Autotest

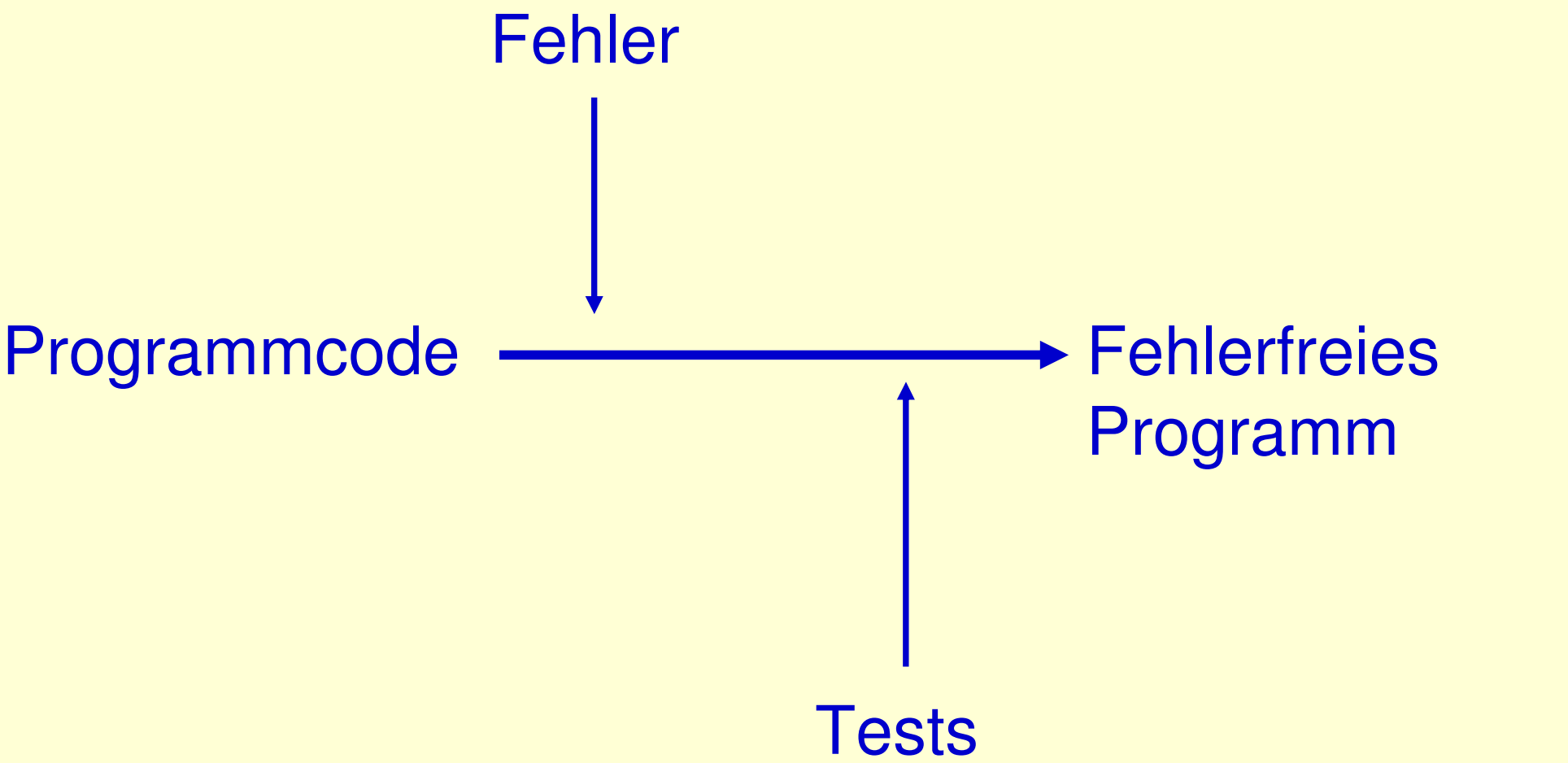
Automatische Testgenerierung mit
Design by Contract

von Simon Greiner
am 12. Juli 2007

Autotest

1. Testen
2. Design by Contract
3. Strategien zur Testerstellung
    1. Zufallsstrategie
    2. Planungsstrategie
4. Zusammenfassung

1. Testen – Warum testen?



1. Testen – Warum testen

- Tests können Fehler finden
- Speicherbar für Regressionstests
- Schnelles Finden von Flüchtigkeitsfehlern

1. Testen - Funktionsweise

- Besteht aus Eingabedaten und erwarteten Ergebnissen

1. Erstellen von Eingabedaten
2. Ausführen der Funktion / Methode
3. Überprüfen der Ausgabedaten und Seiteneffekte auf Korrektheit

1. Testen - Probleme

- Fehlerfreiheit nicht garantiert

- Erstellen von Tests sehr aufwendig
- Oft ungeliebte Pflicht von Programmierern
- Qualität der Tests sehr wichtig

1. Testen – Warum automatisieren

- Testen wird einfacher und schneller
  - ➢Theoretisch unbegrenzt viele Testfälle
- Abgeben einer „lästigen“ Pflicht
- Push-Button Integration in IDEs oder Stand-alone Tests
  - ➢Größere Akzeptanz bei Programmierern
  - ➢Oft umfangreiches Testen
  - ➢(garantierte Testabdeckung)

2. Design by Contract – Das Prinzip

- Methode zur Softwareerstellung
- Vertrag zwischen aufgerufenem und aufrufendem Programmteil
  - Aufrufer garantiert gültige Eingaben (preconditions)
  - Aufrufer garantiert gültigen Zustand des Aufgerufenen (vor Aufruf)
  - Aufgerufener garantiert korrekte Ausgaben und korrekte Seiteneffekte (postconditions)
  - Aufgerufener garantiert gültige Zustände (invariants) nach Ablauf des Aufrufes
- Eigentlich nur in Eiffel komplett implementiert, aber auch in anderen Sprachen realisierbar

2. DbC - Vorteile

- Definition von Schnittstellen beim Design
- Hilfe bei Dokumentation von Code
- Einfacheres Auffinden von Fehlern
- Automatische Testfallerstellung

2. DbC - Elemente

- Precondition
  - Vor Ausführung einer Funktion/Methode
- Postcondition
  - Nach Ausführung einer Funktion/Methode
- Invarianten
  - Vor und nach Ausführung einer Funktion/Methode
- Zusicherung
  - Zu einem bestimmten Zeitpunkt

2. DbC – Eiffel (1)

```
class BANK_ACCOUNT
create
        make
feature {NONE} -- Initialization

        make (a_owner: PERSON) is
                require
                        a_owner_not_void: a_owner /= Void
                do
                        /* implementation */
                ensure
                        owner_set: owner = a_owner
                end
feature -- Access
        owner: PERSON
        currency: CURRENCY
        balance: INTEGER
/* further features */
invariant
        owner_not_void: owner /= Void
        currency_not_void: currency /= Void
        balance >= 0
end
```

2. DbC – Eiffel(2)

```
make (a_owner: PERSON) is
        require
                a_owner_not_void: a_owner /= Void
        do
                /* implementation */
        ensure
                owner_set: owner = a_owner
        end
…
invariant
    owner_not_void: owner /= Void
    currency_not_void: currency /= Void
    balance >= 0
```

2. DbC - Java

```
public class Container
{   int count;
    int capacity;

    classinvariant:
    count>=0 and count<=capacity;

    public int insert(Object a)
    {     precondition: a!=null;
          /* actual implementation */
          return count;
          postcondition: this.contains(a) == true;
          postcondition: old count + 1 == Result;
    }
}
```

2. Testen mit DbC

- Erkennung von Zusicherungsverletzungen
- Erkennung des „Schuldigen“

Bug	Ungültige Eingabedaten
- Postcondition verletzt - Precondition nach erstem Aufruf verletzt - Klasseninvariante verletzt - Zusicherung verletzt - Prozedur wird nicht beendet	-Precondition beim Funktionsaufruf verletzt

3. Testerstellung - Strategien

- Finden möglichst vieler Fehler
- Testen möglichst vieler Klassen / Funktionen /Prozeduren

3.1 Zufallsstrategie - Ablauf

- A.foo(o1, o2) soll getestet werden
  - Erstellen je einer Instanz von A, o1 und o2
  - Verändern einer dieser Instanzen (zufälliger Funktionsaufruf)
  - Ausführen von A.foo(o1, o2) inklusive Dokumentation von aufgerufenen Funktionen zur Diversifikation und eventueller Fehler
  - Testen der nächsten Funktion/Methode

3.1 Zufallsstrategie - Beispieltool

3.1 Zufallsstrategie - Erfolge

- Beispiel: LINKED_LIST aus EiffelBase

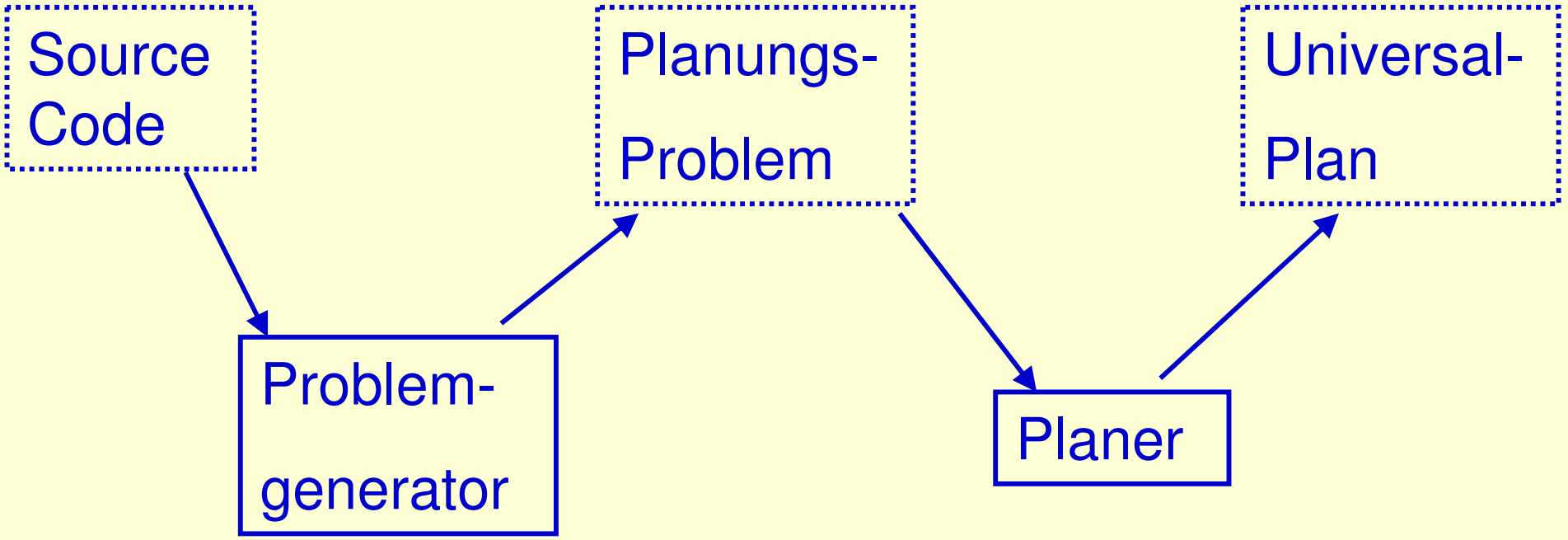
	Durchge-führt	Bestan-den	Ungültige Antwort	Durchge-fallen	Kein gültiger Aufruf
Funktionen / Methoden	195	150	3	7	35

- ➢ Fehler gefunden
- ➢ Strikte Funktionen können nicht getestet werden

3.2 Planungsstratgie - Prinzip

- Ziel: Erreichen von benötigten Zuständen
- Strukturiertes Vorgehen anhand von Plänen
- Auch strikte preconditions erfüllen

3.2 Planungsstrategie – Der Planer



3.2 Planungsstrategie – Elemente des Planungsproblems

- Variablen (Klassenveriablen): Die Variablen der Planungsstrategie
- Aktionen (Methoden): Zustandsübergänge (Veränderungen der Variablen)
- Anfangszustand (Objekt nicht initialisiert)
- Zielzustand (zu testende Precondition): Zu erreichender Zustand einer Klasse

3.2 Planungsstrategie – Beispiel (1)

```
class SIMPLE_LIST

f e a t u r e –– Que r i e s
index : INTEGER
–– Index of item at current cursor
    position

count : INTEGER
–– Number of items in this list

islast : BOOLEAN
–– Is internal cursor on the last item?

off : BOOLEAN
–– Is internal cursor not at avalid
    position?
```

```
feature –– Commands
force
  –– Add an element .
  ensure
  count = old count + 1

finish
  –– Move internal cursor to last item.
  ensure
  count > 0 implies islast

invariant
  count >= 0
  index >= 0
  index <= count + 1
  islast = ( ( count > 0) and ( index = count ) )
  off = ( ( index = 0) or ( index = count + 1) )

  end
```

3.2 Planungsstrategie – Beispiel (2)

- Anfangszustand: Leere Liste

  count = 0 ^ index = 0 ^ ¬is last ^ off

- Zielzustand: Cursor steht auf einer gültigen Position (Precondition für Abfrage des Objektes an der Cursorposition)

  ¬off

3.2 Planungsstrategie – Beispiel (2)

- Effekt force

  (count' = count + 1) ^ (count' + 1 > 0) ^
  (index' + 1 > 0) ^ (index' < count' + 2) ^
  (islast' <=> ((count' > 0) ^ (index' = count'))) ^
  (off' <=> ((index' = 0) v (index' = count' + 1)))

- Effekt finish

  (count' > 0) => (islast' = 1) ^ (count' + 1 > 0) ^
  (index' + 1 > 0) ^ (index' < count' + 2) ^
  (islast' <=> ((count' > 0) ^ (index' = count'))) ^
  (off' <=> ((index' = 0) v (index' = count' + 1)))

3.2 Planungsstrategie – Beispiel (3)

Vorgeschlagene Lösung für bestimmte Zustände

count	index	islast	off	
00	*0	0	1	force
01	**	*	1	finish
1*	**	*	1	finish

Ausgangszustand: count = 0 ^ index = 0 ^ ¬islast ^ off

1. Schritt: force -> count = 1 ^ index = 0 ^ ¬ islast ^ off

2. Schritt: finish -> count = 1 ^ index = i ^ islast ^ ¬off

Der Zielzustand ¬off ist erreicht

3.2 Planungsstrategie – Unsicherheit

- Theoretisch: Variablenänderung in Postcondition nicht angegeben -> Variable ändert sich nicht
- Praktisch: Variablen können sich trotzdem ändern

➢ Unsicherheit, ob Variable sich ändert

➢ Planer muss annehmen dass sich jede Variable ändern kann (zum Vorteil des Planes)

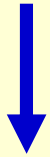
3.2 Planungsstrategie – Ausgabe des Planers

- Schwache Lösung: Der Plan erfüllt das Ziel vielleicht
- Starke Lösung: Der Plan erfüllt das Ziel sicher

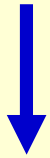
➢ Praxis: Fast nur schwache Lösungen

3.2 Planungsstrategie – Testausführung

Verändern des Objektes nach Vorgabe des Planers



Ausführen des Tests



Vorbedingung nicht erfüllt

3.2 Planungsstrategie – Planen mit Lernen

- Plan liefert ein Ergebnis, das die Vorbedingung nicht erfüllt
- Zustandsübergänge vermeiden, die den Plan nicht „weiterbringen“

3.2 Planungsstrategie – Lernalgorithmus

1. Planungsproblem erstellen
2. Planer liefert schwache Lösung
3. Lösung ausführen und Zustandsübergänge (s -> s‘) speichern
4. Wenn Zielzustand nicht erreicht wird:

   Verändern der Postcondition:

   post := post ^ (s -> s‘)

   Weiter bei 2.

3.2 Planungsstrategie – Resultate

- In vereinzelten Versuchen waren vorher untestbare Methoden jetzt testbar
- Noch nicht implementierbar -> nicht testbar
  - Verwendeter Planer unterstützt keine Methoden mit Argumenten

4. Zusammenfassung

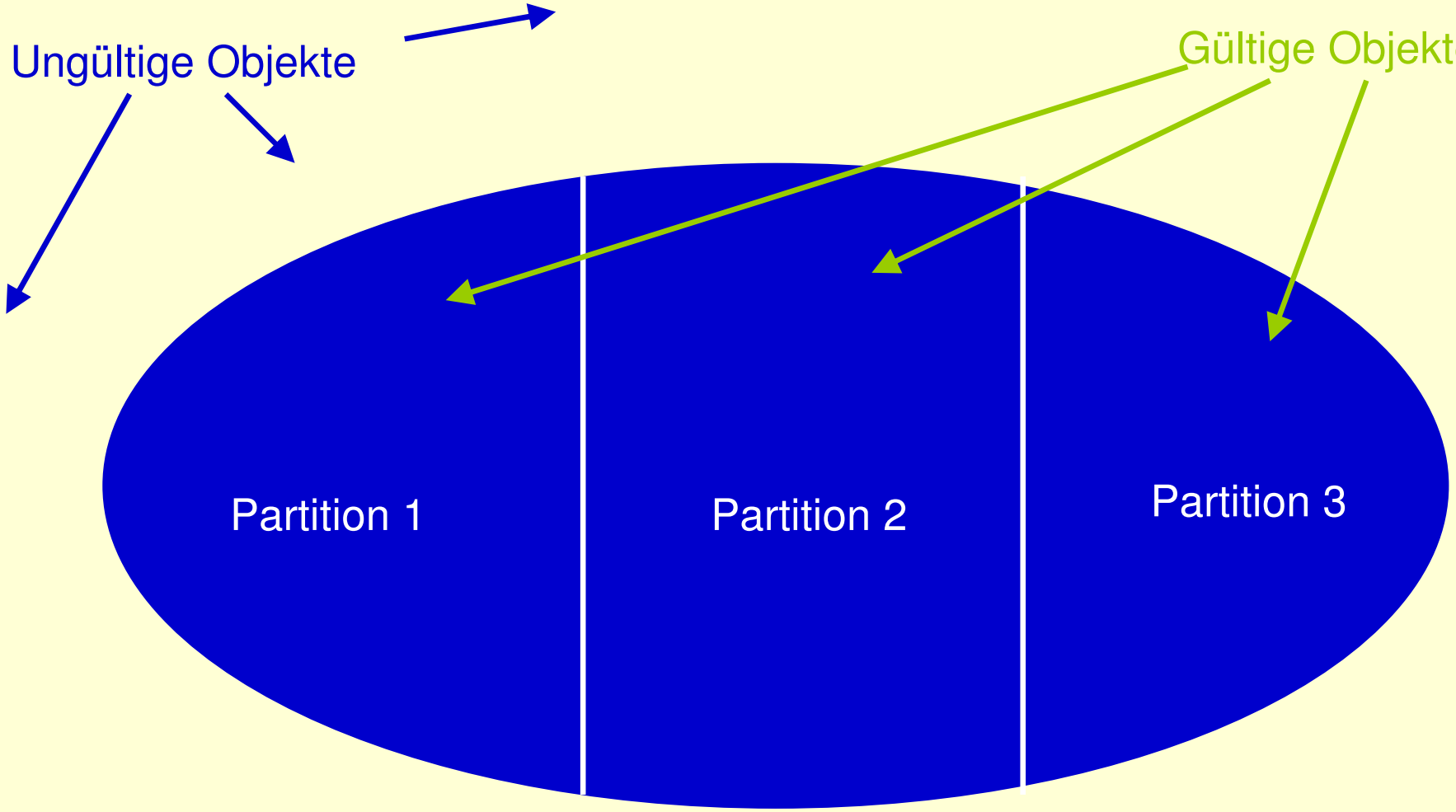
- Automatisches Testen ist möglich
- Methoden funktionieren

- Planungsstrategie muss implementiert/verbessert werden
- Bisher reine Blackbox Tests

5. Äquivalenzpartitionen – Motivation

- Bisher: reines Blackbox testen
- Idee: Hinzufügen von Programmiererwissen
- Vorteil: Bessere Testabdeckung

5. Äquivalenzpartitionen – Partitionen



5. Äquivalenzpartitionen – Erweiterungen (1)

```
public class Container
{
    int count;
    int capacity;
    classinvariant:
    count>=0 and count<=capacity;
    public int insert(Object a)
    {
        precondition: a!=null;
        /* actual implementation */
        return count;
        postcondition: this.contains(a) == true;
        postcondition: old count + 1 == Result;
    }
}
```

Normale Notation

5. Äquivalenzpartitionen – Erweiterungen (2)

```
public class Container
{
  int count; int capacity;

  partitioninvariant empty
    count==0;
  partitioninvariant oneElement
    count==1;
  partitioninvariant severalElements
    count>1,
  classinvariant:
    count>=0 and count<=capacity,
    in[empty oneElement
       severalElements];
```

```
public int insert(Object a)
  {
    precondition: a!=null;

    /* actual implementation */
    return count;

    postcondition:
      this.contains(a) == true;
    postcondition:
      old count + 1 == Result;
    postcondition:
      notin[empty];
  }
}
```

5. Äquivalenzpartitionen – Überlappungen

- Partitionen können sich überlappen
  - Empty
  - oneElement
  - severalElements
  - sameType (Alles Objekte haben den gleichen Typen)
  - differentTypes (Objeke sind von verschiedenen Typen)
- Weitere Schlüsselwörter
  - onlyin[p_1, … , p_n]: Klasse in genau einer der Partitionen p_1 bis p_n

5. Äquivalenzpartitionen – Test

- Jede Klasse in jeder Partition mindestens ein mal testen
- Erzielt eine gewisse Testabdeckung
- Qualität stark abhängig von Programmierer