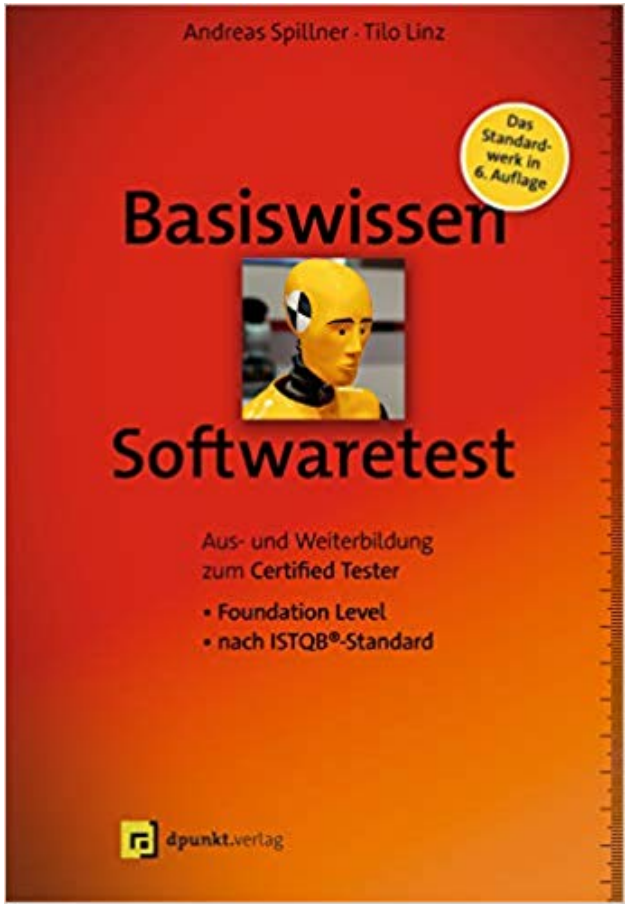
Programmiertechniken 2

Prof. Dr.-Ing. Zhen Ru Dai
zhenru.dai@haw-hamburg.de

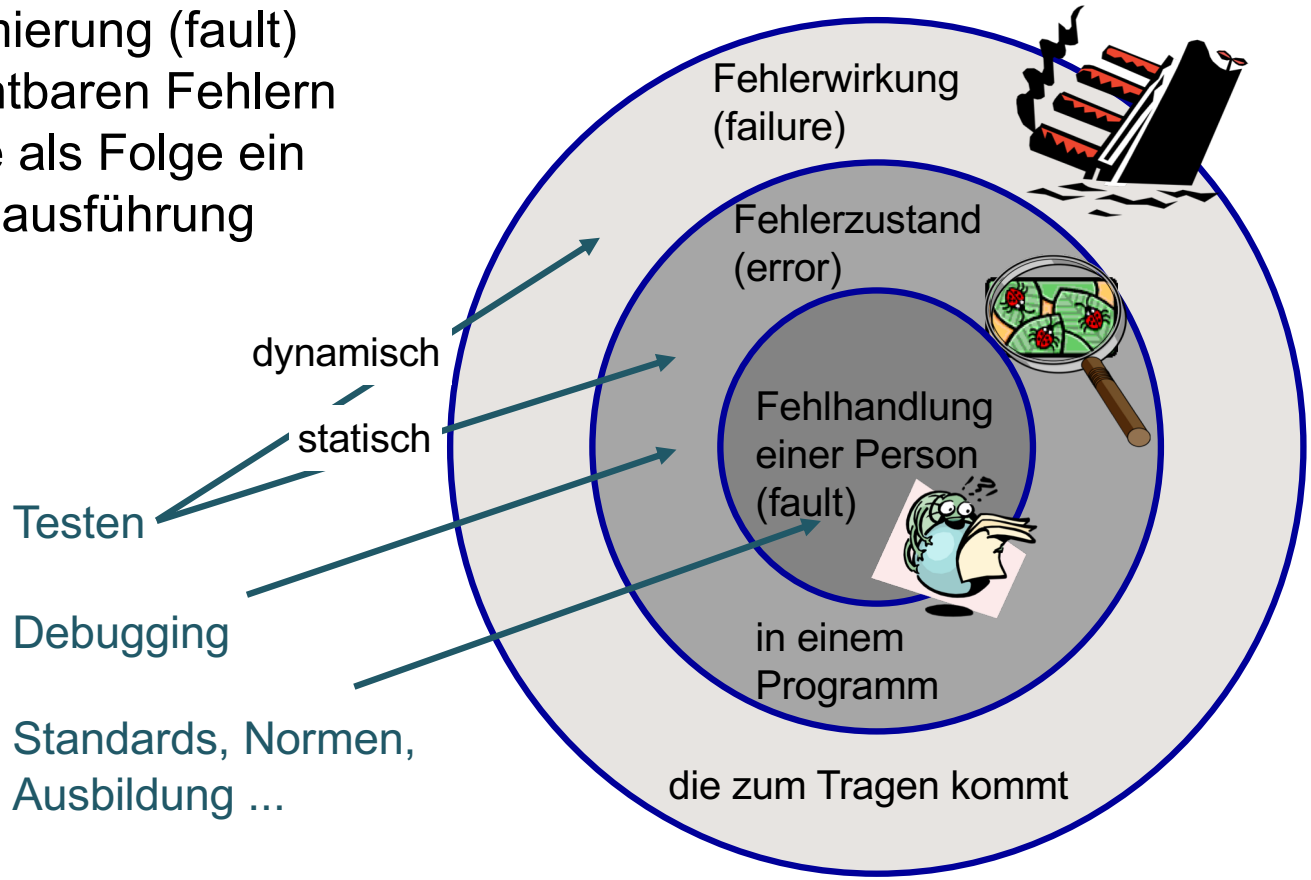
Flipped Classroom

- Spillner, Linz:
  Kapitel 5.1.3. Zustandsbasierter Test



Definition: Failure, Fault und Error

Fehler bei der Programmierung (fault) führen ursächlich zu sichtbaren Fehlern im Programm (error), die als Folge ein Versagen der Programmausführung (failure) bewirken.



Debugging ist eine Entwicklungstätigkeit, bei der die Ursachen einer Fehlerwirkung identifiziert, analysiert und entfernt werden.

[Quelle] Certified Tester Foundational Level

V 2.0 / 2011

Definition: Fehler und Mangel

- Fehler ist die Nichterfüllung einer festgelegten Anforderung
  - Abweichung zwischen Ist- und Soll-Verhalten
  - Voraussetzung: Soll-Verhalten muss vorab beschrieben sein (z.B. in einer Spezifikation)
- Mangel liegt vor, wenn eine Anforderung nicht angemessen erfüllt wird
  - z. B. wenn ein Programm zwar funktioniert, aber der Dialog oder Speichern zu lange dauert, oder das GUI unvollständig (falsche Farben, fehlende Grafiken) ist

[Quelle] Certified Tester Foundational Level

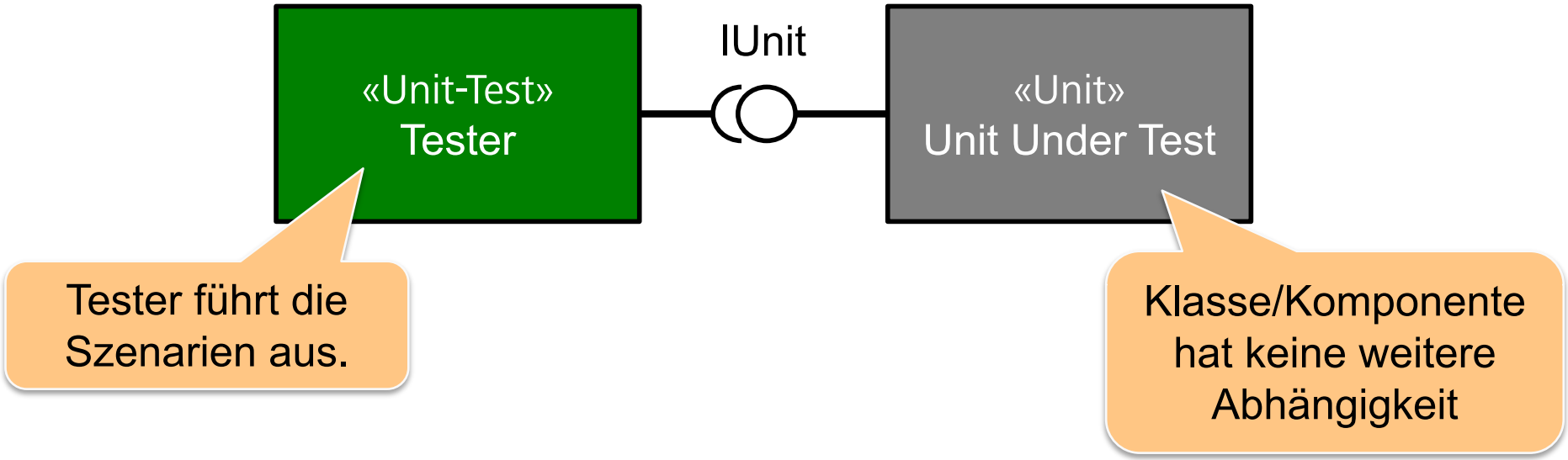
Testen, Verifikation, Validierung

Testen: (Automatisierter) Vergleich der Ist-Situation zur geforderten Soll-Situation. Soll-Situation ist durch Anforderungen definiert. Ein Test prüft exemplarisch Situationen ab.

Verifikation: „Did we build the system right?“
(Haben wir das System richtig realisiert?)

Validation: „Did we build the right system?“
(Haben wir das richtige System realisiert?)

Unit-Testing



Die über das Interface sichtbare Reaktion (Ist-Zustand) wird mit der erwarteten Reaktion (Soll-Zustand) automatisch Verglichen (Assertions).

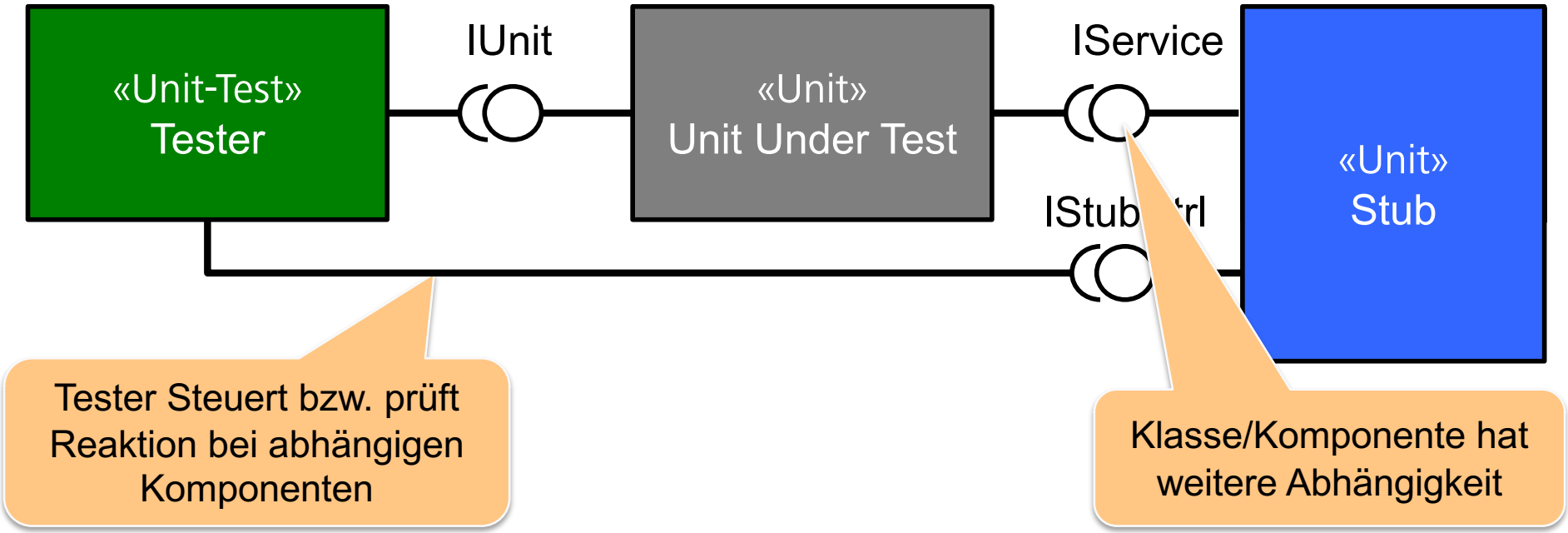
→ Ablauf über Programmcode

→ Erwartungen über Assertions.

```
assert( x==y );
assert( obj->getValue() > 42);
```

assertion.h

Unit-Testing mit Stubs



Benötigt die UUT weitere Komponenten, so werden diese direkt in den Test einbezogen oder werden durch einen **Stub** ersetzt. Der Stub simuliert die Reaktionen der abhängigen Komponente (meist unter Kontrolle des Testers).
→ Tester muss abhängige Komponente austauschen können

Definition Platzhalter/Stub [nach IEEE 610]
Eine rudimentäre oder spezielle Implementierung einer Softwarekomponente, die verwendet wird, um eine noch nicht implementierte Komponente zu ersetzen bzw. zu simulieren.

Zustandsbasiertes Testen

- Besonders geeignet zum Test objektorientierter Systeme
  - Objekte können unterschiedliche Zustände annehmen
  - Die jeweiligen Methoden zur Manipulation der Objekte müssen dann entsprechend auf die unterschiedlichen Zustände reagieren
  - Beim objektorientierten Testen hat der zustandsbasierte Test deshalb eine herausgehobene Bedeutung, da er diesen speziellen Aspekt der Objektorientierung berücksichtigt
- Ziele des zustandsbasierten Tests
  - Nachweis, dass sich das Testobjekt konform zum Zustandsdiagramm verhält (Zustands-Konformanztest)
  - Zusätzlich Test unter nicht-konformanten Benutzungen (Zustands-Robustheitstest)

Arbeitsschritte des zustandsbasierten Tests

1. Fokussierung auf das Zustandsdiagramm
2. Prüfung auf Vollständigkeit
3. Ableiten des Übergangsbaumes für den Zustands-Konformanztest
4. Erweitern des Übergangsbaumes für den Zustands-Robustheitstest
5. Generieren der Botschaftssequenzen und Ergänzen der Botschaftsparameter
6. Ausführen der Tests und Überdeckungsmessung

Beispiel zur Zustandsmodellierung: Stapel (Stack)

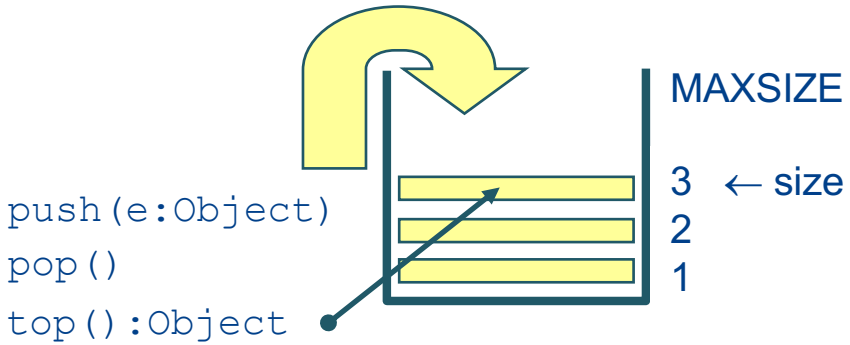
Klasse `Stapel`

Zustandserhaltende Operationen

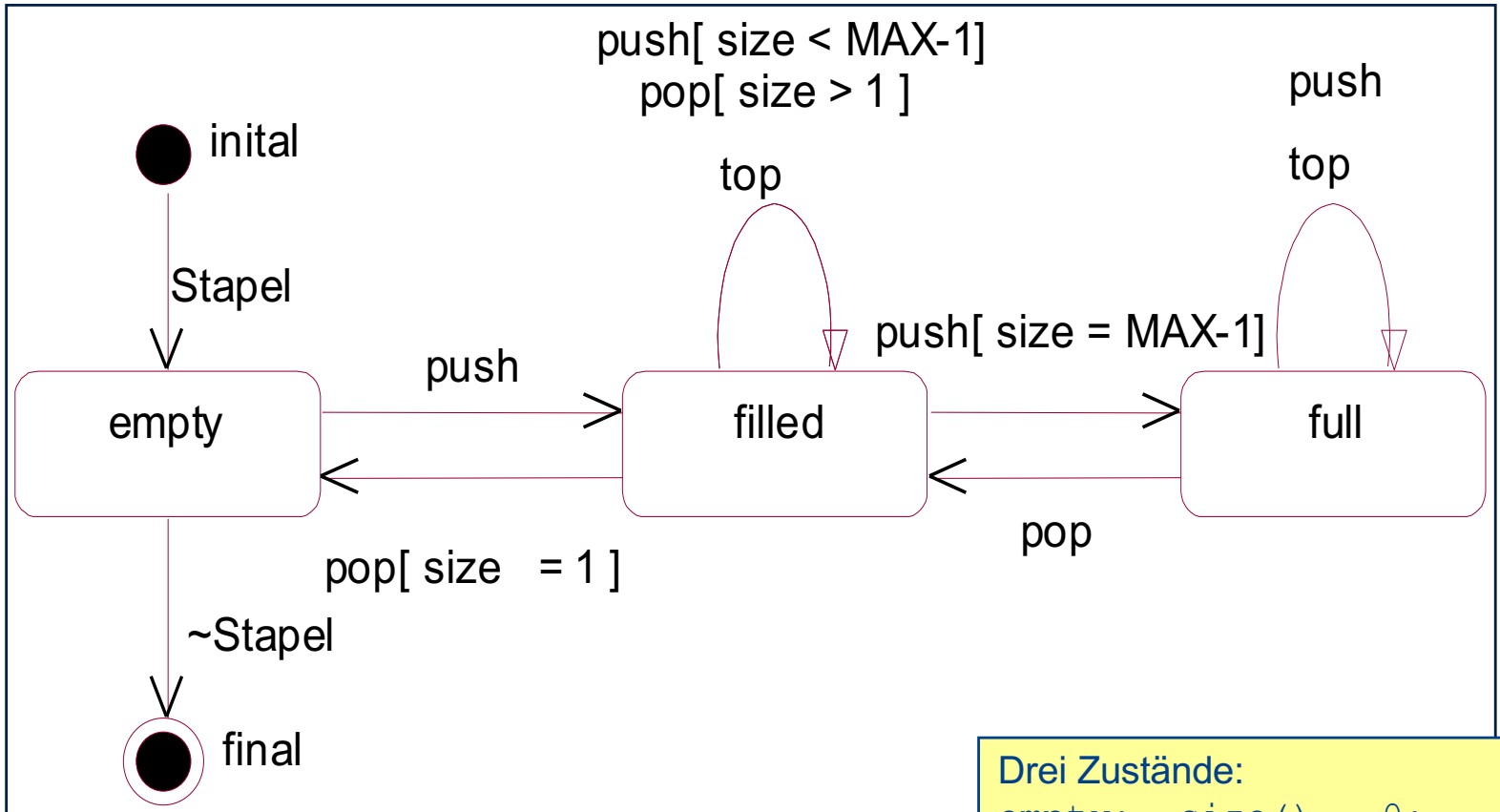
```
size():integer; // Anzahl gestapelter Elemente
MAX():integer;  // Maximale Anzahl
top():Object;   // Zeiger auf oberstes Element
```

Zustandsverändernde Operationen

```
Stapel(Max:integer);   // Konstruktor
~Stapel();             // Destruktor
push(element:Object);  // Stapelt Element
pop(); // Entfernt oberstes Element
```



Beispiel zur Zustandsmodellierung: Stapel (Stack)



Drei Zustände:

```
empty:  size() = 0;
filled: 0 < size() < MAX();
full:   size() = MAX();
```

1. Fokussierung auf das Zustandsdiagramm

Drei Zustände:

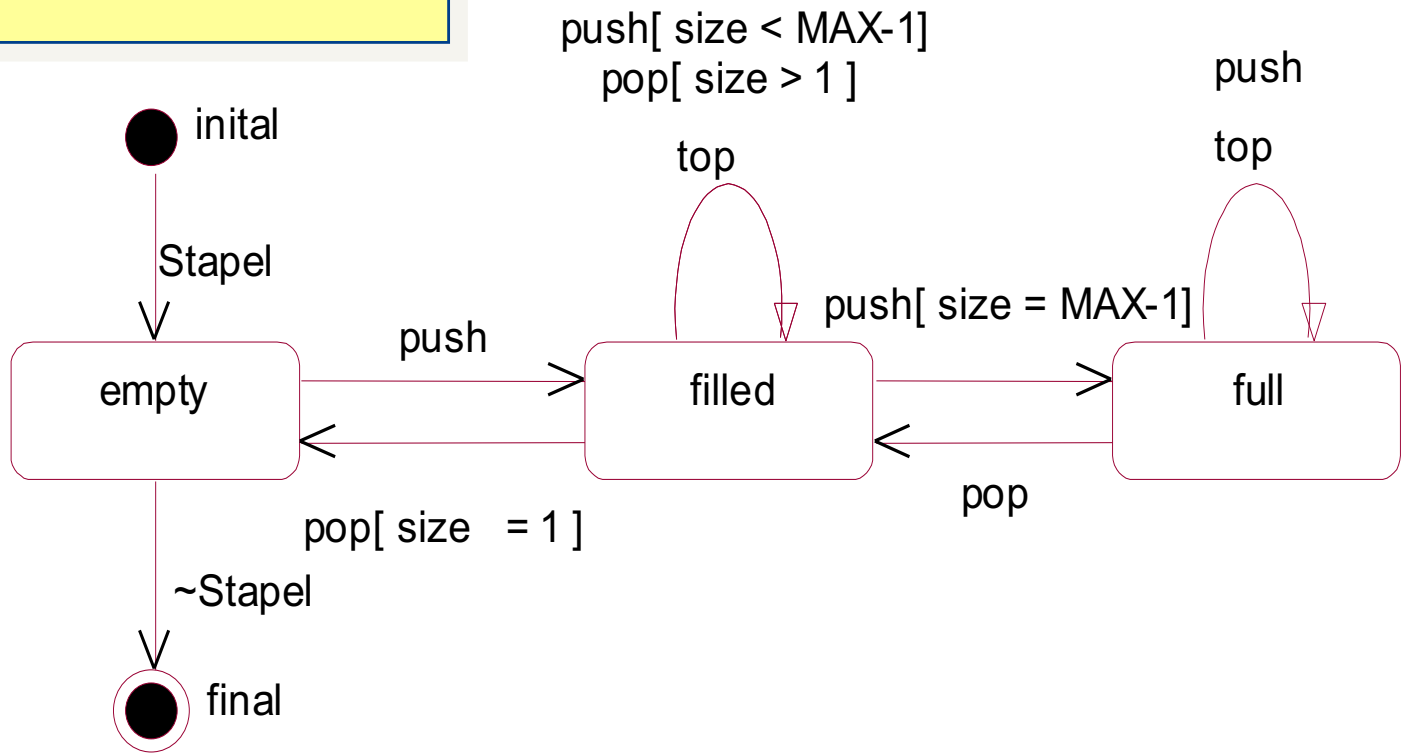
```
empty:   size() = 0;
filled:  0 < size() < MAX();
full:    size() = MAX();
```

Zwei »Pseudo-Zustände«:

```
initial: Vor Erzeugung;
final:   Nach Zerstörung
```

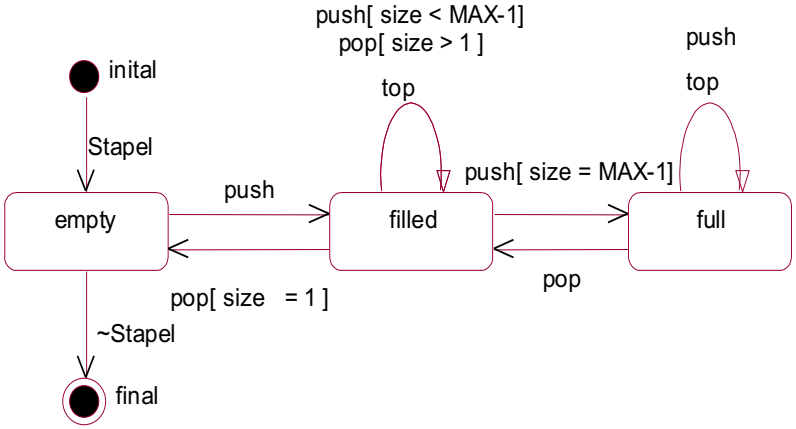
Acht Zustandsübergänge:

```
initial → empty; empty → final
empty → filled; filled → empty (Zyklus!)
filled → full; full → filled   (Zyklus!)
filled → filled; full → full   (Zyklen!)
```



2. Prüfung auf Vollständigkeit

- Zustandsdiagramm hinsichtlich der »Vollständigkeit« untersuchen
  - Ggf. Zustandsübergangstabelle anlegen
  - Ggf. auch die Wächterbedingungen bez. eines Ereignisses auf »Vollständigkeit« und Konsistenz prüfen
  - Nicht spezifizierte Zustands/Ereignis-Paare hinterfragen



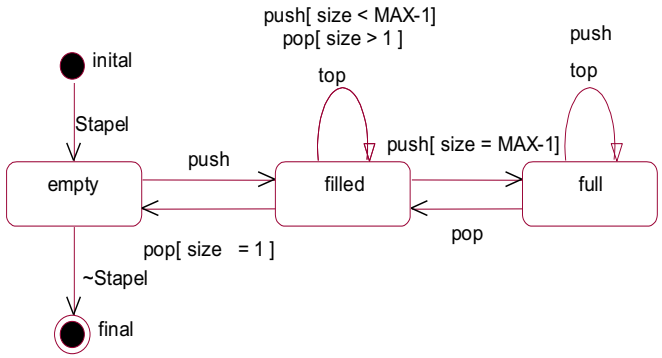
Zustand / Ereignis	initial	empty	filled	full
Stapel()	empty	N/A	N/A	N/A
~Stapel()	N/A	final	?	?
push()	N/A	filled	filled, full	full
pop()	N/A	?	empty, filled	filled
top()	N/A	?	filled	full

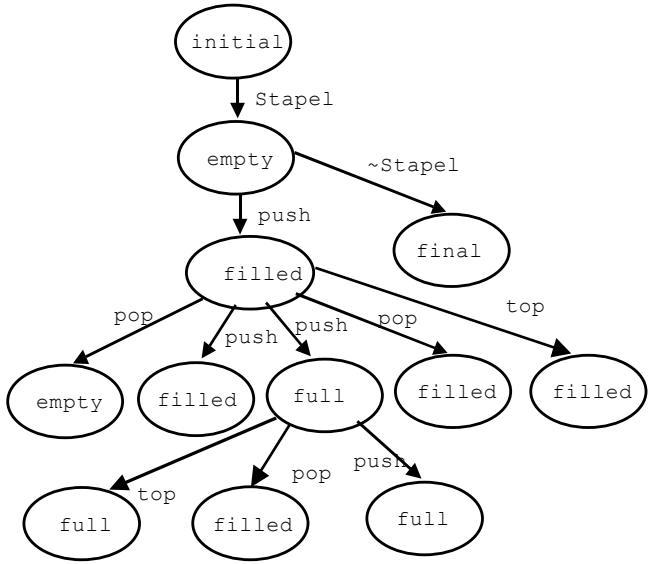
3. Aufbau des Übergangsbaumes: Zustands-Konformanztest

1. Der Anfangszustand wird die Wurzel des Baumes.
2. Für jeden möglichen Übergang vom Anfangszustand zu einem Folgezustand im Zustandsdiagramm erhält der Übergangsbaum von der Wurzel aus einen Zweig zu einem Knoten, der den Nachfolgezustand repräsentiert. Am Zweig wird das Ereignis (Operation) und ggf. die Wächterbedingung notiert.
3. Der letzte Schritt wird für jedes Blatt des Übergangsbaums so lange wiederholt, bis eine der beiden Endbedingungen eintritt:

   Der dem Blatt entsprechende Zustand ist auf einer »höheren Ebene« bereits einmal im Baum enthalten.

   Der dem Blatt entsprechende Zustand ist ein Endzustand und hat somit keine weiteren Übergänge, die zu berücksichtigen wären.

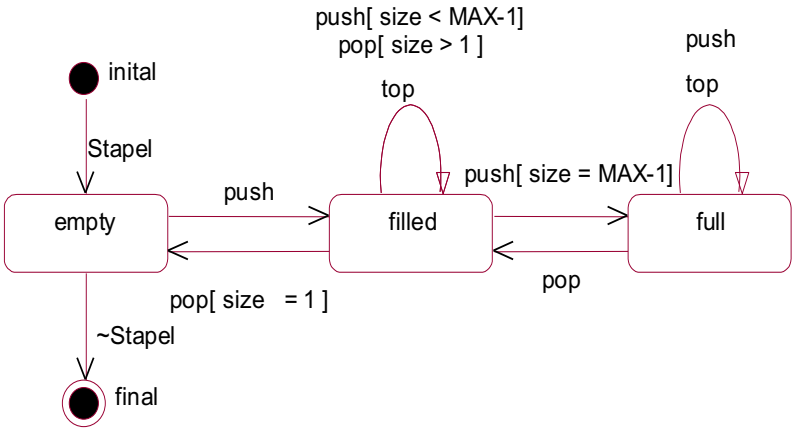


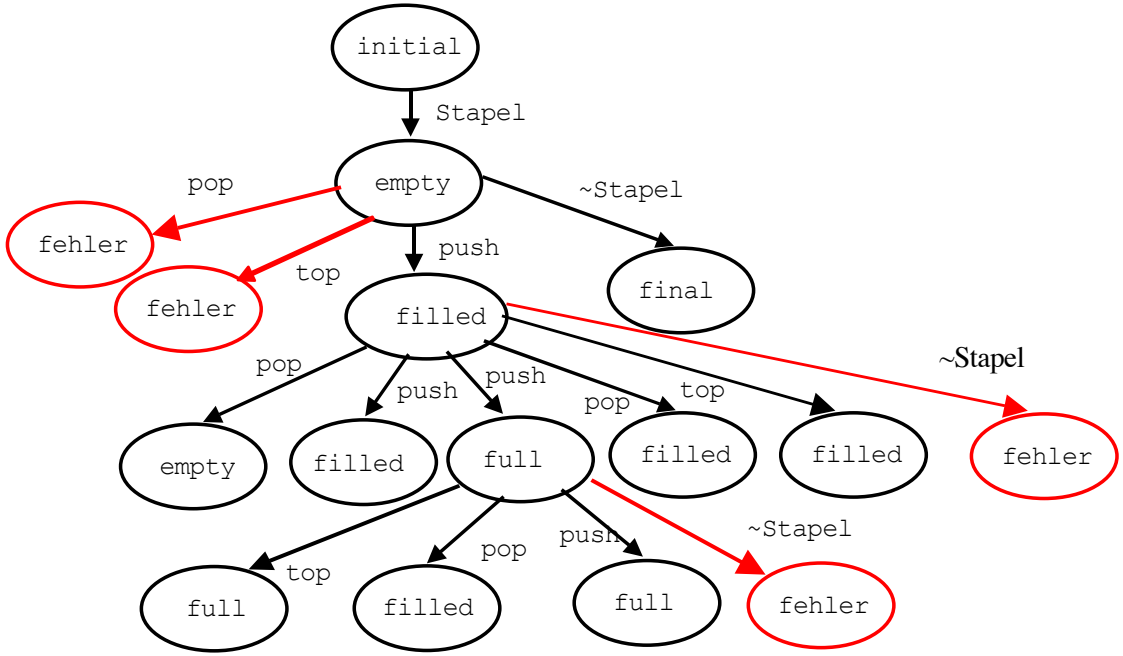


(Wächterbedingungen hier nicht dargestellt)

4. Erweitern des Übergangsbaumes: Zustands-Robustheitstest

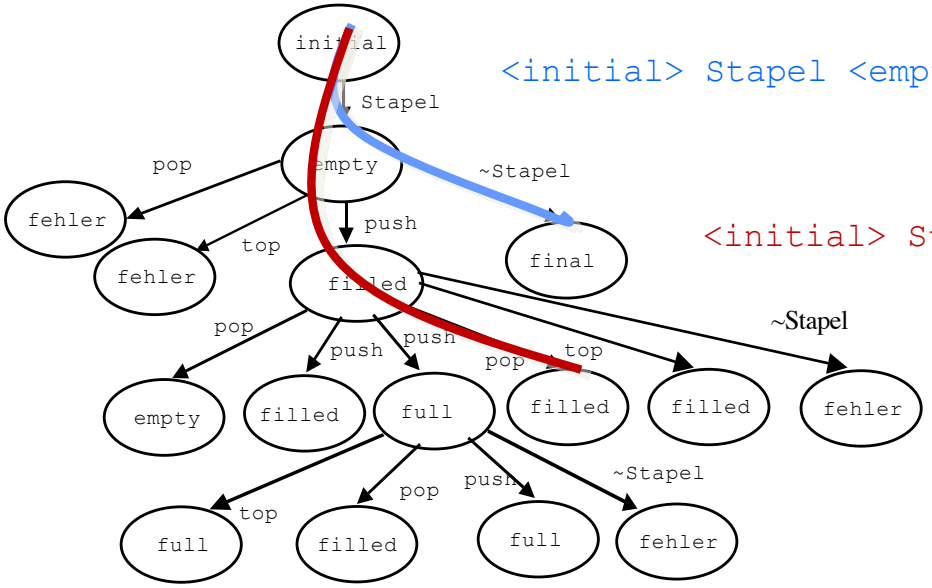
- Robustheit unter spezifikationsverletzenden Benutzungen prüfen
- Für Botschaften, für die aus dem betrachteten Knoten kein Übergang spezifiziert ist, den Übergangsbaum um einen neuen »Fehler«-Zustand erweitern





5. Generieren der Testfälle (1)

- Pfade von der Wurzel zu Blättern im erweiterten Übergangsbaum als Funktions-Sequenzen auffassen
- Stimulierung des Testobjekts mit den entsprechenden Funktionsaufrufen deckt alle Zustände und Zustandsübergänge im Zustandsdiagramm ab
- Ergänzen der Parameter!



<initial> Stapel <empty> ~Stapel <final>

<initial> Stapel <empty> push <filled> pop <filled>

5. Generieren der Testfälle (2)

- Pfade von der Wurzel zu Blättern im erweiterten Übergangsbaum als Funktions-Sequenzen auffassen
- Stimulierung des Testobjekts mit den entsprechenden Funktionsaufrufen deckt alle Zustände und Zustandsübergänge im Zustandsdiagramm ab
- Parameter ergänzen! Wächterbedingungen beachten!

Zustands-Konformanztest:

```
K1 = <initial> new Stapel() <empty> ~Stapel() <final>
K2 = <initial> new Stapel() <empty> push() <filled> pop() <empty>
K3 = <initial> new Stapel() <empty> push() <filled> push() <filled>
K4 = <initial> new Stapel() <empty> push() <filled> pop() <filled>
...
K8 = <initial> new Stapel() <empty> push() <filled> push() <full> push() <full>
```

Zustands-Robustheitstest:

```
R1 = <initial> new Stapel() <empty> pop() <fehler>
R2 = <initial> new Stapel() <empty> top() <fehler>
R3 = <initial> new Stapel() <empty> push() <filled>
          ~Stapel() <fehler>
R4 = <initial> new Stapel() <empty> push() <filled> push() <full> ~Stapel() <fehler>
```

6. Ausführen der Tests

- Testfälle bzw. Botschaftsfolgen in ein Testskript verkapseln
- Unter Benutzung eines Testtreibers ausführen
- Zustände über zustandserhaltende Operationen ermitteln und protokollieren

```
K3' = //<initial>
        Stapel OUT = new Stapel(5)
      //<empty>
        OUT.push(new Object())
      //<filled>
        OUT.push(new Object())
      //<filled>
        assert(OUT.size() == 2);
```