Informatik / Computer Science

TDD and Mock Objects

with Eclipse, CUTE und Mockator

IFS INSTITUTE FOR SOFTWARE

Prof. Peter Sommerlad
Director IFS Institute for Software

Agile PT, Porto June 23rd 2012

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Contact Info: Prof. Peter Sommerlad
phone: +41 55 222 4984
email: ifs@hsr.ch

Linticator

see more on http://linticator.com

Linticator gives you immediate feedback on programming style and common programmer mistakes by integrating Gimpel Software's popular PC-lint and FlexeLint static analysis tools into Eclipse CDT.

PC-lint and FlexeLint are powerful tools, but they are not very well integrated into a modern developer's workflow. **Linticator** brings the power of Lint to the Eclipse C/C++ Development Tools by fully integrating them into the IDE. With Linticator, you get continuous feedback on the code you are working on, allowing you to write better code.

- **Automatic Lint Configuration**
  Lint's configuration, like include paths and symbols, is automatically updated from your Eclipse CDT settings, freeing you from keeping them in sync manually.
- **Suppress Messages Easily**
  False positives or unwanted Lint messages can be suppressed directly from Eclipse, without having to learn Lint's inhibition syntax–either locally, per file or per symbol.
- **Interactive "*Linting*" and Information Display**
  Lint is run after each build or on demand, and its findings are integrated into the editor by annotating the source view with interactive markers, by populating Eclipse's problem view with Lint's issues and by linking these issues with our *Lint Documentation View*.
- **Quick-Fix Coding Problems**
  Linticator provides automatic fixes for a growing number of Lint messages, e.g, making a reference-parameter const can be done with two keystrokes or a mouse-click.

Register at http://linticator.com if you want to try it for 30 days or order via email. Linticator is available for Eclipse CDT 3.4 (Ganymede), 3.5 (Helios) and later. It is compatible with Freescale CodeWarrior.

Pricing for Linticator is **CHF 500.-** per user (non-floating license). A maintenance contract that is required for updates costs 20% of license fee per year. The compulsory first maintenance fee includes 6 month of free updates.
Orders, enquiries for multiple, corporate or site licenses are welcome at ifs@hsr.ch.

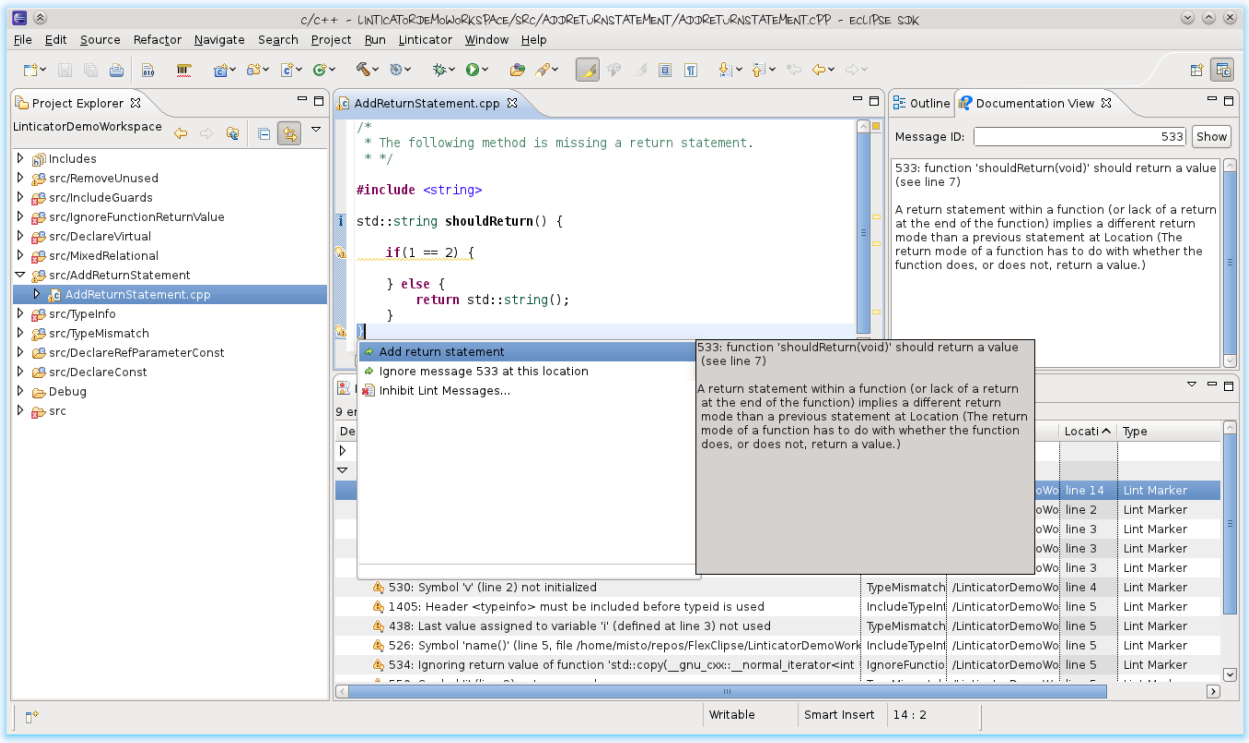
Linticator requires a corresponding PC-Lint (Windows) or FlexeLint license per user.

IFS Institute for Software

IFS is an institute of HSR Rapperswil, member of FHO University of Applied Sciences Eastern Switzerland.

In January 2007 IFS became an associate member of the Eclipse Foundation.

The institute manages research and technology transfer projects of four professors and hosts about a dozen assistants and employees.
http://ifs.hsr.ch/



IFS INSTITUTE FOR SOFTWARE

Contact Info: Prof. Peter Sommerlad
phone: +41 55 222 4984
email: ifs@hsr.ch

Includator

#include Structure Analysis and Optimization for C++ for Eclipse CDT

The **Includator** plug-in analyzes the dependencies of C++ source file structures generated by #include directives, suggests how the *#include structure* of a C++ project can be optimized and performs this optimization on behalf of the developer. The aim of these optimizations is to improve code readability and quality, reduce coupling and thus reduce duration of builds and development time of C++ software.

see more on
http://includator.com
to be released 2012

Includator Features

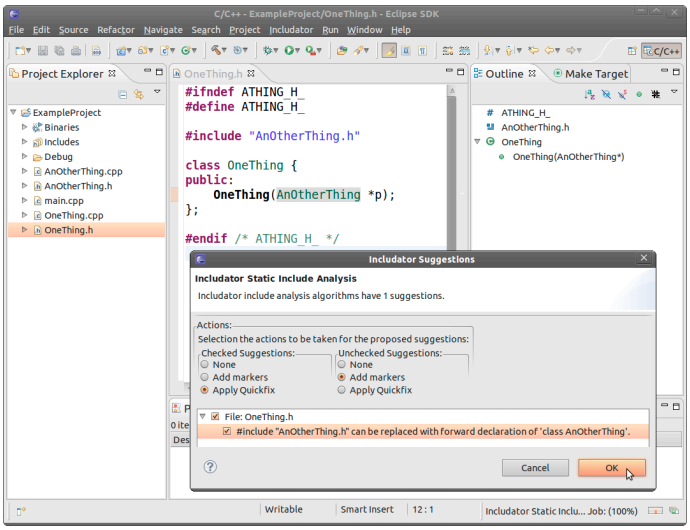
- **Find unused includes**
  Scans a single source file or a whole project for superfluous #include directives and proposes them to be removed. This also covers the removal of #include directives providing declarations that are (transitively) reachable through others.
- **Directly include referenced files**
  Ignores transitively included declarations and proposes to #include used declarations directly, if they are not already included. This provides an “include-what-you-use” code structure.
- **Organize includes**
  Similar to Eclipse Java Development Tool's *Organize imports* feature for Java. Adds missing #include directives and removes superfluous ones.
- **Replace includes with forward declarations**
  Locates #include directives for definitions that can be omitted, when replacing them with corresponding forward declarations instead. This one is useful for minimizing #includes and reducing dependencies required in header files.
- **Static code coverage**
  Marks C++ source code as either *used*, *implicitly used* or *unused* by transitively following C++ elements’ definitions and usages. This helps to trim declarations and definitions not used from your source code. In contrast to dynamic code coverage, such as provided by our CUTE plug-in (http://cute-test.com) it allows to determine required and useless C++ declarations and definitions instead of executed oder not-executed statements.
- **Find unused files**
  Locates single or even entangled header files that are never included in the project’s source files.

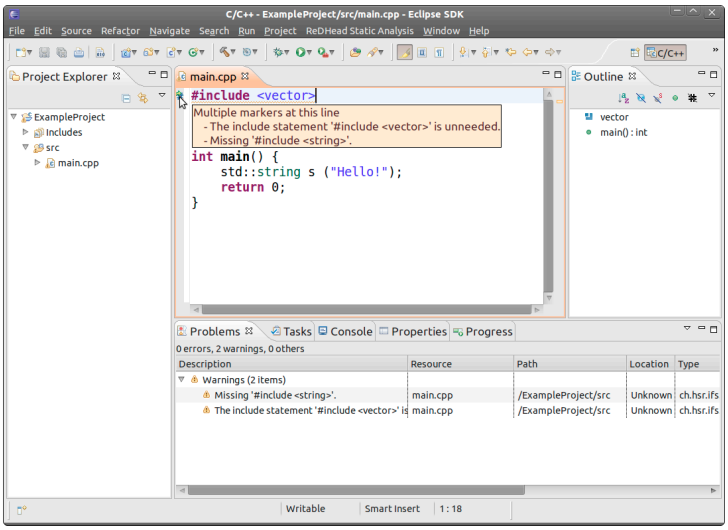
Individual licensing TBA.
Free 30-day trial licenses available by registering at http://includator.com/account/register.

Enquiries for corporate or site licenses are welcome at ifs@hsr.ch.

User Feedback and Participation

Includator is in beta testing. Register at http://includator.com if you are interested in a 30-day trial license or if you want to be notified about Includator’s release.





IFS INSTITUTE FOR SOFTWARE

Contact Info: Prof. Peter Sommerlad
+41 55 222 4984
email: ifs@hsr.ch

Green Bar for C++ with CUTE

Get Swiss Quality CUTE

eclipse FOUNDATION MEMBER

Eclipse plug-in for C++ unit testing with CUTE

Automated unit testing improves program code quality, even under inevitable change and refactoring. As a side effect, unit tested code has a better structure. Java developers are used to this because of JUnit and its tight integration into IDEs like Eclipse. We provide the modern C++ Unit Testing Framework CUTE (C++ Unit Testing Easier) and a corresponding Eclipse plug-in. The **free CUTE plug-in** features:

- wizard creating test projects (including required framework sources)
- test function generator with automatic registration
- detection of unregistered tests with quick-fix for registration
- test navigator with green/red bar (see screen shots to the right)
- diff-viewer for failing tests (see screen shot down to the right)
- code coverage view with gcov (see screen shot below)

Support for TDD and Mock Objects (Mockator)

The CUTE plug-in incorporates support for Test-Driven Development (TDD) in C++. It will support code generation for test doubles and mock objects in spring 2012:

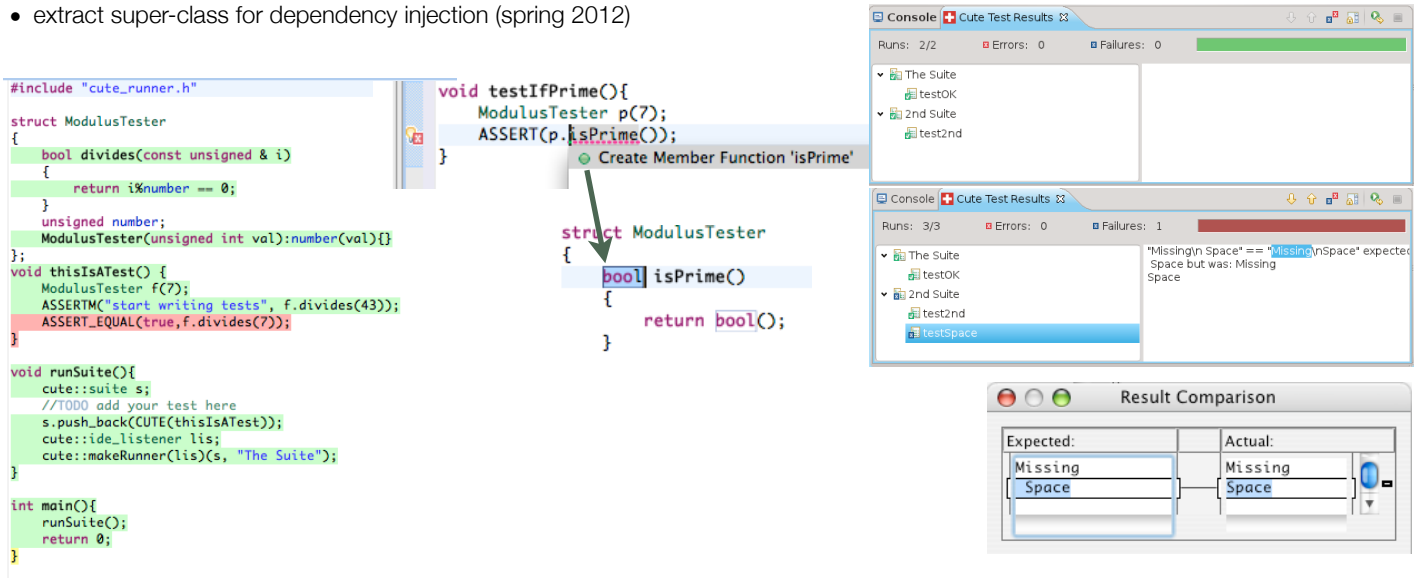
- create unknown function, member function, variable, or type from its use in a test case as a quick-fix (see screen shots below)
- move function or type from test implementation to new header file. (More MOVE Refactorings are becoming available in spring 2012)
- toggle function definitions between header file and implementation file, for easier change of function signature, including member functions (now part of CDT itself)
- extract template parameter for dependency injection, aka instrumenting code under test with a test stub through a template argument (instead of Java-like super-class extraction)
- automatic generation of test stubs and mock classes from test cases (spring 2012)
- extract super-class for dependency injection (spring 2012)

IFS Institute for Software

IFS is an Institute of HSR Rapperswil, member of FHO University of Applied Sciences Eastern Switzerland.
In January 2007 IFS became an associate member of the Eclipse Foundation.
The institute manages research and technology transfer projects of four professors and hosts a dozen assistants and employees.
http://ifs.hsr.ch/

Eclipse update site for installing the free CUTE plug-in:

http://cute-test.com/updatesite



IFS INSTITUTE FOR SOFTWARE

Contact Info: Prof. Peter Sommerlad
+41 55 222 4984
email: ifs@hsr.ch

SConsolidator

More information:

http://SConsolidator.com/

Install the **free SConsolidator plug-in** from the following Eclipse update site:

http://SConsolidator.com/update

SConsolidator has been successfully used to import the following SCons-based projects into Eclipse CDT:

- MongoDB
- Blender
- FreeNOS
- Doom 3
- COAST (http://coast-project.org)

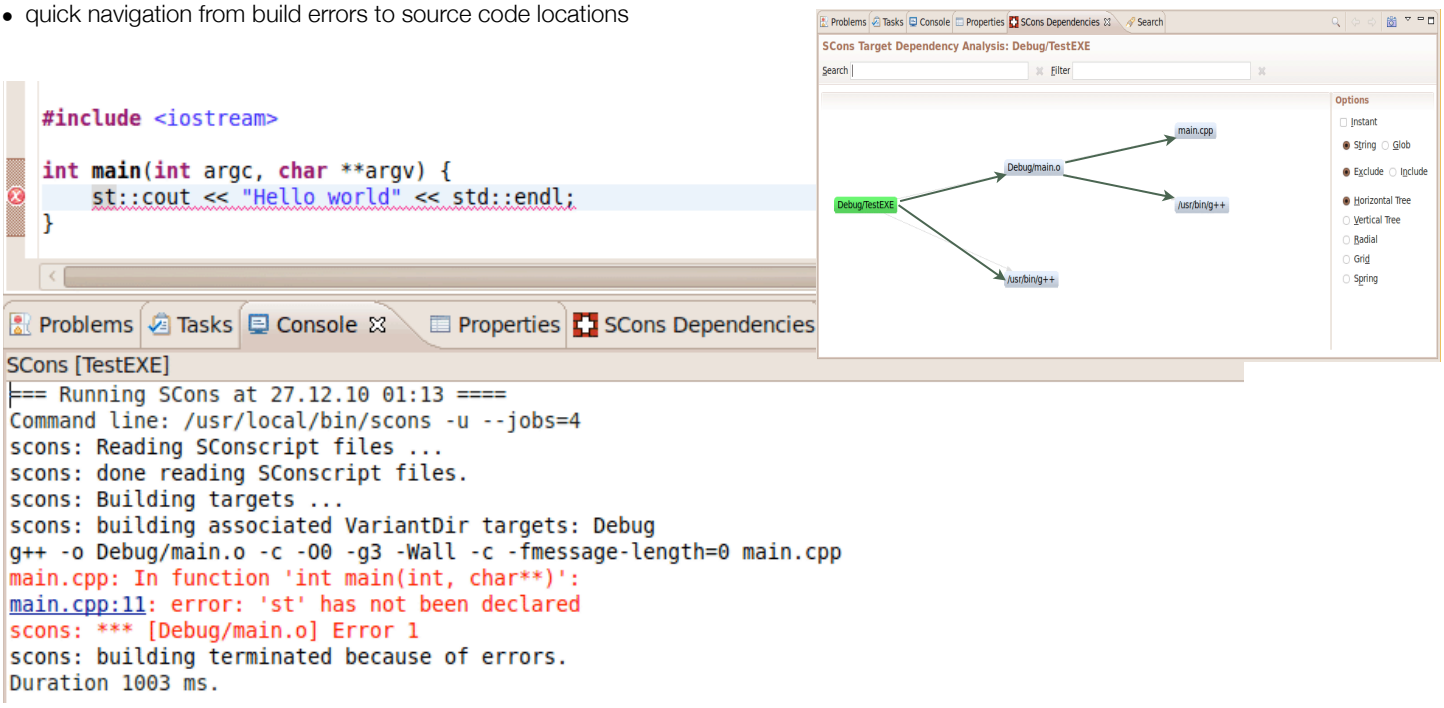
Eclipse CDT plug-in for SCons

SCons (http://www.SCons.org/) is an open source software build tool which tries to fix the numerous weaknesses of *make* and its derivatives. For example, the missing automatic dependency extraction, make's complex syntax to describe build properties and cross-platform issues when using shell commands and scripts. *SCons* is a self-contained tool which is independent of existing platform utilities. Because it is based on Python a SCons user has the full power of a programming language to deal with all build related issues.

However, maintaining a SCons-based C/C++ project with Eclipse CDT meant, that all the intelligence SCons puts into your project dependencies had to be re-entered into Eclipse CDT's project settings, so that CDT's indexer and parser would know your code's compile settings and enable many of CDT's features. In addition, SCons' intelligence comes at the price of relatively long build startup times, when SCons (re-) analyzes the project dependencies which can become annoying when you just fix a simple syntax error.

SConsolidator addresses these issues and provides tool integration for SCons in Eclipse for a convenient C/C++ development experience. The **free** plug-in features:

- conversion of existing CDT managed build projects to SCons projects
- **import of existing SCons projects into Eclipse with wizard support**
- SCons projects can be managed either through CDT or SCons
- interactive mode to quickly build single source files speeding up round trip times
- a special view for a convenient build target management of all workspace projects
- graph visualization of build dependencies with numerous layout algorithms and search and filter functionality that enables debugging SCons scripts.
- quick navigation from build errors to source code locations



Peter Sommerlad
peter.sommerlad@hsr.ch

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

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

Software Quality

Software Quality

Software Quality
"An Elephant in the Room"

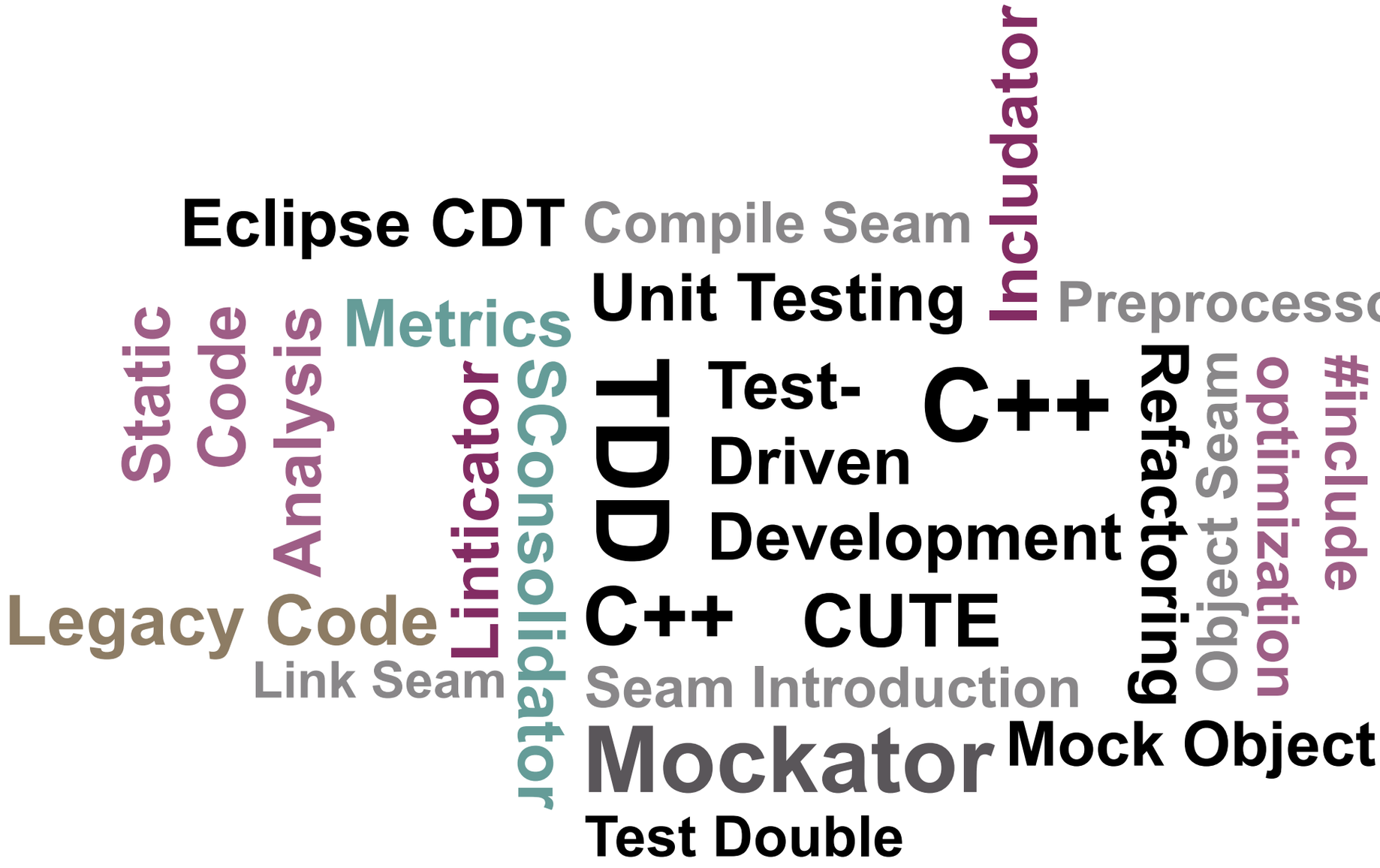
- **classic approach:**
  - manual testing after creation

But:
Small Cute Things

Grow to become larger Problems!

Overview

ScrumTable

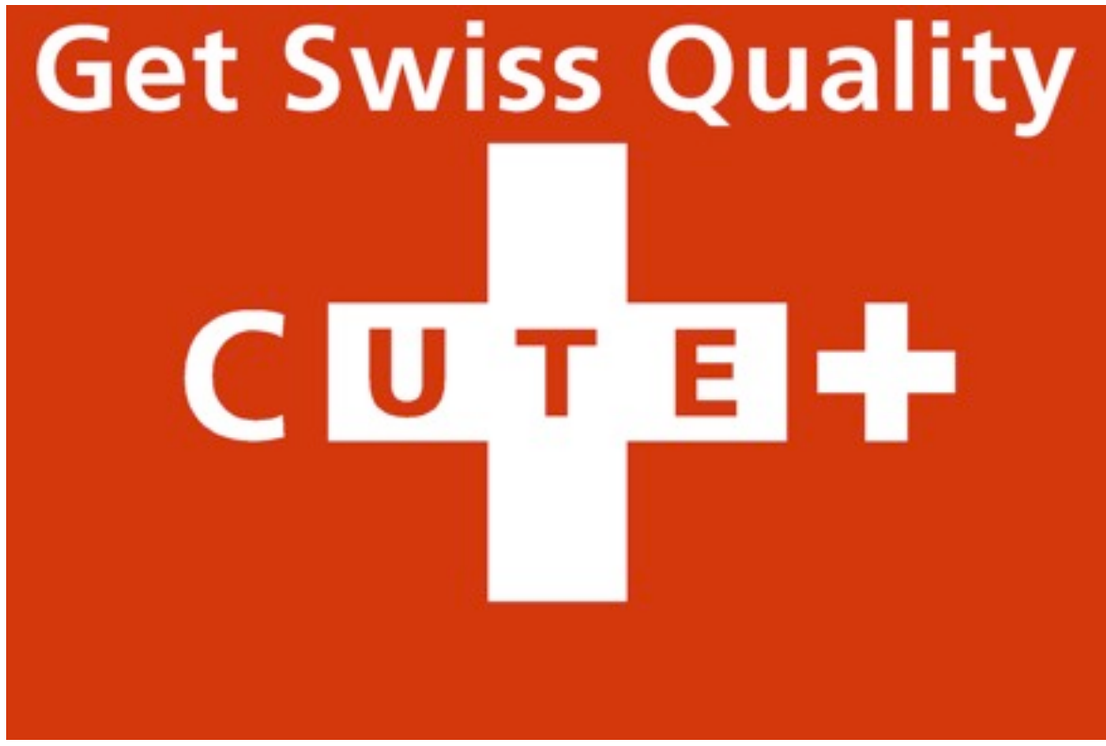


Scala Refactoring

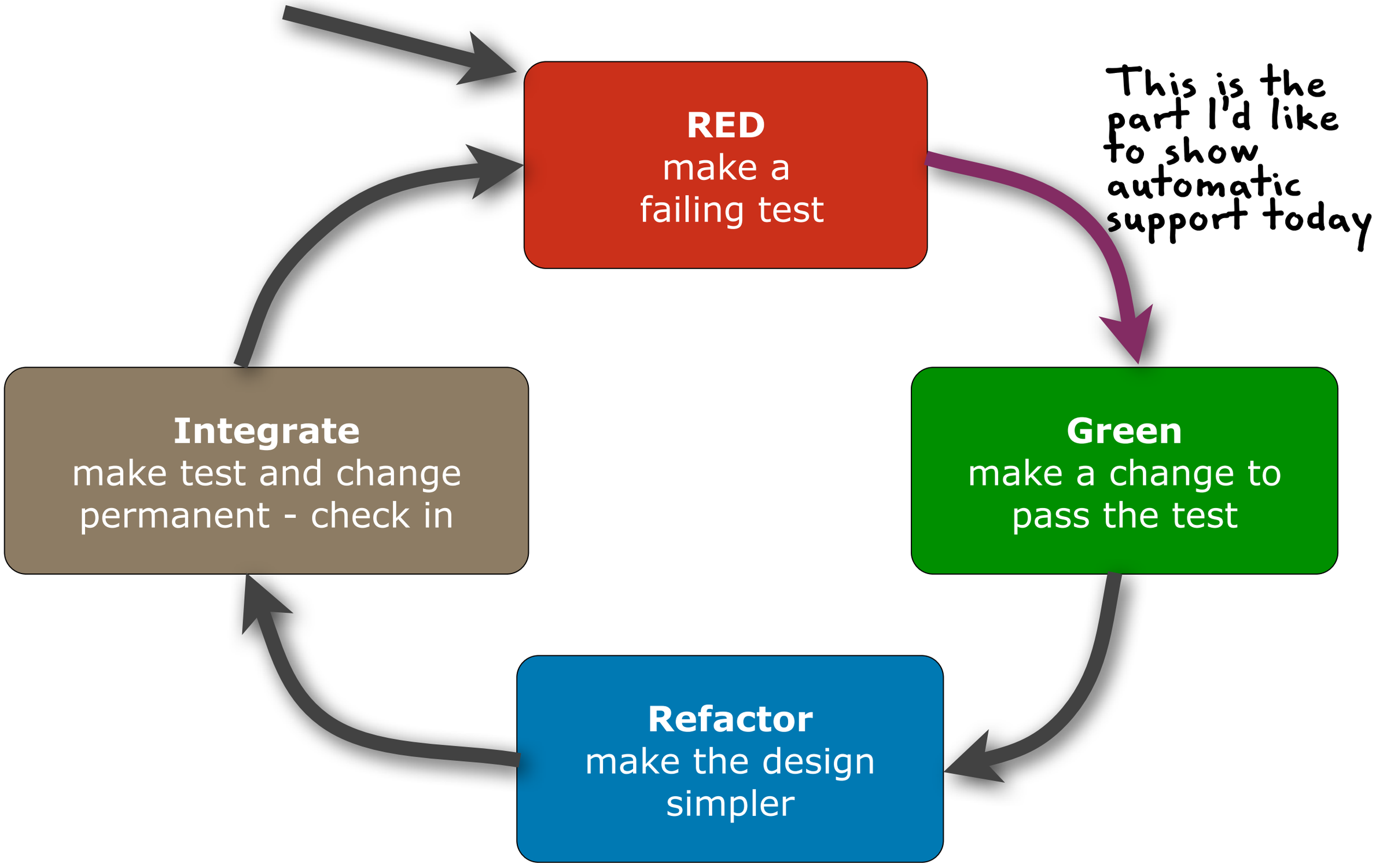
C++ Unit Testing with CUTE in Eclipse CDT

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



TDD Cycle



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

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

- **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

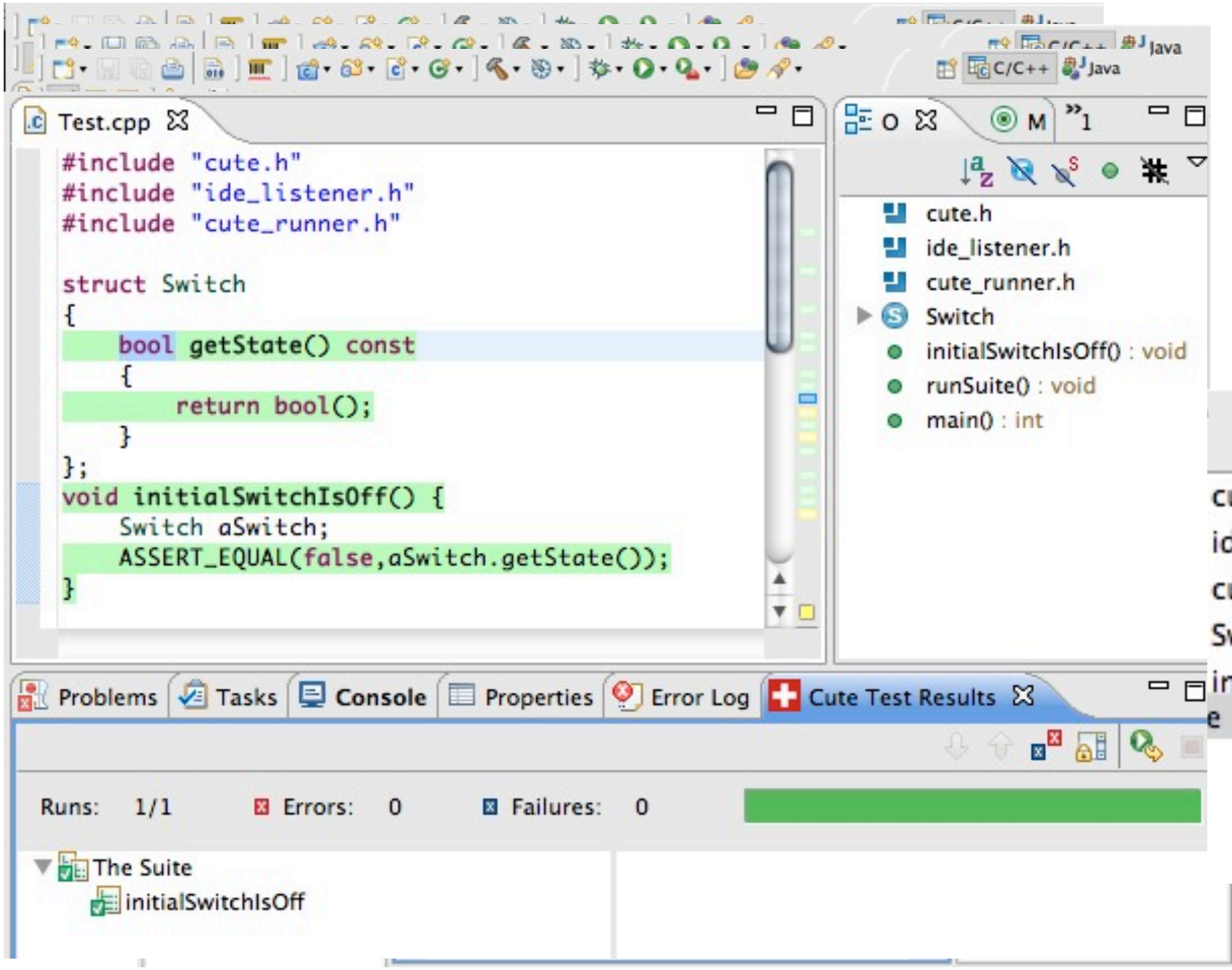
- **Simulation of a Switch (class)**
  - alternative possible, if this seems too simple and time permits

THE LIST FOR CLASS SWITCH
1. CREATE,GETSTATE → OFF
2. TURNON,GETSTATE → ON
3. TURNOFF,GETSTATE → OFF

Demo 

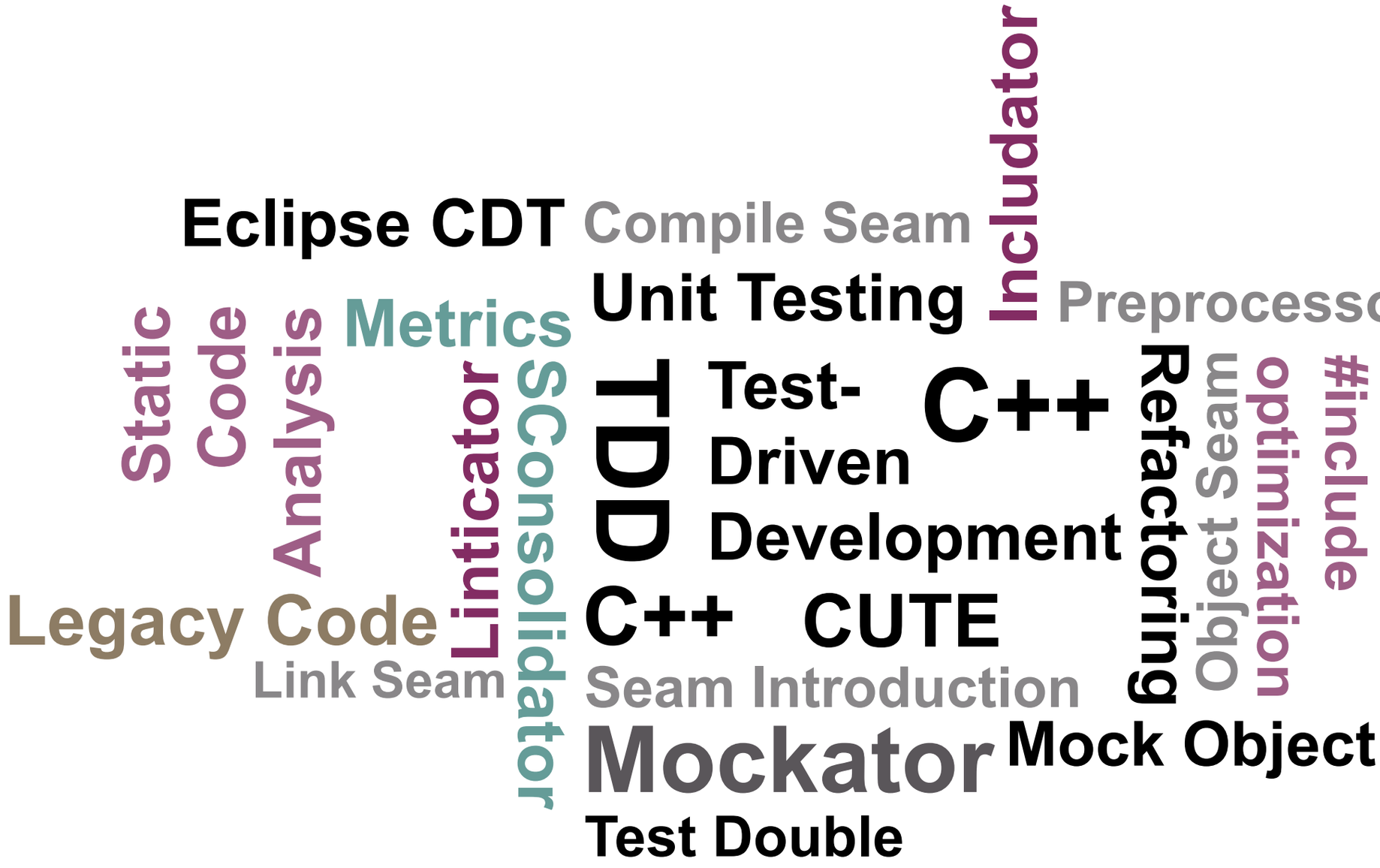
Demo

- **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...**



Overview

ScrumTable



Scala Refactoring

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

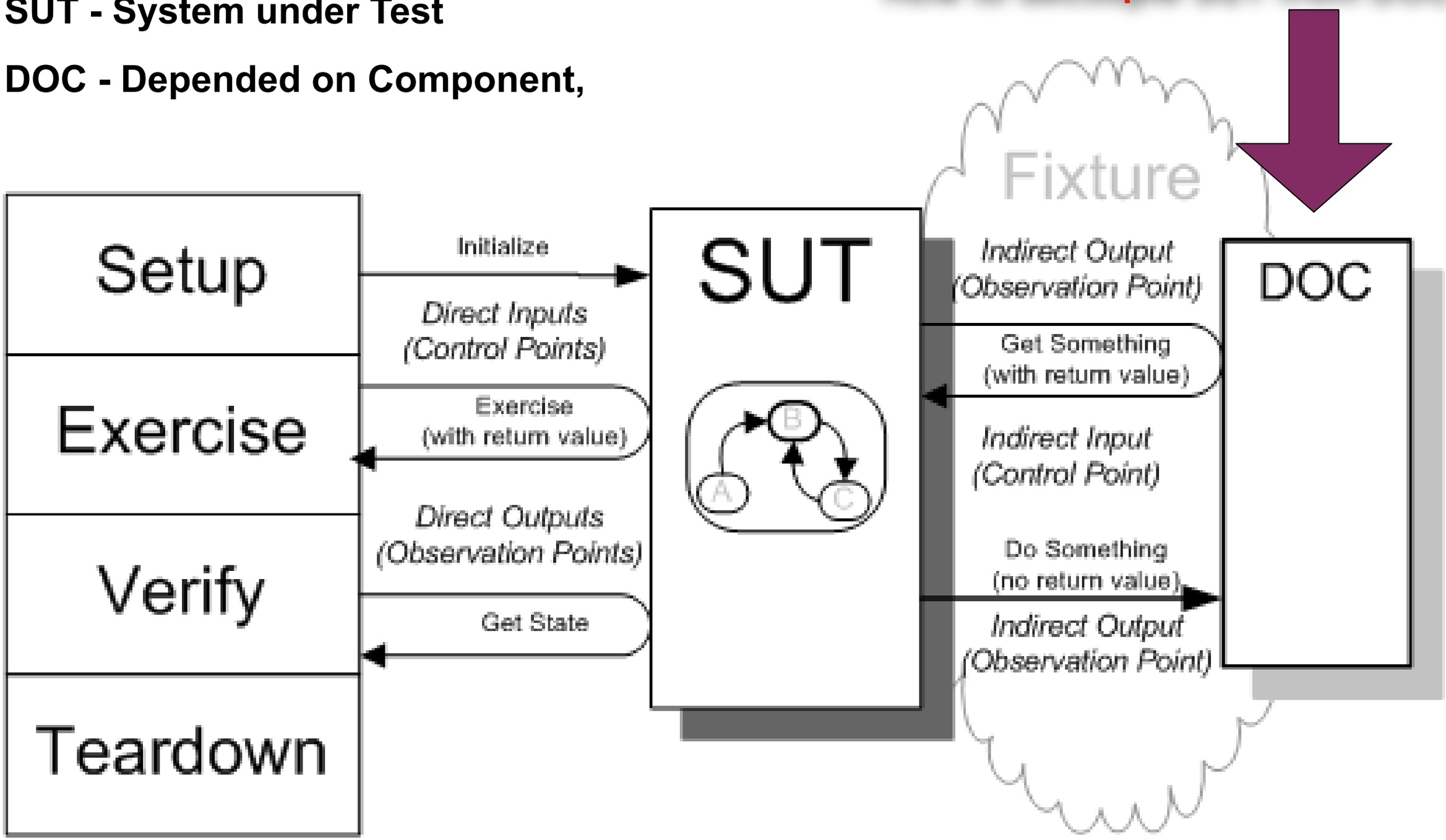
HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

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**

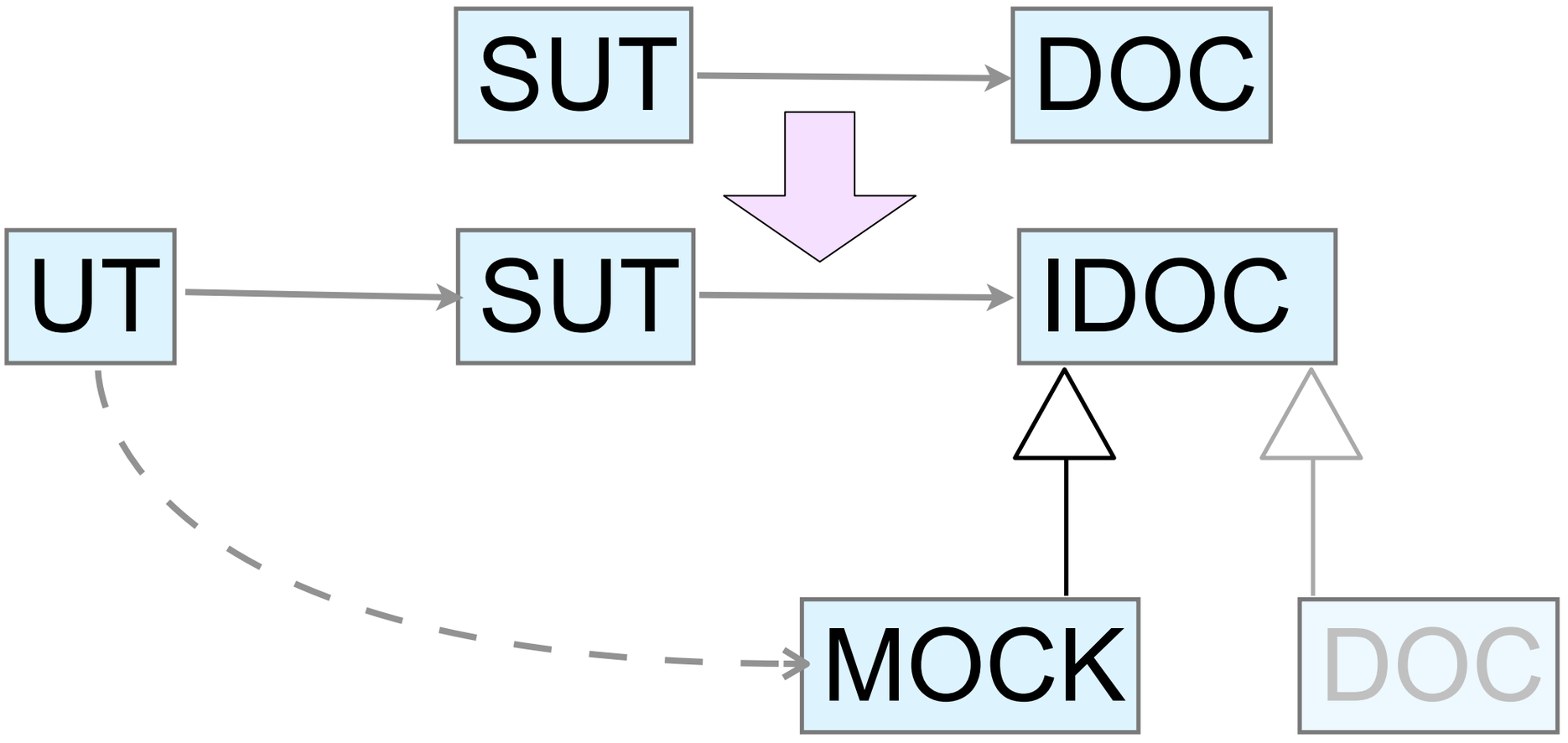
Seams!

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

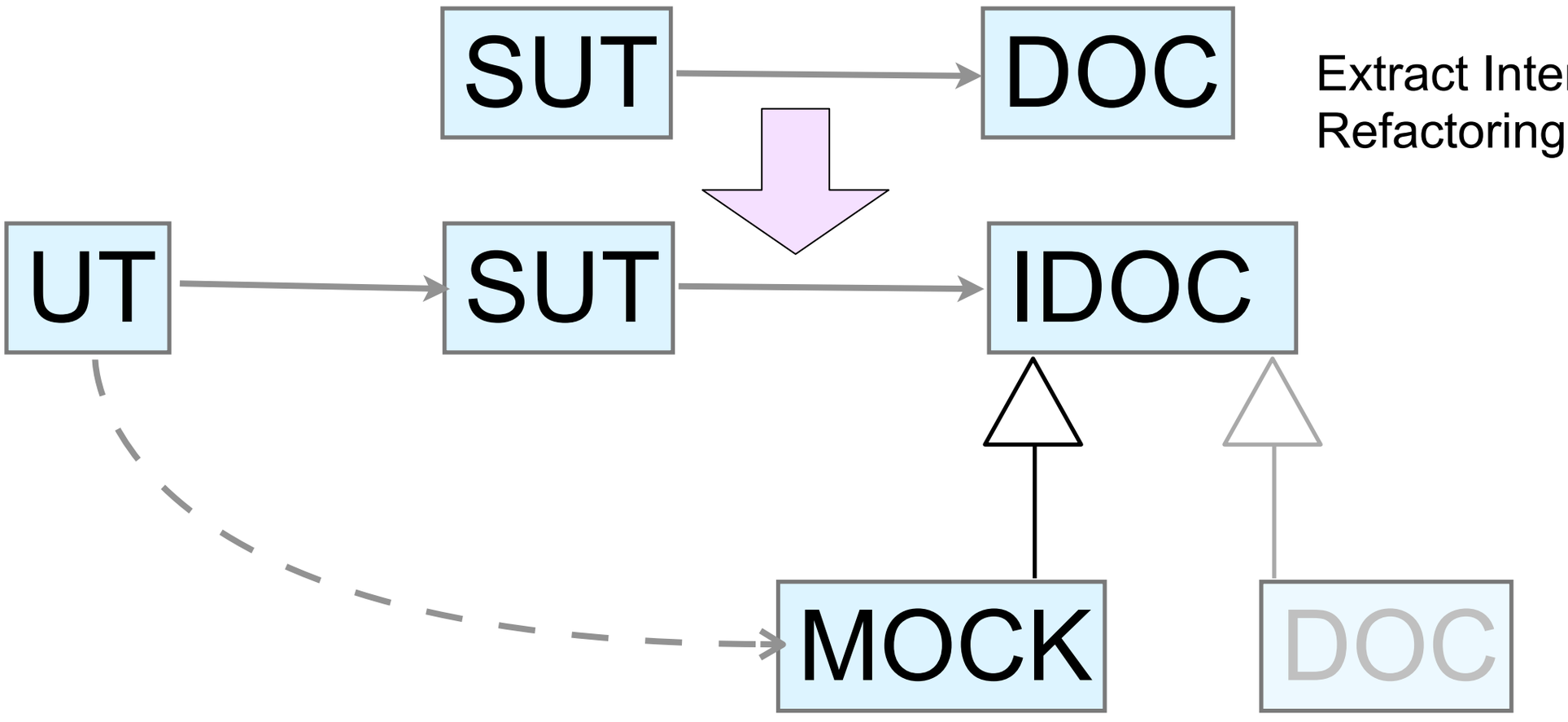
- **classic inheritance based mocking**
  - extract interface for DOC -> IDOC
  - make SUT use IDOC
  - create MOCK implementing IDOC and use it in UT



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

Introduce an Object Seam

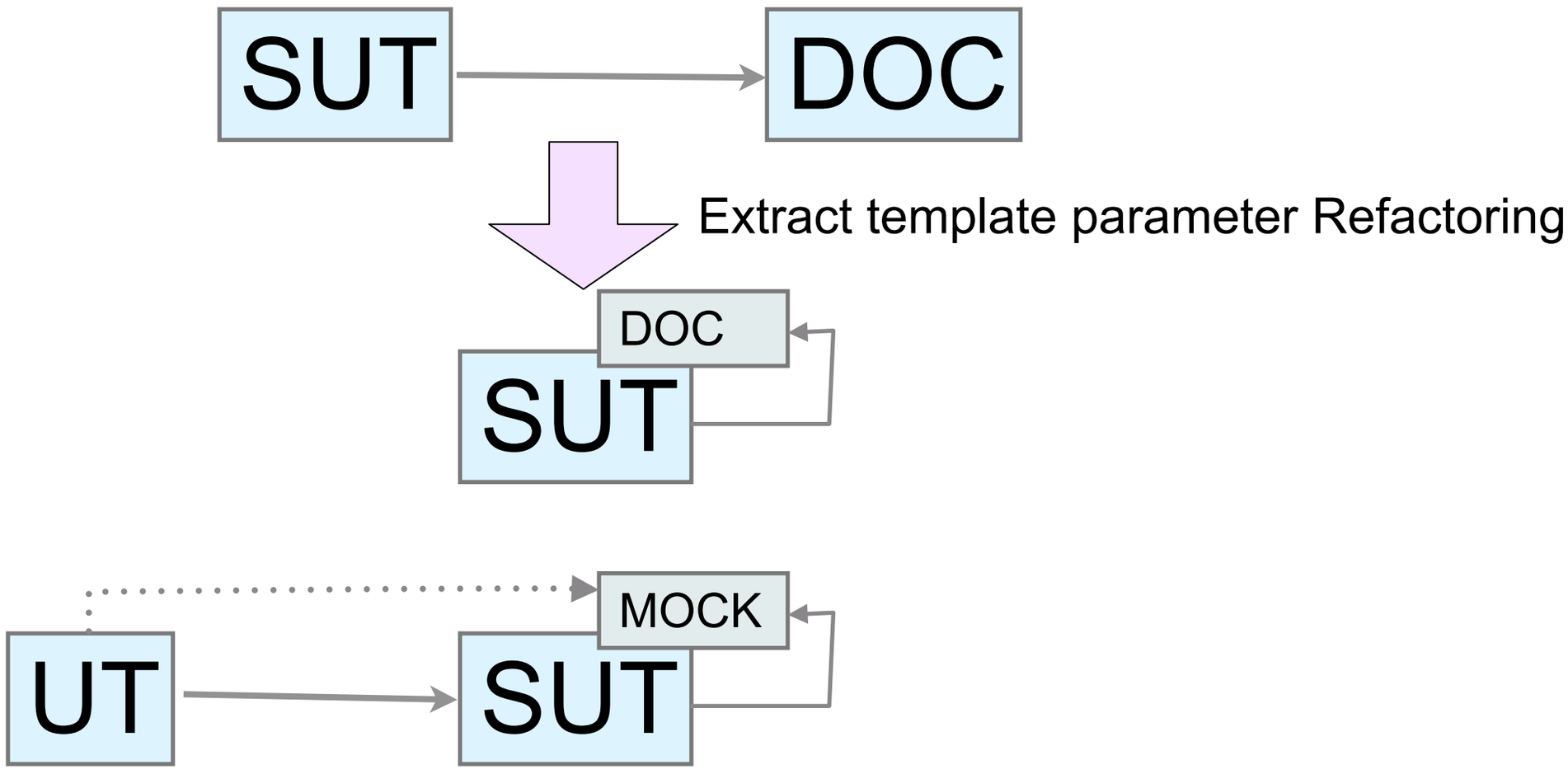
- **classic inheritance based mocking**
  - extract interface for DOC -> IDOC
  - make SUT use IDOC, edit constructor
  - create MOCK implementing IDOC and use it in UT



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

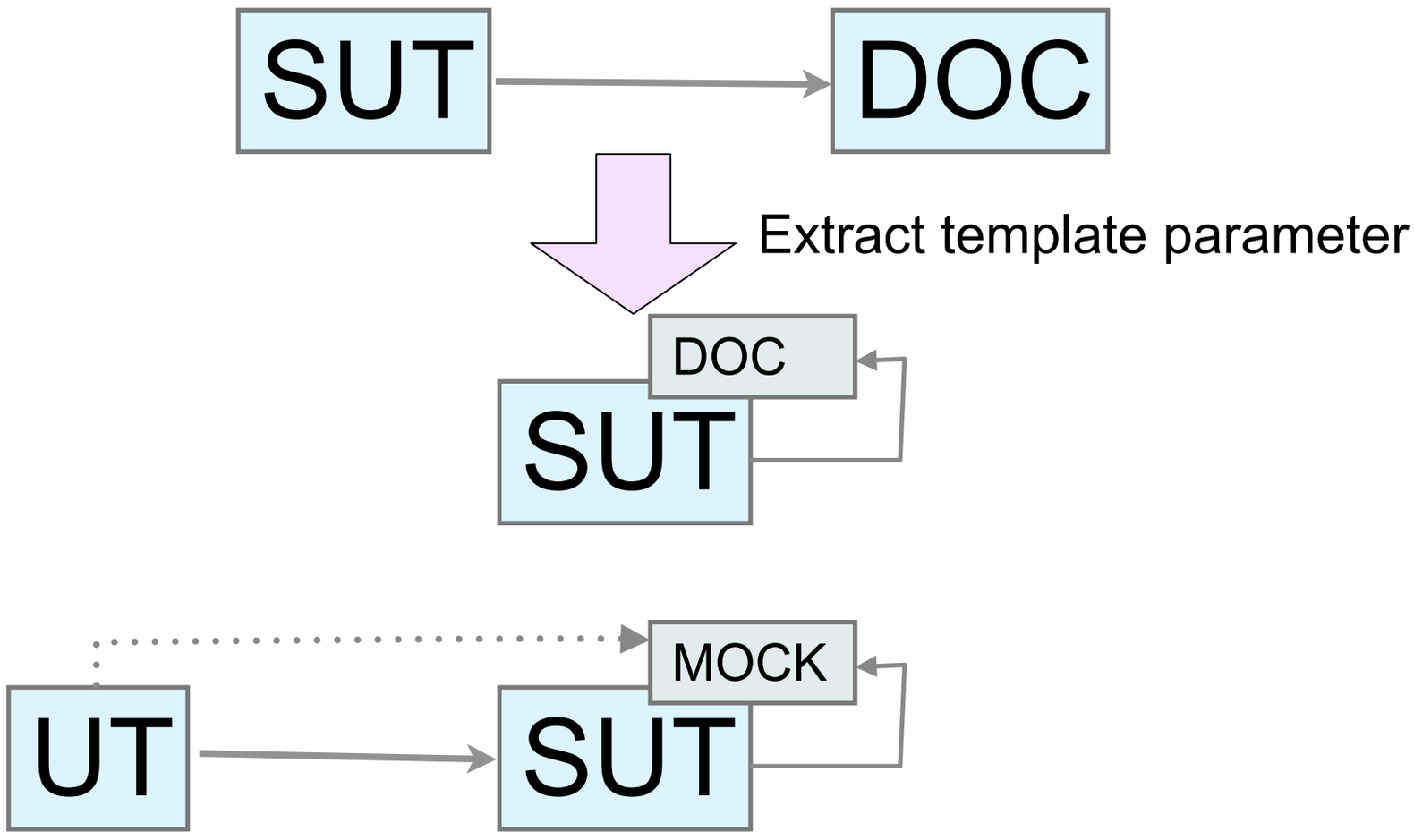
Introduce a Compile Seam

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



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

Refactoring for Mocks in C++

Variations of Mock Objects classics

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?

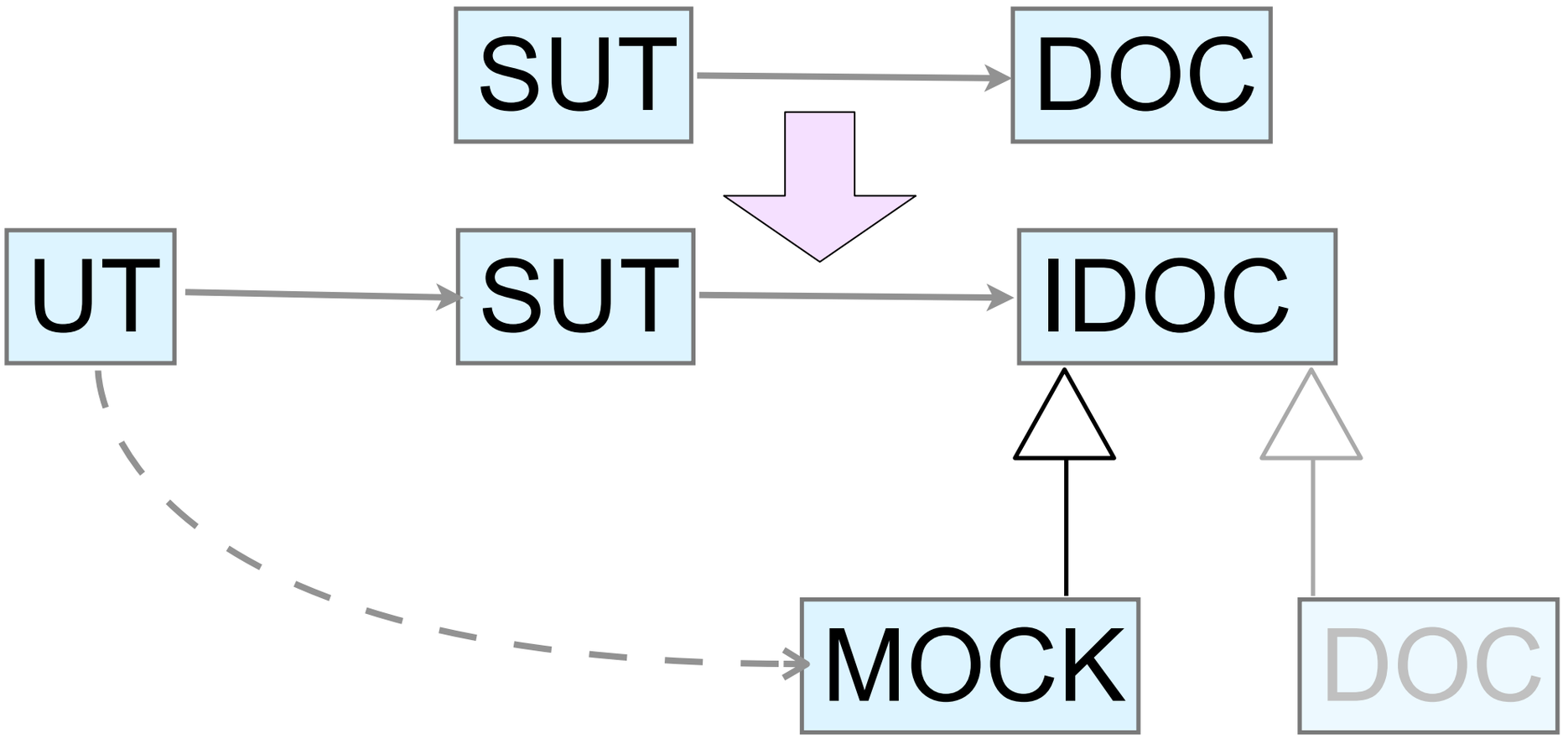
- **Simpler Tests and Design**
  - especially for external dependencies
  - promote interface-oriented design
- **Independent Testing of single Units**
  - isolation of unit under testing
  - or for not-yet-existing units
- **Speed of Tests**
  - no external communication (e.g., DB, network)
- **Check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

Interface-oriented Mock

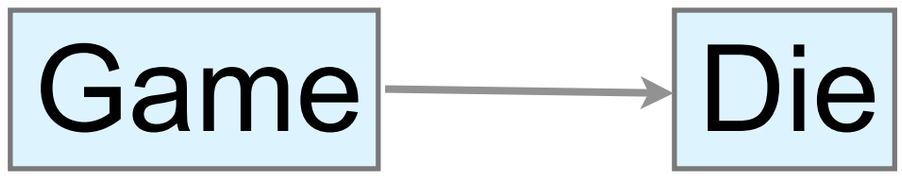
- **classic inheritance based mocking**
  - extract interface for DOC -> IDOC
  - make SUT use IDOC
  - create MOCK implementing IDOC and use it in UT



  - 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?**

- **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?**

Simulation Mock
Test it

- **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 Paramter" 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);
}
```

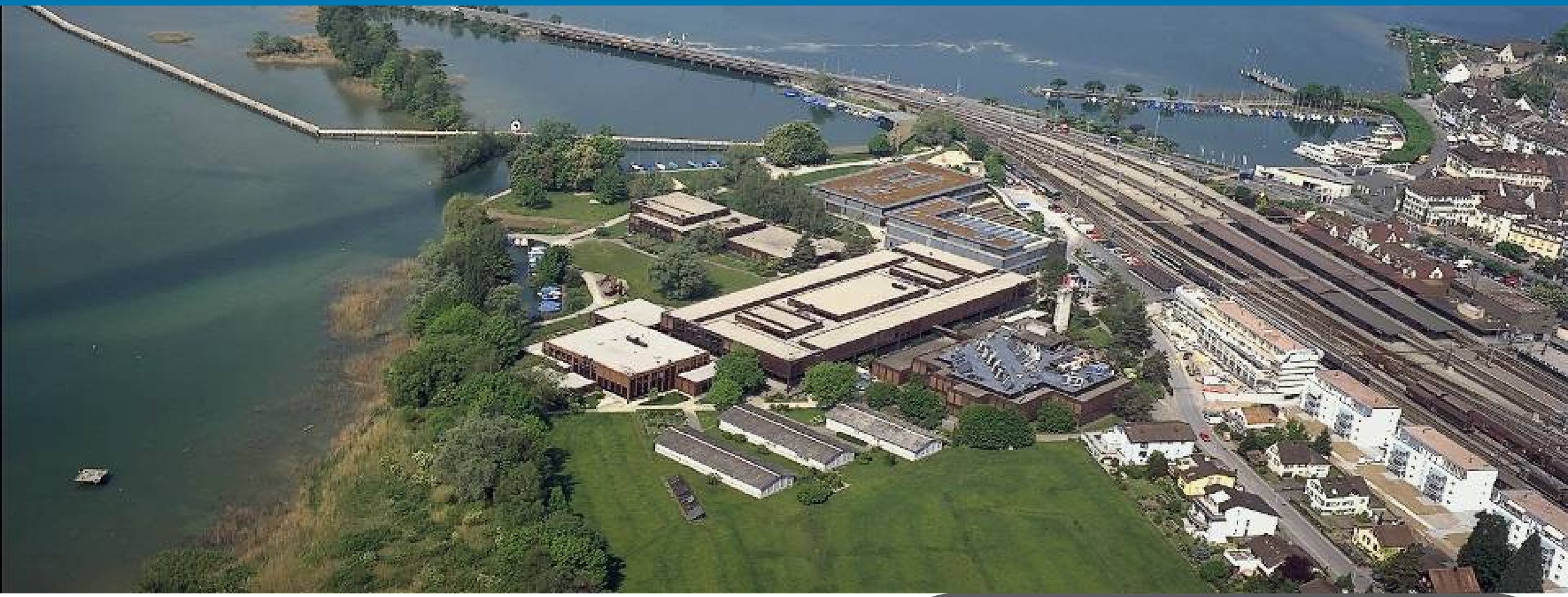
- **Generating trace functions like this one is easy: Ctrl+Alt+T (on Linux)**
  - Mockator passes mockator_malloc.h to the GNU compiler by using its -include option

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** (Mockator will be here 3.Q2012)
- **http://linticator.com http://includator.com**
- **http://sconsolidator.com** (M. Rüegg as well)
- **peter.sommerlad@hsr.ch http://ifs.hsr.ch**

Have Fun with TDD Mockator and Refactoring!