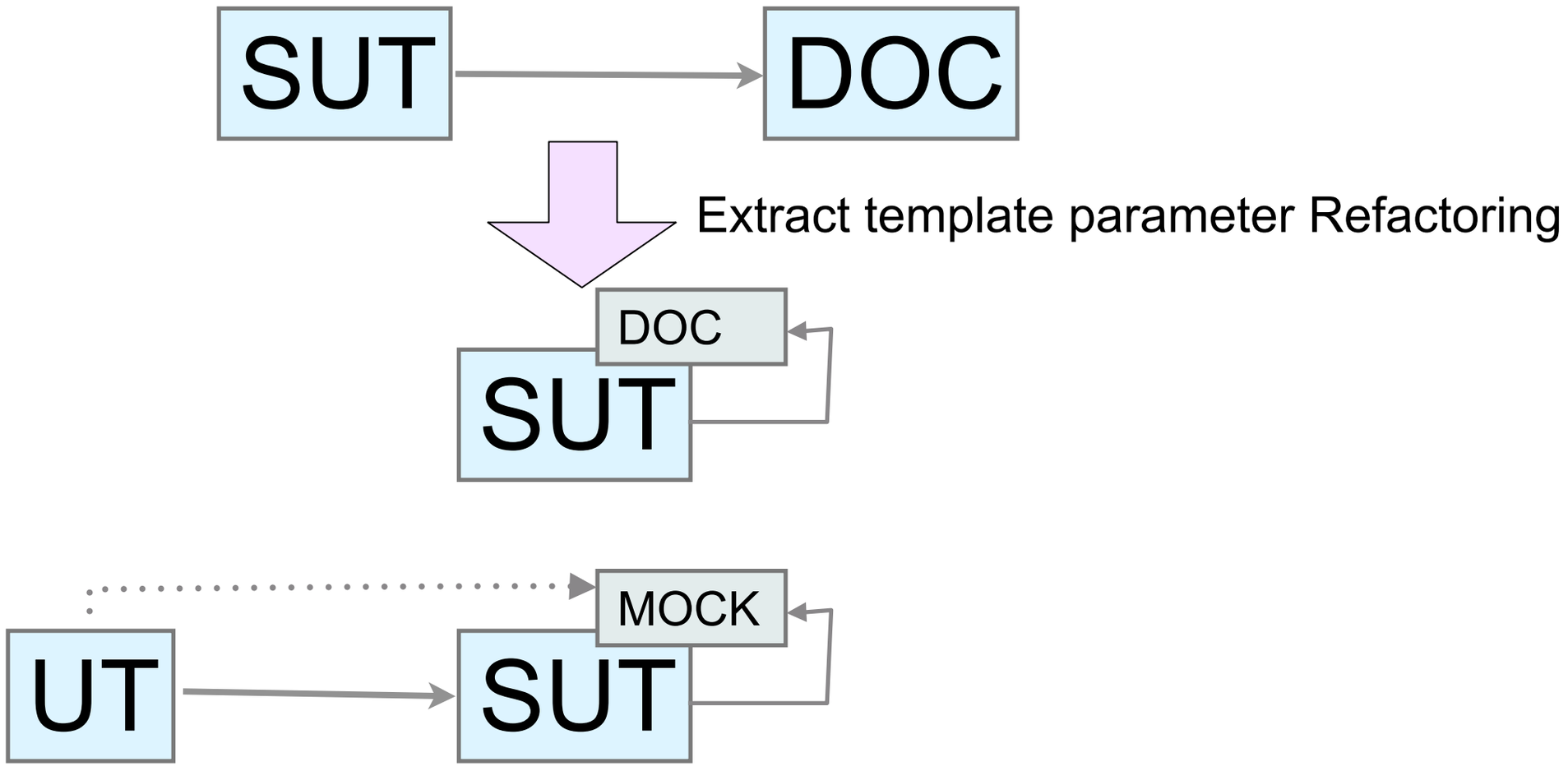
- **classic inheritance based mocking**
  - extract interface for DOC -> IDOC
  - make SUT use IDOC, edit constructor
  - create MOCK implementing IDOC and use it in UT



- in C++ this means overhead for DOC (virtual functions)!

Introduce a Compile Seam

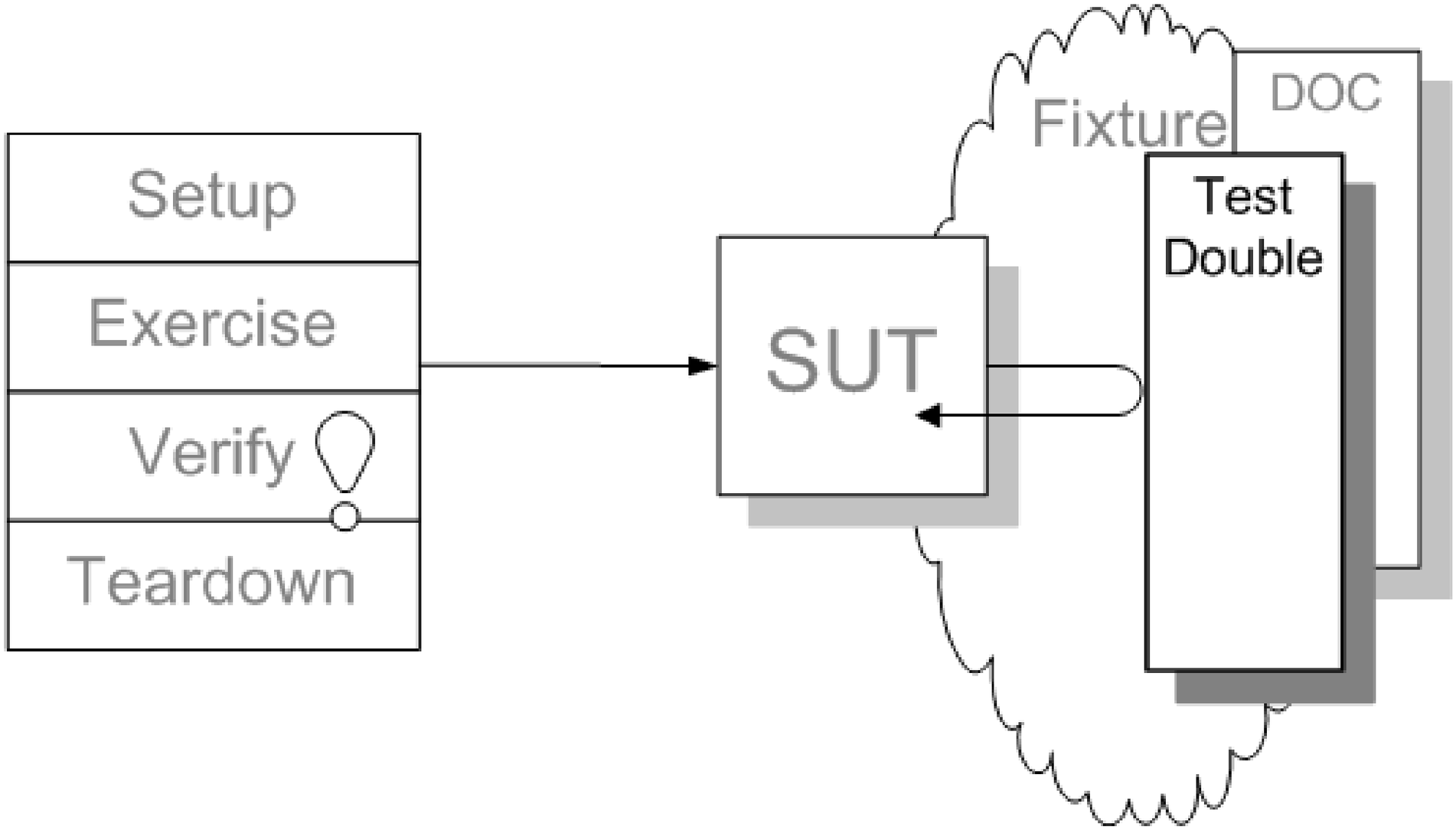
- **C++ template parameter based mocking**
  - make DOC a default template argument of SUT



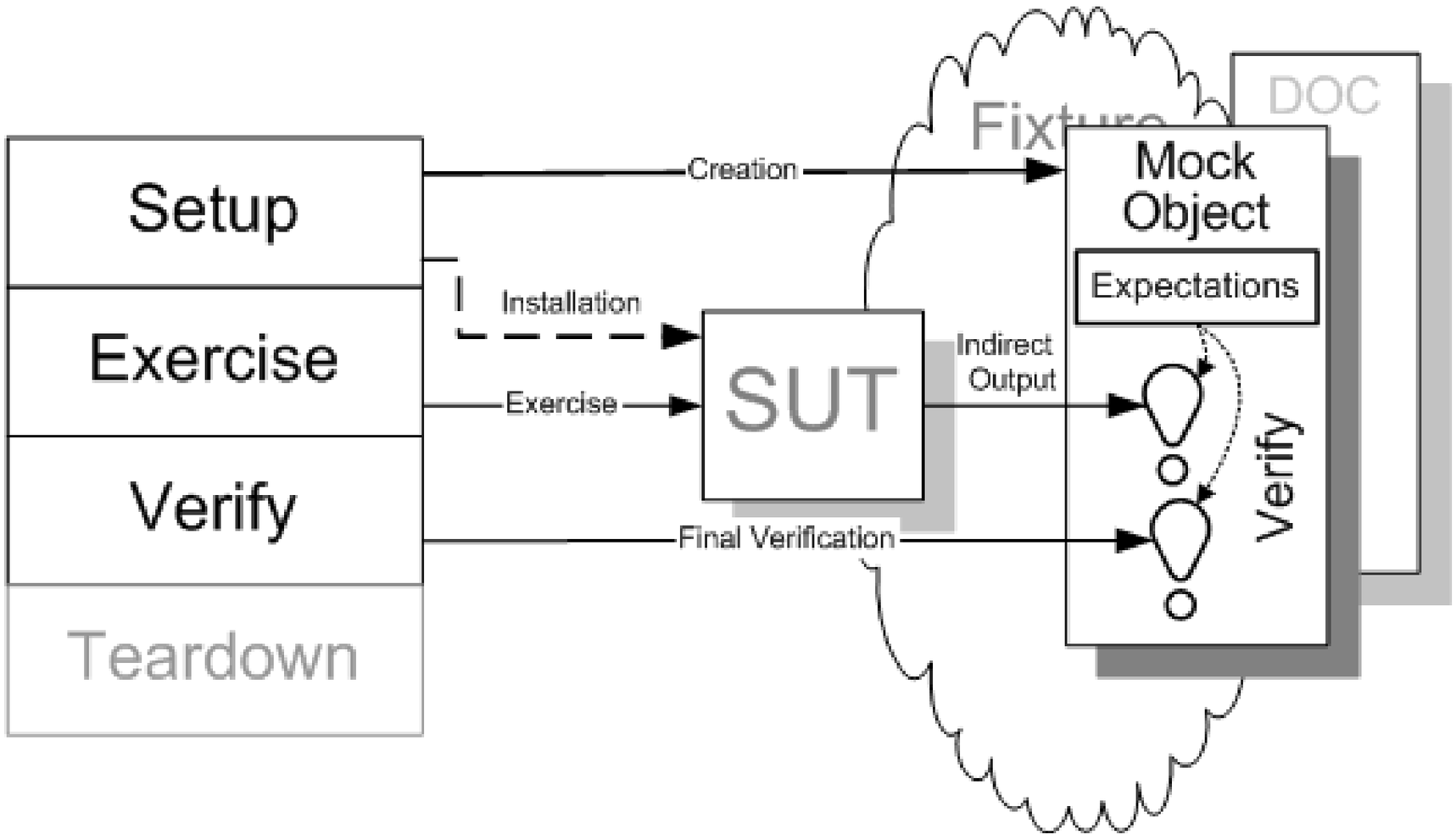
C/C++ specifics: Link Seam, Preprocessor

- **absolutely no change on existing code of SUT needed!**
- **remove Dependency on system/library functions**
  - shadowing through providing an alternative implementation earlier in link sequence
    - avoid dependency on system or non-deterministic functions, e.g. rand(), time(), or "slow" calls
  - wrapping of functions with GNU linker --wrap option, allows calling original also
    - good for tracing, additional checks (e.g., valgrind)
  - wrapping functions within dynamic libraries with dlopen/dlsym&LD_PRELOAD
  - problem: C++ name mangling if done by hand (solved by Mockator)
- **replace calls through #define preprocessor macro**
  - as a means of last resort, many potential problems

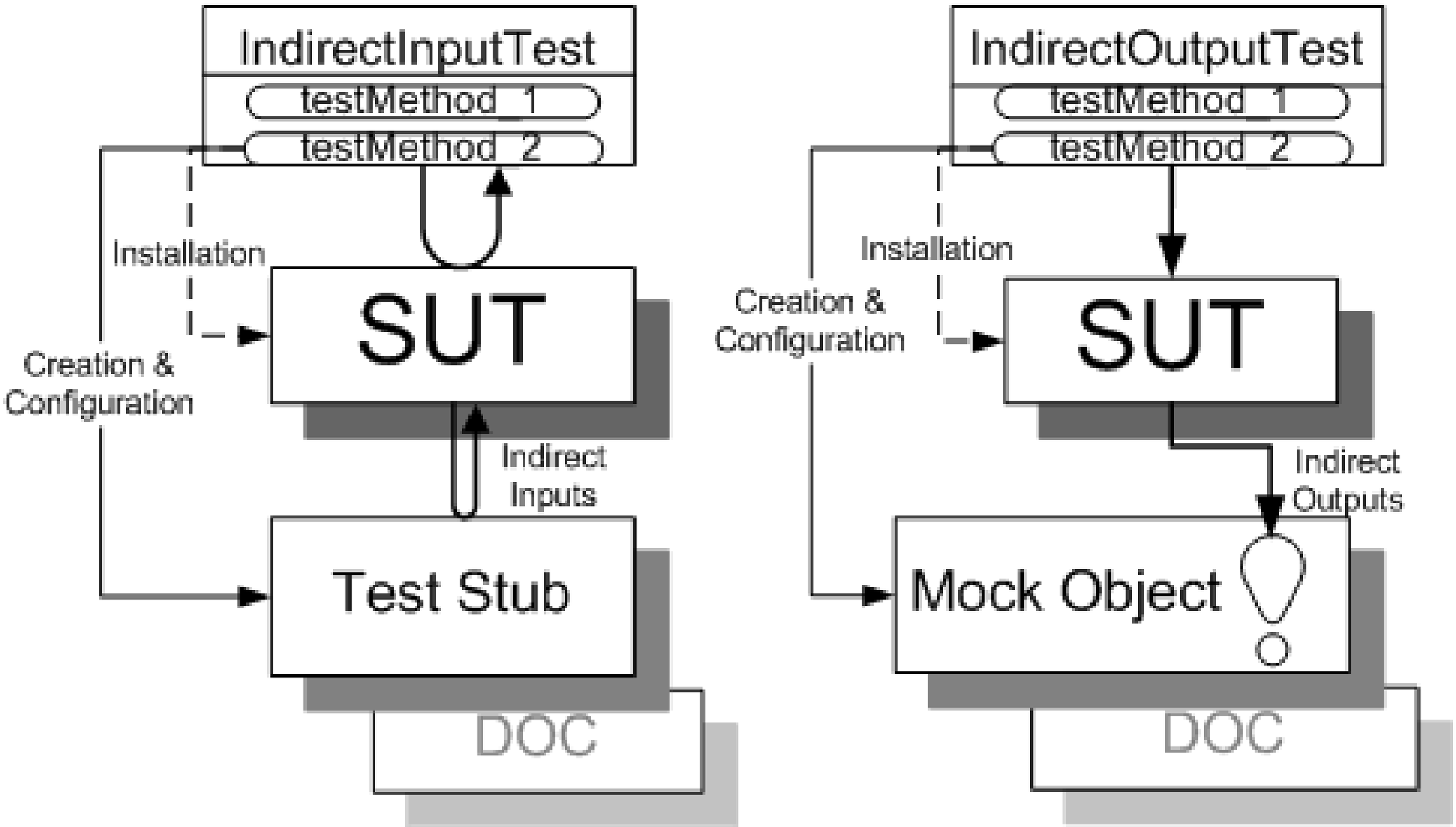
- **How can we verify logic independently when code it depends on is unusable?**
- **How can we avoid slow tests?**



- **How do we implement behavior verification for indirect outputs of the SUT?**



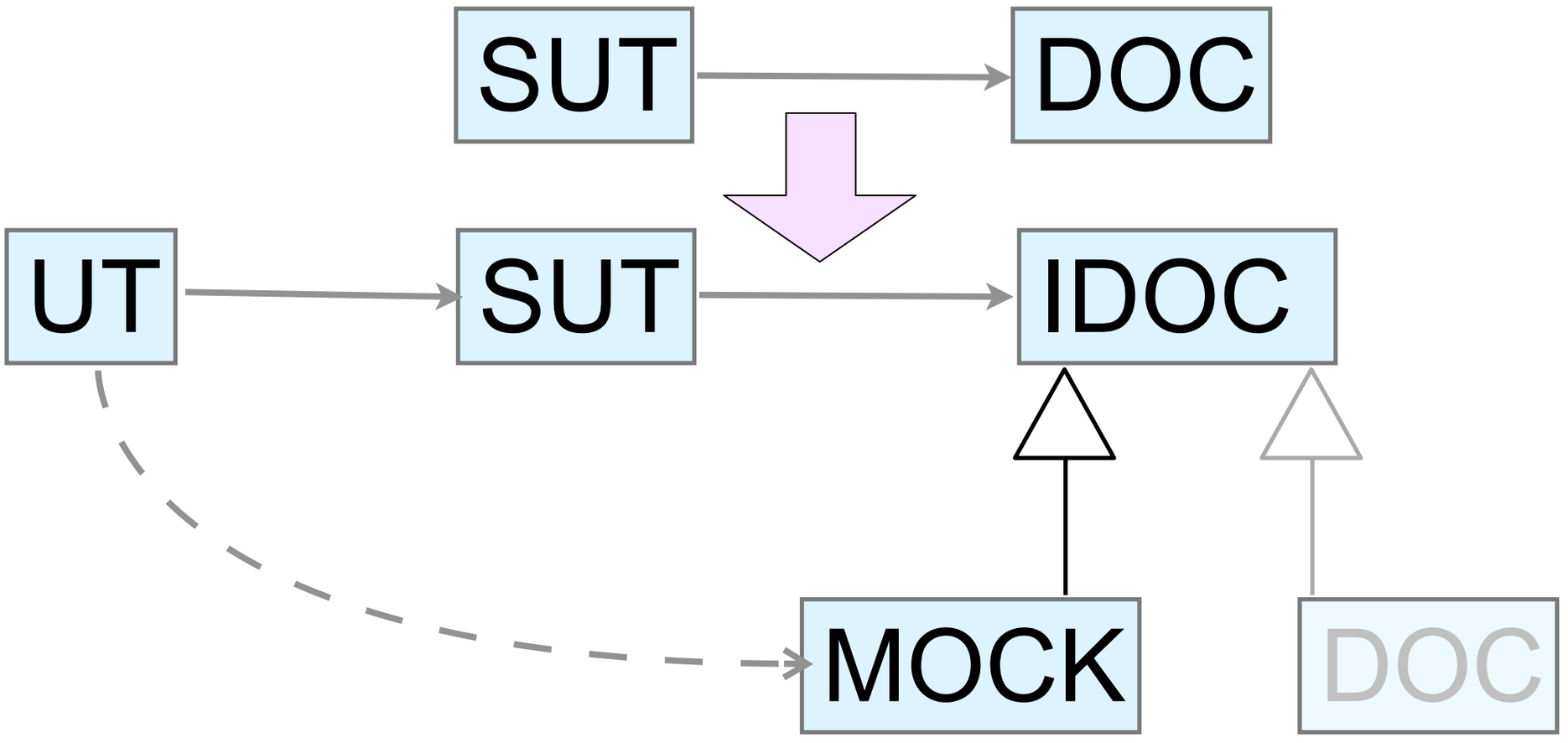
Difference between Test-Stub and Mock-Object



- **xunitpatterns.com**

Interface-oriented Mock

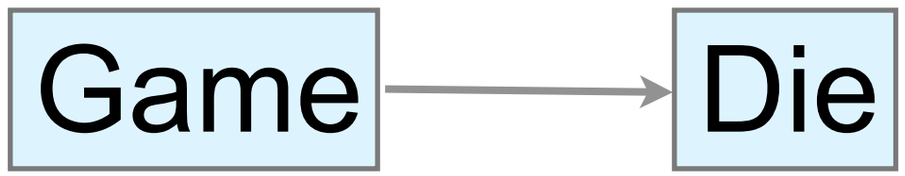
- **classic inheritance based mocking**
  - extract interface for DOC -> IDOC
  - make SUT use IDOC
  - create MOCK implementing IDOC and use it in UT



  - in C++ this means overhead for DOC (virtual functions)!

Demo/Exercise
Code in need for Mocking

- **A very simple game, roll dice, check if you've got 4 and you win, otherwise you loose.**



- **We want to test class Die first:**

```
#include <cstdlib>

struct Die
{
   int roll() { return rand()%6 + 1; }
};
```

How to test Game?


```
#include "Die.h"
class GameFourWins
{
   Die die;
public:
   GameFourWins();
   void play();
};
```

```
void GameFourWins::play(){
   if (die.roll() == 4) {
      cout << "You won!" << endl;
   } else {
      cout << "You lost!" << endl;
   }
}
```

```
#include "Die.h"
#include <iostream>

class GameFourWins
{
   Die die;
public:
   GameFourWins();
   void play(std::ostream &os = std::cout);
};
```

```
void GameFourWins::play(std::ostream &os){
   if (die.roll() == 4) {
      os << "You won!" << endl;
   } else {
      os << "You lost!" << endl;
   }
}
```

Test with a Mock ostream

- **We now can use a ostrstream to collect the output of play() and check that against an expected value:**

```
void testGame() {
   GameFourWins game;
   std::ostringstream os;
   game.play(os);
   ASSERT_EQUAL("You lost!\n",os.str());
}
```

- **What is still wrong with that test?**

Simulation Mocks
Interface-oriented

- **deliver predefined values**
  - we need that for our Die class
- **Introduce an Interface**

```
struct DieInterface
{
   virtual ~DieInterface(){}
   virtual int roll() =0;
};

struct Die: DieInterface
{
   int roll() { return rand()%6+1; }
};
```

- **now we need to adjust Game as well to use DieInterface& instead of Die**
- **Mockator Pro plug-in will make those code conversions automatic (Summer 2012)**

Simulation Mocks
preparing SUT

- **Changing the interface, need to adapt call sites**
- **theDie must live longer than Game object**

```
class GameFourWins
{
   DieInterface &die;
public:
   GameFourWins(DieInterface &theDie):die(theDie){}
   void play(std::ostream &os = std::cout);
};
```

- **now we can write our test using an alternative implementation of DieInterface**
- **would using pointer instead of reference improve situation? what's different?**

- **This way we can also thoroughly test the winning case:**

```
struct MockWinningDice:DieInterface{
   int roll(){return 4;}
};

void testWinningGame() {
   MockWinningDice d;
   GameFourWins game(d);
   std::ostringstream os;
   game.play(os);
   ASSERT_EQUAL("You won!\n",os.str());
}
```

A C++ alternative using templates

- **advantages: no virtual call overhead, no extra Interface extraction**
  - transformation provided by our "Introduce Typename Template Parameter" Refactoring
- **drawback: inline/export problem potential**

```
template <typename Dice=Die>
class GameFourWinsT
{
   Dice die;
public:
   void play(std::ostream &os = std::cout){
     if (die.roll() == 4) {
     os << "You won!" << std::endl;
     } else {
       os << "You lost!" << std::endl;
     }
   }
};
typedef GameFourWinsT<Die> GameFourWins;
```

Mock via template parameter

- **The resulting test looks like this:**

```
struct MockWinningDice{
   int roll(){return 4;}
};
void testWinningGame() {
   GameFourWins<MockWinningDice> game;
   std::ostringstream os;
   game.play(os);
   ASSERT_EQUAL("You won!\n",os.str());
}
```

- **should we also mock the ostream similarly?**

Call Tracing Mocks

- **We want also to count how often our dice are rolled. How to test this?**

```
struct MockWinningDice:DieInterface{
   int rollcounter;
   MockWinningDice():rollcounter(0){}
   int roll(){++rollcounter; return 4;}
};
void testWinningGame() {
   MockWinningDice d;
   GameFourWins game(d);
   std::ostringstream os;
   game.play(os);
   ASSERT_EQUAL("You won!\n",os.str());
   ASSERT_EQUAL(1,d.rollcounter);
   game.play(os);
   ASSERT_EQUAL(2,d.rollcounter);
}
```

Using C++ template Parameters for Mocking

- **C++ template parameters can be used for mocking without virtual member function overhead and explicit interface extraction.**
  - no need to pass object in as additional parameter
  - unfortunately no default template parameters for template functions (yet)
- **You can mock**
  - Member Variable Types
  - Function Parameter Types
- **Mocking without template inline/export need is possible through explicit instantiations**

Mockator Demo - Compile Seam using Templates

- **Extract Template Parameter**
  - part of CUTE plug-in
- **Use Template in Test**
  - introduce Mock object for DIE
    - create test double class...
    - add missing member function
    - implement test double code
- **Add Mock Object Support**
  - check for expected calls (C++11)
- **additional features**
  - dependency injection through
    - templates
    - abstract interface classes

```
void testWinningGame(){
    INIT_MOCKATOR();
    static std::vector<calls> allCalls(1);
    struct WinningDie
    {
        const int mock_id;
        WinningDie()
        :mock_id(++mockCounter_)
        {
            allCalls.push_back(calls());
            allCalls[mock_id].push_back(call{"WinningDie()"});
        }

        int roll() const
        {
            allCalls[mock_id].push_back(call{"roll() const"});
            return 4;
        }

    };
    GameT<WinningDie> game;
    ASSERTM("Winning Game", game.play());
    calls expected = {{"WinningDie()"}, {"roll() const"}};
    ASSERT_EQUAL(expected,allCalls[1]);
}
```

Mockator - Preprocessor Seam

- **mockator_malloc.h**

```
#ifndef MOCKATOR_MALLOC_H_
#define MOCKATOR_MALLOC_H_
#include <cstdlib >
int mockator_malloc(size_t size, const char *fileName, int lineNumber);
#define malloc(size_t size) mockator_malloc((size),__FILE__,__LINE__)
#endif
```

- **mockator_malloc.c**

```
#include "mockator_malloc.h"
#undef malloc
int mockator_malloc(size_t size, const char * fileName, int lineNumber) {
//TODO your tracing code here
   return malloc(size);
}
```

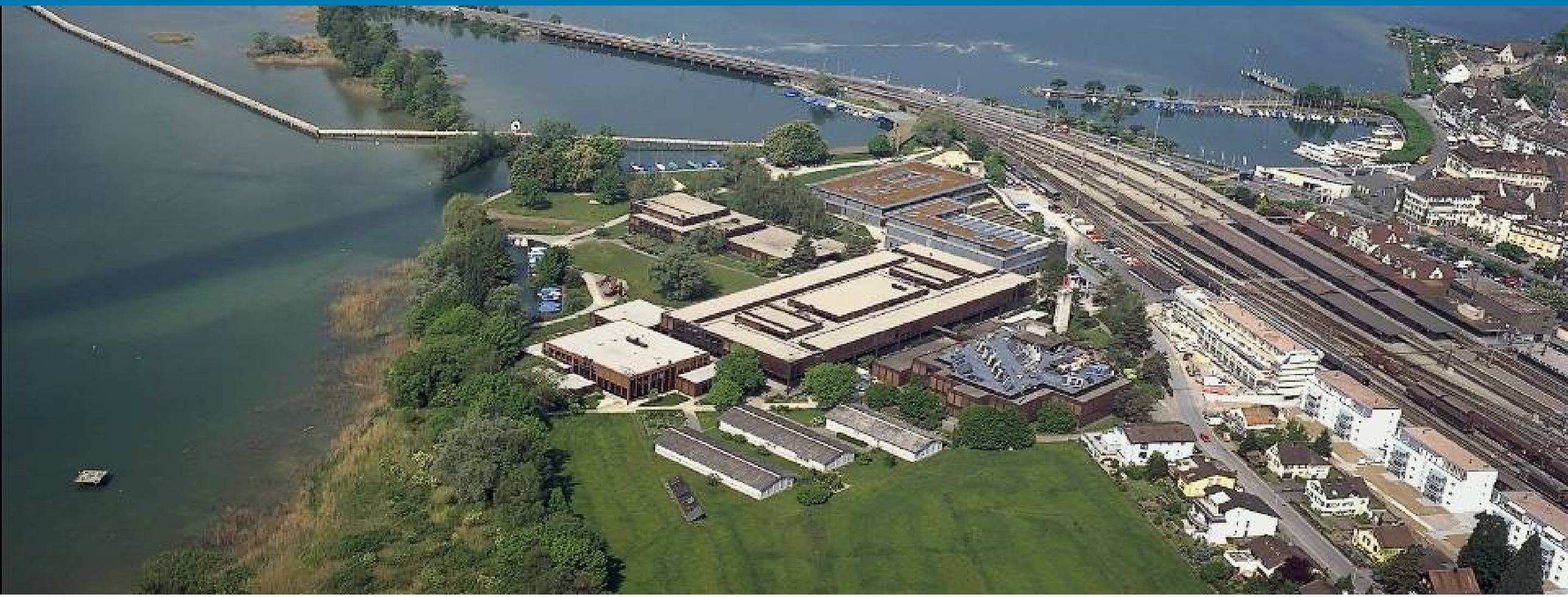
- **Generating trace functions like this one is easy: Ctrl+Alt+T (on Linux)**
  - Mockator passes mockator_malloc.h to the GNU compiler by using its -include option

Mockator - Link Seam - Shadowing & Wrapping Library Functions

- **Shadowing on Linux only using GCC Linker**
- **Allows wrapping of functions in shared libraries**
  - Both Linux and MacOS X supported
- **Mockator does all the hard work like**
  - creating a shared library project,
  - adding dependencies to the dl library,
  - creating run-time configurations with the following env values, etc.,
    - Linux: LD PRELOAD=libName.so MacOS X:
    - DYLD FORCE FLAT NAMESPACE=1
    - DYLD INSERT LIBRARIES=libName.dylib

```
int foo(int i) {
  static void *gptr = nullptr;
  if(!gptr)  gptr = dlsym(RTLD_NEXT,"_Z3fooi");
  typedef int (*fptr)(int);
  fptr  my_fptr = reinterpret_cast<fptr>(gptr);
  // TODO  put your code here
  return my_fptr(i);
}
```

Questions ?



- http://cute-test.com - Unit Testing
- http://mockator.com - Seams and Mocks
- http://linticator.com - Lint Integration
- http://includator.com - #include optimization
- http://sconsolidator.com - SCons integration

Have Fun with TDD and Refactoring!

Or get it all at once in a modern C++ IDE

Cevelop

Your C++ code deserves it

Download IDE at:
www.cevelop.com