CERN-ITC Code Analysis Project 2004-2007

Paolo Tonella
ITC-irst, Centro per la Ricerca Scientifica e Tecnologica
Povo, Trento, Italy
tonella@itc.it

ITC-irst

Research Centre conducting research in Computer Science, Microsystems and Surface Physics.

- Interested in collaborations on advanced research topics.
- Develops research prototypes within funded research projects.

Software Engineering research at ITC-irst

STAR

Software Technology Advanced Research

`http://star.itc.it`

- Source code analysis and manipulation.
- Testing.
- Programming languages.

Past CERN-ITC project 1999-2003

- Reverse engineering of UML diagrams from C++ code

Paolo Tonella, Alessandra Potrich,
Reverse Engineering of Object Oriented Code
ISBN: 0-387-40295-0
224 pages, 80 illustrations, Hardcover.
Series: Monographs in Computer Science
Springer-Verlag, 2005.

- Automated verification of coding conventions
  - Involved experiments: Alice, ATLAS
  - Prototype tool: **RuleChecker**
  - Tool users: Alice, ATLAS, IT-Control (PVSS), Root, etc.
  - Availability: http://spi.cern.ch/ → External Software

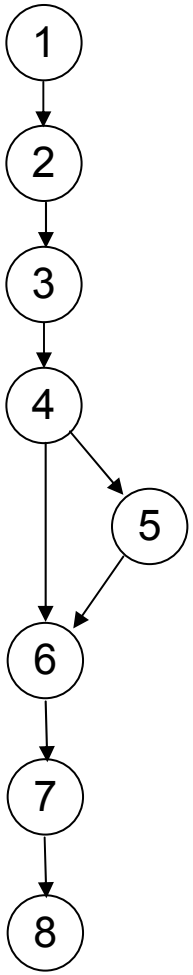
New CERN-ITC project 2004-2007

- Automated test case generation
- Migration to Aspect Oriented Programming
- Code smell detection

Automated test case generation

Coverage testing

```
1   program P
2   begin
3     input(x);
4     if (x > 0)
5         x++;
6     end if
7     print(x);
8   end program
```



A program can be delivered only when all its statements (or branches, paths, etc.) have been traversed in some test case.

1. Statement coverage
2. Branch coverage
3. Condition coverage
4. Path coverage

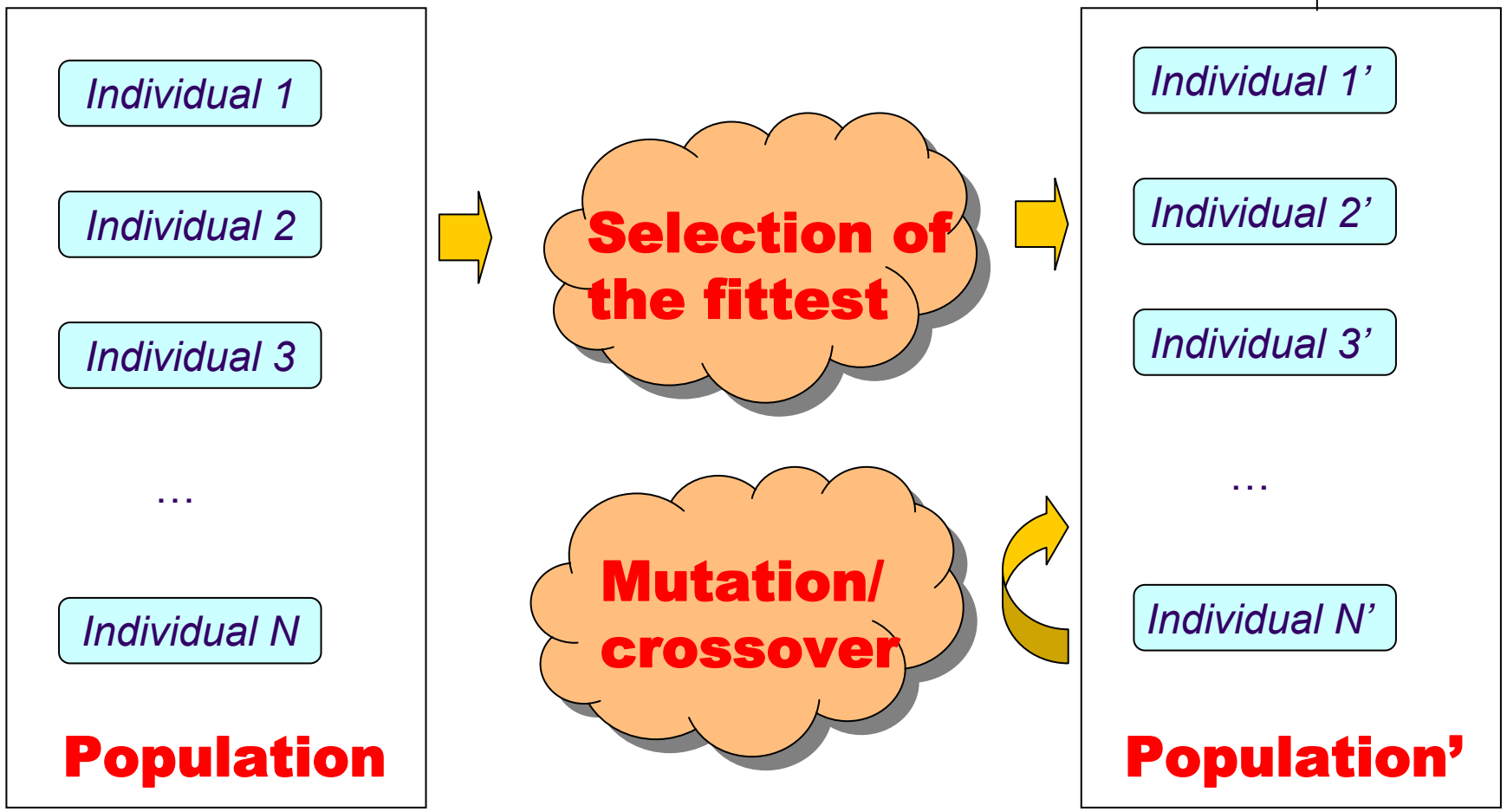
```
x = 1
```

⇒ Coverage: 1

```
x = 0
x = 1
```

⇒ Coverage: 1, 2, 3, 4

Genetic algorithms



After some evolutionary steps, the fittest individual approximates the searched optimum of the objective function

Evolutionary testing

- Test cases are individuals of a population
- Chromosomes encode test input values:
  (v1, v2, …, vN)
- Test cases are evolved by means of mutation (e.g., change value) and crossover (e.g., swap input value tails)
- The fittest individuals are the test cases that get closer to the (current) target of test execution (e.g., covering a given branch)

Chromosomes

Procedural code: (v1, v2, …vN)

Object-Oriented code: $x0=A(): $x0.f(): $x0.g() @ v1, v2, …vN

$a=A(int):$b=B():$b.f(int):$a.m(int, $b) @ -1, 2, 5

chromosome variables

input values

```
class TestA extends TestCase {
    public void testCase1() {
        A a = new A(-1);
        B b = new B();
        b.f(2);
        a.m(5, b);
    }
}
```

Random chromosome construction

1. A constructor for the object under test is randomly selected: `$a=A(int)@-1`
2. The invocation of the method under test is appended:
   `$a=A(int):$a.m(int,$b)@-1,5`
3. All required object constructions are inserted:
   `$a=A(int):$b=B():$a.m(int,$b)@-1,5`
4. Method invocations to change the state of the created objects are randomly inserted:
   `$a=A(int):$b=B():$b.f(int):$a.m(int,$b)@-1,2,5`

Steps 3 and 4 are repeated until all chromosome variables used as method or constructor parameters are properly initialized (**well-formedness** of the resulting chromosome).

Random generation of input values

Default, parameterized and customized input generators:

A.m(int)	Default integer generator: uniform selection in [0, 100]
A.m(int[-2;2])	Parameterized integer generator: uniform selection in [-2, 2]
A.m(int[MyIntGenerator])	Customized integer generator: method newIntValue() from class MyIntGenerator is called to obtain the value
A.m(bool)	Default boolean generator: **`true`** and **`false`** are equally likely
A.m(string)	Default string generator: characters are uniformly chosen from [a-zA-Z0-9], with the string length decaying exponentially
A.m(string[DateGenerator])	Customized string generator: only strings representing legal dates are produced (e.g., “3/3/2003”)

Fitness

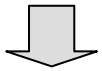
Selection of the fittest test cases requires:

- Generation of the execution trace for each test case
- Overlap between the execution trace and the control/call dependences leading to the target (branch not yet covered)

Mutation operators

Change input value:

$a=A(int):$b=B():$b.f(int):$a.m(int, $b) @ -1, 2, 5



$a=A(int):$b=B():$b.f(int):$a.m(int, $b) @ -1, 4, 5

Mutation operators

Crossover:

Well-formedness must be maintained

$a=A(int):$b=B():$b.g():$a.m(int, $b) @ 0, -3

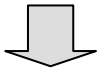
$a=A():$b=B():$b.f(int):$a.m(int, $b) @ -1, 2

$a=A(int):$b=B():$b.g():$b.f(int):$a.m(int, $b) @ 0, -1, 2

$a=A():$b=B():$a.m(int, $b) @ -3

$a=A():$b=B(int):$c=C(int):$b.h($c):$b.f():$a.m(int, $b) @ 1, 4, 5

$a=A(int, int):$b=B():$a.m(int, $b) @ 0, 3, 6



$a=A():$b=B(int):$a.m(int, $b) @ 1, 6

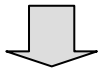
$a=A(int, int):$b=B():$c=C():$b.h($c):$b.f():$a.m(int, $b) @ 0, 3, 5

new

Mutation operators

Constructor change:

$a=A(int):$b=B():$b.f(int):$a.m(int, $b) @ -1, 2, 5

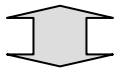


$a=A():$b=B():$b.f(int):$a.m(int, $b) @ 2, 5

Mutation operators

Insertion/removal of method call:

$a=A(int):$b=B():$b.f(int):$a.m(int, $b) @ -1, 2, 5

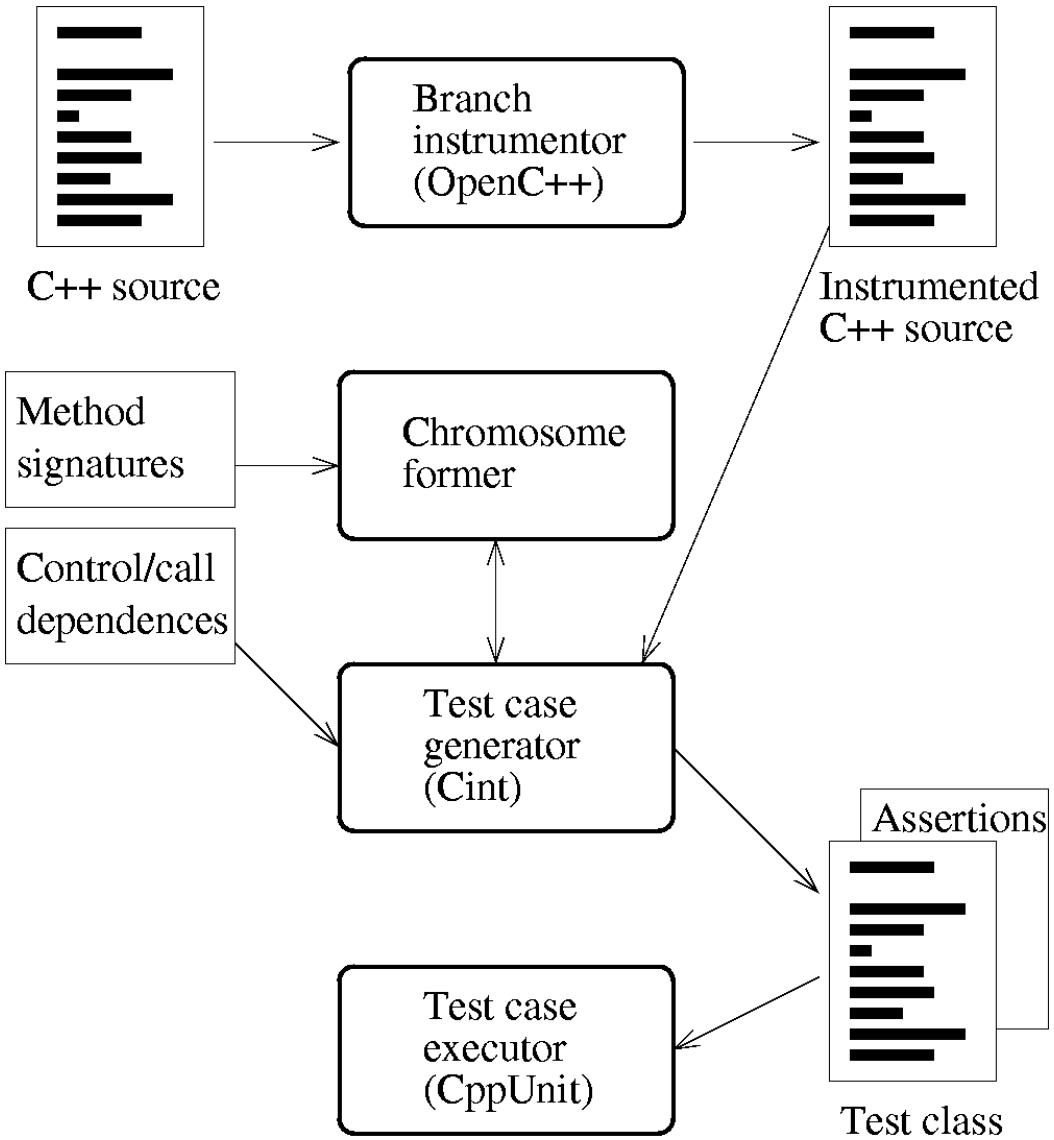


$a=A(int):$b=B():$a.m(int, $b) @ -1, 5

Evolution

- During chromosome evolution, each test case covering a previously uncovered branch is added to the final test suite
- The final test suite is minimized by means of a greedy heuristics

eToc++: a tool for the evolutionary testing of C++ classes



Branch instrumentor

- Every control flow branch has to be uniquely identified and traced upon execution
- The reflection capabilities of OpenC++ (source-to-source transformation) are exploited to instrument the code, by adding tracing instructions
- The instrumented code is printed to file

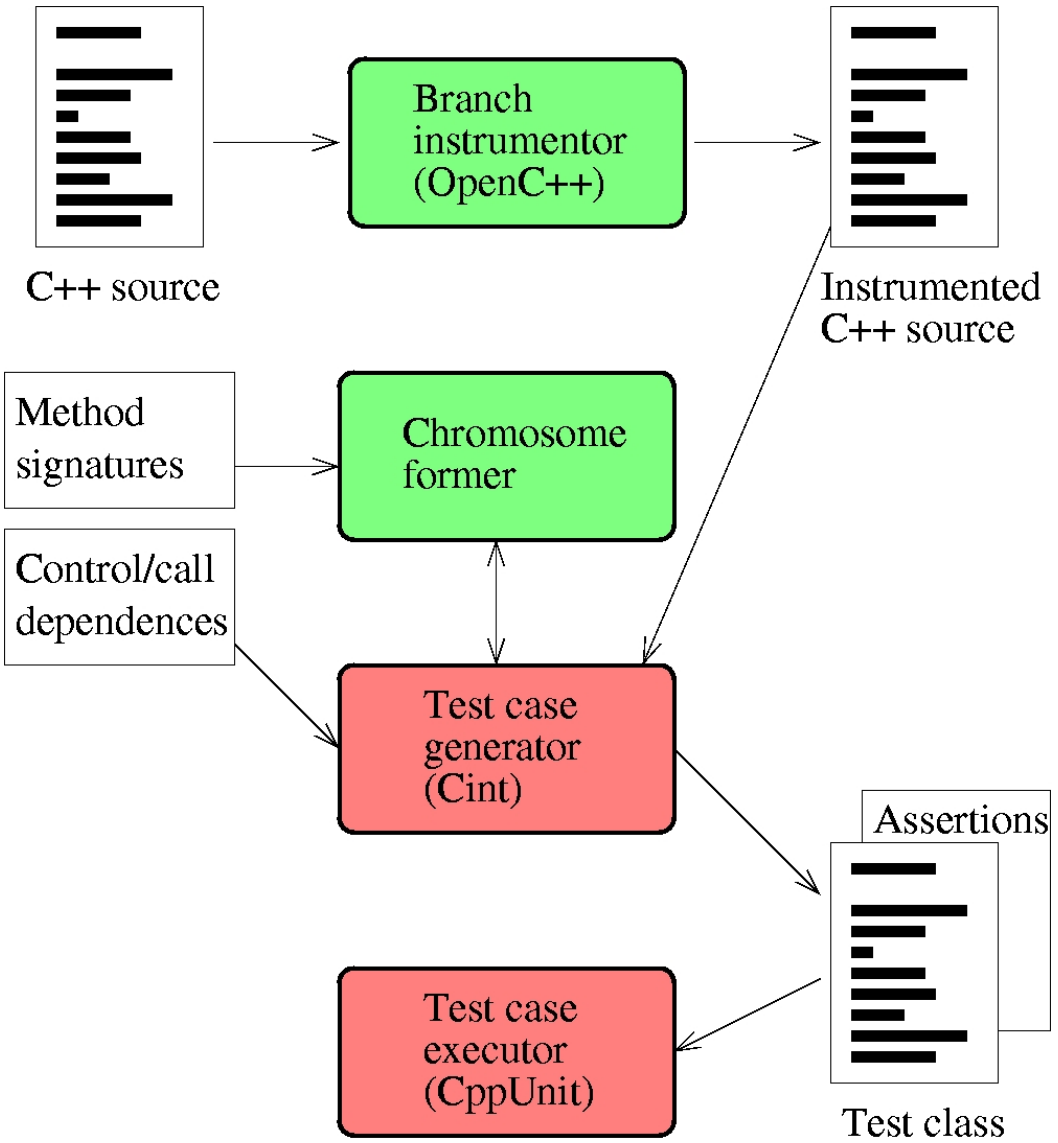
Chromosome former

- Builds chromosomes randomly for the initial population
- Changes chromosomes according to the mutation operators
- Produces input values using the default, parameterized or customized generators

Test case generator

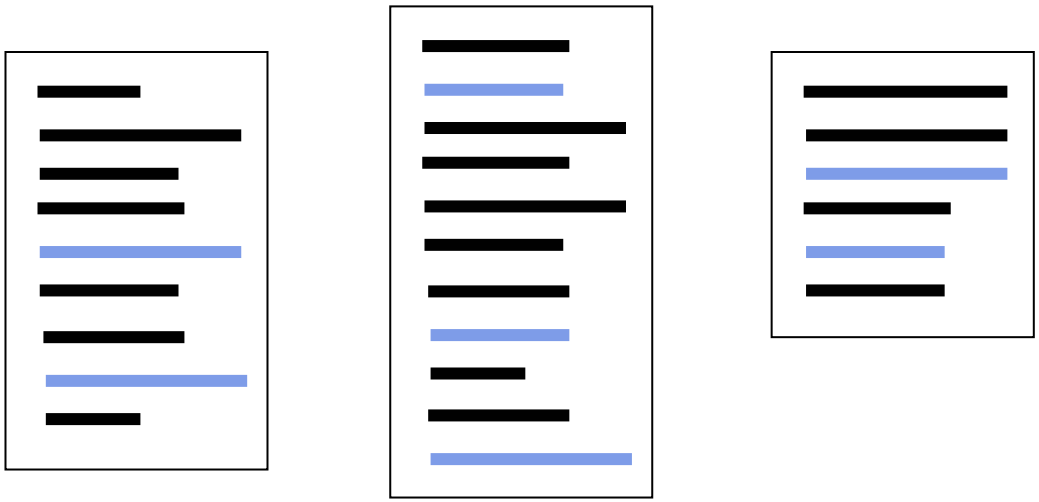
- Implements the genetic algorithm
- Uses Cint to execute test cases encoded as chromosomes and to determine the execution traces
- Applies a greedy minimization procedure on the resulting test suite
- Produces a CppUnit test class as output
- Assertions must be eventually added manually

State of the project



Migration to Aspect Oriented Programming

Crosscutting concerns



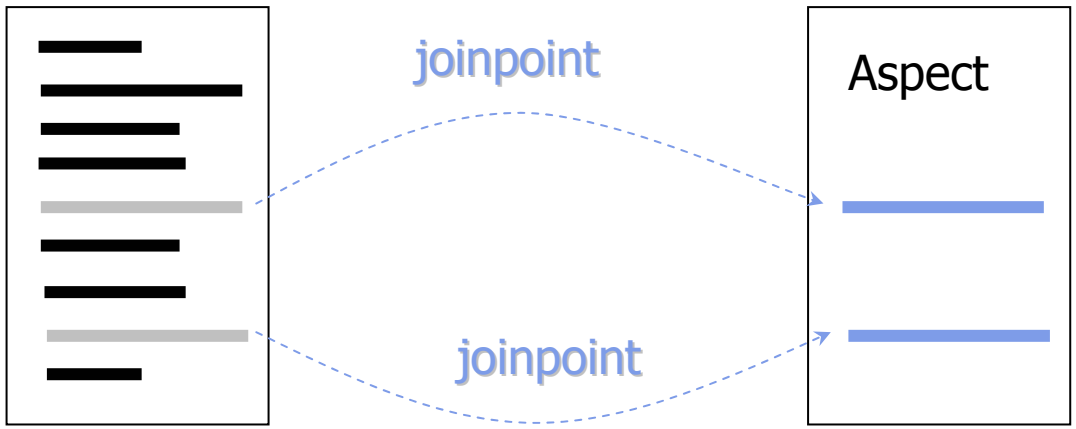
Example:

Synchronization code in a multi-thread application

- Crosscutting concerns are inherent in any complex applications
- Aspects provide a mechanism to factorize them

Join points

- A *join point* is a well-defined point in the program flow, where execution can be intercepted by an aspect.



Pointcuts

- *Pointcuts* select certain join points and values at those points.

Unnamed pointcuts

```
call(void Point.setX(int))
```

```
call(void Point.setX(int)) ||
call(void Point.setY(int))
```

Named pointcut

```
pointcut move():
        call(void FigureElement.setXY(int,int)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int)) ||
        call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point));
```

Pointcuts


```
call(void Figure.make*(..))
```

```
call(public * Figure.* (..))
```

Wildcards in pointcuts

```
cflow(move()) &&
call(void Figure.get*(..))
```

join points occurring in the context of another pointcut

Advices

- *Before advice* runs when a join point is reached and before the computation proceeds.
- *After advice* runs after the computation 'under the join point' finishes.
- *Around advice* runs when the join point is reached, and blocks the computation under the join point, until an explicit *proceed* instruction is executed.

```
after(): move() {
      System.out.println("A figure element moved.");
}
```

Exposing context in pointcuts

- Pointcuts can expose part of the execution context at their join points.
- Values exposed by a pointcut can be used in the body of the advice.

```
pointcut setXY(FigureElement fe, int x, int y):
        call(void FigureElement.setXY(int, int)) &&
        target(fe) && args(x, y);

after(FigureElement fe, int x, int y): setXY(fe, x, y) {
        System.out.println(fe + " moved to (" + x +
        ", " + y + ").");
}
```

Introductions

An *introduction* can

- add methods to an existing class
- add fields to an existing class
- extend an existing class with another
- implement an interface in an existing class
- convert checked exceptions into unchecked exceptions

```
aspect CloneablePoint {

  declare parents: Point implements Cloneable;

  declare soft: CloneNotSupportedException:
      execution(Object clone());

  Object Point.clone() {
      return new Point(x, y);
  }
}
```

Example: Observer design pattern

```
aspect PointObserving {

  private Vector Point.observers = new Vector();

  public static void addObserver(Point p, Screen s) {
        p.observers.add(s);

  }

  public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);

  }

  pointcut changes(Point p): target(p) &&
        call(void Point.set*(int));

  after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
                updateObserver(p, (Screen)iter.next());

        }

  }

  static void updateObserver(Point p, Screen s) {
        s.display(p);

  }

}
```

Aspects to be investigated during the project

1. **Debug**: printout of function calls, including input and output values (before/after the call)
2. **Counter**: how often is a function called and from where.
3. **Timer**: how much time is spent in a function
4. **Memory**: how much memory is used/allocated by functions
5. **Histogramming**: fill debug values, counters, timings into a histogram

AspectC++

- General-purpose aspect-oriented extension to C/C++
- The compiler is freely available under the GPL license
- http://www.aspectc.org/

State of the project

- Preliminary assessment of **AspectC++** gave positive results
- Prototype implementation of the **debug** aspect in AspectC++

Code smell detection

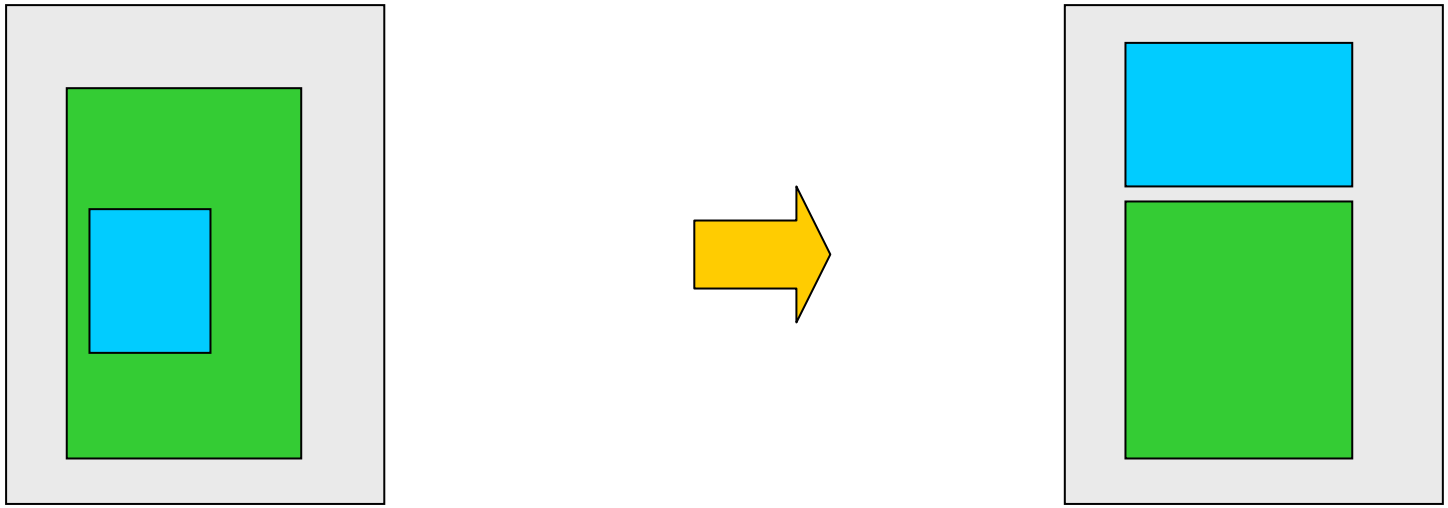
Refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, while it improves its internal structure.

- Disciplined way to clean up code.
- The design of the system is improved after the code has been written.
- Design occurs continuously during development.
- The risks associated with the production of a good design from the very beginning are reduced.

Extract method

Refactoring Extract Method: *When a sequence of logically related statements can be grouped together, they can be turned into the body of a method, whose name should explain the isolated behavior. Referenced variables should be made available as parameters and/or return values, if not visible.*

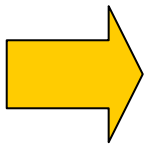


Renaming

Good code should communicate what it is doing clearly, and variable names are a key to clear code. Anybody can write code that a computer can understand. Good programmers write code that humans can understand.

Refactoring Renaming: *If the name of an entity does not reveal its purpose, it should be changed. All references to such an entity must be changed accordingly. Moreover, conflicts with existing entities must be avoided when choosing the new name.*

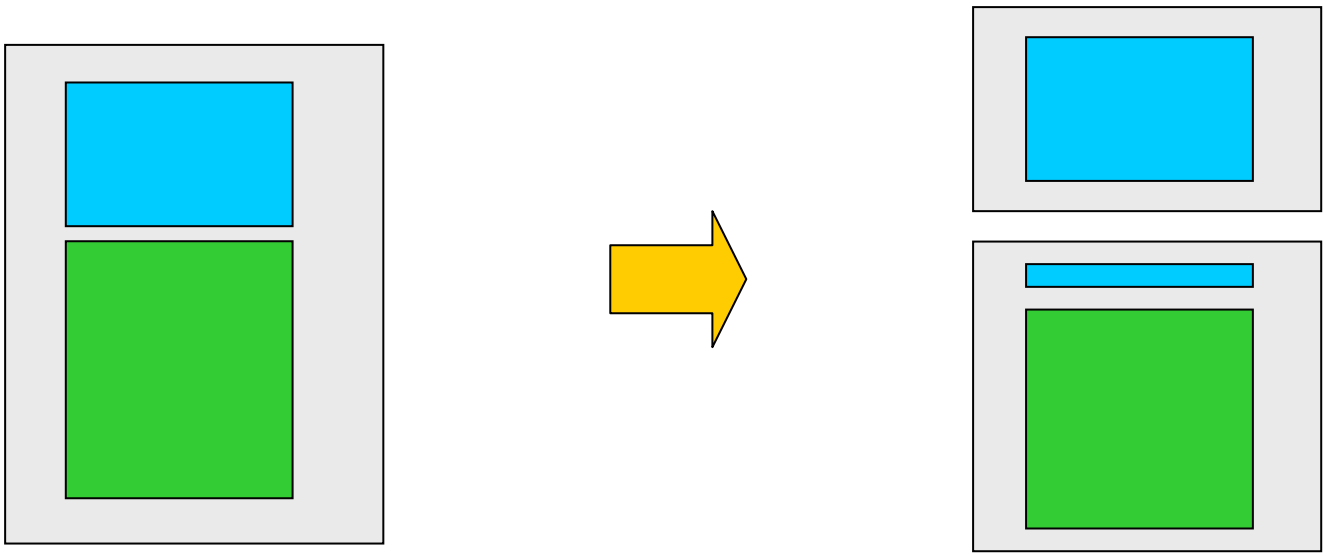
```
...
amountFor(Rental each)
```



```
...
amountFor(Rental aRental)
```

Move method

Refactoring **Move Method**: *If a method is, or will be, using or used by more features of another class than the class in which it is defined, a new method with a similar body can be created in the class it uses most. The old method can either be turned into a simple delegation, or it can be removed altogether.*



Introducing polymorphism

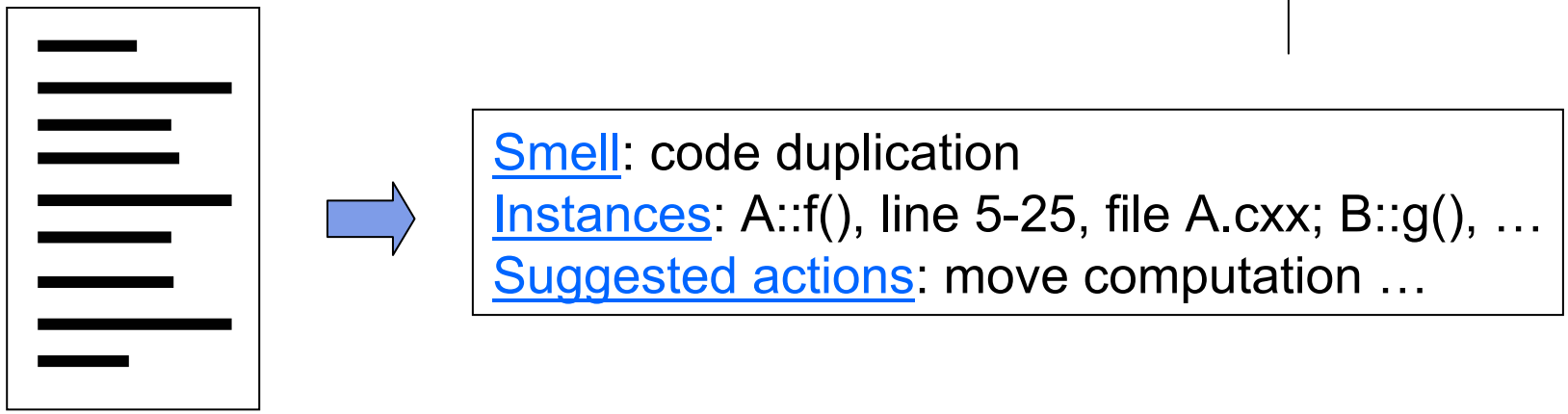
Polymorphism allows avoiding an explicit conditional when the behavior of an object depends on its type.

- If conditional code is present, each time a new type is added, all conditionals sparsed in the code have to be found and updated.
- On the contrary, if conditional code is replaced with polymorphism, it is sufficient creating a new subclass and providing the appropriate methods.
- Clients of a class don't need to know about the subclasses, thus reducing the dependencies in the system and simplifying its update.

Code smells

- Indicators of areas of the programs where refactoring could be beneficial
- Smells can be detected automatically, but the final assessment of the actual need of refactoring is manual (both false positives and false negatives are expected)

Smell detector



- Suggest improvements to programmers
- Identify problematic sub-systems (those with more smell detected)
- Guide code inspection
- Help reasoning on overall design

Preliminary list of code smells

1. Code duplication
2. Long methods
3. Parallel inheritance hierarchies
4. Message chains
5. Large classes with low cohesion
6. Feature envy (high coupling)
7. Switch statements
8. Data class
9. Refused bequest (inherited features remain unused)
10. Data clumps (data clusters not grouped into a class)

State of the project

- The reflection capabilities of **OpenC++** have been assessed and judged adequate for the smell detection task
- A **preliminary list of code smells** was defined (see previous slide)

Conclusions

- The project will give us the opportunity to investigate advanced research topics
- Project deliverables consist of research prototypes to be possibly integrated into the CERN software process infrastructure
- Working with large and complex C++ systems will give us the possibility to conduct interesting empirical studies on the usefulness of the investigated techniques