INSTITUTE
FOR
SOFTWARE

Better Software: Simpler Faster

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

INFORMATIK

CUTE
C++ Unit Testing Easier

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

Peter Sommerlad

peter.sommerlad@hsr.ch

INSTITUTE FOR SOFTWARE

- **Work Areas**
  - Refactoring Tools (C++,Ruby, Python,...) for Eclipse
  - **Decremental Development (make SW 10% its size!)**
  - Modern Software Engineering
  - Patterns
    - Pattern-oriented Software Architecture (POSA)
    - 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

Agenda

INSTITUTE FOR SOFTWARE

- What is Unit Testing?
- Why CUTE?
- Writing Simple Tests
- Running Tests: Green Bar for C++
- Writing Advanced Tests
- CUTE's Design - GoF for C++ templates

Is that Testing?

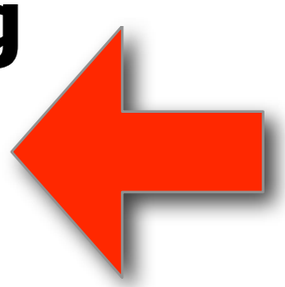
INSTITUTE FOR SOFTWARE

- **"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

What is Testing?

INSTITUTE FOR SOFTWARE

- **All on the previous slide, but much more!**
- **Manual Testing**
  - sometimes useful and needed
    - UI testing, usability testing, user testing with a plan
  - but automation is much better!
    - no ad-hoc testing!
- **Automated Testing**
  - unit tests



Today's topic

  - functional tests
  - integration, load and performance tests
  - code quality tests (lint, compiler, code checkers)

What does Testing mean?

INSTITUTE FOR SOFTWARE

- **You want a correctness guarantee!**
  - how do you define "correctness"?
  - "correctness" against what specification?
  - what kind of guarantee?
    - 6 months, 2 years, lifetime?
- **Alternatives to Testing?**
  - code reviews
  - walkthroughs
  - inspection
  - mathematical proofs of correctness
    - hard, hard to specify understandable specifications
    - but, can be used to construct code with proven correctness

Unit Testing

INSTITUTE FOR SOFTWARE

- **Is not "Testing" in the classic sense:**

> Program testing can be used to show the presence of bugs, but never to show their absence! - *E.W. Dijkstra*

But

- **Is Built-In Quality Assurance**
- **Allows Regression Testing**
- **Enables Refactoring**
- **Is Change Insurance**
- **Improves Built Automation**

Why a Unit Testing Framework?

INSTITUTE FOR SOFTWARE

- **A lot of repetitive code when writing tests**
  - AAA, logging failures and errors
- **JUnit (Java) and SUnit (Smalltalk) as templates for other OO-languages**
- **C++ lacks convenient reflection mechanism, but has macros instead**
- **Several C++ frameworks mimick JUnit**
  - CPPUnitLite, CPPUnit, boost/test, Aeryn
  - but modern C++ is not Java, JUnit's design has come of age
- **Therefore**
  - CUTE: C++ Unit Testing Easier

Test Automation

INSTITUTE FOR SOFTWARE

- **Advantages**
  - repeatability - regression
    - insurance for change, portability, extension
    - no (or very low) cost for re-testing
  - well-defined specification given by executable tests
    - test-code is program code with well defined semantics
  - repeatability, repeatability, repeatability, ...
- **Drawbacks**
  - need to write and maintain also test code
    - tests also require refactoring
  - test code is program code
    - is the right thing tested? (instead of implemented?)

Why and When?

INSTITUTE FOR SOFTWARE

- **Become "test-infected". Once you are used to unit testing your code, you get addicted.**
  - That's a fact I observed many times.
  - You'll regret every piece of code you want to change where you don't have tests for
- **Write your tests close to writing your code!**
  - Some say: Test-First or Test-Driven Design (TDD)
  - modern: Behavior-Driven Design (BDD)
  - Retrofitting existing code with tests will show you its design deficiencies
    - hard to write tests -> entangled design, too complex
    - easy to write tests -> orthogonal design, simpler
- **At least write tests before you change code!**

Structure of a typical Unit Testing Framework

INSTITUTE FOR SOFTWARE

- **Test Assertion / Check statement**
  - o used in
- **Test (Member-)Function**
  - o defined in
- **TestCase Subclass bundling Tests**
  - o its objects contained in
- **Test Suite collecting test objects**
  - o executed by
- **Test Runner (often in a main() function)**
  - o delivers result
- **OK or Failure**

Example of a CUTE test

INSTITUTE FOR SOFTWARE

```
std::string::size_type
operator% (std::string const &l,
           std::string const &r) { ... } // find r in l

void testFindOperator(){
  std::string s("Hallo");
  std::string tobefound("ll");
  std::string::size_type
     pos= s % tobefound; // the new %
  ASSERT(pos == 2); // the position
}
```

Principle of Automated Tests Triple-A (AAA)

INSTITUTE FOR SOFTWARE

1. **Arrange**
   - initialize object(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"

AAA in our Example

INSTITUTE
FOR
SOFTWARE

```
std::string::size_type
operator% (std::string const &l,
           std::string const &r) { ... } // find r in l

void testFindOperator(){
  std::string s("Hallo");          // Arrange
  std::string tobefound("ll"); // Arrange
  std::string::size_type
     pos= s % tobefound;           // Act
  ASSERT(pos == 2);                // Assert
}
```

Using CUTE

INSTITUTE FOR SOFTWARE

```
#include "cute.h"
```

- ASSERT(condition);
  - fails if condition is false
- ASSERT_EQUAL(expected,actual);
  - fails if exected is not equal to actual
  - special three parameter cases for double ASSERT_EQUAL(exp,act,delta)
- **add a message by appending** M
  - ASSERTM(msg,condition)
  - ASSERT_EQUALM(msg,exp,act)
- FAIL(); FAILM(msg)
  - fails always, use to mark unwritten tests
  - or for checking exceptions

CUTE vs. CPPUnitLite

INSTITUTE FOR SOFTWARE

- **CUTE**
- **header-only**
  - no library to link
- **no need to inherit from TestCase base class**
  - Functor classes can be used or void functions
- **uses modern C++ features and boost**
  - requires boost installation for compiler (headers sufficient)
  - requires modern standard C++ compiler
    - embedded compilers lack about 10 years behind

- **CppUnitLite (and others)**
- **header+library**
  - need to link against (static) library
  - C-ish legacy code (stdio)
- **tests must inherit from TestCase**
  - macro eases/hides this
- **small enough to include with project**
- **extendable and adaptable**
- **might run on embedded hardware**
- **CHECK_EQUAL has problems with integer values (of different kind)**

How many unit tests should I write?

INSTITUTE FOR SOFTWARE

- **Test anything that might break**
  - don’t write tests for code that cannot break
- **Test everything that does break**
  - for every bug, write a test demonstrating it
- **New code is guilty until proven innocent**
- **At least as much test code as production code**
- **Run local tests with each compile**
  - don’t write new code when tests are failing
- **Run all tests before check-in to repository**
  - better: also run them after check in on your build server

Use your Right-BICEP [PragProg]

INSTITUTE FOR SOFTWARE

- **Are the results right?**
  - ASSERT_EQUAL(42,7*6)
- **Are all boundary conditions CORRECT?**
  - 0, 1, 0xffffffff
- **Can you check inverse relationships?**
  - sqrt(x)*sqrt(x) == x
- **Can you cross-check results using other means?**
- **Can you force error conditions to happen?**
  - y/x, x=0
- **Are performance characteristics within bounds?**

CORRECT Boundary Conditions

INSTITUTE FOR SOFTWARE

- **Conformance**
  - e.g., check email address: foo@bar.com
- **Ordering**
  - is sequence relevant? what if out of order?
- **Range**
  - is the domain range correct
- **Reference**
  - expectations on environment
- **Existence**
  - is some parameter/variable defined, null, existent
- **Cardinality**
  - off-by one errors, 0,1, many

Test Fixtures

INSTITUTE FOR SOFTWARE

- **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
  - CUTE employs constructor and destructor of a testing class with per test object incarnation

Test Fixture with CUTE

INSTITUTE
FOR
SOFTWARE

```
#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

INSTITUTE FOR SOFTWARE

- `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

Goals for CUTE's Design

INSTITUTE
FOR
SOFTWARE

- **Simplicity**
- **easier to use than CPPUnits**
- **low learning curve**
  - no need to understand inheritance or templates for simple users
  - intuitive ASSERTions
- **dependencies only to std classes (in the future)**
  - uses boost:: parts proposed for std
    - boost::function, boost::bind
- **header-only distribution**
  - no dependency with linking to library or objects

Structure of a typical Unit Testing Framework

INSTITUTE FOR SOFTWARE

- **Test Assertion / Check statement**
  - ASSERT() ASSERT_EQUAL() - macros
- **Test (Member-)Function**
  - void operator()(), void function()
- **~~TestCase Subclass bundling Tests~~**
  - class cute::test - encapsulates a test function
- **Test Suite collecting test objects**
  - cute::suite - simple std::vector<cute::test>
- **Test Runner (often in a main() function)**
  - cute::runner<cute::listener>() - runs tests
- **OK or Failure -- cute::listener**
  - plug-in for Eclipse OR VS debug console

CUTE's Design
class cute::test

INSTITUTE FOR SOFTWARE

- **represent a test object -> requirements:**
  - should be identified by a name
    - std::string name();
  - must run, doesn't return a value
    - void operator()();
  - failure noted by throwing a defined exception
  - error noted by throwing an unexpected exception
  - keeps the code to run
    - boost::function<void()> theTest
    - will be std::function in the future
  - should fit into a std datastructure, i.e. std::vector
    - value class: copy-ctor, assignment
    - default implementation should be sufficient

CUTE's Design
class cute::test

INSTITUTE FOR SOFTWARE

- **creating a test object**
  - naming can rely on C++ reflection capabilities
    - ```
      template <typename VoidFunctor>
      test(VoidFunctor const &t,
           std::string name =
               demangle(typeid(VoidFunctor).name()))
      :theTest(t),name_(name){}
      ```
    - for test functor classes, naming is automatic
    - constructor as template member function (!)
    - de-mangling required for compiler specific typeid.name()
  - or a macro
    - `#define CUTE(name) cute::test((&name),(#name))`
    - for test functions a name must be provided
      ```

CUTE's Design cute::test_failure

- **explain reason for failure**
  - std::string reason;
- **need to know source position**
  - std::string filename; int lineno;
- **throw test_failure if assertion is false:**
  - ```
    #define ASSERTM(msg,cond)  if (!(cond)) \
       throw cute::test_failure((msg),__FILE__,__LINE__)
    #define ASSERT(cond) ASSERTM(#cond,cond)
    ```
  - macro provides only feasible access to position
    ```

Design Patterns with templates

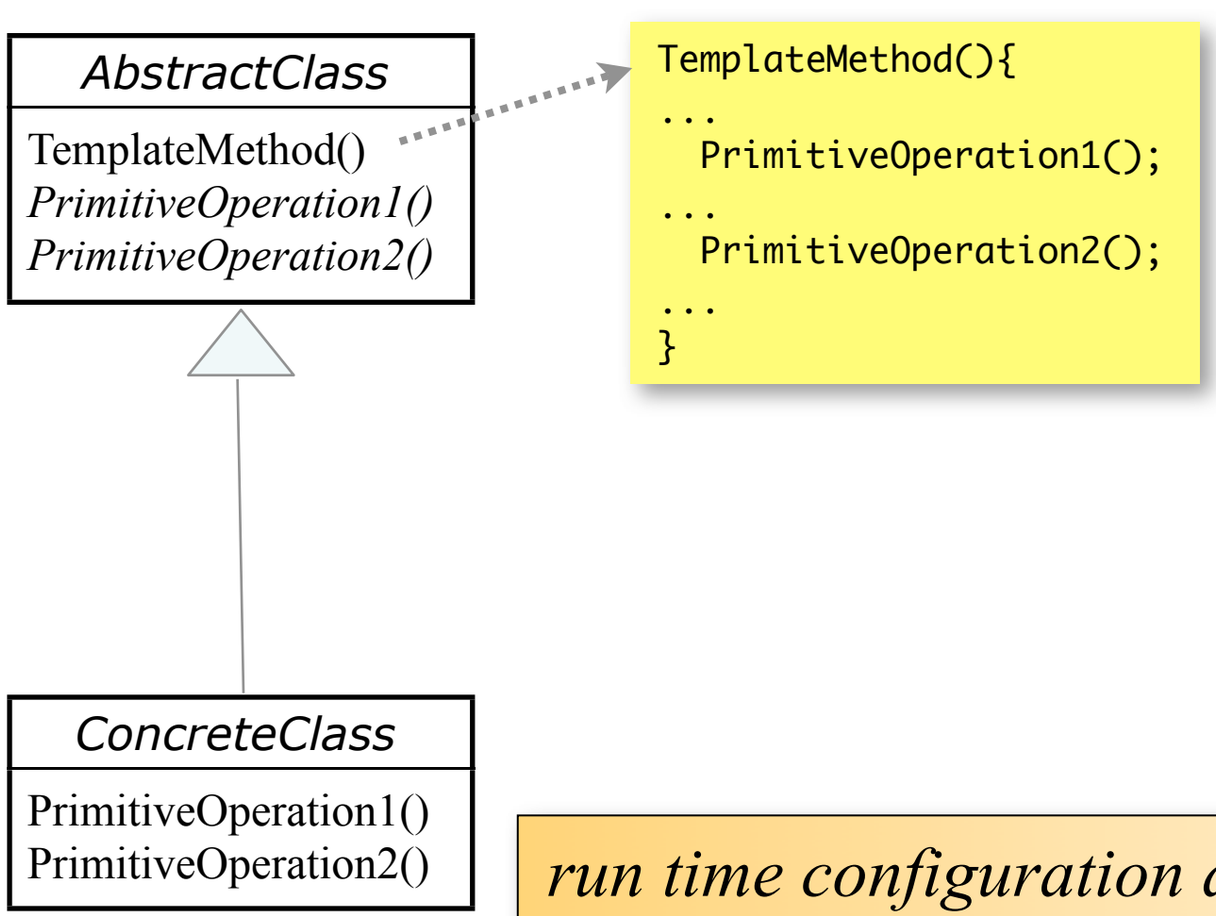
INSTITUTE FOR SOFTWARE

- **cute::runner uses a combination of Design Patterns Template Method and Strategy**
  - but instead of subclassing cute::runner, you provide the strategy object as a template parameter (Listener) that runner inherits from.
- **To add functionality to listeners, CUTE uses an adaptation of the Decorator Design Pattern**
  - but instead of wrapping an instance, listener-decorators again, take a to-be-superclass as template paramater
- **This Decorator chain can be terminated by the default strategy showing an incarnation of the Null Object Design Pattern**

Template Method

INSTITUTE FOR SOFTWARE

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

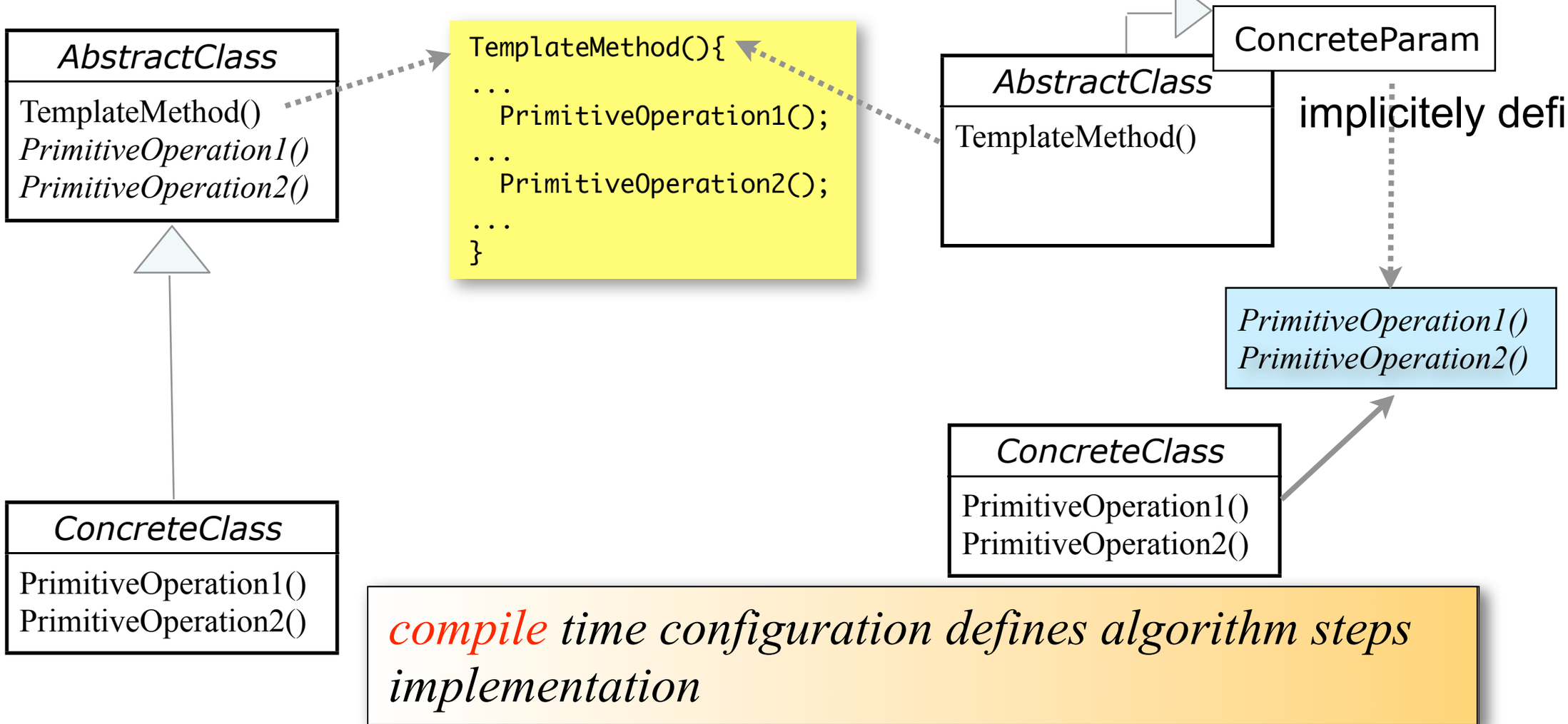


run time configuration defines algorithm steps implementation

Template Method

INSTITUTE FOR SOFTWARE

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets superclass redefine certain steps of an algorithm without changing the algorithm's structure.



compile time configuration defines algorithm steps implementation

template class cute::runner

INSTITUTE FOR SOFTWARE

```
template <typename Listener=null_listener>
class runner : Listener{
  bool runit(test const &t){
    try {
      Listener::start(t);
      t(); // run the test
      Listener::success(t,"OK");
      return true;
    } catch (cute::test_failure const &e){
      Listener::failure(t,e);
    } catch (std::exception const &exc){
      Listener::error(t,test::demangle(exc.what()).c_str());
    } catch(...) {
      Listener::error(t,"unknown exception thrown");
    }
    return false;
  }
....
};
```

Null-Object Pattern

Template Method Pattern

Null Object
null_listener

INSTITUTE FOR SOFTWARE

```
struct null_listener{ // defines Contract of runner parameter
      void begin(suite const &s, char const *info){}
      void end(suite const &s, char const *info){}
      void start(test const &t){}
      void success(test const &t,char const *msg){}
      void failure(test const &t,test_failure const &e){}
      void error(test const &t,char const *what){}
};
```

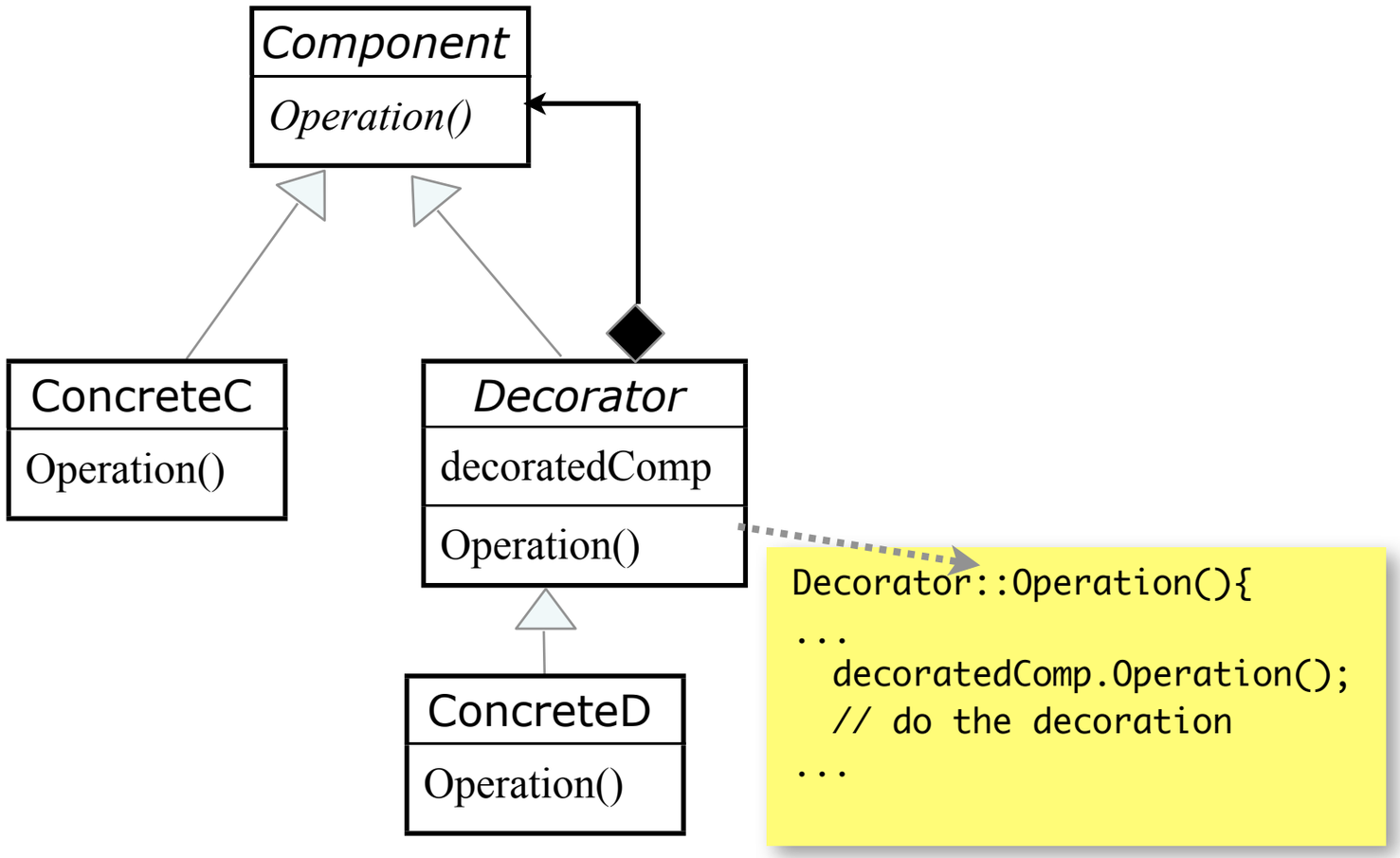
Null-Object Pattern

Template Method Pattern
Interface

Decorator

INSTITUTE FOR SOFTWARE

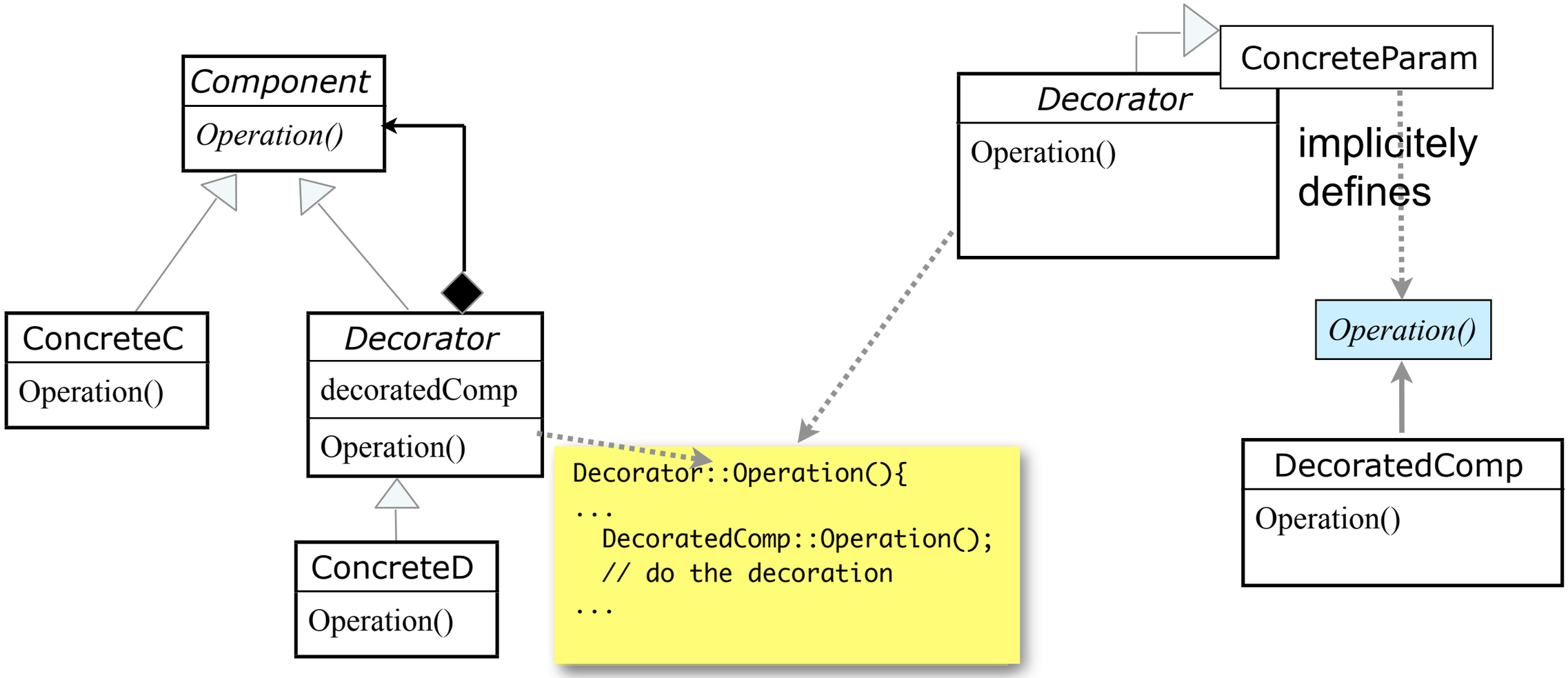
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Decorator

INSTITUTE FOR SOFTWARE

Attach additional responsibilities to an object statically. Decorators provide a flexible alternative via subclassing for extending functionality.



Decorator counting_listener

INSTITUTE FOR SOFTWARE

```
template <typename Listener=null_listener>
struct counting_listener:Listener{
  counting_listener() :Listener()
      ,numberOfTests(0),successfulTests(0),failedTests(0){}
      void start(test const &t){
                  ++numberOfTests;
                  Listener::start(t);
      }
      void success(test const &t,char const *msg){
                  ++successfulTests;
                  Listener::success(t,msg);
      }
      void failure(test const &t,test_failure const &e){
                  ++failedTests;
                  Listener::failure(t,e);
      }
...
};
```

Null-Object Pattern

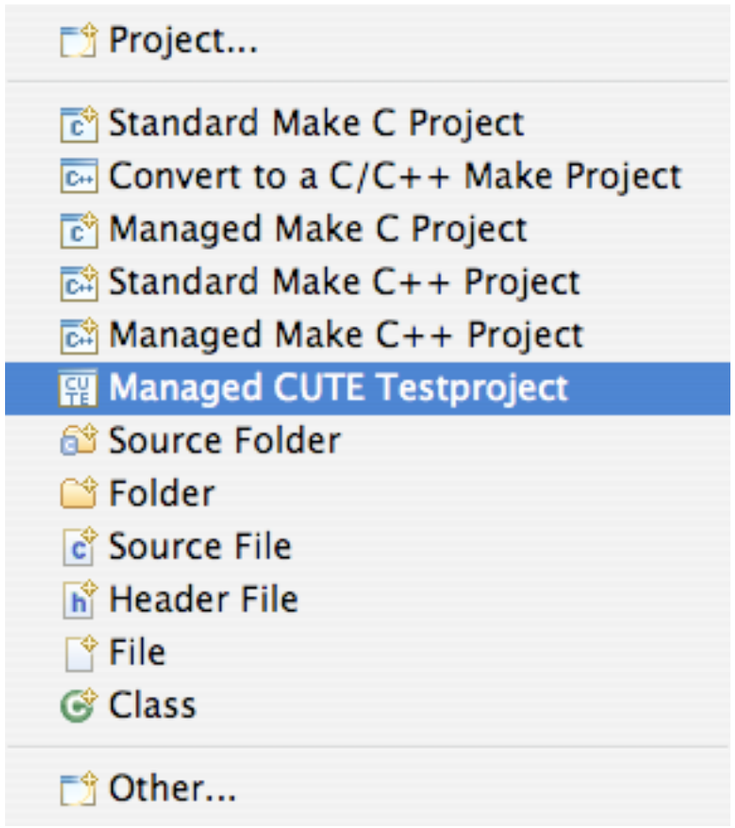
Template Method Pattern
Interface

Green-Bar for C++
CUTE Plug-In for Eclipse

INSTITUTE
FOR
SOFTWARE

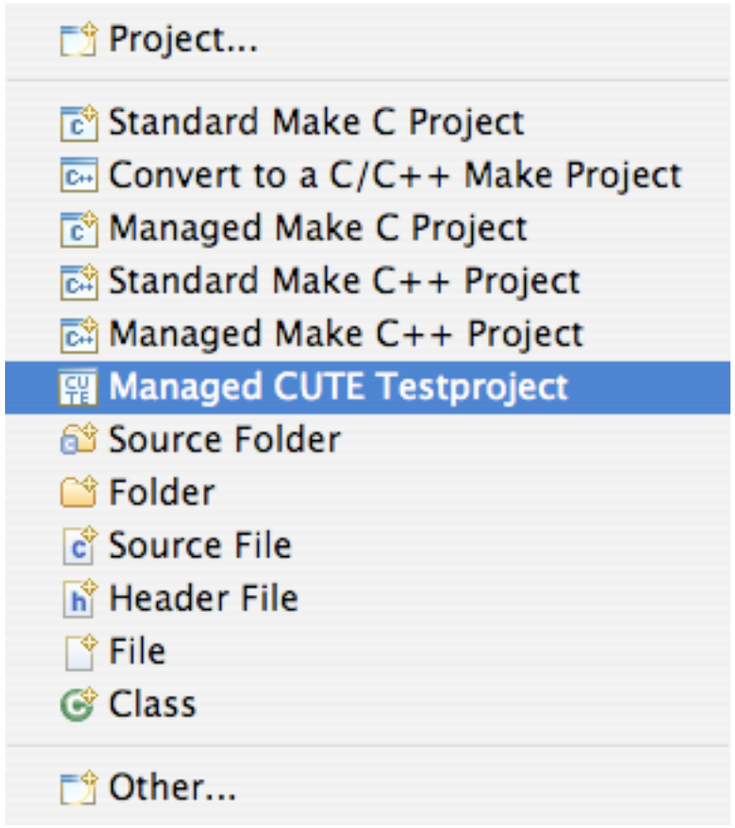
Green-Bar for C++
CUTE Plug-In for Eclipse

INSTITUTE
FOR
SOFTWARE



Green-Bar for C++
CUTE Plug-In for Eclipse

INSTITUTE FOR SOFTWARE



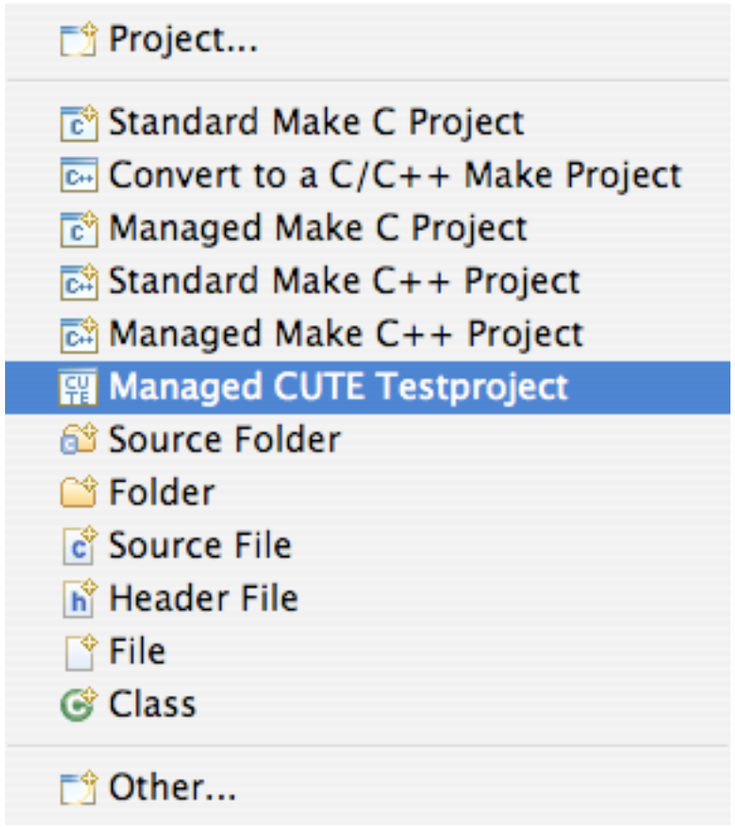

```
void testFindOperator(){
    std::string s("Hallo");
    std::string tobefound("ll");
    std::string::size_type pos= s % tobefound;
    ASSERT(pos == 2);
}


void runSuite(){
    cute::suite s;
    //TODO add your test here
    s += CUTE(testFindOperator);
    cute::eclipse_listener lis;
    cute::makeRunner(lis)(s, "The Suite");
}

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

Green-Bar for C++
CUTE Plug-In for Eclipse

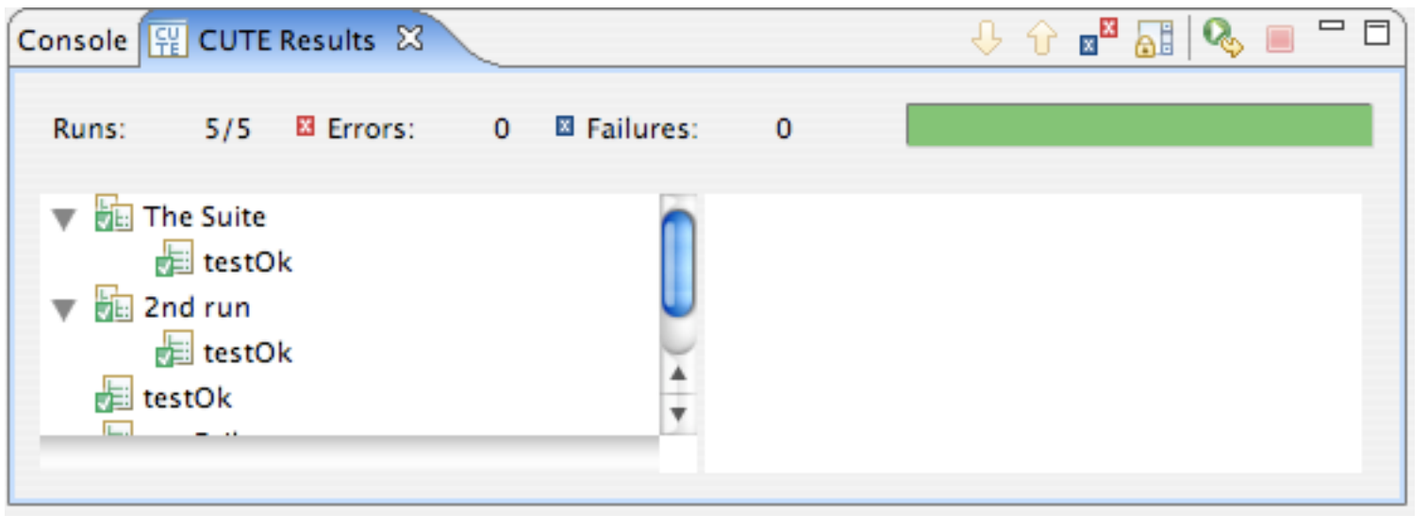
INSTITUTE FOR SOFTWARE




```
void testFindOperator(){
    std::string s("Hallo");
    std::string tobefound("ll");
    std::string::size_type pos= s % tobefound;
    ASSERT(pos == 2);
}


void runSuite(){
    cute::suite s;
    //TODO add your test here
    s += CUTE(testFindOperator);
    cute::eclipse_listener lis;
    cute::makeRunner(lis)(s, "The Suite");
}

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



Outlook/Questions

INSTITUTE
FOR
SOFTWARE

- ...