Konformitäts- / Konventionsanalyse

- Der Compiler als statisches Analysewerkzeug
- Der Compiler führt auf jeden Fall eine Syntaxanalyse durch
- Abhängig vom Compiler werden weitere Prüfungen durchgeführt
  - Typprüfung
  - Ermittlung von nicht initialisierten Variablen
  - Erstellung von Verwendungsnachweisen
  - Über- oder Unterschreitung von Feldgrenzen
  - Nicht erreichbarer Code
  - ...

Datenfluss-anomalieanalyse

Datenflussanomalieanalyse

Compilerunterstützung ist abhängig von der Programmiersprache

- Frage: Gibt es bei der Verarbeitung der Daten Auffälligkeiten, die auf einen Fehler hindeuten?
- Eine Anomalie ist eine (aus statischer Sicht) auffällige Anweisungs-sequenz
  - Eine Anomalie muss nicht zwingend zu einem Fehler führen
  - Eine Anomalie muss genauer untersucht werden, um einen Fehler auszuschließen
- Eine Datenflussanomalie äußert sich in unstimmigen Variablenzugriffen
- Aus Eingabedaten werden über eine Sequenz von Anweisungen und Variablenzugriffen Ausgabedaten berechnet
- Ein spezielle Bearbeitungssequenz von Daten entspricht einem Pfad im Kontrollflussgraphen des Programms

- Mit einer Variablen *x* können auf einem Pfad verschiedene Aktionen ausgeführt werden

  wenn der Kontext klar ist, dann kann die Angabe der Variablen auch entfallen

  - *x* wird definiert (*d(x)*)
    - Jede Wertzuweisung führt zu einer neuen Definition der Variablen
    - Das Anlegen einer neuen Variablen führt nicht zwingend zu einer Initialisierung des Speicherplatzes (z.B. in C)
    - Hinweis: Eine Deklaration ist nur die Bekanntgabe eines Namens und führt nicht zur Reservierung von Speicherplatz
  - *x* wird referenziert (*r(x)*)
    - Eine Referenzierung kann ein *c-use* oder ein *p-use* sein
  - *x* wird undefiniert (*u(x)*)
    a) Der Speicherplatz wird der Variablen entzogen (z.B. der Gültigkeitsbereich einer lokalen Variablen wird verlassen)
    b) Der reservierte Speicherplatz wird nicht initialisiert

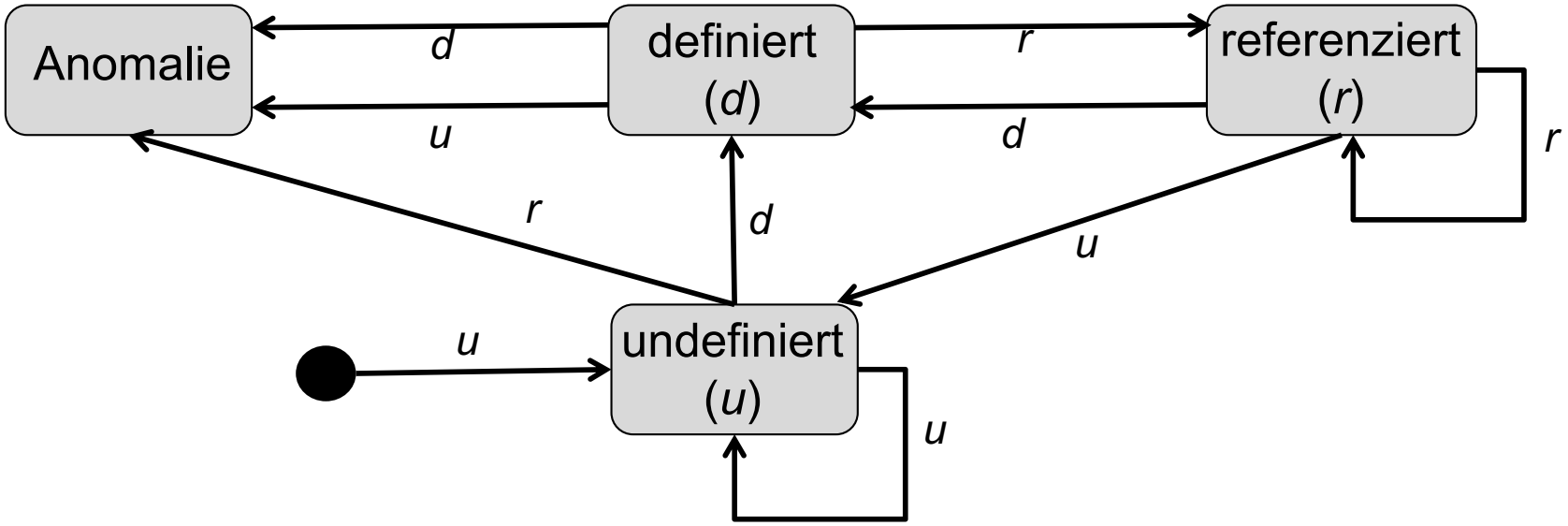
- Die folgenden Zugriffssequenzen auf eine Variable x stellen keine Anomalien dar

Zugriffssequenz	Beschreibung
dr	Variable bekommt neuen Wert und wird danach verwendet
rd	Variable wird verwendet und bekommt danach einen neuen Wert
rr	Variable wird zweimal hintereinander verwendet
ru	Variable wird verwendet und danach gelöscht
ud	Eine undefinierte Variable wird initialisiert
uu	Eine undefinierte Variable wird gelöscht

- Die folgenden Zugriffssequenzen stellen Anomalien dar

Zugriffssequenz	Beschreibung
ur	Variable wird ohne Initialisierung verwendet
du	Variable bekommt einen Wert und wird dann direkt gelöscht
dd	Variable wird zweimal hintereinander überschrieben

– Die Datenflussanomalieanalyse kann durch den folgenden Zustandsautomaten beschrieben werden



nach P. Liggesmeyer. *Software Qualität*. Spektrum Verlag, 2002

– Über den analogen endlichen Automaten kann eine reguläre Grammatik definiert werden
– Eine Datenflussanomalieanalyse ist damit effizient durch den Compiler realisierbar

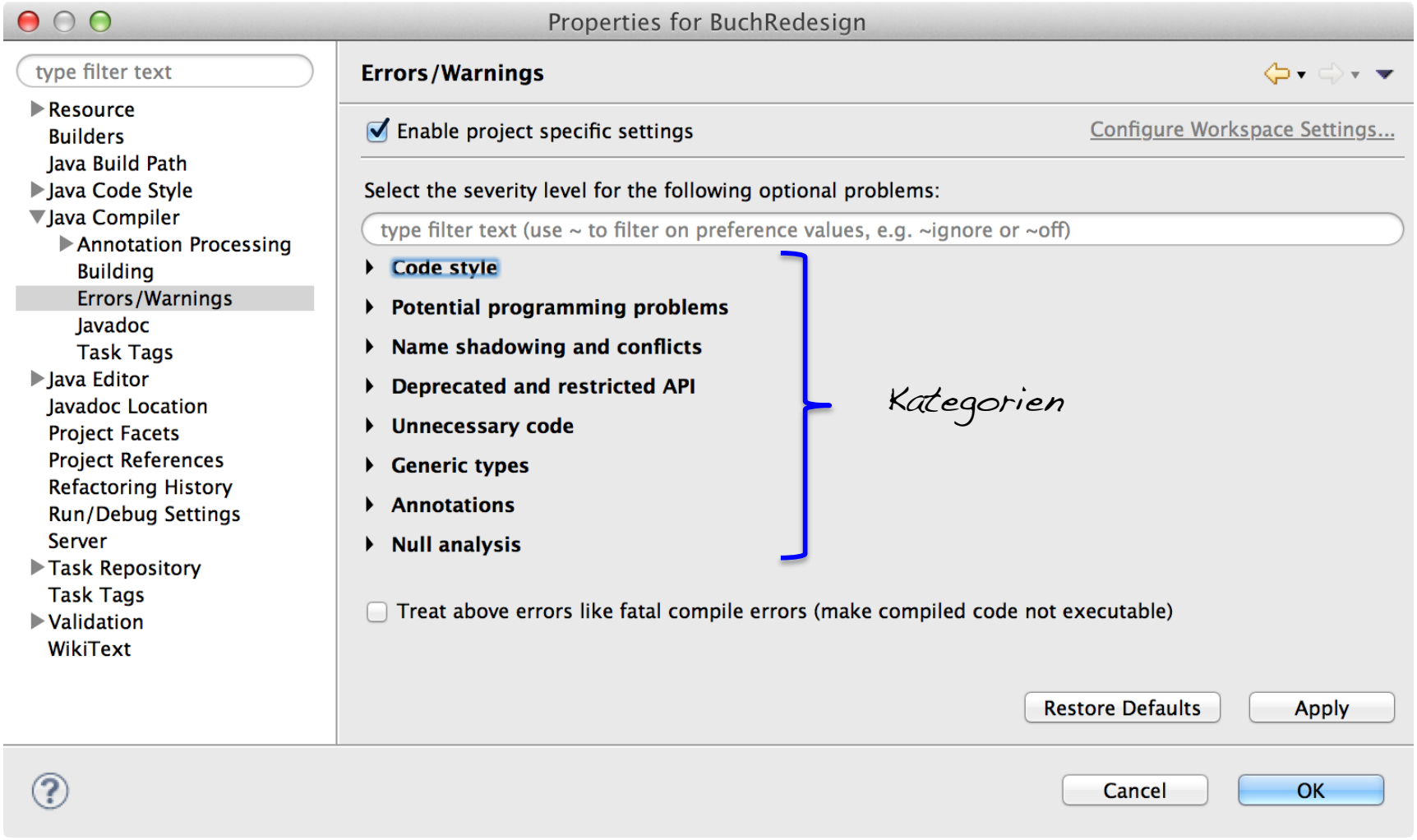
– Aufgabe: Führen Sie für die folgende Implementierung der Funktion `minmax` eine Datenflussanomalieanalyse durch! Die Funktion `minmax` erhält zwei Zahlen (*by reference*) übergeben und sortiert die Zahlen in eine aufsteigende Reihenfolge.

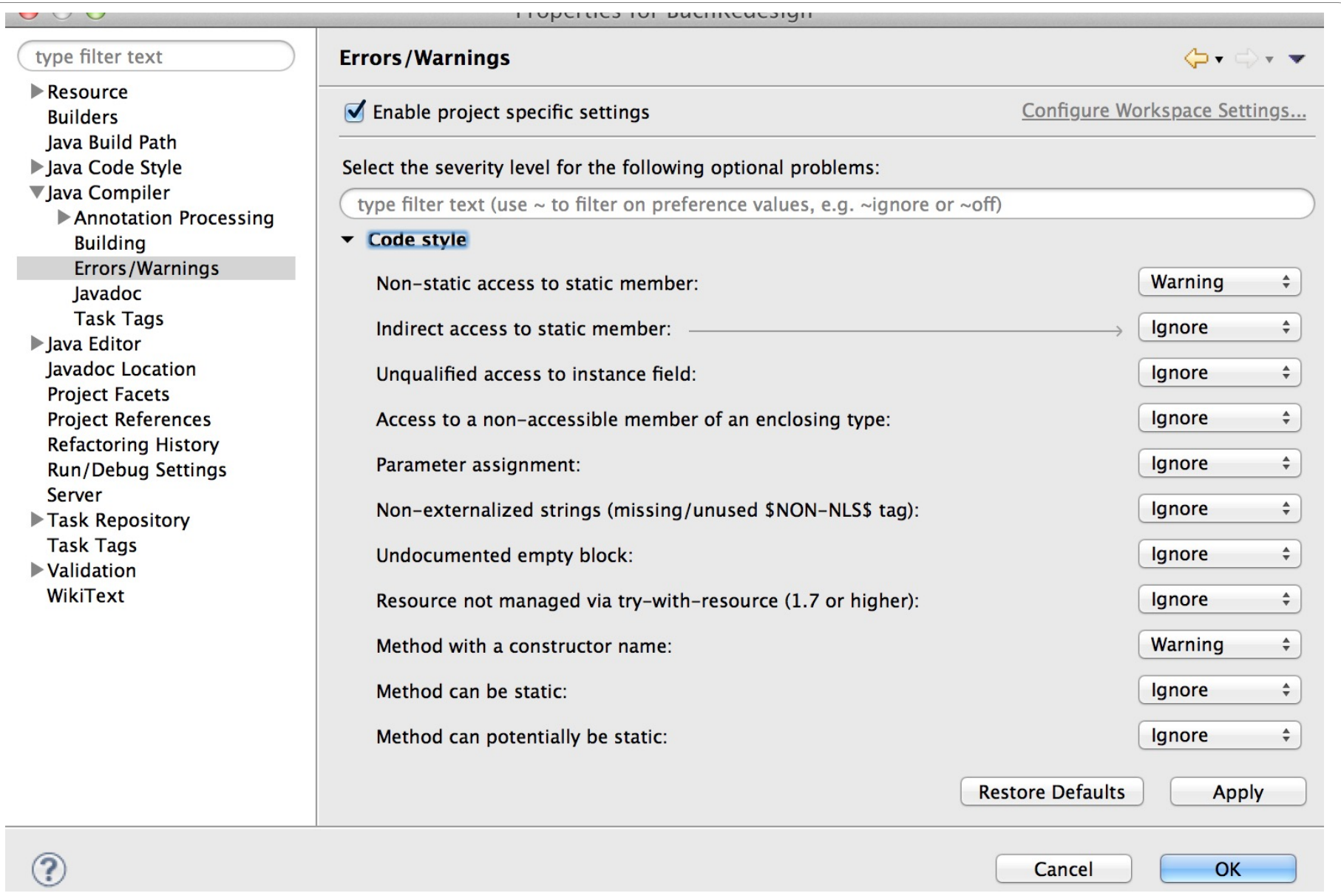
Aufgabe nach P. Liggesmeyer. *Software Qualität*. Spektrum Verlag, 2002

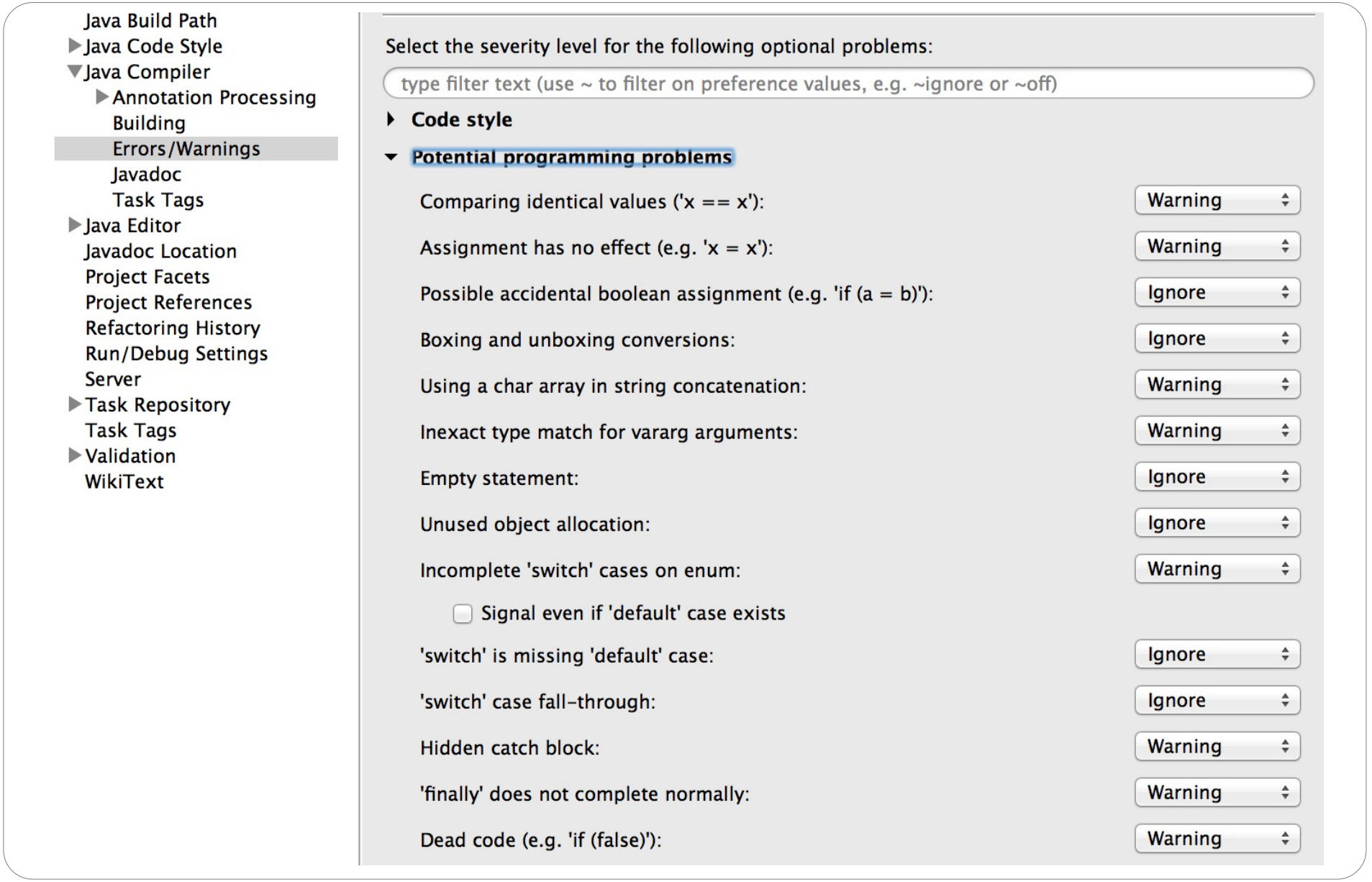
```
void minmax (int& min, int& max){
   int temp;

   if (min > max){
      max = temp;
      max = min;
      temp = min;
   }
}
```

Programmiersprache C++

– Beispiel: Statische Prüfungen des Java Compiler (Konfiguration über Eclipse)




type filter text
Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Annotation Processing
Building
Errors/Warnings
Javadoc
Task Tags
Java Editor
Javadoc Location
Project Facets
Project References
Refactoring History
Run/Debug Settings
Server
Task Repository
Task Tags
Validation
WikiText
Errors/Warnings
Enable project specific settings
Configure Workspace Settings...
Select the severity level for the following optional problems:
type filter text (use ~ to filter on preference values, e.g. ~ignore or ~off)
Code style
Non-static access to static member: Warning
Indirect access to static member: Ignore
Unqualified access to instance field: Ignore
Access to a non-accessible member of an enclosing type: Ignore
Parameter assignment: Ignore
Non-externalized strings (missing/unused $NON-NLS$ tag): Ignore
Undocumented empty block: Ignore
Resource not managed via try-with-resource (1.7 or higher): Ignore
Method with a constructor name: Warning
Method can be static: Ignore
Method can potentially be static: Ignore
Restore Defaults
Apply
Cancel
OK

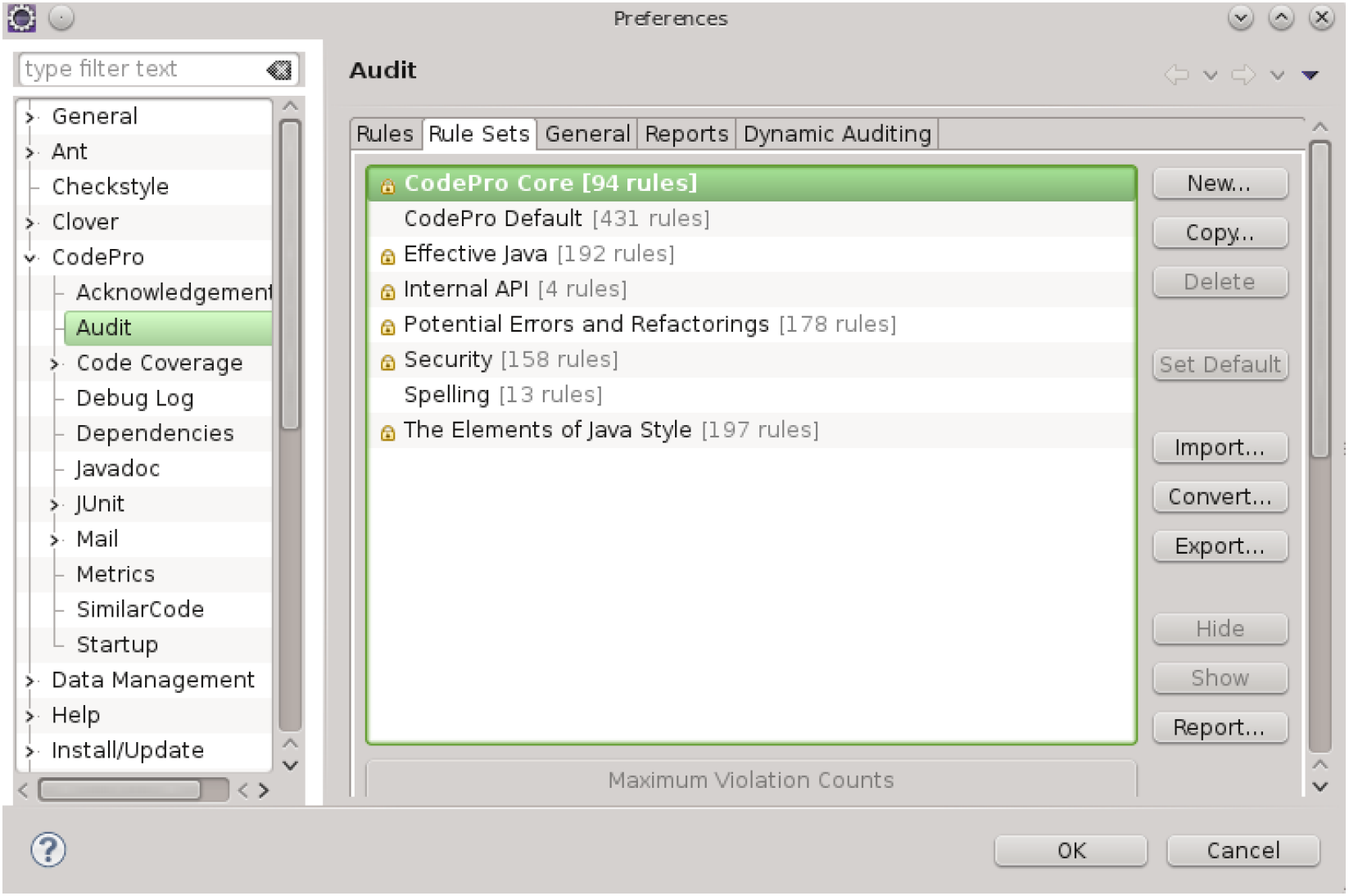

Java Build Path
Java Code Style
Java Compiler
Annotation Processing
Building
Errors/Warnings
Javadoc
Task Tags
Java Editor
Javadoc Location
Project Facets
Project References
Refactoring History
Run/Debug Settings
Server
Task Repository
Task Tags
Validation
WikiText
Select the severity level for the following optional problems:
type filter text (use ~ to filter on preference values, e.g. ~ignore or ~off)
Code style
Potential programming problems
Comparing identical values ('x == x'): Warning
Assignment has no effect (e.g. 'x = x'): Warning
Possible accidental boolean assignment (e.g. 'if (a = b)'): Ignore
Boxing and unboxing conversions: Ignore
Using a char array in string concatenation: Warning
Inexact type match for vararg arguments: Warning
Empty statement: Ignore
Unused object allocation: Ignore
Incomplete 'switch' cases on enum: Warning
Signal even if 'default' case exists
'switch' is missing 'default' case: Ignore
'switch' case fall-through: Ignore
Hidden catch block: Warning
'finally' does not complete normally: Warning
Dead code (e.g. 'if (false)'): Warning

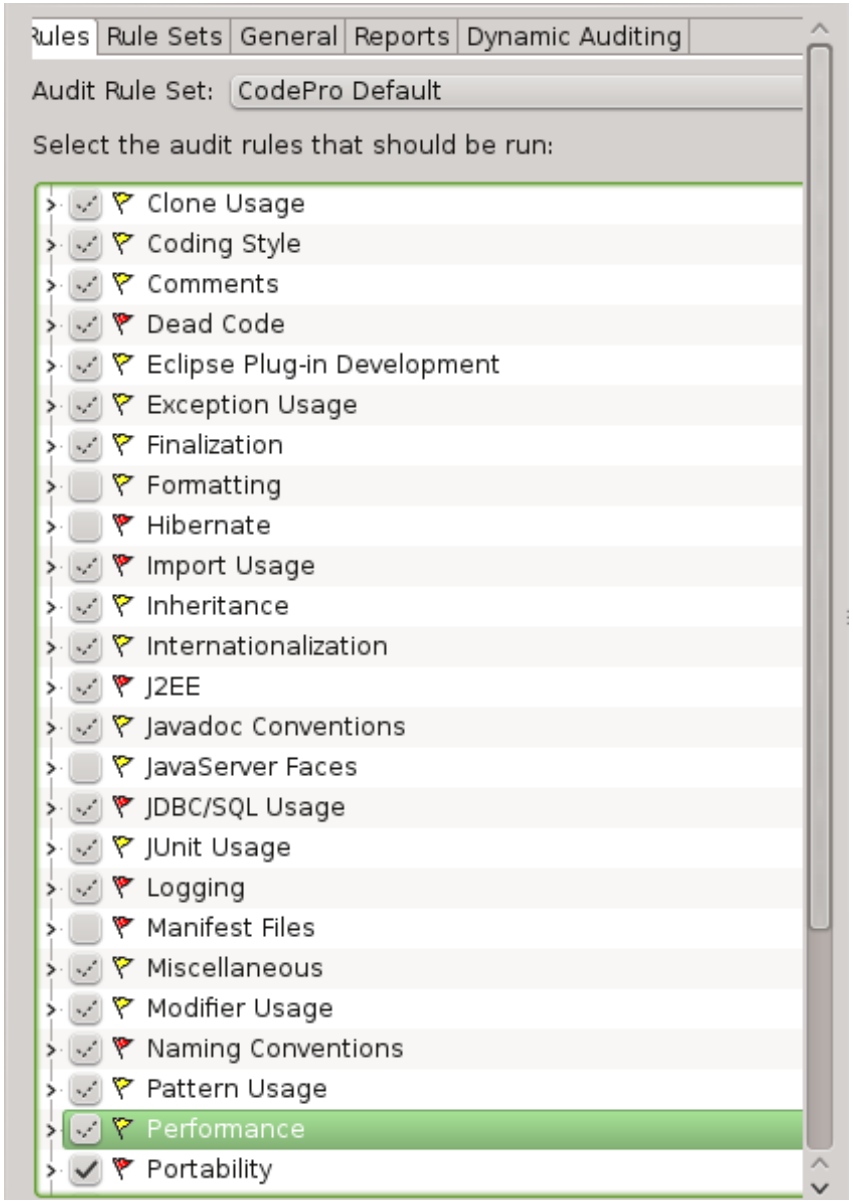
- Statische Analyse mit speziellen Werkzeugen (Sourcecode-Checker)
  - PMD
  - Checkstyle
  - CodePro Audit/Metrik
  - ...
- Die statischen Sourcecode-Checker führen Prüfungen nach (umfangreichen) Regelsätzen durch
  - Es können eigene Regelsätze konfiguriert werden
  - Regelverletzungen können als Warnungen oder Compilefehler klassifiziert werden
  - Es wird eine Auflistung der Regelverletzungen erstellt

Automatisierte Prüfung auf Einhaltung von Codierungsrichtlinien

- Folgende Prüfungen können z.B. vorgenommen werden:
  - Duplizierter Quellcode
  - nicht erreichbarer Code
  - Grad der Kommentierung
  - lange Klassen und Methoden
  - hohe Komplexität (zyklomatische Komplexität, Schwierigkeit)
  - unbenutzte Elemente
  - Literale in Abfragen/Berechnungen
  - fehlendes `final` für Methodenparameter und Attribute
  - Leere catch-Blöcke
  - Fehlendes Logging für gefangene Ausnahmen
  - ungenutzte lokale Variablen
  - Sichtbarkeit von Attributen
  - fehlender *load factor* bei Erzeugung einer `HashMap`
  - Klassennamen beginnen mit Kleinbuchstaben
  - ...

– Beispiel: CodePro Audit Regelsätze




Rules
Rule Sets
General
Reports
Dynamic Auditing
Audit Rule Set:
CodePro Default
Select the audit rules that should be run:
Clone Usage
Coding Style
Comments
Dead Code
Eclipse Plug-in Development
Exception Usage
Finalization
Formatting
Hibernate
Import Usage
Inheritance
Internationalization
J2EE
Javadoc Conventions
JavaServer Faces
JDBC/SQL Usage
JUnit Usage
Logging
Manifest Files
Miscellaneous
Modifier Usage
Naming Conventions
Pattern Usage
Performance
Portability

– Beispiel: Berechnung von Metriken mit CodePro

Metric	Value
Abstractness	6.6%
Average Block Depth	0.96
Average Cyclomatic Complexity	1.38
Average Lines Of Code Per Method	7.61
Average Number of Constructors Per Type	0.43
Average Number of Fields Per Type	0.90
Average Number of Methods Per Type	2.23
Average Number of Parameters	0.46
Comments Ratio	1.9%
Efferent Couplings	13
Lines of Code	711
Number of Characters	19,891
Number of Comments	14
Number of Constructors	13
Number of Fields	33
Number of Lines	892
Number of Methods	67
Number of Packages	11
Number of Semicolons	404
Number of Types	30
Weighted Methods	111