C++ Refactoring and TDD with Eclipse CDT and CUTE
http://ifs.hsr.ch/cdtrefactoring/updatesite/
http://ifs.hsr.ch/cute/updatesite/

Prof. Peter Sommerlad
HSR - Hochschule für Technik Rapperswil
Institute for Software
Oberseestraße 10, CH-8640 Rapperswil
peter.sommerlad@hsr.ch
http://ifs.hsr.ch
http://wiki.hsr.ch/PeterSommerlad
Work Areas
- Refactoring Tools (C++, Ruby, Python, Groovy, PHP, JavaScript,...) for Eclipse
- **Decremental Development** (make SW 10% its size!)
- Modern Software Engineering
- Patterns
  - POSA 1 and Security Patterns

Background
- Diplom-Informatiker Univ. Frankfurt/M
- Siemens Corporate Research Munich
- itopia corporate information technology, Zurich (Partner)
- Professor for Software HSR Rapperswil, Head 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
Assumptions

• I assume you are familiar with object-oriented concepts of class, constructor/destructor, member functions.
• I assume some basic familiarity with Unit Testing
• I assume you are familiar with standard C++ or intervene otherwise!
  o if I use a C++ feature you do not know or understand, please interrupt! I’ll take a detour.
  o Many C++ programmers got stuck with C++ of the 1990s, feel free to ask for an “upgrade” on the go.
Goals

- You’ll learn about (some) TDD patterns and TDD principles
- You’ll get a brief intro to Test Doubles and Mock Objects
- You’ll participate in Test-driven Design in C++ using Eclipse CDT, CUTE and our C++ Refactoring plug-in (at least as an Observer)
- I want to show you what we’ve created to ease C++ development with CDT.
Unit-Testing Principles (already known?)

- Test anything that might break
- Test everything that does break
- New code is guilty until proven innocent
- Write at least as much test code as production code
- Run local tests with each compile
- Run all tests before check-in to repository
Vicious Circle: Testing - Stress

- Automate tests and run them often!
How do I write good Unit Tests?

- Ask yourself the following questions: (among others about your coding)
  - 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?
Why even more on Test Automation?

- 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?
Principle of Automated Tests
Triple-A (AAA)

1. Arrange
   - initialize system(s) under test
2. Act
   - call functionality that you want to test
3. Assert
   - assert that results are as you expect

Remember: "Triple-A: arrange, act, assert"
Terminology
xunitpatterns.com

- SUT system under test
Test Case Structure: Four Phase Test

- compare that to AAA ---> another similarity
- Source: xunitpatterns.com
Test-Driven Development

Exploiting Unit Tests...
Test-Driven Development [Beck-TDD]

- There are several books on test-driven design (or TDD)
  - Kent Beck, Dave Astels, Gerard Meszaros
- TDD is not a testing technique, but a coding and design technique
  - nevertheless TDD patterns help you writing tests, regardless if you follow TDD or not
- TDD relies heavily on Refactoring
  - we (IFS) try hard to provide you with such Refactoring automation for C++ as well as you might be used to with Java or Ruby. (plus Refactoring for Python (PyDev), Groovy, PHP, JavaScript)
TDD

[Kevlin Henney]

- TDD has emerged from the many practices that form Extreme Programming's core
  - Focused on code-centric practices in the micro process rather than driving the macro process

- TDD can be used in other macro-process models
  - TDD is not XP, and vice versa
  - TDD is not just unit testing

- BDD (Behaviour Driven Design)
  - Follow-up to TDD
  - since TDD is not about Testing
TDD Practices and Characteristics

**Essential Test-Driven Development Practices**
- Test-bounded design increments
- Programmer testing responsibility
- Active test writing
- Refactoring
- Automated tests
- Example-based test cases

**Build and Release Practices**
- Fine-grained versioning
- Continuous integration
- Defined stable increments

**Team-Related Practices**
- Pair programming
- Shared coding guidelines

provided by [Kevlin Henney]
TDD Patterns

Writing Tests & Habits

• Isolated Tests
  o write tests that are independent of other tests

• Test List
  o use a list of to-be-written tests as a reminder
  o only implement one failing test at a time

• Test First
  o write your tests before your production code

• Assert First
  o start writing a test with the assertion
  o only add the acting and arrangement code when you know what you actually assert
Demo TDD V1
Generate Roman Numbers

- generate roman numbers as strings from an integer representation

  - start with the following list of tests
  - create a new CUTE projects
  - write test, implement function, refactor, repeat
  - make up new tests as you go and see need

THE LIST FOR ROMAN NUMBERS (V0)

1 ➔ I
0 ➔ EMPTY STRING
2 ➔ II
...

“Red-bar” Patterns
Finding Tests 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
Demo TDD V2
(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
“Red Bar” Patterns (2)

- **Regression Test**
  - For every bug report write tests showing the bug

- **Break**
  - Enough breaks are essential. When you are tired you loose concentration and your judgement gets worse. This results in more errors, more work, and makes you more fatigue. (vicious circle!)

- **Do Over**
  - If you recognize your design and tests lead nowhere, DELETE your code! A fresh start earlier is often better.
“Green Bar” - Patterns
Make your Tests succeed

- **Fake It ('Til You Make It)**
  - It is OK to “hack” to make your test succeed.
  - 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**
  - Implement functions with many elements first for one element (or none) correctly
TDD Patterns

Habits

● **Child Test**
  ○ If a test case gets too large, “remove” it, redo the core, get “green-bar”, and then introduce the “full” case again, get “green-bar”

● **Broken Test**
  ○ If you have to stop programming or take a break, leave a broken test to remind you where you left.
    ➢ but only do Clean Check-in!

● **Clean Check-in**
  ○ Do only (and may be always) check-in your code and tests when you have a green bar.
Test Double Pattern
xunitpatterns.com

- How can we verify logic independently when code it depends on is unusable?
- How can we avoid Slow Tests?
Test Double Patterns [Beck-TDD]

- **Mock Object**
  - Decouple a class under test from its environment

- **Self Shunt**
  - Use the test case class itself as a Mock Object

- **Log String**
  - Test temporal dependencies of calls by concatenating call info in a string, e.g., using Self Shunt

- **Crash Test Dummy**
  - How do you test exceptions that are hard to force, but might occur during production?
  - Use a dummy/Mock Object that throws an exception instead of the real object.
Mock Object xunitpatterns

- How do we implement Behavior Verification for indirect outputs of the SUT?
- How can we verify logic independently when it depends on indirect inputs from other software components?
Difference Test-Stub and Mock-Object
There is a standard Schema to test some code if it raises a specific exception:

```cpp
void testAnException() {
    std::vector<int> v; // arrange
    try {
        v.at(0); // act
        FAILM("expected out_of_range exception"); // assert
    } catch(std::out_of_range &) {} 
}
```

CUTE encapsulates this to

```cpp
void testAnException() {
    std::vector<int> v;
    ASSERT_THROWS(v.at(0),std::out_of_range);
}
```
Example Crash-Test Dummy in C++

```cpp
struct out_of_memory : std::exception{};

template <typename T>
struct failingallocator : std::allocator<T> {
    typedef typename std::allocator<T>::pointer pointer;
    typedef typename std::allocator<T>::size_type size_type;
    pointer allocate(size_type n, std::allocator<void>::const_pointer hint=0) {
        //return std::allocator<T>::allocate(n,hint);
        throw out_of_memory();
    }
}; // "Crash-Test-Dummy" allocator

void testFailingAllocation() {
    std::vector<int, failingallocator<int>> v;
    ASSERT_THROWS(v.reserve(5), out_of_memory);
}

void testFailingAllocationCtor() {
    std::vector<int, failingallocator<int>> v(5); // will throw in ctor!
}

void runSuite() {
    cute::suite s;
    s.push_back(CUTE(testFailingAllocation));
    s.push_back(CUTE_EXPECT(CUTE(testFailingAllocationCtor), out_of_memory));
    ...
```
Why Test Doubles and Mock Objects? [PragUnit]

- The real object has **nondeterministic behavior** (it produces unpredictable results, like a stock-market quote feed.)
- The real object is **difficult to set up**.
- The real object has **behavior that is hard to trigger** (for example, a network error).
- The real object is **slow**.
- The real object has (or is) a **user interface**.
- The test needs to **ask** the real object about **how it was used** (for example, a test might need to confirm that a callback function was actually called).
- The real object **does not yet exist** (a common problem when interfacing with other teams or new hardware systems).
TDD Expression Evaluator

• Thanks to Hubert Matthews for his last year workshop where I tried TDD on this problem.

• Wanted:
  • A volunteer keeping track of tests to write: The List
  • Your help in implementing and refactoring
    ➢ just call, ask, and answer
    ➢ I am your (sometimes intelligent) typing machine (and guide)
How to write CUTE Tests?

optional slides...
Test Fixtures

- Often several test cases require identical arrangements of testee objects

Reasons
- "expensive" setup of objects
- no duplication of code (DRY principle)

Mechanisms
- JUnit provides setup() and teardown() methods
- CPPUnitLite does not provide this
  - other CPPUnit variants do as virtual functions
- CUTE employs constructor and destructor of a testing class with per test object incarnation
  - no need for inheritance and virtual member functions
  - just employ C++ standard mechanisms
#include "cute.h"
#include "cute_equals.h"
struct ATest {
    CircularBuffer<int> buf;
    ATest():buf(4){}
    void testEmpty(){   ASSERT(buf.empty());}
    void testNotFull(){   ASSERT(!buf.full());}
    void testSizeZero(){   ASSERT_EQUAL(0,buf.size());}
};

#include "cute_testmember.h"
...
s.push_back(CUTE_SMEMFUN(ATest,testEmpty));
s.push_back(CUTE_SMEMFUN(ATest,testNotFull));
s.push_back(CUTE_SMEMFUN(ATest,testSizeZero));
...
Member Functions as Tests in CUTE

- **CUTE_SMEMFUN(TestClass, memfun)**
  - instantiates a new object of TestClass and calls memfun on it ("simple" member function)

- **CUTE_MEMFUN(testobject, TestClass, memfun)**
  - uses pre-instantiated testobject as target for memfun
    - this is kept by reference, take care of its scoping/lifetime
    - allows reuse of testobject for several tests and thus of a fixture provided by it.
  - allows for classes with complex constructor parameters

- **CUTE_CONTEXT_MEMFUN(context, TestClass, memfun)**
  - keeps a copy of context object and passes it to TestClass' constructor before calling memfun on it
    - avoids scoping problems
    - allows single-parameter constructors
Refactoring forMocks 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 too 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
  o especially for external dependencies
  o promote interface-oriented design

• Independent Testing of single Units
  o isolation of unit under testing
  o or for not-yet-existing units

• Speed of Tests
  o no external communication (e.g., DB, network)

• Check usage of third component
  o is complex API used correctly

• Test exceptional behaviour
  o 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

وية

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

```cpp
#include <cstdlib>

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

#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 ostream to collect the output of play() and check that against an expected value:

```cpp
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*

```cpp
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
Simulation Mocks
preparing SUT

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

```cpp
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!
",os.str());
    }
A C++ alternative using templates

- **advantage:** no virtual call overhead
- **drawback:** inline/export problem potential

```cpp
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:

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

```cpp
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.
  o no need to pass object in as additional parameter
  o unfortunately no default template parameters for template functions (yet)

● You can mock
  o Member Variable Types
  o Function Parameter Types

● Mocking without template inline/export need is possible through explicit instantiations
Summary Mock Objects

- Mock Objects are important for isolating unit tests
  - or speeding them up
- They can lead to better, less-coupled design
  - separation of concerns
- Overdoing mocking can be dangerous
  - go for simplicity!
- C++ offers additional ways to introduce mock objects through templates
  - also through #define and typedef!
Outlook/Questions

...
References

[Becck-TDD]
  o Kent Beck: Test-Driven Design

[PragUnit]
  o Andy Hunt, Dave Thomas: Pragmatic Unit Testing

[Kevlin Henney]
  o JUTLAND:
    Java Unit Testing: Light, Adaptable 'n' Discreet

[Dave Astels] - TDD
  o Test Driven Development: A Practical Guide
  o [video.google.com](http://video.google.com/videoplay?docid=8135690990081075324) - on BDD

[Dan North] - Behaviour Driven Development
  o [http://dannorth.net/introducing-bdd/](http://dannorth.net/introducing-bdd/)

[Gerard Meszaros] - xUnit Test Patterns
  o [http://xunitpatterns.com](http://xunitpatterns.com)
  o very good overview of the problems of and with test automation and their solutions