INSTITUTE FOR SOFTWARE

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL

INFORMATIK

AgilePT 2010

CUTE GUTs for GOOD
Good Unit Tests drive Good OO Design

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

Plus SCRUM Multi-Touch Table Demo Video

Peter Sommerlad

peter.sommerlad@hsr.ch

INSTITUTE FOR SOFTWARE

- **Work Areas**
  - Refactoring Tools (C++, Scala, Groovy, Ruby,...) for Eclipse
  - **Decremental Development** (make SW 10% its size!) + Tools!
  - **C++ Standardization**
  - Patterns and Software Engineering
    - Pattern-oriented Software Architecture (POSA)
    - Security Patterns
- **Background**
  - Diplom-Informatiker (Univ. Frankfurt/M, Germany)
  - Siemens Corporate Research - Munich
  - itopia corporate information technology, Zurich (Partner)
  - Professor for Software HSR Rapperswil, Switzerland 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

What is GOOD?
GOOd (OO) Design

INSTITUTE FOR SOFTWARE

- **Simple**
  - o C.A.R Hoare and E. Dijkstra
- **Encapsulation and Information Hiding**
  - o D. Parnas
- **High Cohesion & Low Coupling**
  - o L. Constantine
- **DRY - Don't Repeat Yourself**
  - o Pragmatic Programmers (A. Hunt, D. Thomas)
- **SOLID**
  - o R. Martin (Uncle Bob)
- **Relatively easy to detect violation, BUT also too easy to violate**

Famous Quotes by Sir C.A.R.(Tony) Hoare

- Inside every large program,
  there is a small program trying to get out.

- There are two ways of constructing a software design:
  - one way is to make it so simple that there are obviously no deficiencies, and
  - the other way is to make it so complicated that there are no obvious deficiencies.
- The first method is far more difficult.

SOLID principles

SRP - Single Responsibility Principle

WENGER
SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

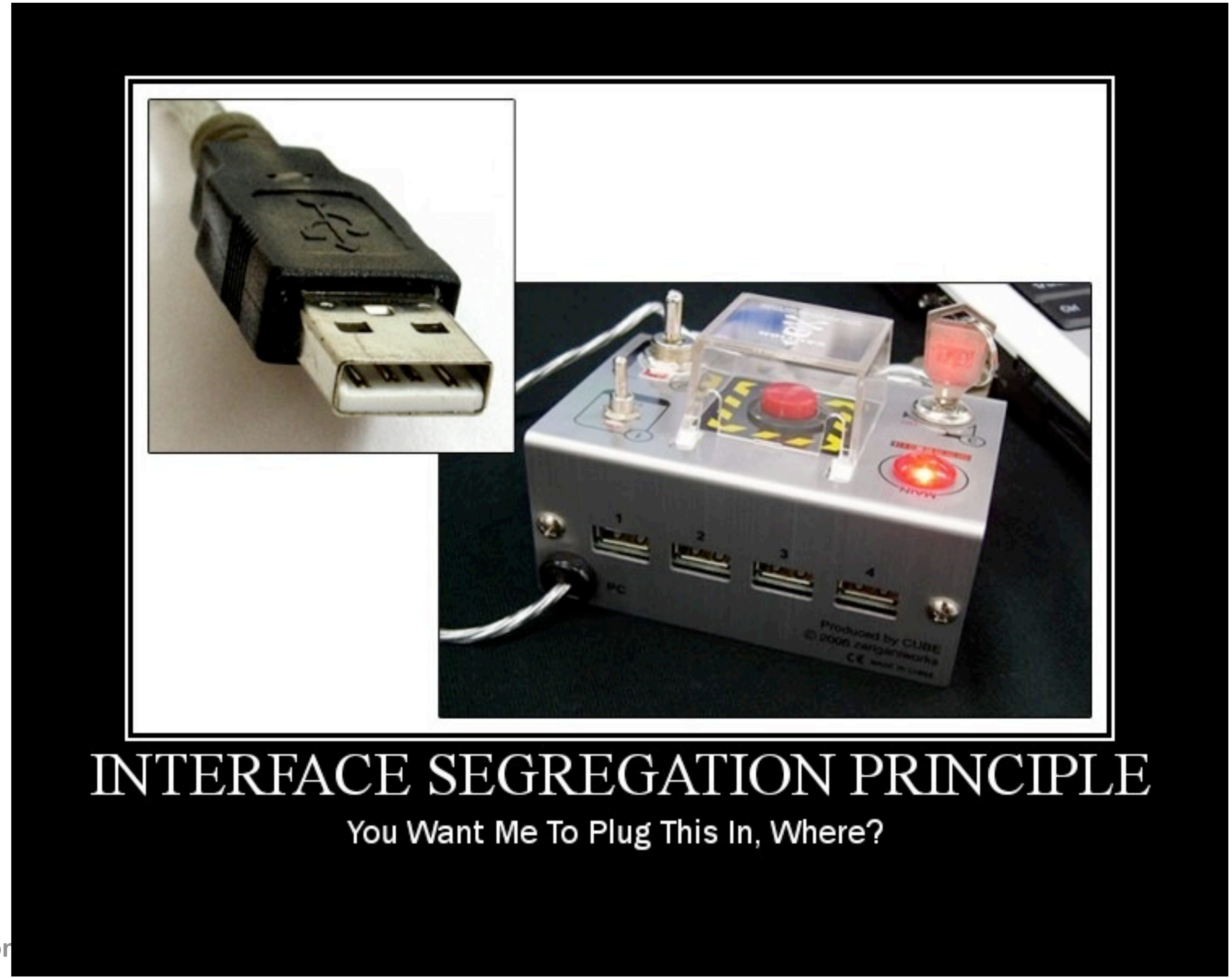
OCP - Open Closed Principle

LSP - Liskov Substitution Principle

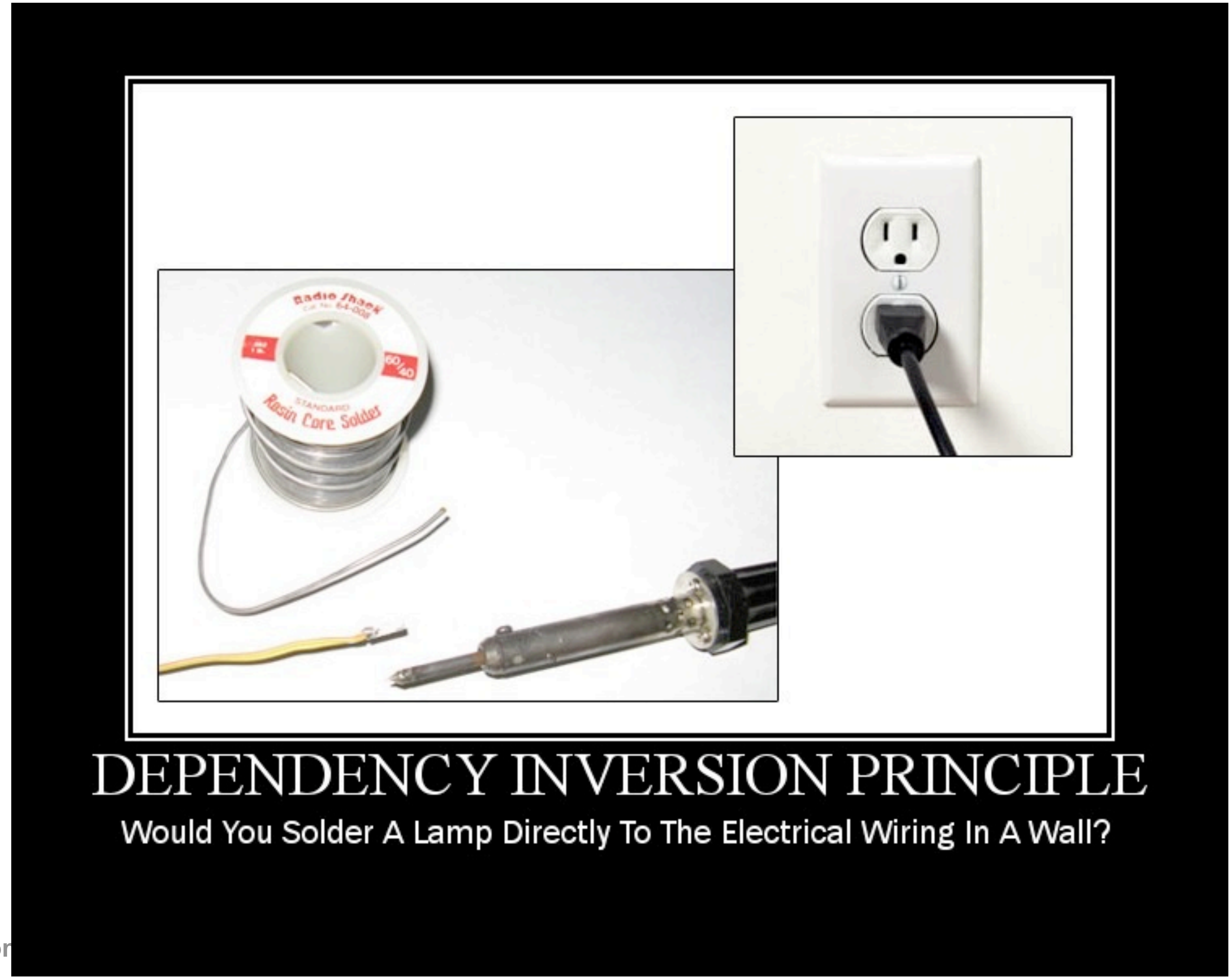
LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

ISP - Interface Segregation Principle

INSTITUTE FOR SOFTWARE


INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

DIP - Dependency Inversion Principle


DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

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

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

Caution: sales pitch ahead!

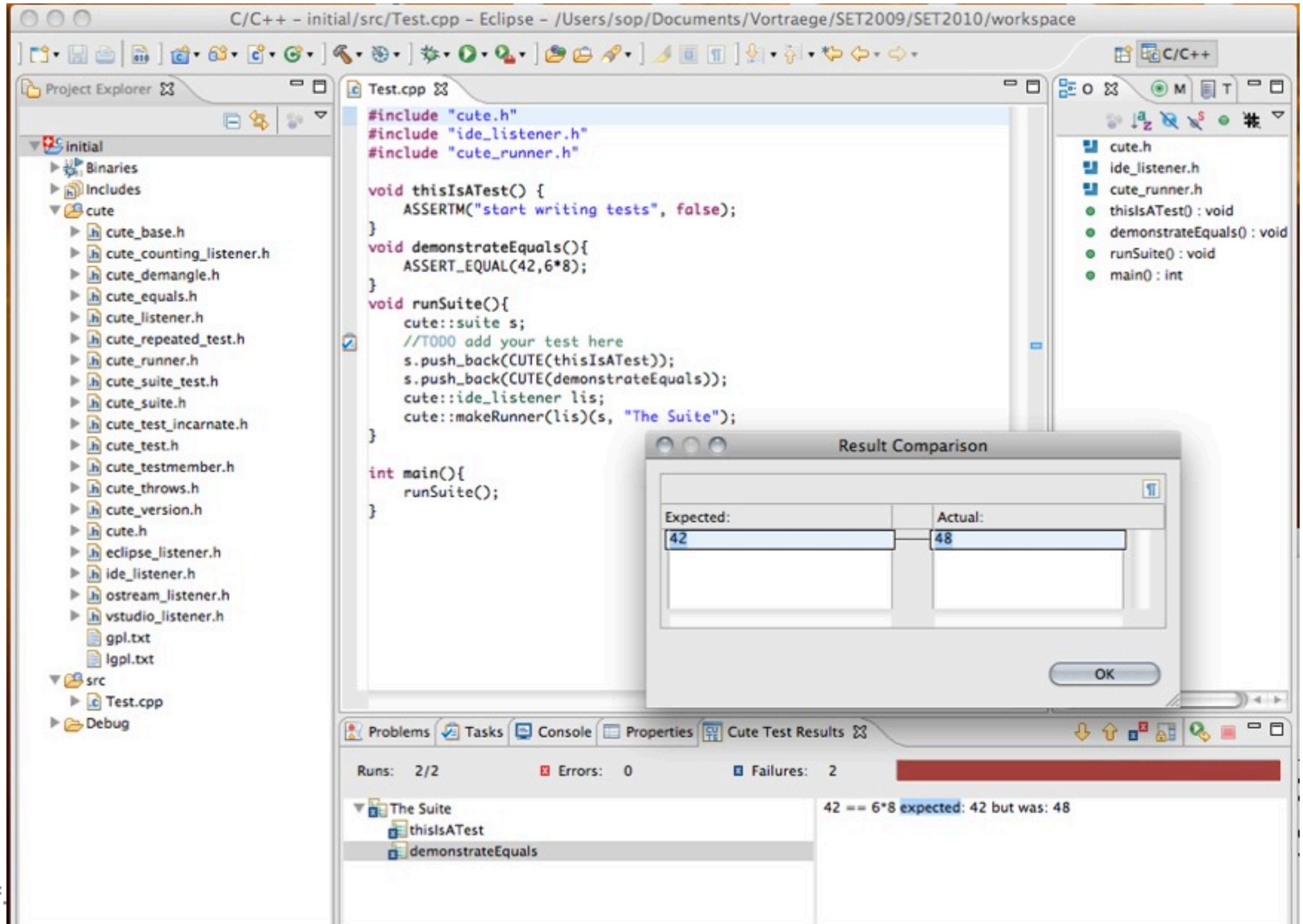
What is CUTE? C++ Unit Testing Easier

- **A simple to use C++ Unit Testing framework**
  - Header-only distribution! no library to link against
  - simple test functions, explicit test registration
  - 5 macros to learn: FAIL, ASSERT, ASSERT_EQUAL, ASSERT_EQUAL_DELTA, ASSERT_THROWS
    - 5 variations with suffix M to provide additional message
  - customizable output
- **an accompanying Eclipse CDT plug-in**
  - code-generation for test and test case registration
  - red-green bar viewer with test navigation and equality failure diff-viewer
  - tests also run in MS VS 2003/2008/2010

Why CUTE and not CPPUnit/GTest?

- **CPPUnit and GoogleTest are JUnit clones**
  - o try to re-create features available in Java (and alike) but not suitable to (standard) C++
  - o complicated use and design (GTest)
  - o provide too much than needed regularly
  - o too many fancy macros, restricted customizability
- **C++ is not Java**
  - o values are first class citizens, objects second class
    - ➢ automatic copy and assignment
    - ➢ deterministic life-time of variables and values
  - o (generic) types create values
    - ➢ and provide customization hooks
  - o (generic) functions are (almost) first class

CUTE Plug-in



C/C++ - initial/src/Test.cpp - Eclipse - /Users/sop/Documents/Vortraege/SET2009/SET2010/workspace

Project Explorer

- initial
  - Binaries
  - Includes
  - cute
    - cute_base.h
    - cute_counting_listener.h
    - cute_demangle.h
    - cute_equals.h
    - cute_listener.h
    - cute_repeated_test.h
    - cute_runner.h
    - cute_suite_test.h
    - cute_suite.h
    - cute_test_incarnate.h
    - cute_test.h
    - cute_testmember.h
    - cute_throws.h
    - cute_version.h
    - cute.h
    - eclipse_listener.h
    - ide_listener.h
    - ostream_listener.h
    - vstudio_listener.h
    - gpl.txt
    - lgpl.txt
  - src
    - Test.cpp
  - Debug

Test.cpp

```cpp
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"

void thisIsATest() {
    ASSERTM("start writing tests", false);
}
void demonstrateEquals(){
    ASSERT_EQUAL(42,6*8);
}
void runSuite(){
    cute::suite s;
    //TODO add your test here
    s.push_back(CUTE(thisIsATest));
    s.push_back(CUTE(demonstrateEquals));
    cute::ide_listener lis;
    cute::makeRunner(lis)(s, "The Suite");
}

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

Outline:

- cute.h
- ide_listener.h
- cute_runner.h
- thisIsATest() : void
- demonstrateEquals() : void
- runSuite() : void
- main() : int

Result Comparison

Expected:	Actual:
42	48

OK

Problems | Tasks | Console | Properties | Cute Test Results

```
Runs: 2/2        Errors: 0        Failures: 2
```

- The Suite
  - thisIsATest
  - demonstrateEquals

42 == 6*8 expected: 42 but was: 48

C++ Code Coverage with CUTE Eclipse plug-in

INSTITUTE FOR SOFTWARE

- **The CUTE Eclipse plug-in also provides code coverage visualization**
  - o for GCC gcov
  - o like eclEMMA for Java
- **Run tests with code coverage shows uncovered production code**
  - o and also not-run test code
- **We also are creating a plug-in for C/C++ header file optimization that visualizes "static coverage"**
  - o this allows you to find unused declarations and definitions in your (header) files

C++ static code analysis Gimpel Software's lint

INSTITUTE FOR SOFTWARE

- **(Agile) Java programmers (should) use FindBugs**
  - o static analysis tool that detects common programming mistakes
- **(Agile) .NET programmers (should) use FXCop**
- **C/C++ programmers (might) use PC-Lint (Windows) or FlexeLint (other OSs)**
  - o lint's output is text-only and can be overwhelming
- **IFS' students created a FlexeLint CDT plug-in**
  - o visualizes lint messages in Problems View and editor
  - o provides Quick-fixes for correcting errors/ suppressing false positives
  - o will be available commercially (by end of 2010)

Agile C++ and IFS

INSTITUTE FOR SOFTWARE

- **CUTE testing framework**
  - o free open source
- **CUTE Eclipse CDT plug-in with code coverage**
  - o free open source
- **C++ Refactoring in Eclipse CDT**
  - o free open source (some features not yet integrated)
  - o more useful C++ refactorings to come
- **Lint viewer plug-in for Eclipse CDT**
  - o plan to make it commercially available
- **ReDHead header file optimization plug-in for CDT**
  - o plan to make it commercially available
  - o organize #include like Java "organize imports"

End of sales pitch :-)

An Observation

INSTITUTE FOR SOFTWARE

- If the design of code is not GOOD
  - then writing automated (unit) tests for it is hard to impossible

and vice versa

- If it is hard to write automated (unit) tests
  - then the design of the code is often bad

Unit tests are a good indicator of design quality!

My Assumption

- **Writing automated unit tests improves design**
  - almost automatically

- **under the pre-requisite that we refactor the code accordingly**
  - sometimes needed up front to achieve initial testability
    - there is a whole book by Michael Feathers on that topic: "Working effectively with Legacy Code"

Example:
A hard to test Date class

INSTITUTE FOR SOFTWARE

```
#ifndef DATE_H_
#define DATE_H_

class Date {
   int day, month, year;
   static const int daysPerMonth[];
public:
   Date(); // today
   Date(int day, int month, int year);
   virtual ~Date();
   void print();
   void add(Date const &period);
   void add(int days);
};

#endif /* DATE_H_ */
```

- How can we check if Date() actually fills in the date correctly?
- How can we check that adding days or another Date is correct?
- making everything public is not "nice"

A test program for Date

INSTITUTE FOR SOFTWARE

```
#include "Date.h"

int main(){
   Date d;
   d.print();
   d.add(1); // tomorrow
   d.print();
   d.add(Date(1,0,0)); // the day after
   d.print();
   d.add(Date(0,1,0)); // next month
   d.print();
   d.add(Date(0,0,1)); // next year
   d.print();
}
```

- **What does it tell us?**
- **Can we be sure it works?**
- **What's bad about it?**
- **Is this really a GUT?**

Date's implementation reveals more ugliness

INSTITUTE FOR SOFTWARE

```
#include "Date.h"
#include <ctime>
#include <iomanip>
const int Date::daysPerMonth[]
    ={31,28,31,20,31,30,31,31,30,31,30,31};

Date::Date() {
    time_t tnow=time(0);
    struct tm now(*localtime(&tnow));
    day = now.tm_mday;
    month = now.tm_mon;//+1;
    year = now.tm_year;//+1900;
}

Date::Date(int day, int month, int year)
:day(day),month(month),year(year)
{}

Date::~Date() {
    // TODO Auto-generated destructor stub
}
void Date::add(const Date & other)
{
    day += other.day;
    month += other.month;
    year += other.year;
}
```

```
void Date::print()
{
    std::cout << std::setfill('0')
    << std::setw(2) << day << "."
    << std::setfill('0') << std::setw(2)
    << month <<"."<<std::setw(4)<<year<<"\n";
}
void Date::add(int days)
{
    day += days;
    while (days > daysPerMonth[month-1]
          ||days>29&&month==2&&!(year%4)){
        days -=daysPerMonth[month-1];
        if (month==2 && !(year%4)) days--;
        month++;
        while (month>12){
            month = 1;
            year++;
        }
    }
}
```

Try to write tests

INSTITUTE FOR SOFTWARE

- **A first CUTE test**
  - o constructor wouldn't throw -> create a Date.
  - o not very interesting, do not want to check for internals (might change -> test case breaks)
- **need to refactor first**
  - o need means to check Date's output
  - o observation print() depends on global variable cout -> pass in std::ostream& as parameter

```
void Date::print()
{
   std::cout << std::setfill('0')
   << std::setw(2) << day << "."
   << std::setfill('0') << std::setw(2)
   << month <<"."<<std::setw(4)<<year<<"\n";
}
```

Example
enable output checking

INSTITUTE FOR SOFTWARE

```
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"

#include "Date.h"
void constructAndOutputDate() {
    Date d(18,5,2010);
    std::ostringstream out;
    d.print(out);
    ASSERT_EQUAL("18.05.2010",out.str());
}

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

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

```
#ifndef DATE_H_
#define DATE_H_
#include <iosfwd>

class Date {
	int day, month, year;
	static const int daysPerMonth[];
public:
	Date();
	Date(int day, int month, int year);
	virtual ~Date();
	void print();
	void print(std::ostream &out)const;
	void add(Date const &other);
	void add(int days);
};

#endif /* DATE_H_ */
```

```
void Date::print(){
    print(std::cout);
}
void Date::print(std::ostream &out)const
{
    out << std::setfill('0')
    << std::setw(2) << day << "."
    << std::setfill('0') << std::setw(2)
    << month <<"."<<std::setw(4)<<year;
}
```

add print(std::ostream&) overload

- **extract std::cout dependency**
- **class now better usable**
  - can output Date values through std::cerr, std::clog, stringstreams, files, etc.

- **const'ness of member function print enables even more uses**
  - ➢ should add const to print() also

- **Only checking Date's output is too little testing**
  - would be better if we could ASSERT_EQUAL on Date values
    - ➢ introduce operator== on Date's

Example introduce operator==

```
void equalsDateIsReflexive() {
   Date d(18, 5, 2010);
   ASSERT_EQUAL(d,d);
}
void equalsDateTwoDates() {
   Date d(18, 5, 2010);
   ASSERT_EQUAL(Date(18,5,2010),d);
}
void equalsDateDifferentDatesAreUnequal(){
   ASSERT(Date(18,5,2010)!=Date(19,5,2010));
}
```

```
class Date {
      int day, month, year;
      static const int daysPerMonth[];
public:
      Date();
      Date(int day, int month, int year);
      virtual ~Date();
      void print();
      void print(std::ostream &out)const;
      void add(Date const &other);
      void add(int days);
   bool operator==(Date const &other) const;
   bool operator!=(Date const &other) const {
      return !(*this == other);
   }
};
#endif /* DATE_H_ */
```

- **Date now better usable**
- **more to do**
  - e.g., operator<()
    - ➢ use boost/operators.hpp to automatically add further relational ops

```
bool Date::operator==(const Date & other) const
{
    return day==other.day
        && month == other.month
        && year == other.year;
}
```

  - ➢ but first let's fix other problems

Other Observations

- **C++ uses (overloaded) operators for addition, subtraction and for output**
- **adding Dates doesn't make sense**
  - need something similar representing time periods
    - Introduce class Period
  - Subtracting 2 Dates should result in a Period
- **Default date of "today" hard to test, because of environment dependency.**
- **Adjustment of days, months and years inconsistent (not shown today -> Homework)**
  - does not work with negative "days"
  - tuple representation might not be optimal for that

More interactive examples

INSTITUTE FOR SOFTWARE

- **demo in Eclipse CDT with CUTE plug-in**

Conclusion

- **GUTs are beneficial for GOOD**
  - Even if tests are added after the fact they can help improving your design
    - However, Refactoring is essential
- **CUTE is easy to use (especially with Eclipse)**
  - simpler than alternatives (CPPUnit, GTest)
  - more modern C++ (values, std:: library, no explicit memory management needed)
  - requires boost and/or std::tr1 or C++0x
    - USE_TR1, USE_STD0X macros control impl. used
  - used in teaching and by international users
    - open source
  - provides also test coverage view with gcov

Outlook

- **C++ as a language does not stand in your way if you want to be Agile**
  - o the language (especially with the new standard to be finished soon) combines high-level abstractions without performance penalties or platform limitations (i.e., VM availability)
- **We are creating tools for catching up with an agile working style for C++ developers**
  - o and are filling some gaps with really innovative solutions, i.e., with our ReDHeaD (**ReD**uce **Hea**der **D**ependencies) plug-in
-

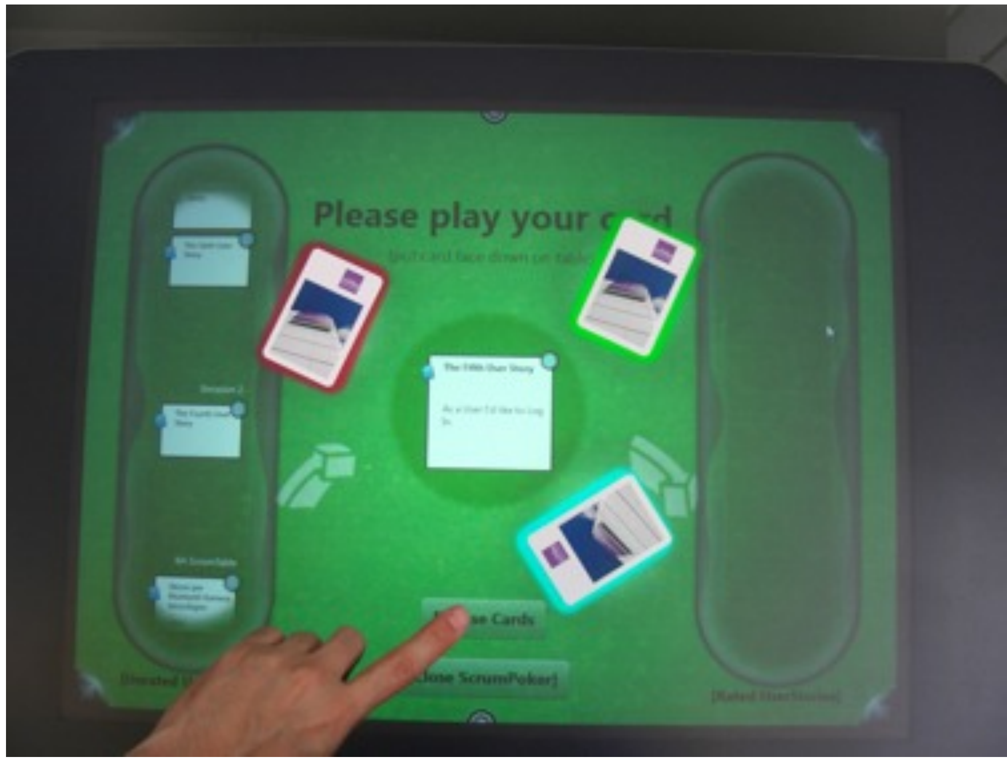
Sales pitch again, sorry

SCRUM
on multi-touch table

Bachelor Thesis 2010 Scrum Table

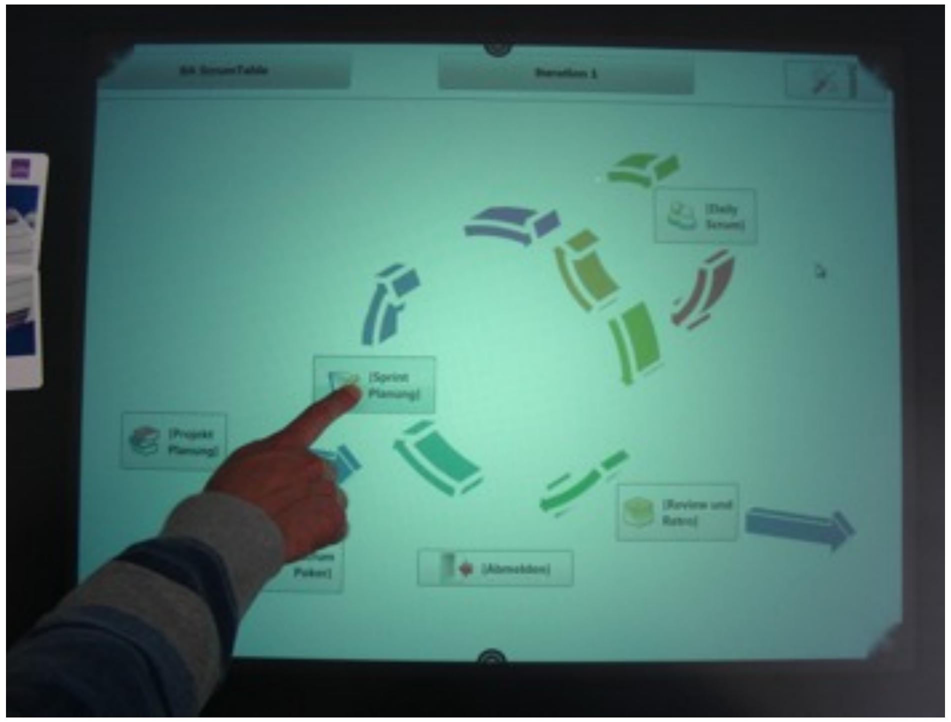
INSTITUTE FOR SOFTWARE

- **Goal: Simpler & more efficient SCRUM project management**
- **User Interface Technology: Microsoft Surface**
- **Project Repository: MS Team Foundation Server 2010**
  - very un-agile UI in its plain form
    - i.e. multiple dialogs needed to create a single story card/backlog item
- **Videos:**
  - http://www.youtube.com/watch?v=upr6ifM4cl4 ***watch***
  - http://www.youtube.com/watch?v=FvGs3PJu5Iw ***watch***

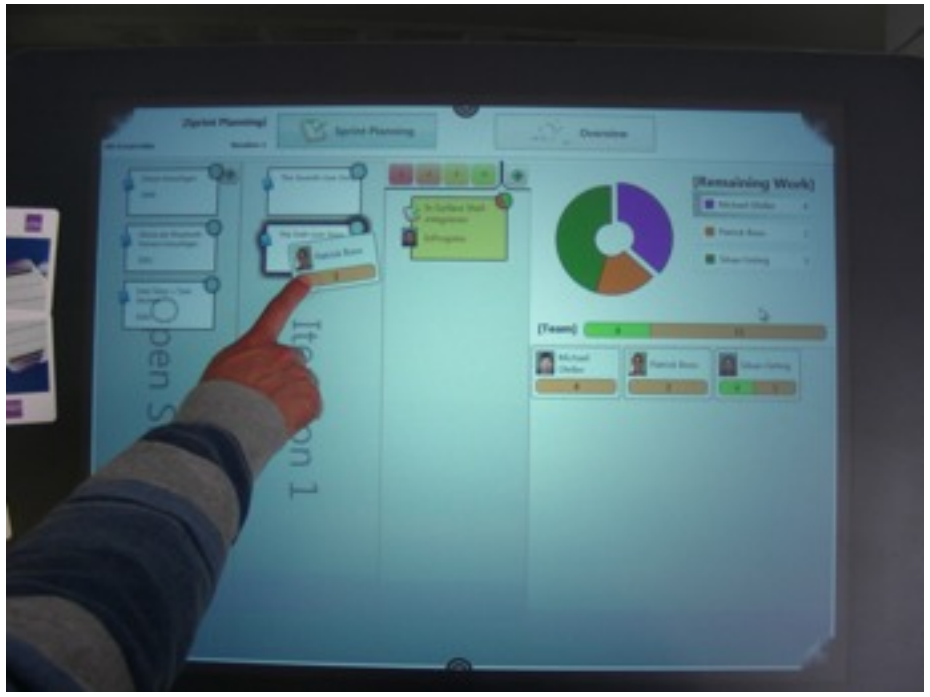


Scrum Table Screen Shots

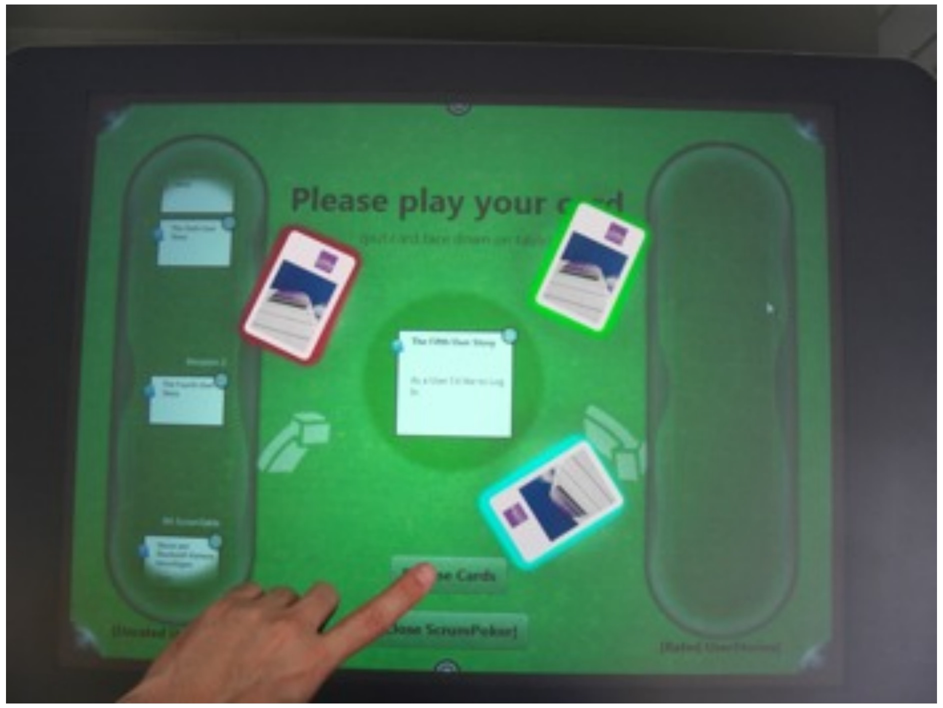
Process Overview



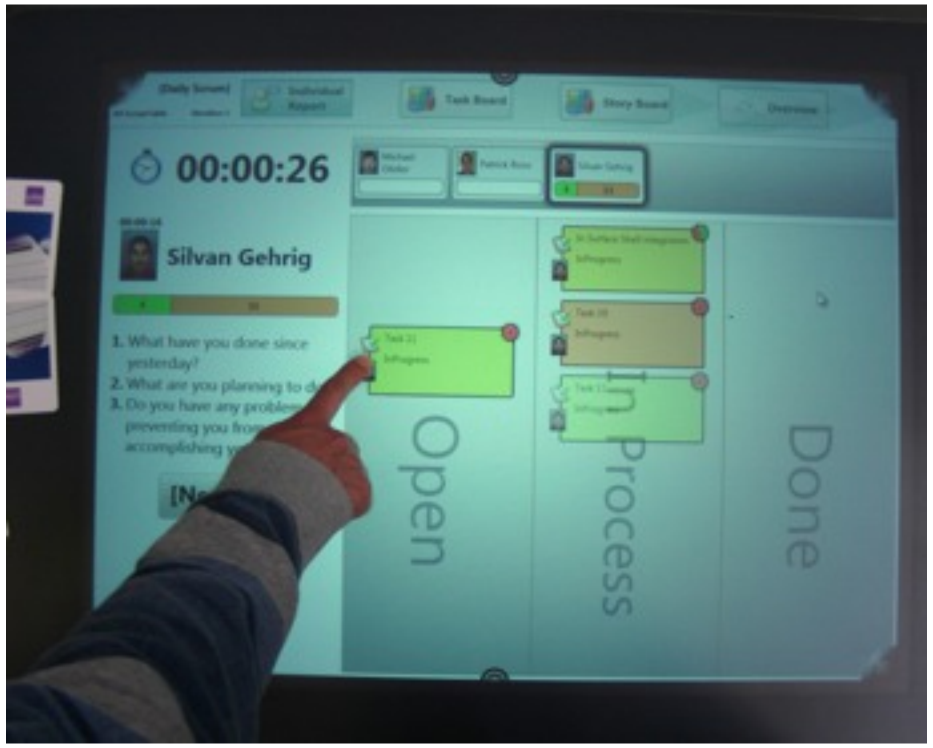
Sprint Planning



Scrum Poker

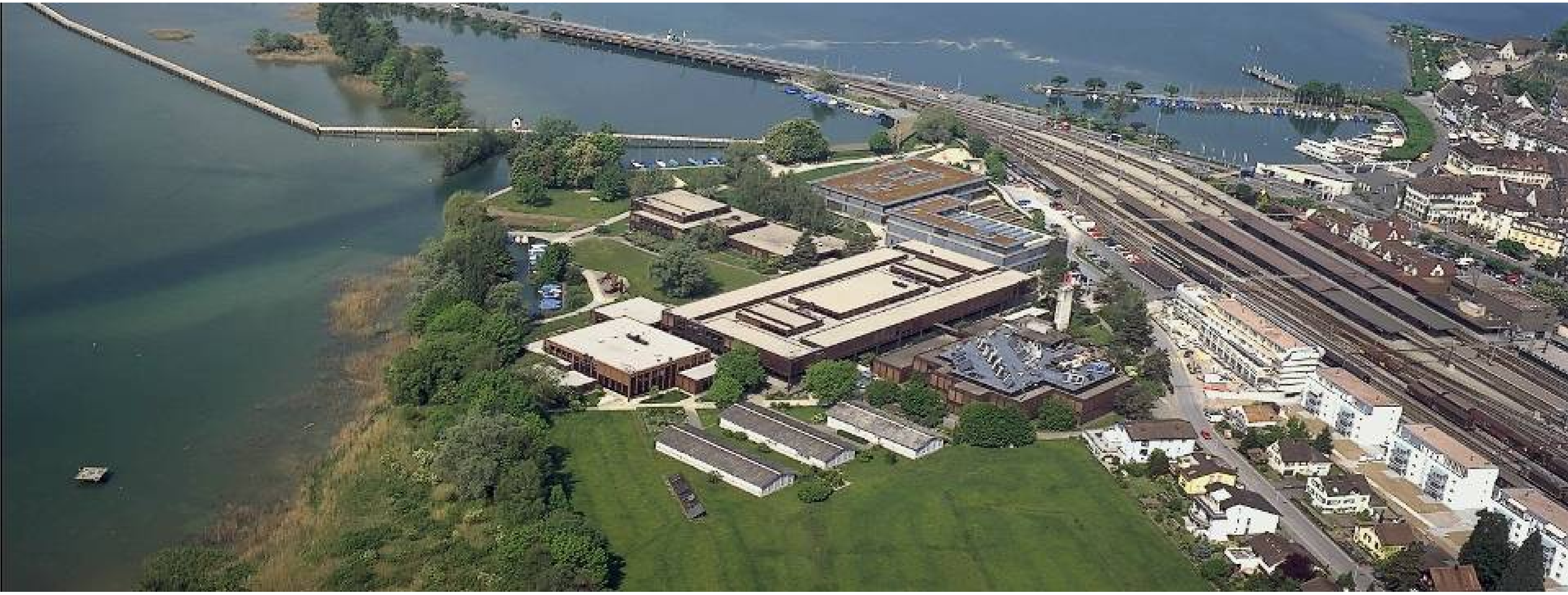


Daily Scrum



Questions?

INSTITUTE FOR SOFTWARE



- more on CUTE at http://r2.ifs.hsr.ch/cute
  - and http://ifs.hsr.ch/Cute.5820.0.html
- or contact me at **peter.sommerlad@hsr.ch**