Entwicklung eines Werkzeugs zur Überdeckungsmessung für strukturorientierte Programmtests

Ronny Treyße

HU Berlin

Gliederung

- Überblick über das Projekt
- Die Komponenten
- Die Schnittstellen
- Die Umsetzung für Java und C++
- Die Benutzerinteraktion per GUI
- Verbindung zu ATOSj
- Ausblick

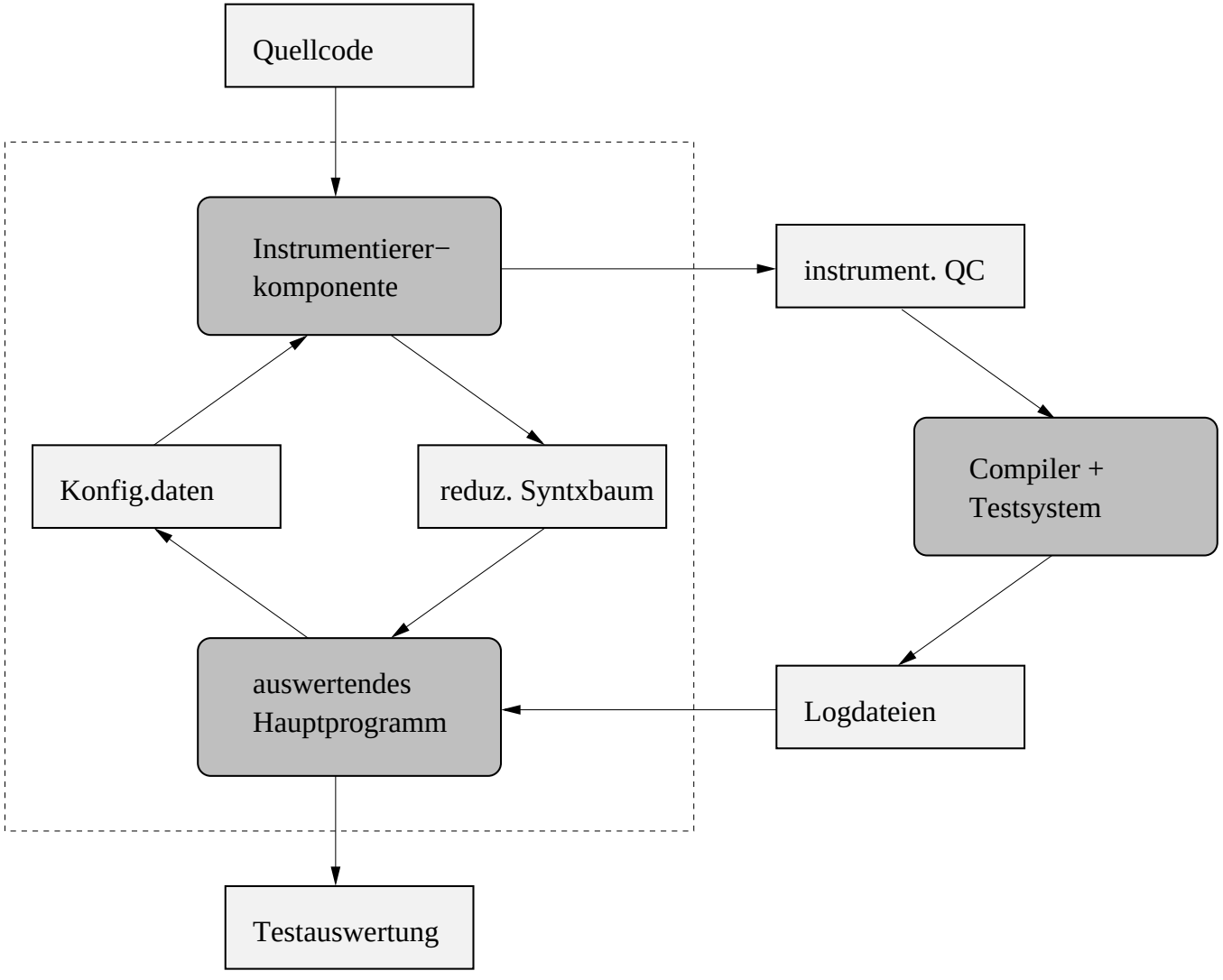
Überblick über das Projekt

- Ziel: Bewertung strukturorientierter Programmtests
- Programmiersprachen: Java und C++
- folgende Überdeckungsmaße sollen unterstützt werden:
  - Anweisungsüberdeckung (C0)
  - Zweigüberdeckung (C1)
  - einfache (C2) und mehrfache (C3) Bedingungsüberdeckung
  - minimal mehrfache Bedingungsüberdeckung
  - MC/DC-Überdeckung
  - Boundary-Interior-Pfadtest

Überblick über das Projekt

- Zweitteilung der Aufgabenstellung:
  - Parser-Subsystem
    - Parsen und Instrumentieren von Quellcode (Java, C++)
    - Erstellen eines Abstrakten Syntaxbaumes (AST)
    - Einlesen der Logdatei
  - GUI und Auswertung (Hendrik Seffler)
    - Projektverwaltung und Nutzerinteraktion
    - grafische Darstellung der Überdeckungsmaße
    - Testfallverwaltung
    - Report-Erstellung

Überblick über das Projekt



Parser-Subsystem: Die sprachunabh. Komponenten

- ASTManager:
  - verwaltet AST-Baum
  - vermittelt Zugriff auf Parser, Logreader
  - veranlasst Parsen, Instrumentieren und Loglesen
  - verarbeitet Konfigurationen für die Instrumentierung
- grundlegender AST-Baum:
  - bietet sprachunabhängigen Zugriff auf die geparste Quelldatei
  - fast komplett abstrakt
- LogReader:
  - liest standardisiertes Logfile in AST-Baum ein

Parser-Subsystem: Die sprachabh. Komponenten

- Parser:
  - parst Quelldatei (gegebenenfalls zusätzliche Dateien)
  - erstellt kompletten AST-Baum
- abgeleitete AST-Baum:
  - sorgt für sprachspezifische Verknüpfungen im AST-Baum
  - ermöglicht Instrumentierung in der Zielsprache
- LoggingKomponente:
  - stellt Logging-Funktionalität in der Zielsprache zur Verfügung

Schnittstellen zum GUI-/Auswertungssystem

- Parserkomponente kann sowohl alleinstehend als auch von der Auswertungskomponente angesprochen werden
- Funktionalität gegenüber Auswertungskomponente:
  - Initialisierung
  - Auslösen des Parsevorganges
  - Konfiguration der Instrumentierung
  - Auslösen der Instrumentierung
  - Auslösen des Loglesens
  - Übergabe projektbezogener Daten und des AST-Baumes

Konfiguration des Parsersystems

- die Arbeit des Instrumentierers lässt sich wie folgt spezifizieren:
  - Sprache des Quellcodes
  - Globales Instrumentierungslevel
  - Zuordnung einzelner Klassen und Funktionen zu Instrumentierungsleveln
  - Liste der zu verarbeitenden Dateien
  - Angabe eines Präfixes für generierte Variablennamen zur Vermeidung von Konflikten
  - Wahl des Namensformat für erstellte Dateien
- die Spezifikation kann über die GUI oder eine Konfigurationsdatei erfolgen

Instrumentierungslevel

- die Instrumentierungslevel haben dabei folgende Bedeutung:
- Level 0 - keine Instrumentierung
- Level 1 - einfach Instrumentierung, betrifft nur CFG-relevante Statements (C0 und C1)
- Level 2 - zusätzlich Instrumentierung für Bedingungsüberdeckungen (alle Überdeckungsmaße)
- Level 3 - vollständige Instrumentierung (jedes einzelne Statement)

Der Parsergenerator JavaCC

- JavaCC ist populärster Parsergenerator für Javaprojekte, wird aktiv supported
- ist OpenSource und unter BSD-Lizenz
- ist Top-Down-Parser, d.h. generiert LL(1)-Parser, aber ermöglicht einfach lokalen, syntaktischen und semantischen Lookahead
- bietet Unterstützung durch andere Java-Tools, z.B. JJTree zur Syntaxbaum-Konstruktion
- ToDo: aktuellere Grammatik für Java im Projekt umsetzen

Die Umsetzung für Java

- Grammatik
  - Java ist prinzipiell einfach zu parsen, da klar und einfach definiert
  - vorgefundene Java-Grammatik für JavaCC ist aktuell und wird weiterhin gewartet
- Problem bei der AST-Erstellung
  - schwierigstes Problem liegt in der Unterscheidung von boolschen und binären Operatoren
  - betrifft alle Bedingungsauswertungen
  - vollständige Lösung scheint relativ schwierig zu sein
  - (trifft auch auf C++ zu)

Problem der Operatorunterscheidung

- Beispiel:
  - `if( a == b ) ...`
- Frage: ist hier == boolscher oder binärer Operator?
- Antwort muss statisch erfolgen, damit der AST korrekt konstruiert werden kann
- Lösungsansatz: in Symboltabelle wird der Variablentyp vermerkt und beim Parsen verglichen
- Problem: Operand kann nichttrivial sein: `ClassX.getVariable()` ...

Die Umsetzung für C++

- Grammatik
  - C++ generell extrem schwer zu parsen, eigentlich unparsebar
  - vorgefundene Grammatik ist Date 97 ?!
  - ist für die Arbeit auf präprozessierten Quellcode gedacht
  - verarbeitet kein `namespace`, `typename`, `type_id`, keine differenzierten Casts (`dynamic_cast`, ...), ...
  - bei Testläufen kleinere Unverträglichkeiten bei speziellen Konstrukten

Probleme bei C++

- Typ-Erkennung für JavaCC-Grammatik
  - Unterscheidung zwischen Typ-Identifiers und Variablen-Identifiers notwendig
  - Beispiel:
    - `<ID> <ID> ( <ID> ) ;`
  - I Konstruktorinitialisierung, z.B.:
    - `ClassX instance_A(initValue);`
    - d.h.: `<Typ> <ID> ( <ID> ) ;`
  - II Funktionsdefinition, z.B.:
    - `ClassX function(ClassY);`
    - d.h.: `<Typ> <ID> ( <TYP> ) ;`

Probleme bei C++ - Lösungsansätze

- Lösungsansatz: Symboltabelle für ALLE Typen
- empfohlene Lösung:
  - Parsen präprozessierter Quelldateien
  - unpraktisch, da der Präprozesser den Quellcode stark verändern kann
- alternativer Ansatz:
  - Parsen der Header nach Typen zur Erstellung der Symboltabelle
  - unangenehm, da Headerdateien oft nichtstandardisierte Makrodefinitionen enthalten

Probleme bei C++ - Lösung

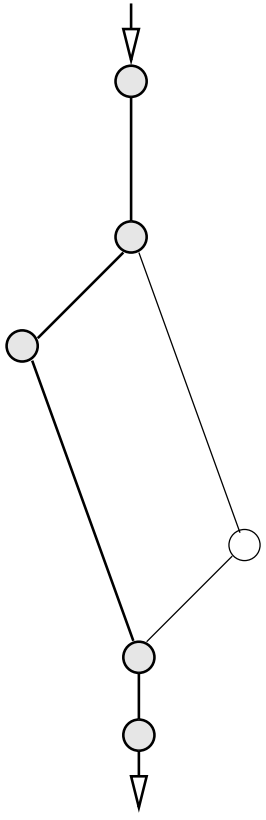
- praktikable Lösung - zwei Parserdurchläufe
  - 1. Parsen der präprozessierten Quelldateien nur nach Typen
  - Ergebnis: vollständige Symboltabelle für den Quellcode
  - 2. Parsen der originalen Quelldateien mit Hilfe der Symboltabelle
  - Ergebnis: AST des originalen Quellcodes

Benutzerinteraktion per GUI

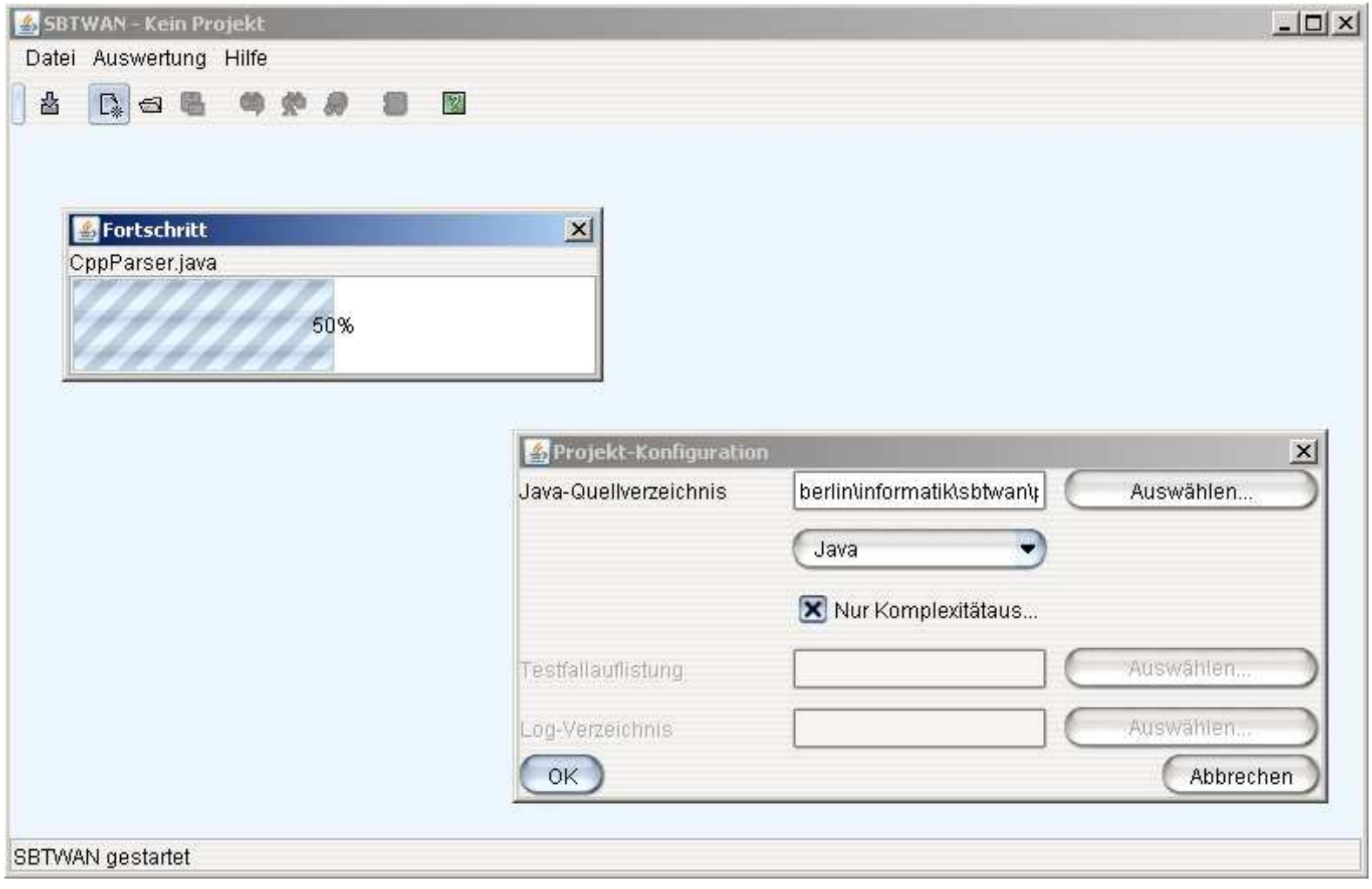
- vor dem Programmtest:
  - Projekterstellung
  - Einlesen der Quelldateien und unmittelbare Darstellung von Architekturinformationen, statischen Komplexitätsmaße und das KFG
  - Konfiguration und Instrumentierung
- nach dem Programmtest:
  - Einlesen der Logfiles und Testfalldefinitionen
  - Berechnung der Überdeckungsmaße und ihre Darstellung am KFG und Quellcode in Abh. einer Testfallmenge
  - Erzeugung eines Reports

Schema der Überdeckungsdarstellung

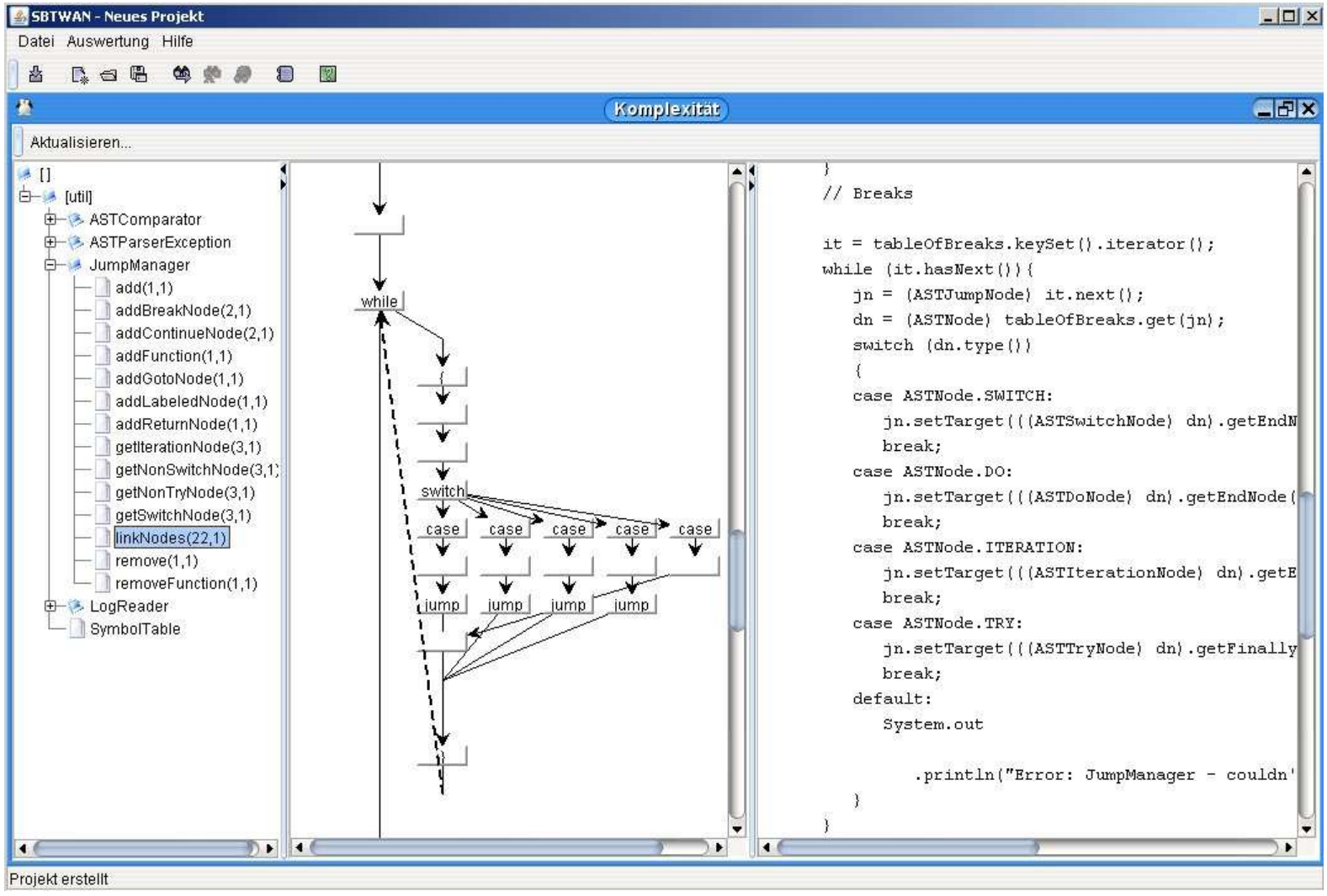
Testfälle	Klassenstruktur	Sourcecode	Kontrollflussgraph

Test 1
Test 2
Test 3
Test 4
	glob. Funktionen
	main()
	foo()
	Klassen
	class A
	foo_1()
	foo_2()
	class B
	foo_1()
	...
		`int main() {`
		`...`
		`if(a==1) {`
		`a.foo_2();`
		`}`
		`else {`
		`a.foo_1();`
		`}`
		`b.foo_2();`
		`}`
			

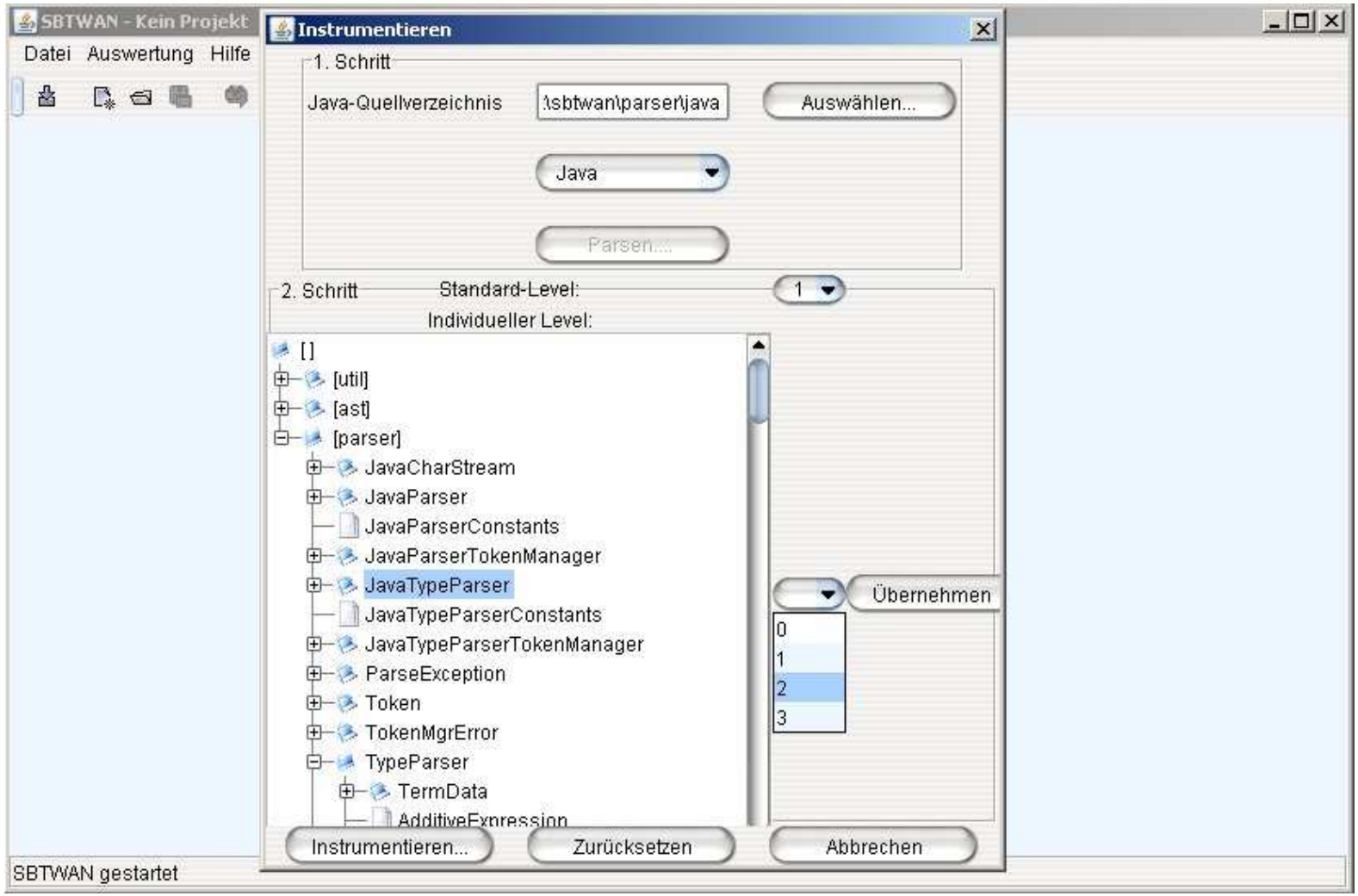
Projekterstellung



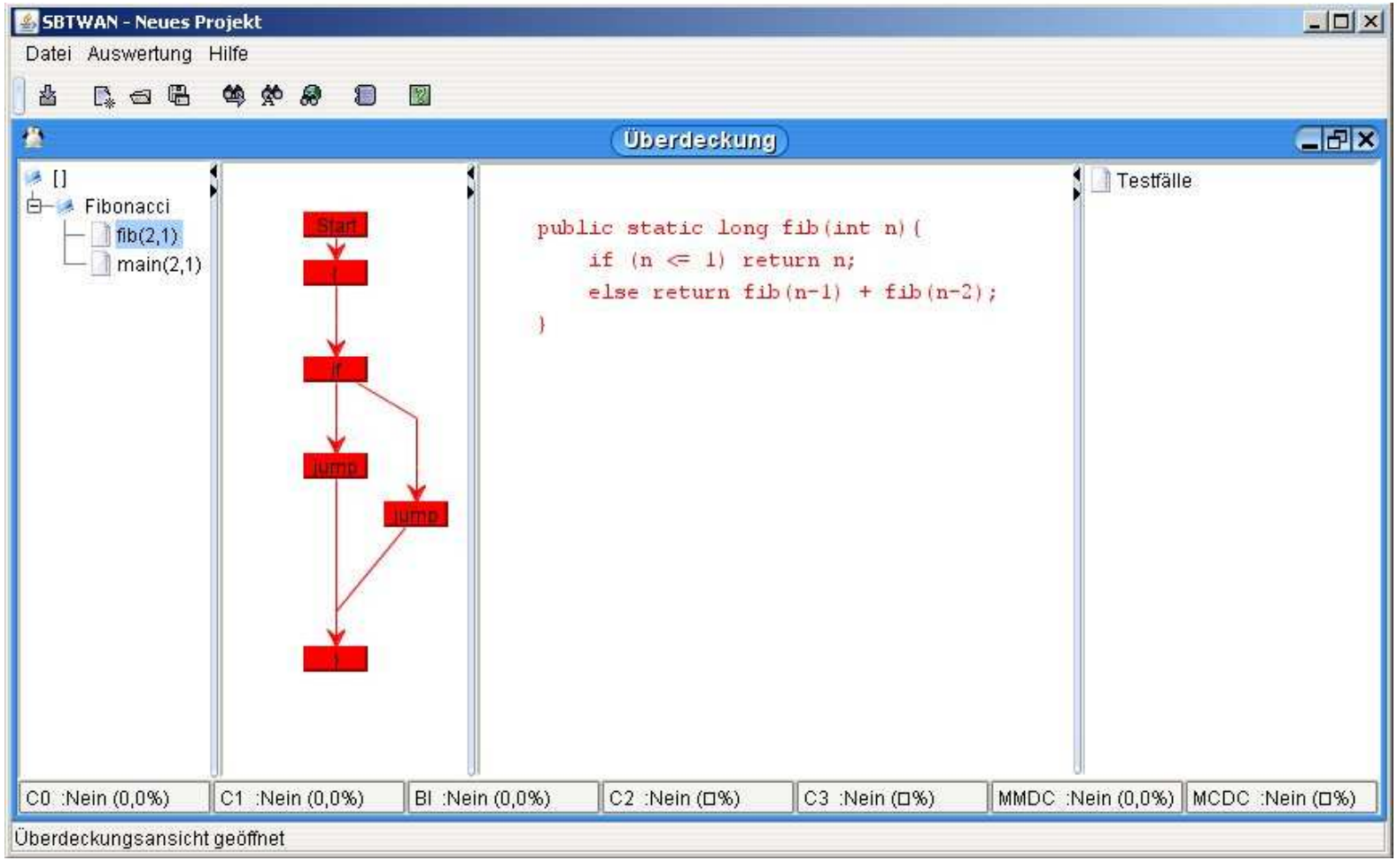
Statische Komplexitätsmaße



Konfiguration der Instrumentierung



Darstellung der Quellcode-Überdeckung



Verbesserungsmöglichkeiten

- Sicherheit: nicht alle Exceptions werden behandelt
- Information: Definition der Komplexitätmaße für Klassen
- Nutzerfreundlichkeit: Projektmanagement intuitiver gestalten
- bisher von Hand: Testfalldefinition (XML)

sbtwan und ATOSj

- inwieweit ist Kooperation sinnvoll?
- Kontra: Überdeckungsmaße jenseits von Methodenüberdeckung sind für den oberflächenbas. Regressionstest irrelevant, da zu genau (Tegos)
- Pro: Instrumentierung lässt sich sehr variabel einstellen, nicht vollständige Überdeckung ist das Ziel sonder Information über Aussagequalität des Tests
- Pro: Einsatz in der Lehre wäre mit aufeinander abgestimmten Programmen in jedem Fall praktischer
- aber beide Werkzeuge sollten in einer autonom funktionstüchtigen Version vorhanden sein!

Verknüpfung mit ATOSj

- mögliche Verknüpfungsszenarien:
  - Integration der vollständigen Funktionalität in ATOSj
  - Darstellung einer abgespeckten Überdeckungsmessung in ATOSj
  - lediglich Austausch über Testfälle

Ausblick

- Vorschläge für weiterführende Arbeiten durch Herrn Seffler:
  - flexiblere und informationsträchtigere Darstellung der Überdeckungsmaße, des KFG und des Quellcodes
  - variablerer Umgang mit Projektdateien und Testfallmengen
  - Auswertung der Informationen aus den Testfällen zu Profiling-Zwecken
  - Konstruktion von Aufrufgraphen
  - Unterstützung bei der Optimierung von Testfallmengen (z.B. für MC/DC)