Getting Legacy C/C++ under Tests

Peter Sommerland and Michael Rüegg

ACCU 2013 / Bristol, UK / April 13, 2013

Who are we?

Figure : University of Applied Sciences Rapperswil, Switzerland

Software Quality

An elephant in the room

Manual testing or no testing at all

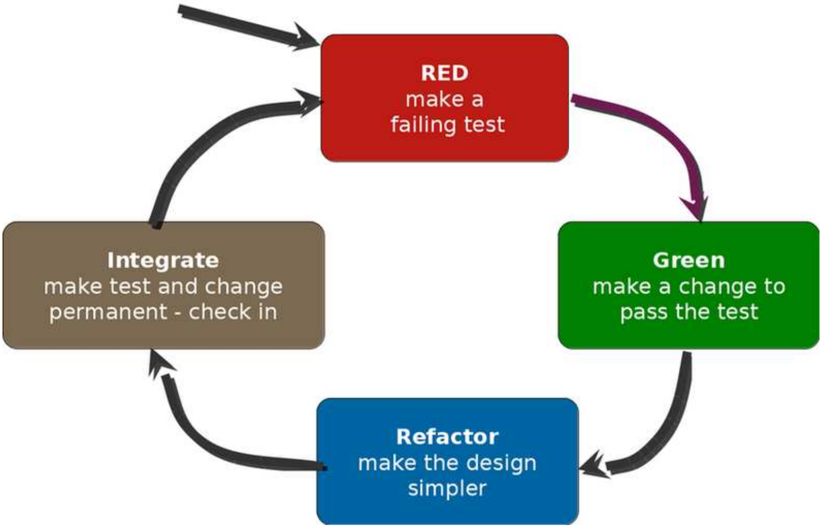
Manual testing or no testing at all

But: Small cute things. . .

Grow to become larger problems!

Part I - TDD with CUTE

TDD Cycle

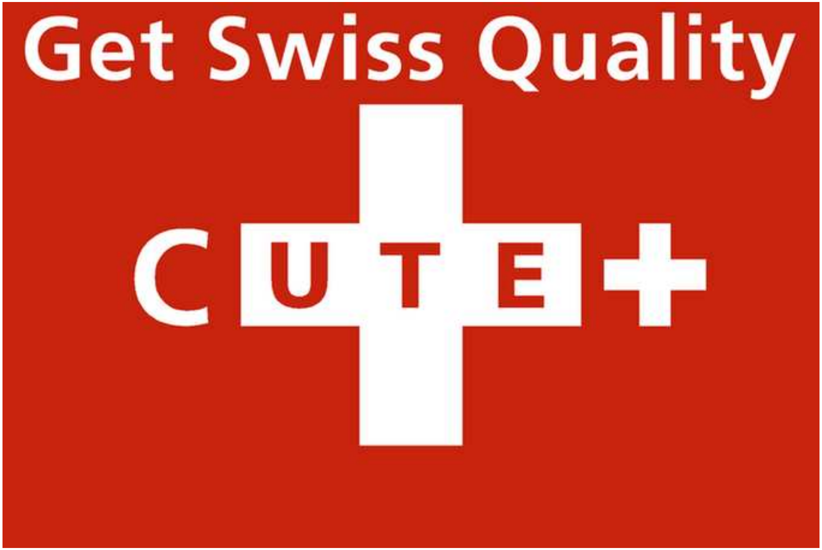

RED
make a failing test
Green
make a change to pass the test
Refactor
make the design simpler
Integrate
make test and change permanent - check in

TDD with C++ can work! => CUTE

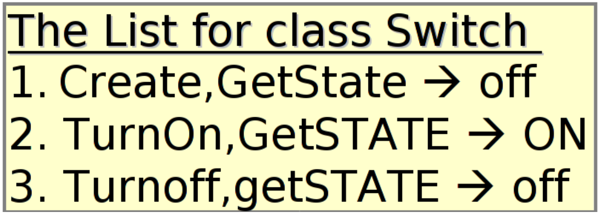
- Unit testing library for C/C++ with tool support for Eclipse C/C++ Development Tooling project (**CDT**)
- Simple to use: **a test is a function**
- Designed to be used with IDE support
- Deliberate minimization of `#define` macro usage => macros make life harder for C/C++ IDEs

Where can I get it?

- Grab it for free from `http://cute-test.com`



TDD Demo: Simulation of a switch



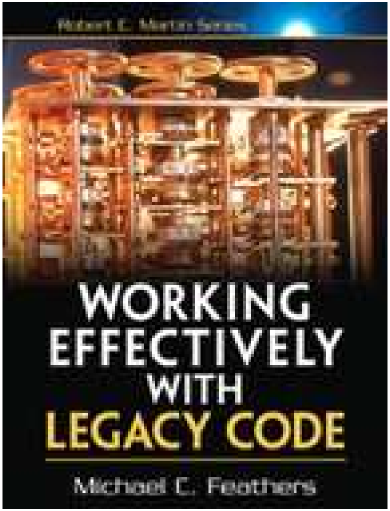
The List for class Switch

1. Create,GetState → off
2. TurnOn,GetSTATE → ON
3. Turnoff,getSTATE → off

Demo

Part II - Getting legacy code under test with seams

What is legacy code?



- Definition of Michael Feathers: Code *without* unit tests

Dependencies and the legacy code dilemma

- Perils of dependencies in software
- **Triggers** for changing existing code (Feathers): adding new functionality, fixing a bug, applying refactorings and code optimisations
- The Legacy code **dilemma**
- We do **not** want to **change** the code **inline**
- There is hope: **Seams** - but they are hard and cumbersome to create by hand! => **refactorings and IDE support necessary**

Our contribution: Mockator

- **Refactorings** and **toolchain support** for achieving seams in C/C++
- **Eclipse plug-in** for (CDT)
- **C++ mock object library** (header-only) with Eclipse support

What is a Seam?

- Term introduced by Michael Feathers in **Working Effectively With Legacy Code**: *"A place in our code base where we can alter behaviour without being forced to edit it in that place."*
- **Inject dependencies from outside** to improve the design of existing code and to enhance testability
- Every seam has an **enabling point**: the place where we can choose between one behaviour or another
- Different kinds of seam types in C++: **object**, **compile**, **preprocessor** and **link seams**

4 Phases, SUT, DOC

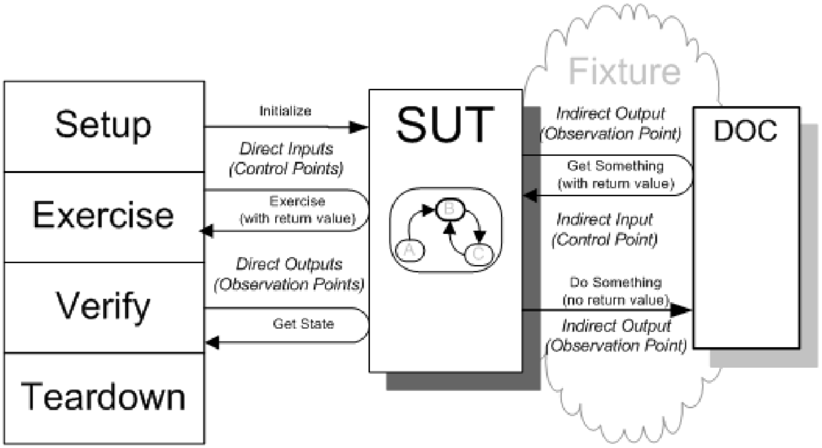

Setup
Exercise
Verify
Teardown
Initialize
Direct Inputs (Control Points)
Exercise (with return value)
Direct Outputs (Observation Points)
Get State
SUT
A
B
C
Fixture
Indirect Output (Observation Point)
Get Something (with return value)
Indirect Input (Control Point)
Do Something (no return value)
Indirect Output (Observation Point)
DOC

Figure : Source: xunitpatterns.com

How to decouple SUT from DOC?

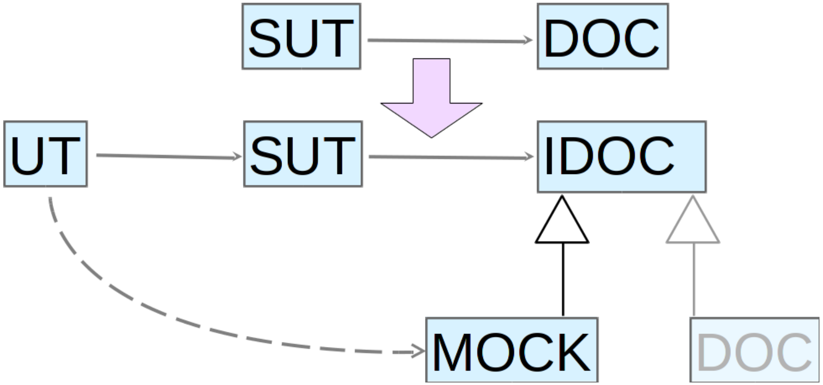
- Introduce a seam: **makes DOC exchangeable**!
- C++ provides different mechanisms:
  - **Object seam** (classic OO seam)
    - Introduce interface - change SUT to use interface instead of DOC directly
    - Pass DOC as a (constructor) argument
  - **Compile seam** (use template parameter)
    - Make DOC a default template argument

Starting Position

```
// Real object 'Die' makes it hard to test the
// system under test (SUT) 'GameFourWins'
// in isolation
struct Die {
  int roll() const { return rand() % 6 + 1; }
};
struct GameFourWins {
  void play(std::ostream& os = std::cout) {
    if (die.roll() == 4) {
      os << "You won!" << std::endl;
    } else {
      os << "You lost!" << std::endl;
    }
  }
private:
  Die die;
};
```

Object Seam

- Based on **subtype / inclusion polymorphism**
- Used refactoring: **Extract interface** (Fowler)



Example

- **Enabling point**: DI via ctor / member function:

```
struct IDie { // extracted interface
  virtual ~IDie() { }
  virtual int roll() const =0;
};
//NEW: GameFourWins(IDie& d) : die{d} {}
void testGameFourWins() {
  struct : IDie {
    int roll() const {
      return 4;
    }
  } fake;
  GameFourWins game{fake}; // enabling point
  std::ostringstream oss;
  game.play(oss);
  ASSERT_EQUAL("You won!", oss.str());
}
```

Refactoring excursus

- William F. Opdyke: *"Refactorings always yield legal programs that perform operations equivalent to before the refactoring."*
- Most important point: **Functionality preservation**

Can you spot the problem?

```
struct AlwaysSixDie : Die {
  int roll() const {
    return 6;
  }
};
struct Croupier {
  void sixWinsJackpot(Die const& die) {
    if (die.roll() == 6) { /* jackpot */ }
  }
};
//NEW:
struct Die : IDie { /* .. */ };
void quiz() {
  Croupier croupier;
  AlwaysSixDie die;
  croupier.sixWinsJackpot(die); // huuh?
}
```

Object Seam

- **Trade-offs**:
  - Run-time overhead of calling virtual member functions
  - Tight coupling
  - Enhanced complexity and fragility
- **Demo**

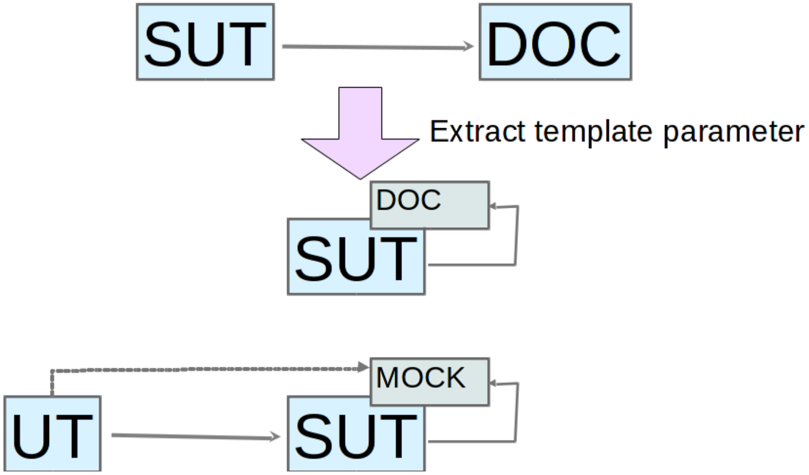
Compile Seam

- Based on **static / parametric** polymorphism
- Compile-time duck typing:

```
template <typename T> void foo(T t)
```

- `t` can be of any type as long as it provides the operations executed on it in `foo` (known as the **implicit interface**)
- Used refactoring: **Extract template parameter**
- **Enabling point**: Template instantiation

Compile Seam



Compile Seam

```
template <typename Dice=Die> // compile seam
struct GameFourWinsT {
  void play(std::ostream &os = std::cout){
    if (die.roll() == 4) {
      os << "You won!" << std::endl;
    } else {
      os << "You lost!" << std::endl;
    }
  }
private:
  Dice die;
};
// do not break existing code
typedef GameFourWinsT<> GameFourWins;
```

Compile Seam

```
void testGameFourWins() {
  struct FakeDie {
    int roll() const {
      return 4;
    }
  };
  GameFourWinsT<FakeDie> game; // enabling point
  std::ostringstream oss;
  game.play(oss);
  ASSERT_EQUAL("You won!\n", oss.str());
}
```

C++11 Excursion: Local Classes

- With C++98/03: local classes had no linkage => could not be used as template arguments
- With C++11: awkward restriction has been removed
- **Still no first-class citizens**:
  - Access to automatic variables prohibited
  - Not allowed to have static members
  - Cannot have template members

Compile Seam

- **Advantages**: No run-time overhead, compile-time duck typing (no interface burden)
- **Disadvantages**: Increased compile-times, (sometimes) reduced clarity
- **Demo**

What if we cannot change our SUT?

- Preprocessor seam
- Link seams:
  - Shadow functions
  - GNU's wrap function
  - Runtime function interception with ld.so
- Absolutely **no changes on existing code** of SUT needed!

Preprocessor Seam

- Use of the C preprocessor **CPP**: Replace calls through `#defines`
- Useful for tracing function calls with debug information

```
// myrand.h
#ifndef MYRAND_H_
#define MYRAND_H_
int my_rand(const char* fileName, int lineNr);
#define rand() my_rand(__FILE__, __LINE__)
#endif
// myrand.cpp
#include "myrand.h"
#undef rand
int my_rand(const char* fileName, int lineNr){
  return 3;
}
```

- **Enabling point**: compiler options to include header file or to define macros (GCC -**include** option)

Preprocessor Seam

- **Trade-offs**: Many! As a means of last resort only!
  - Preprocessor lacks type safety causing hard to track bugs
  - Recompilations necessary
  - Redefinition of member functions is not possible
  - ...
- **Demo**

Link Seams

- **Goal:** Avoid dependency on system or non-deterministic functions, e.g. `rand()`, `time()`, or *slow* calls
- Tweak build scripts by using your **linker's options**
- Three kinds with GNU toolchain:
  - Shadowing functions through **linker order**
  - Wrapping functions with **GNU's wrap option**
  - Run-time function interception of **ld**
- **Enabling point**: linker options
- Constraints: All link seams do not work with **inline functions**

Shadow Function

- Based on **linking order**: linker takes symbols from object files instead the ones defined in libraries
- Place the object files **before** the library in the linker call
- Allows us to shadow the real implementation:

```
// shadow_roll.cpp
#include "Die.h"
int Die::roll() const {
  return 4;
}

$ ar -r libGame.a Die.o GameFourWins.o
$ g++ -Ldir/to/GameLib -o Test test.o \
  > shadow_roll.o -lGame
```

Shadow Function

- Mac OS X GNU linker needs the shadowed function to be defined as a weak symbol:

```
struct Die {
  __attribute__((weak)) int roll() const;
};
```

- **Trade-off**: No possibility to call the original function
- **Demo**

Wrap Function

- Based on **GNU's linker option wrap**
- Possibility to call the original / wrapped function
- Useful to intercept function calls (kind of monkey patching):

```
FILE* __wrap_fopen(const char* path,
                   const char* mode) {
  log("Opening %s\n", path);
  return __real_fopen(path, mode);
}
```

- *"Use a wrapper function for symbol. Any undefined reference to symbol will be resolved to `__wrap_symbol`. Any undefined reference to `__real_symbol` will be resolved to symbol."* - LD's manpage

Wrap Function

- Watch out for C++ **mangled names**!
- Example with Itanium's ABI:

```
$ gcc -c GameFourWins.cpp -o GameFourWins.o
$ nm --undefined-only GameFourWins.o | \
  > grep roll
U_ZNK3Die4rollEv

extern "C" {
  extern int __real__ZNK3Die4rollEv();
  int __wrap__ZNK3Die4rollEv() {
    return 4;
  }
}

$ g++ -Xlinker -wrap=_ZNK3Die4rollEv \
  > -o Test test.o GameFourWins.o Die.o
```

Wrap Function

- **Trade-offs**:
  - Only works with GNU's linker on Linux (Mac OS X not supported)
  - Does not work with functions in shared libraries
- **Demo**

Run-time function interception

- **Alter the run-time linking behaviour** of the loader **ld.so**
- Usage of the environment variable `LD_PRELOAD` the loader `ld.so` interprets
- Manpage of `ld.so`: *"A white space-separated list of additional, user-specified, ELF shared libraries to be loaded before all others. This can be used to selectively override functions in other shared libraries."*
- Instruct the loader to prefer our code instead of libs in `LD_LIBRARY_PATH`

Run-time function interception

- Used by many C/C++ programs (e.g., Valgrind)
- **Not done yet:** would not allow us to call the original function
- **Solution:** use `dlsym` to lookup original function by name
- Takes a handle of a dynamic library (e.g., by `dlopen`)
- Use pseudo-handle `RTLD_NEXT`: next occurence of symbol

Run-time function interception

```
#include <dlfcn.h>
int rand(void) {
  typedef int (*funPtr)(void);
  static funPtr origFun = 0;
  if (!origFun) {
    void* tmpPtr = dlsym(RTLD_NEXT, "rand");
    origFun = reinterpret_cast<funPtr>(tmpPtr);
  }
  int notNeededHere = origFun();
  return 3;
}
$ g++ -shared -ldl -fPIC foo.cpp -o libFoo.so
$ LD_PRELOAD=path/to/libRand.so executable
```

Run-time function interception

- Mac OS X users: Note that environment variables have different names!
- `LD_PRELOAD` is called `DYLD_INSERT_LIBRARIES`
- Additionally needs the environment variable `DYLD_FORCE_FLAT_NAMESPACE` to be set
- **Demo**

Trade-offs

- **Advantages**:
  - Allows wrapping of functions in shared libraries
  - No recompilation / relinking necessary
  - Source code must not be available
  - Linux and Mac OS X supported
- **Disadvantages**
  - Not reliable with member functions
  - Not possible to intercept `dlsym` itself
  - Ignored if the executable is a setuid or setgid binary

Seams - What have we achieved?

- With object and compile seams:
  - **No fixed / hard-coded dependencies** anymore
  - Dependencies are **injected** instead
  - **Improved design** and **enhanced testability**
- Preprocessor seam is primarily a debugging aid
- Link seams help us in replacing or intercepting calls to libraries

Part III - Mock objects

Test double pattern

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

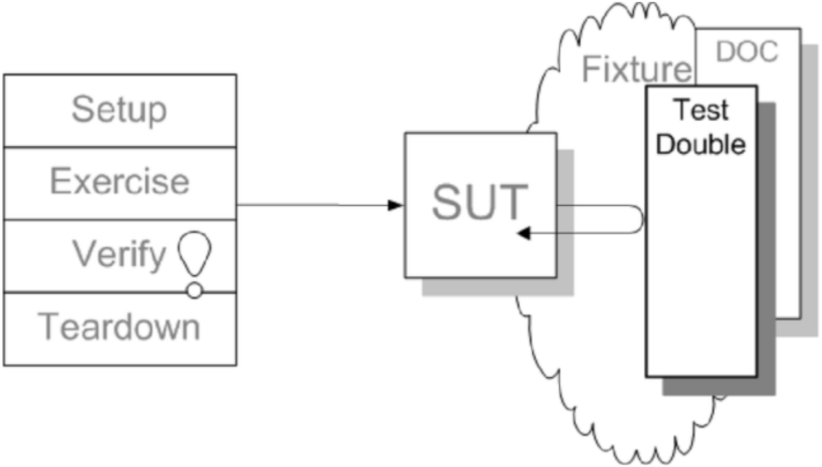


Figure : Source: xunitpatterns.com

Mock object

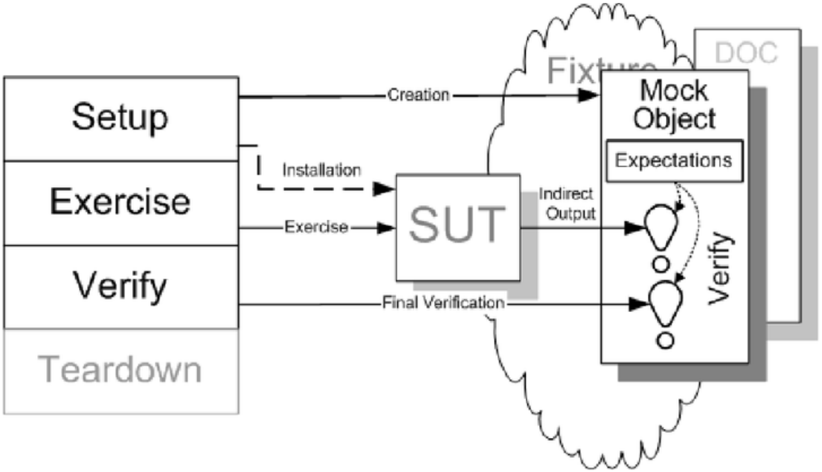


Figure : Source: xunitpatterns.com

Test stub vs. mock object

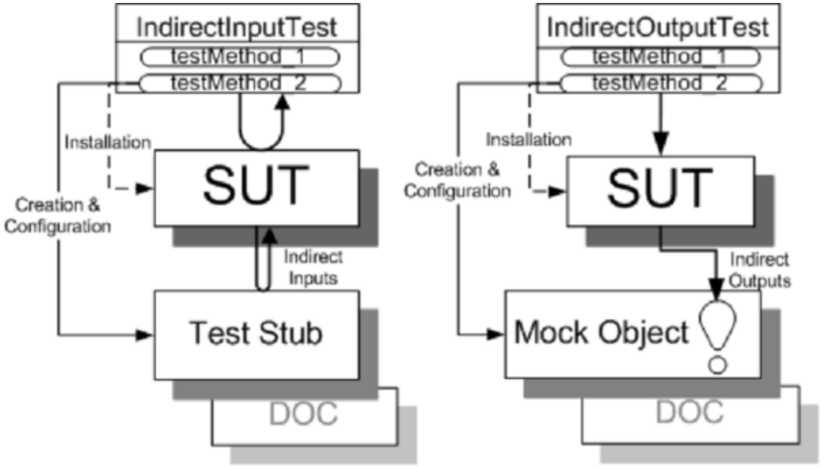


Figure : Source: xunitpatterns.com

Why the need for mock objects?

- **Simpler** tests and design
- Promote **interface-oriented design**
- Independent testing of single units
- **Speed** of tests
- Check usage of third component (is complex API used correctly?)
- Test exceptional behaviour (especially when such behaviour is hard to trigger)

Types of test doubles

- There exist different categories of test doubles 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

Mockator

- **Mock functions and objects** with IDE support
- Mock functionality is not hidden from the user through macros => **better transparency**
- Conversion from fake to mock objects possible
- Support for regular expressions to match calls with expectations
- **Demo**

Wrap-up

Future work for Mockator

- Support **other toolchains** beside GCC (Clang, MS)
- Gain more **practical experience** (e.g., embedded software industry)
- Support **other programming languages**

Conclusions

- TDD in C++ does not need to be a pain: **Try CUTE**
- **Seams** help in making legacy code testable and lead to better software design
- Our refactorings and toolchain support makes seams easier to apply
- Next step is often the use of **test doubles**
- **Mockator** contains a mock object library with code generation for fake and mock objects

Questions?

INSTITUTE FOR SOFTWARE

Figure : http://ifs.hsr.ch

- CUTE: `www.cute-test.com`
- Mockator: `www.mockator.com`
- *Credits*: TDD and mock slides from Peter Sommerlad

IFS INSTITUTE FOR SOFTWARE

Free Trial

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

Linticator

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) up to 4.2 (Juno). It is compatible with Freescale CodeWarrior and other Embedded C/C++ IDEs based on Eclipse CDT.

More information:
http://linticator.com

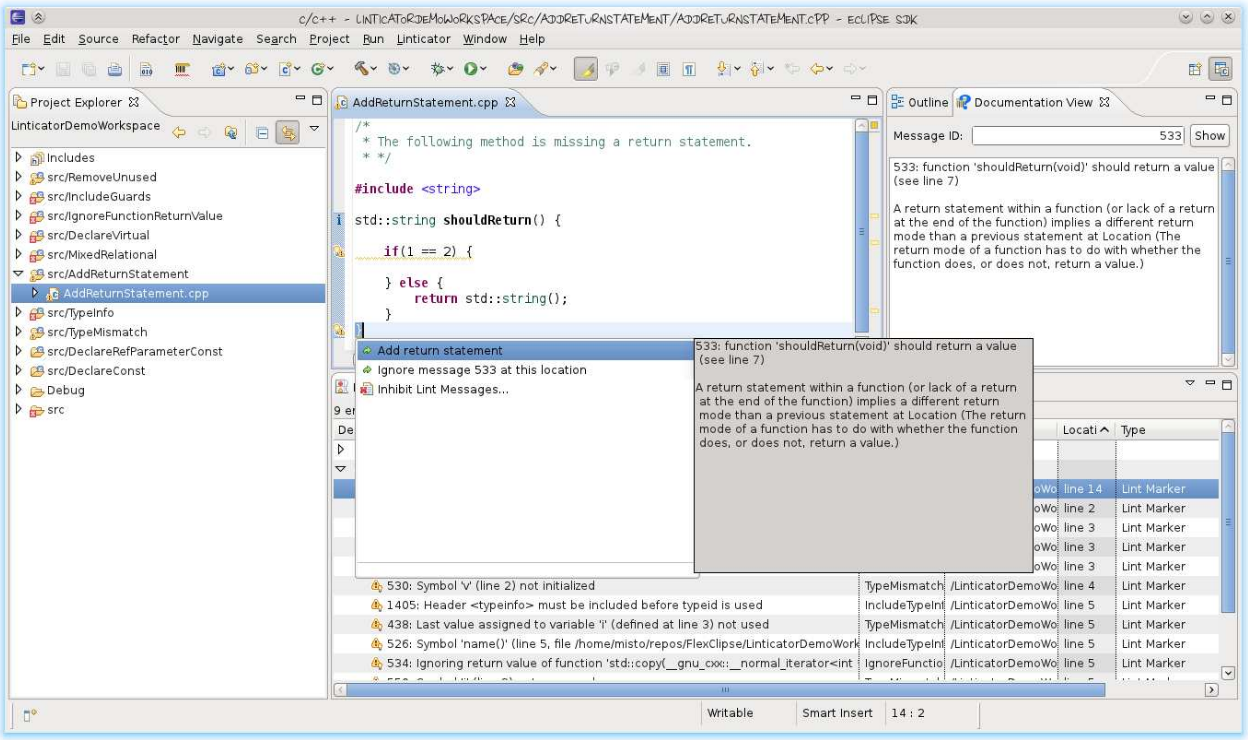
Pricing for Linticator is **CHF 500.-** per user (non-floating license). A maintenance contract that is required for updates costs 20% of license fee per year. The compulsory first maintenance fee includes 6 month of free updates.
Orders, enquiries for multiple, corporate or site licenses are welcome 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 a dozen assistants and employees. Contact us if you look for **Code Reviews**, **UI**, **Design** and **Architecture Assessments.**
http://ifs.hsr.ch



```
/*
 * The following method is missing a return statement.
 * */

#include <string>

std::string shouldReturn() {

    if(1 == 2) {

    } else {
        return std::string();
    }
}
```

533: function 'shouldReturn(void)' should return a value (see line 7)

A return statement within a function (or lack of a return at the end of the function) implies a different return mode than a previous statement at Location (The return mode of a function has to do with whether the function does, or does not, return a value.)

			Locati	Type
		oWo	line 14	Lint Marker
		oWo	line 2	Lint Marker
		oWo	line 3	Lint Marker
		oWo	line 3	Lint Marker
		oWo	line 3	Lint Marker
530: Symbol 'v' (line 2) not initialized	TypeMismatch	/LinticatorDemoWo	line 4	Lint Marker
1405: Header <typeinfo> must be included before typeid is used	IncludeTypeInf	/LinticatorDemoWo	line 5	Lint Marker
438: Last value assigned to variable 'i' (defined at line 3) not used	TypeMismatch	/LinticatorDemoWo	line 5	Lint Marker
526: Symbol 'name()' (line 5, file /home/misto/repos/FlexClipse/LinticatorDemoWork	IncludeTypeInf	/LinticatorDemoWo	line 5	Lint Marker
534: Ignoring return value of function 'std::copy(__gnu_cxx::__normal_iterator<int	IgnoreFunctio	/LinticatorDemoWo	line 5	Lint Marker

IFS INSTITUTE FOR SOFTWARE

Free Trial

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.

More information:
http://includator.com

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.
- **Find unused files**
  Locates single or even entangled header files that are never included in the project's source files.
- **Replace includes with forward declarations (EXPERIMENTAL)**
  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 (EXPERIMENTAL)**
  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 or not-executed statements.

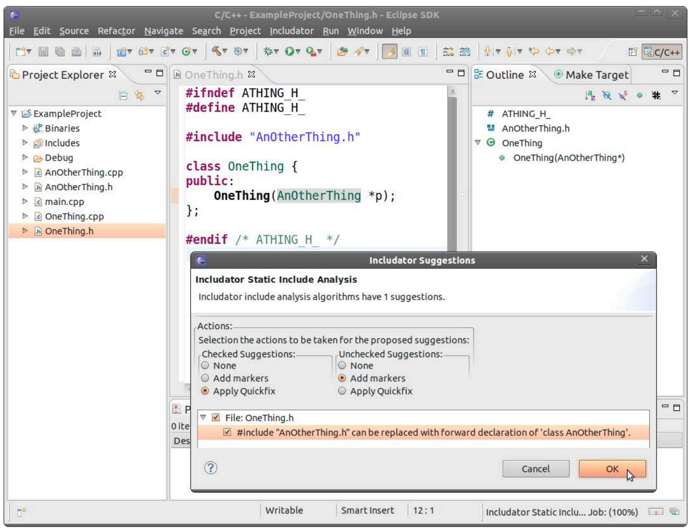
Includator is **CHF 830.-** per user (non-floating license). A maintenance contract is required for updates at 20% of the license fee per year. The compulsory first maintenance fee includes 6 month of free updates. **Orders, enquiries for multiple, corporate or site licenses are welcome** at ifs@hsr.ch.

Enquiries for corporate or site licenses are welcome at ifs@hsr.ch. **Significant volume discounts** apply.

Get Swiss Quality
CUTE+

User Feedback and Participation

Includator is free to try and test. Register at http://includator.com for a 30-day trial license.



```
#ifndef ATHING_H_
#define ATHING_H_

#include "AnOtherThing.h"

class OneThing {
public:
    OneThing(AnOtherThing *p);
};

#endif /* ATHING_H_ */
```

Includator Suggestions

Includator Static Include Analysis

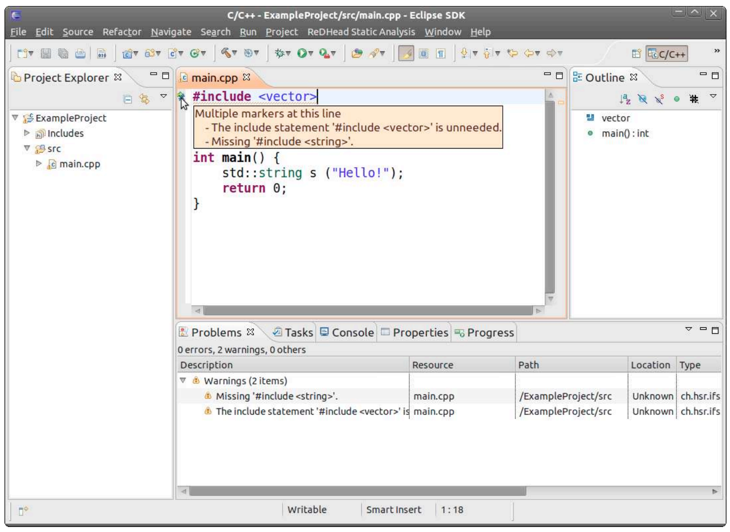
Includator include analysis algorithms have 1 suggestions.

Actions:
Selection the actions to be taken for the proposed suggestions:

Checked Suggestions:	Unchecked Suggestions:
None	None
Add markers	[✓ Add markers]
[✓ Apply Quickfix]	Apply Quickfix

☑ File: OneThing.h
☑ #include "AnOtherThing.h" can be replaced with forward declaration of 'class AnOtherThing'.

Cancel
OK
Writable
Smart Insert
12 : 1
Includator Static Inclu... Job: (100%)


C/C++ - ExampleProject/src/main.cpp - Eclipse SDK
File Edit Source Refactor Navigate Search Run Project ReDHead Static Analysis Window Help
Project Explorer
ExampleProject
Includes
src
main.cpp
main.cpp
Outline
vector
main() : int

```
#include <vector>
int main() {
    std::string s ("Hello!");
    return 0;
}
```

Multiple markers at this line
- The include statement '#include <vector>' is unneeded.
- Missing '#include <string>'.

Problems
Tasks
Console
Properties
Progress

0 errors, 2 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (2 items)				
Missing '#include <string>'.	main.cpp	/ExampleProject/src	Unknown	ch.hsr.ifs
The include statement '#include <vector>' is	main.cpp	/ExampleProject/src	Unknown	ch.hsr.ifs

INSTITUTE FOR SOFTWARE

Free

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

Green Bar for C++ with CUTE

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 Test-driven Development (TDD) and automatic Code Generation

The CUTE plug-in incorporates support for Test-Driven Development (TDD) in C++ and preview of Refactoring features developed by IFS and its students.

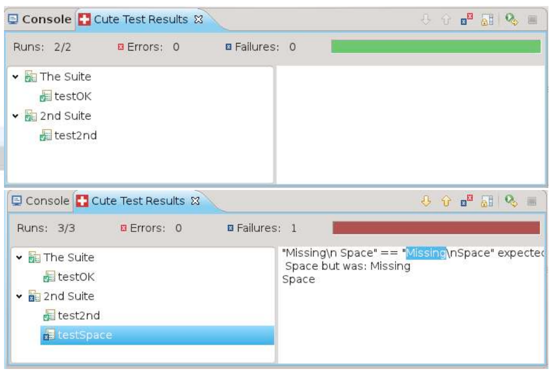
- create unknown function, member function, variable, or type from its use in a test case as a quick-fix (see screen shots below)
- move function or type from test implementation to new header file, after completion TDD cycle.
- toggle function definitions between header file and implementation file, for easier change of function signature, including member functions (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)
- check out Mockator flyer page for further code refactoring and generation features.

More information:
http://cute-test.com

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. Contact us if you look for **Code Reviews**, **UI**, **Design** and **Architecture Assessments.**
http://ifs.hsr.ch

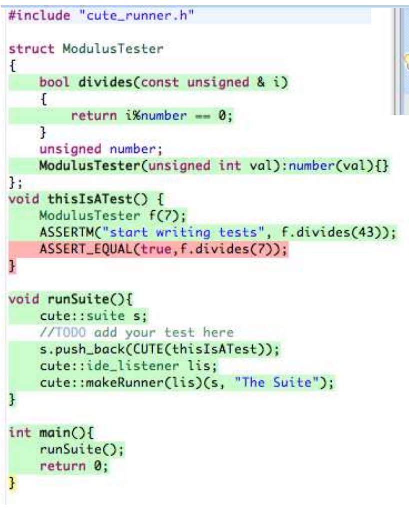
Eclipse update site for installing the free CUTE plug-in:
http://cute-test.com/updatesite



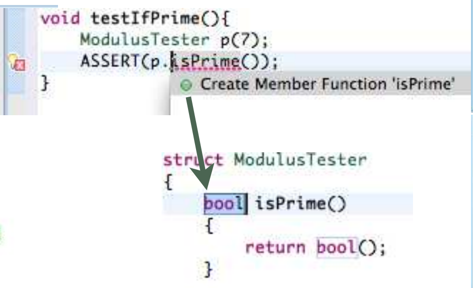
Console | Cute Test Results

```
Runs:  2/2     Errors:  0     Failures:  0
▾ The Suite
    testOK
▾ 2nd Suite
    test2nd
```

```
Runs:  3/3     Errors:  0     Failures:  1
▾ The Suite
    testOK
▾ 2nd Suite
    test2nd
    testSpace

"Missing\n Space" == "Missing\nSpace" expected
 Space but was: Missing
Space
```



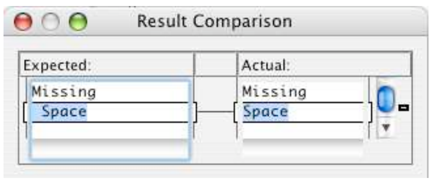
```cpp
#include "cute_runner.h"

struct ModulusTester
{
    bool divides(const unsigned & i)
    {
        return i%number == 0;
    }
    unsigned number;
    ModulusTester(unsigned int val):number(val){}
};
void thisIsATest() {
    ModulusTester f(7);
    ASSERTM("start writing tests", f.divides(43));
    ASSERT_EQUAL(true,f.divides(7));
}

void runSuite(){
    cute::suite s;
    //TODO add your test here
    s.push_back(CUTE(thisIsATest));
    cute::ide_listener lis;
    cute::makeRunner(lis)(s, "The Suite");
}

int main(){
    runSuite();
    return 0;
}
```



```cpp
void testIfPrime(){
    ModulusTester p(7);
    ASSERT(p.isPrime());
}
```

Create Member Function 'isPrime'

```cpp
struct ModulusTester
{
    bool isPrime()
    {
        return bool();
    }
```



Result Comparison

Expected:	Actual:
Missing	Missing
Space	Space

IFS INSTITUTE FOR SOFTWARE

Free

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

Mockator - Eclipse CDT Plug-in for C++ Seams and Mock Objects

eclipse FOUNDATION MEMBER

Refactoring Towards Seams

Breaking dependencies is an important task in refactoring legacy code and putting this code under tests. Michael Feathers' seam concept helps in this task because it enables to inject dependencies from outside to make code testable. But it is hard and cumbersome to apply seams without automated refactorings and tool chain configuration assistance. Mockator provides support for seams and creates the boilerplate code and necessary infrastructure for the following four seam types:

- ***Object seam***: Based on inheritance to inject a subclass with an alternative implementation. Mockator helps in extracting an interface class and in creating the missing test double class including all used virtual member functions.
- ***Compile seam***: Inject dependencies at compile-time through template parameters. Apply the "Extract Template Parameter" refactoring and Mockator creates the missing test double template argument class including all used member functions.
- ***Preprocessor seam***: With the help of the C++ preprocessor, Mockator redefines function names to use an alternative implementation without changing original code.
- ***Link seam***: Mockator supports three kinds of link seams that also allow replacing dependencies without changing existing code:
  - Shadow functions through linking order (override functions in libraries with new definitions in object files)
  - Wrap functions with GNU's linker option -wrap (GNU Linux only)
  - Run-time function interception with the preload functionality of the dynamic linker for shared libraries (GNU Linux and MacOS X only)

Creating Test Doubles Refactorings

Mockator offers a header-only mock object library and an Eclipse CDT plug-in to create test doubles for existing code in a simple yet powerful way. It leverages new C++11 language facilities while still being compatible with C++03. Features include:

- Mock classes and free functions with sophisticated IDE support
- Easy conversion from fake to mock objects that collect call traces
- Convenient specification of expected calls with C++11 initializer lists or with Boost assign for C++03. Support for regular expressions to match calls.

Integrates easily with CUTE!

Get Swiss Quality CUTE

```cpp
struct Die {
    int roll() const {
        return rand() % 6 + 1;
    }
};
template<typename Dice = Die>
struct GameFourWinsT {
    void play(std::ostream& os = std::cout) {
        if (die.roll() == 4) {
            os << "You won!" << std::endl;
        } else {
            os << "You lost!" << std::endl;
        }
    }
private:
    Dice die;
};
typedef GameFourWinsT<> GameFourWins;
```

```cpp
void testGameFourWins() {
    INIT_MOCKATOR();
    static std::vector<calls> allCalls { 1 };
    struct MockDie {
        const size_t mock_id;
        MockDie(): mock_id { reserveNextCallId(allCalls) } {
            allCalls[mock_id].push_back(call { "MockDie()" });
        }
        int roll() const {
            allCalls[mock_id].push_back(call { "roll() const" });
            return 4;
        }
    };
    GameFourWinsT<MockDie> game;
    std::ostringstream oss;
    game.play(oss);
    ASSERT_EQUAL("You won!\n", oss.str());
    calls expectedMockDie = { { "MockDie()" }, { "roll() const" } };
    ASSERT_EQUAL(expectedMockDie, allCalls[1]);
}
```

More information:
http://Mockator.com

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. Contact us if you look for **Code Reviews**, **UI**, **Design** and **Architecture Assessments.**

http://ifs.hsr.ch

Also check out IFS' other plug-ins
http://cute-test.com
http://sconsolidator.com
http://linticator.com
http://includator.com

Eclipse update site for installing Mockator for free:
http://mockator.com/update/indigo
http://mockator.com/update/juno

INSTITUTE FOR SOFTWARE

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

Free

SConsolidator

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

More information:
http://SConsolidator.com

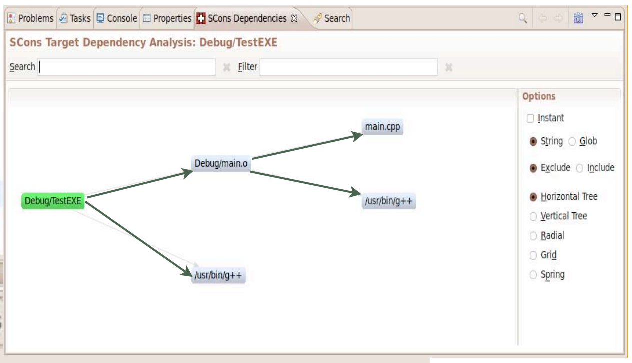
Install the **free SConsolidator plug-in** from the following Eclipse update site:

http://SConsolidator.com/update

Also check out IFS' other plug-ins at:
http://cute-test.com
http://mockator.com
http://linticator.com
http://includator.com

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

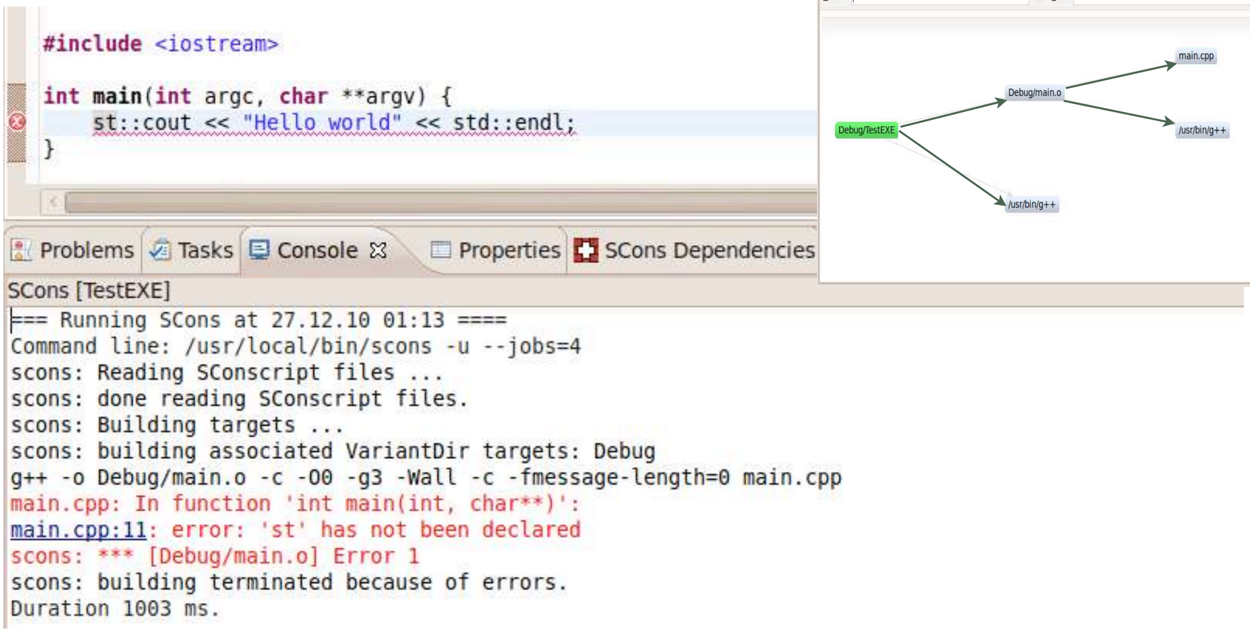
- MongoDB
- Blender
- FreeNOS
- Doom 3
- COAST (http://coast-project.org)



SCons Target Dependency Analysis: Debug/TestEXE

Search | Filter

Instant; [✓ String] Glob; [✓ Exclude] Include; [✓ Horizontal Tree] Vertical Tree Radial Grid Spring



```
#include <iostream>

int main(int argc, char **argv) {
    st::cout << "Hello world" << std::endl;
}
```

SCons [TestEXE]

```
=== Running SCons at 27.12.10 01:13 ====
Command line: /usr/local/bin/scons -u --jobs=4
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: Debug
g++ -o Debug/main.o -c -O0 -g3 -Wall -c -fmessage-length=0 main.cpp
main.cpp: In function 'int main(int, char**)':
main.cpp:11: error: 'st' has not been declared
scons: *** [Debug/main.o] Error 1
scons: building terminated because of errors.
Duration 1003 ms.
```