Technische Universität Kaiserslautern

Grundlagen Software Engineering

Prüfen objektorientierter Software

Technische Universität Kaiserslautern

Prüfen objektorientierter Software

- Regeln für die Entwicklung
- Eigenschaften objektorientierter Systeme
- Objektorientierter Modultest: Klassentest
- Objektorientierter Integrationstest
- Objektorientierter Systemtest

Prüfen objektorientierter Software
Objektorientierung und Qualitätssicherung

+ Ansatzpunkt: Konzeptionelle Fehler
+ Verwendung von kommerziellen Bibliotheken (ausgetestet)
+ Wiederverwendung (*Reuse*)
+ Durchgängiges Konzept für Analyse, Entwurf, Implementierung
+ Klares Konzept (Regeln des Modells)
+ Paradigma für Problemanalyse
- Einziges Paradigma für Problemanalyse (Paradigma-Blindheit)

Prüfen objektorientierter Software
Objektorientierte Programmierung und Qualitätssicherung

+ Durchgängiges Konzept für Analyse, Entwurf, Implementierung
+ Hohe Produktivität (Bibliotheken)
+ Datenabstraktion
- Enorme Analyseprobleme beim Test (Dynamik ist statisch schwierig nachvollziehbar)
- Dynamisches Binden bereitet Probleme mit Echtzeit
- Polymorphismus
- Komplexitäten durch Vererbung

Entwickeln und Prüfen objektorientierter Software
Regeln für die Entwicklung: Analyse und Entwurf

- Modularisierung ist die Haupteigenschaft der Objektorientierung, die positiv auf die Prüfung wirkt
  - Module sind klar identifiziert (Klassen)
  - Unabhängige Testbarkeit der Klassen aufgrund der Abgeschlossenheit
- Konsequenz: Modularisierungskonzept der Objektorientierung so nutzen, dass für die Prüfung der Software die Voraussetzungen erfüllt sind (muss spätestens beim Entwurf bedacht werden)
  - Modularisierungskonzept entsprechend der Objektorientierung konsequent durchhalten (z. B. Bilden von Datenabstraktionen durch Zusammenfassen logisch zusammengehöriger Daten und Funktionen in Klassen)
  - Abgeschlossenheitskonzept nicht durchbrechen (z. B. keine friend-Klassen in C++)
  - Keine Zugriffe auf Attribute unter Umgehung der dafür zuständigen Methoden
  - Durch konsequente Anwendung der Verfeinerung OOA, OOD, OOP Konsistenz der Programmstruktur mit der Spezifikationsstruktur herstellen

Entwickeln und Prüfen objektorientierter Software
Regeln für die Entwicklung: Analyse und Entwurf

- Vererbung in Kombination mit Polymorphismus ist eine kritische Eigenschaft der Objektorientierung für den Test (Verständlichkeit der Struktur wird verringert)
  - Vererbung vorsichtig verwenden
  - Keine zu tiefen Vererbungshierarchien
  - Mehrfachvererbung nicht zu häufig verwenden
  - Konsequentes Durchhalten einer bestimmten Vererbungshierarchie (typischerweise vom allgemeinen zum speziellen)

Entwickeln und Prüfen objektorientierter Software
Regeln für die Entwicklung: Implementierung

- Beachten der oben erwähnten Regeln auch für die Implementierung
- Steigerung der Beobachtbarkeit der Klasseninterna (z. B. Werte der Klassenattribute) durch Verwendung von so genannten Zusicherungen (sind in C++ Bestandteil des Sprachumfangs)
- In komplexen oder kritischen Teilen der Software verständliche, einfache Programmierung gegenüber eleganten Lösungen vorziehen
- Dynamisches Binden in zeitkritischen Softwareteilen nicht nutzen

Prüfen objektorientierter Software
Eigenschaften objektorientierter Systeme

- Objekte und Klassen sind komplizierter als Funktionen
- Untergrenze der Komplexität entspricht der von Datenabstraktionen bzw. abstrakten Datentypen in klassischen Softwareentwicklungen
- Objekte bzw. Klassen besitzen
  - Beziehungen
  - Eigenschaften
  - Bestandteile bzw. sind selbst Bestandteil
  - einen Zustand
- Sie können
  - Botschaften versenden
  - Operationen ausführen

Prüfen objektorientierter Software
Eigenschaften objektorientierter Systeme

- Sie besitzen
  - Vorbedingungen
  - Nachbedingungen
  - Invarianten
  - Ausnahmebehandlungen (exceptions)
- Die Abläufe zwischen Teilen von Objekten sind kompliziert

⇒ In objektorientierten Softwaresystemen sind wesentliche Teile des Komponententests identisch mit Integrationstestschritten für klassische Softwaresysteme

Prüfen objektorientierter Software
Objektarten

- Essentielle Objekte
  - Modellieren Teile der Anwendung, werden als Teil der Systemanforderung identifiziert
- Nicht essentielle Objekte entstehen während späterer Phasen der Softwareentwicklung (Entwurf, Implementierung)
  - Sind Bestandteil der technischen Realisierung eines Softwaresystems
  - Sind oft Standardkomponenten

Technische Universität Kaiserslautern

Prüfen objektorientierter Software

Objektorientierter Modultest

Technische Universität Kaiserslautern

Objektorientierter Modultest

- Objektorientierter Modultest: Klassentest
- Funktionsorientierte Prüfung einzelner Operationen
- Funktionsorientierte Prüfung von Operationen im Kontext ihrer Klasse
- Strukturorientierte Prüfung
  - Kontrollflussorientierter Test
  - Datenflussorientierter Test
- Eine geeignete Testvorgehensweise
- Testen abstrakter und parametrisierter Klassen

Objektorientierter Modultest: Klassentest
Prüfung einzelner Operationen

- Das Prüfen von Methoden wird im Wesentlichen analog zum Modultest in der klassischen Softwareentwicklung durchgeführt
- Aber Operationen
  - besitzen oft eine sehr einfache Kontrollstruktur
  - sind stark abhängig von den Attributen des Objekts
  - besitzen starke Abhängigkeiten untereinander
- Die Prüfung einzelner Operationen (Funktionstest bzw. Strukturtest) ist sinnvoll unter folgenden Voraussetzungen
  - Die Operation besitzt eine gewisse Mindestkomplexität
  - Die Operation besitzt keine zu starken Abhängigkeiten zu anderen Teilen des Objektes
- Im Regelfall sind Operationen nicht sinnvoll einzeln prüfbar

Objektorientierter Modultest: Klassentest
Prüfung einzelner Operationen
Spezifikation des Operation "Rückgeld_auszahlen"

- Die Klasse Rückgeldauszahler enthält die Information über die für die Rückgeldauszahlung verfügbaren Münzen nach Art und Anzahl. Insgesamt sind max. 50 Münzen à 2 €, 100 Münzen à 1 €, 100 Münzen à 0,50 €, 100 Münzen à 0,20 € und 200 Münzen à 0,10 € möglich. Kleinere Münzen als 0,10 € werden nicht verarbeitet. Die Operation `Rückgeld_auszahlen ()` ermittelt die Art und Anzahl der auszuzahlenden Münzen nach den folgenden Regeln
  - Gezahlt wird mit der geringsten Anzahl Münzen, d. h. der Rückgeldbetrag wird zunächst gegebenenfalls mit 2 €-Münzen beglichen, dann mit 1 €-Münzen, anschließend mit 0,50 €-Münzen, 0,20 €-Münzen und 0,10 €-Münzen.
  - Falls eine benötigte Münzsorte nicht mehr verfügbar ist, so wird mit der nächstkleineren Sorte zurückgegeben.
  - Falls weniger als zwanzig 10-Cent-Münzen verfügbar sind, so wird die Botschaft `Kein_Wechselgeld (ja)` versandt, um die *Passend zahlen-Anzeige* zu aktivieren. Dies geschieht auch, wenn ein Gesamtbetrag des Wechselgelds mit Ausnahme der 2 €-Münzen von 5 € unterschritten wird. Sonst wird die Botschaft `Kein_Wechselgeld (nein)` versandt.

Objektorientierter Modultest: Klassentest
Funktionsorientierte Prüfung einzelner Operationen

☐ Funktionale Äquivalenzklassenaufstellung:

Bedingung	**gültig**		**ungültig**	
Rückgeld	Rückgeld ≥ 2 €	2 € > Rückgeld ≥ 1 €	< 0 €	
	1 € > Rückgeld ≥ 0,50 €	0,50 € > Rückgeld ≥ 0,20 €		
	0,20 € > Rückgeld ≥ 0,10 €	0,10 € > Rückgeld ≥ 0 €		
Münzvorrat 2 €	50 ≥ Münzvorrat > 0	Münzvorrat = 0	< 0	> 50
1 €	100 ≥ Münzvorrat > 0	Münzvorrat = 0	< 0	> 100
0,50 €	100 ≥ Münzvorrat > 0	Münzvorrat = 0	< 0	> 100
0,20 €	100 ≥ Münzvorrat > 0	Münzvorrat = 0	< 0	> 100
0,10 €	200 ≥ Münzvorrat ≥ 20	20 > Münzvorrat ≥ 0	< 0	> 200
Passend zahlen	Anzahl 0,10 € < 20 Gesamtbetrag < 5 €	Anzahl 0,10 € ≥ 20 und Gesamtbetrag ≥ 5 €		

Objektorientierter Modultest: Klassentest
Funktionsorientierte Prüfung einzelner Operationen

☐ Testfälle für gültige Äquivalenzklassen:

Testfall-Nr.	1	2	3	4	5	6
Rückgeld	2 €	1 €	0,50 €	0,20 €	0,10 €	0,00 €
Münzvorrat 2 €	50	1	0	10	0	10
1 €	100	1	0	10	0	10
0,50 €	100	1	0	10	0	10
0,20 €	100	1	10	0	9	10
0,10 €	200	20	1	19	4	40
Passend zahlen	Betr.≥ 5,- Anz. 0,10 ≥ 20	Betr.< 5,- Anz. 0,10 ≥ 20	Betr.< 5,- Anz. 0,10 < 20	Betr.≥ 5,- Anz. 0,10 < 20	Betr.< 5,- Anz. 0,10 < 20	Betr.≥ 5,- Anz. 0,10 ≥ 20
Ergebnis	1 * 2 € *Passend zahlen* nicht aktivieren	1 * 1 € *Passend zahlen* aktivieren	2 * 0,20 € 1 * 0,10 € *Passend zahlen* aktivieren	2 * 0,10 € *Passend zahlen* aktivieren	1 * 0,10 € *Passend zahlen* aktivieren	*Passend zahlen* nicht aktivieren

Objektorientierter Modultest: Klassentest
Funktionsorientierte Prüfung einzelner Operationen

- Testfälle für ungültige Äquivalenzklassen:

Testfall-Nr.	7	8	9	10	11	12	13	14	15	16	17
Rückgeld	1 €	1 €	1 €	1 €	1 €	1 €	1 €	1 €	1 €	1 €	-0,10 €
Münzvorrat 2 €	51	10	20	10	10	-1	20	30	20	14	10
1 €	10	101	20	10	10	10	-1	10	10	10	10
0,50 €	10	1	101	10	10	10	30	-1	22	24	10
0,20 €	10	1	10	101	9	10	40	2	-1	8	10
0,10 €	30	42	30	40	201	10	50	49	30	-1	50

Objektorientierter Modultest: Klassentest
Funktionsorientierte Prüfung von Operationen im Kontext ihrer Klasse

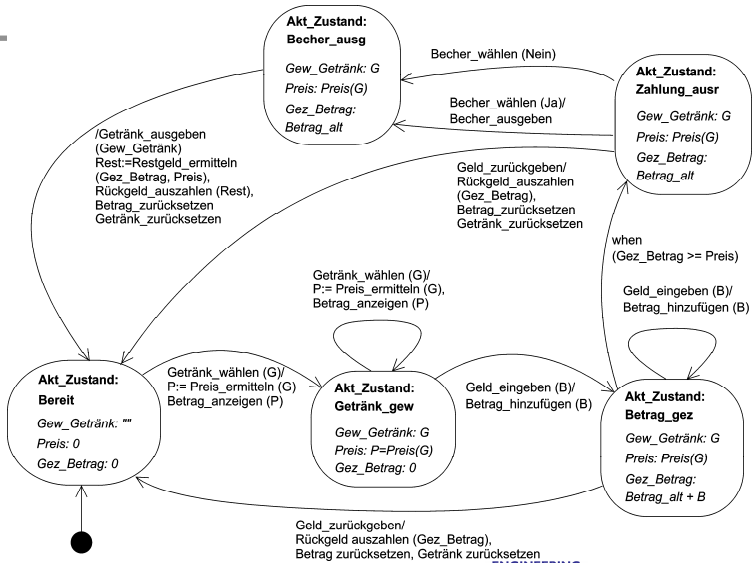
- Im Regelfall müssen Operationen im Kontext ihrer Klasse geprüft werden
- Die Operationen eines Objekts einer Klasse stehen über gemeinsam genutzte Attribute in Wechselwirkung
- Die Werte der Attribute definieren den momentanen Zustand des Objekts

⇒ Zustandsautomat ist ein geeignetes Spezifikationshilfsmittel
⇒ Zustandsautomat kann als Basis der Prüfung dienen

- Anwendungsgebiet: Prüfung von Operationssequenzen

Objektorientierter Modultest: Klassentest
Funktionsorientierte Prüfung von Operationen im Kontext ihrer Klasse

- Beispiel: Getränke-automaten-steuerung



Objektorientierter Modultest: Klassentest
Funktionsorientierte Prüfung von Operationen im Kontext ihrer Klasse

- Hierarchie von Vollständigkeitskriterien
  - Mindestens einmalige Abdeckung aller Zustände
  - Mindestens einmaliges Durchlaufen aller Zustandsübergänge
  - Mindestens einmaliges Erzeugen aller Ereignisse an allen Zustandsübergängen
- Hierarchie
  - Alle Ereignisse $\supseteq$ Alle Zustandsübergänge $\supseteq$ Alle Zustände
- Wichtig: Prüfung der Fehlerbehandlung nicht vergessen

Objektorientierter Modultest: Klassentest
Funktionsorientierte Prüfung von Operationen im Kontext ihrer Klasse

- Der Test aller Zustandsübergänge ist z.B. mit den folgenden Testfällen möglich
  1. Bereit, Getränk wählen -> Getränk gewählt, Getränk wählen -> Getränk gewählt, Geld eingeben -> Betrag gezahlt, Gezahlter Betrag < Preis des gewählten Getränks und Geld eingeben -> Betrag gezahlt, Geld zurückgeben -> Bereit
  2. Bereit, Getränk wählen -> Getränk gewählt, Geld eingeben -> Betrag gezahlt, Gezahlter Betrag >= Preis des gewählten Getränks -> Zahlung ausreichend, Geld zurückgeben -> Bereit
  3. Bereit, Getränk wählen -> Getränk gewählt, Geld eingeben -> Betrag gezahlt, Gezahlter Betrag >= Preis des gewählten Getränks -> Zahlung ausreichend, Becher wählen(Ja) -> Becher ausgeben, - -> Bereit
  4. Bereit, Getränk wählen -> Getränk gewählt, Geld eingeben -> Betrag gezahlt, Gezahlter Betrag >= Preis des gewählten Getränks -> Zahlung ausreichend, Becher wählen(Nein) -> Becher ausgeben, - -> Bereit
- Die Bestimmung der Werte für Schnittstellenparameter kann durch Äquivalenzklassenanalyse geschehen

Objektorientierter Modultest: Klassentest
Strukturorientierte Prüfung

- Kontrollflußorientierte Testtechniken (z.B. Zweigüberdeckungstest) sind relativ ungeeignet, weil sie die Kopplungen zwischen Operationen eines Objekts durch gemeinsam genutzte Attribute nicht beachten; aber fast alle Testwerkzeuge für objektorientierte Sprachen unterstützen den Zweigüberdeckungstest
- Datenflußorientierte Testtechniken sind besser geeignet, weil sie Kopplungen zwischen Operationen beachten
- Die Attribute werden von Operationen geschrieben (*def*) und gelesen (*use*). Ein Datenflußtest auf Basis der Attribute fordert den Test von Interaktionen über die gemeinsam genutzten Daten.
  (Beispiel: Definition und Benutzung der Attribute "Gew_Getränk" und "Preis" im Zustand "Getränk_gew" der Getränkeautomatensteuerung)

Objektorientierter Modultest: Klassentest
Strukturorientierte Prüfung

Implementierung der Operation “Rueckgeld_auszahlen”:

```
int Rueckgeld_auszahlen (int Betrag)
    // Betrag in Cent
    // Zahlt Münzen aus; gibt Restbetrag zurück;
    // Meldet „Kein Wechselgeld”, gibt im Fehlerfall -1 zurück
{
    int Anz2, Anz1, Anz050, Anz020, Anz010 ;
    int Vorrat2, Vorrat1, Vorrat050, Vorrat020, Vorrat010;
    Vorrat2 = Euro2.Anzahl();
    Vorrat1 = Euro1.Anzahl();
    Vorrat050 = Euro050.Anzahl();
    Vorrat020 = Euro020.Anzahl();
    Vorrat010 = Euro010.Anzahl();
```


Objektorientierter Modultest: Klassentest
Strukturorientierte Prüfung

```
    // Bereichsprüfung
    if ((Betrag < 0) || (Vorrat2 < 0)|| (Vorrat1 < 0) || (Vorrat050 < 0)
            || (Vorrat020 < 0) || (Vorrat010 < 0)) return (-1);
    Anz2 = Betrag / 200;
    if (Anz2 <= Vorrat2)
    {
            Euro2.Entnehmen (Anz2);
            Betrag = Betrag - Anz2 * 200;
    }
    else
    {
            Euro2.Entnehmen (Vorrat2);
            Betrag = Betrag - Vorrat2 * 200;
    }
```

Objektorientierter Modultest: Klassentest
Strukturorientierte Prüfung

```
Anz1 = Betrag / 100;
if (Anz1 <= Vorrat1)
{
        Euro1.Entnehmen (Anz1);
        Betrag = Betrag - Anz1 * 100;
}
else
{
        Euro1.Entnehmen (Vorrat1);
        Betrag = Betrag - Vorrat1 * 100;
}
Anz050 = Betrag / 50;
if (Anz050 <= Vorrat050)
{
        Euro050.Entnehmen (Anz050);
        Betrag = Betrag - Anz050 * 50;
}
else
{
        Euro050.Entnehmen (Vorrat050);
        Betrag = Betrag - Vorrat050 * 50;
}
```


Objektorientierter Modultest: Klassentest
Strukturorientierte Prüfung

```
Anz020 = Betrag / 20;
        if (Anz020 <= Vorrat020)
{
        Euro020.Entnehmen (Anz020);
        Betrag = Betrag - Anz5 * 20;
}
else
{
        Euro020.Entnehmen (Vorrat020);
        Betrag = Betrag – Vorrat020 * 20;
}
Anz010 = Betrag / 10;
if (Anz010 <= Vorrat010)
{
        Euro010.Entnehmen (Anz010);
        Betrag = Betrag - Anz010 * 10;
}
else
{
        Euro010.Entnehmen (Vorrat010);
        Betrag = Betrag - Vorrat010 * 10;
}
```

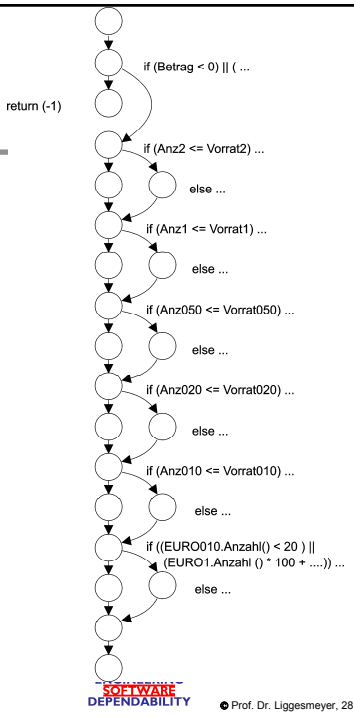
Objektorientierter Modultest: Klassentest
Strukturorientierte Prüfung

```
if ((Euro010.Anzahl() < 20) ||
     (Euro1.Anzahl() * 100 + Euro050.Anzahl() * 50 +
     Euro020.Anzahl() * 20 + Euro010.Anzahl() * 10 < 500))
      {
            Kundenbedieneinheit.Kein_Wechselgeld (ja)
      }
     else
      {
      Kundenbedieneinheit.Kein_Wechselgeld (nein)
      }
     return (Betrag);
};
```

- Übung:
  Prüfen Sie, ob die mit dem funktionalen Äquivalenzklassenverfahren gebildeten Testfälle für die Operation "`Rueckgeld_auszahlen()`" eine vollständige Zweigüberdeckung ergeben. Bilden Sie ggf. weitere Testfälle um dieses Ziel zu erreichen.

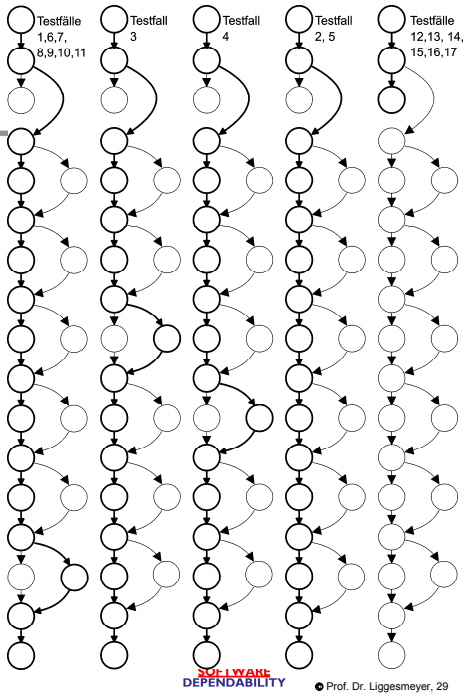
Objektorientierter Modultest: Klassentest
Kontrollflussorientierter Test

- Kontrollflussgraph der Operation „`Rueckgeld_auszahlen()`"



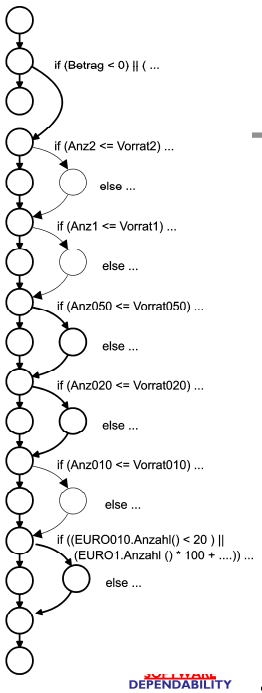
Objektorientierter Modultest: Klassentest Kontrollflussorientierter Test

- Durch funktionsorientierte Testfälle durchlaufene Pfade



Objektorientierter Modultest: Klassentest Kontrollflussorientierter Test

- Insgesamt ausgeführte Zweige und Anweisungen der Operation „Rueckgeld_auszahlen()“



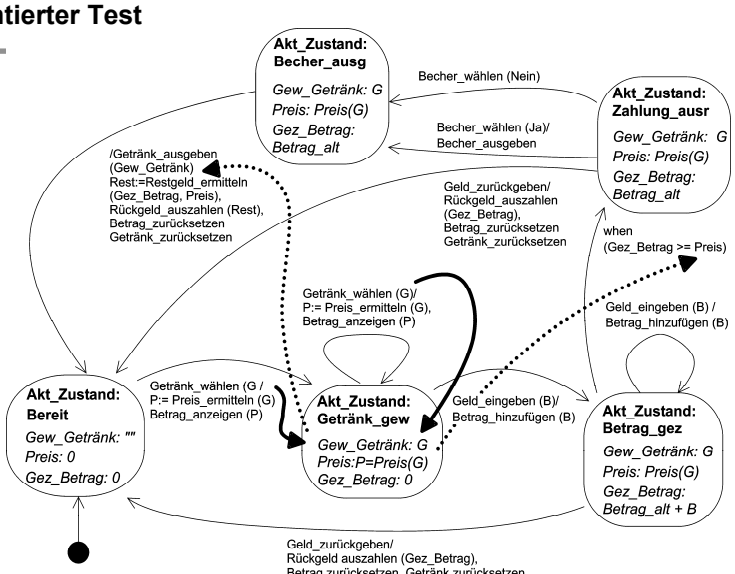
Objektorientierter Modultest: Klassentest
Kontrollflussorientierter Test

Zusätzliche Testfälle für die Erreichung einer vollständigen Zweigüberdeckung

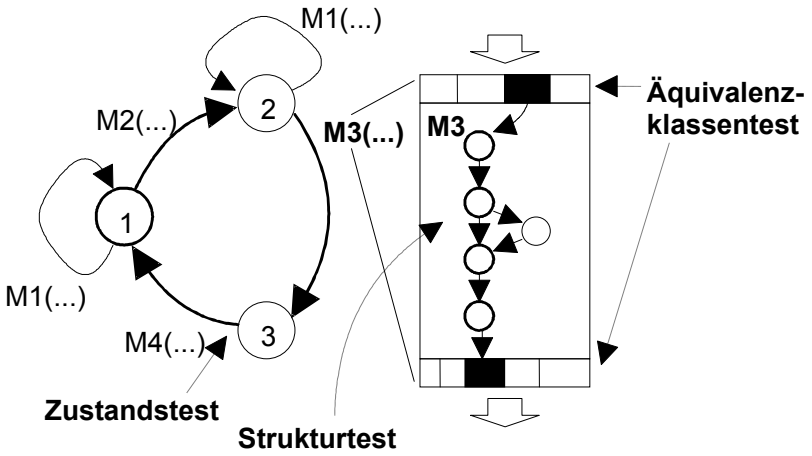
Testfall-Nr.	18	19	20
Rückgeld	2 €	1 €	1,10 €
Münzvorrat 2 €	0	10	0
1 €	20	0	10
0,50 €	20	30	0
0,20 €	0	5	10
0,10 €	100	40	0
Passend zahlen	Betr. ≥ 5,- Anz. 0,10 ≥ 20	Betr. ≥ 5,- Anz. 0,10 ≥ 20	Betr.≥ 5,- Anz. 0,10 < 20
Ergebnis	2 * 1 € *Passend zahlen* nicht aktivieren	2 * 0,50 € *Passend zahlen* nicht aktivieren	1 * 1 € *Passend zahlen* aktivieren

Objektorientierter Modultest: Klassentest
Datenflussorientierter Test

- Schreib- und Lesezugriffe der Attribute "Gew_Getränk" und "Preis" im Zustand "Getränk_gew" der Getränkeauto-matensteuerung



Objektorientierter Modultest: Klassentest
Eine geeignete Testvorgehensweise



Objektorientierter Modultest: Klassentest
Prüfung "generischer" Klassen

- Problem
  - Abstrakte und parametrisierte Klassen gestatten keine (direkte) Erzeugung von Objekten
  - Vielfalt der erzeugbaren Objekte erhöht die Komplexität
  - Getestet werden können nur konkrete Objekte. Frage: Welche?
  - Abstrakte und parametrisierte Klassen verhalten sich zu konkreten Klassen wie normale Klassen zu ihren Objekten

Objektorientierter Modultest: Klassentest
Prüfung "generischer" Klassen

- Abstrakte Klassen
  - Legen die syntaktische und semantische Schnittstelle für Operationen fest, ohne eine Implementation anzubieten
  - Die Implementation für abstrakte Methoden wird in einer Unterklasse der abstrakten Klasse gegeben
  - Strukturieren eine Menge von Klassen
- Parametrisierte Klassen
  - Enthalten formale Klassenparameter, die durch aktuelle Werte (Klassen) ersetzt werden müssen

Objektorientierter Modultest: Klassentest
Testen abstrakter und parametrisierter Klassen

- Instanzierung einer konkreten Klasse
- Test dieser Klasse wie eine gewöhnliche Klasse
- Fragen
  - Welche Instanzierung ist zu wählen?
  - Wie ist beim Testen methodisch vorzugehen?
- Regel: Erzeugung einer möglichst einfachen konkreten Klasse, d. h.:
  - Abstrakte Klassen
    - Realisierung von Implementationen für abstrakte Methoden
      - Falls möglich, leere Implementationen

      Sonst:
      - So einfach wie möglich, mit der Nebenbedingung, dass die Spezifikation erfüllt wird; nur so kompliziert wie erforderlich, um einen sinnvollen Test zu gewährleisten.
  - Parametrisierte Klassen
    - Wahl von Parametern, die den Test möglichst einfach gestalten (z. B. "Stapel für Integer")

Technische Universität Kaiserslautern

Prüfen objektorientierter Software

Objektorientierter Integrationstest

Technische Universität Kaiserslautern

Objektorientierter Integrationstest

- Integrationstest von Basisklassen
- Integrationstest von dienstanbietenden abgeleiteten Klassen
- Integrationstest von Dienstaufrufen aus abgeleiteten Klassen
- Integrationstest dienstanbietender und dienstnutzender Unterklassen
- Vererbung und Integrationstest: Zusammenfassung

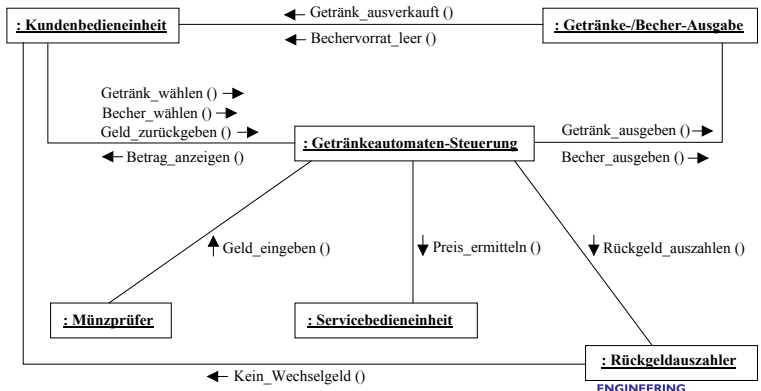
Objektorientierter Integrationstest
Integrationstest von Basisklassen

- Voraussetzung: Getestete Basisklassen aus dem Modultest
- Fragen
  - Sind die Aufrufe von Diensten des Dienstanbieters seitens des Dienstbenutzers korrekt?
  - Funktioniert die Übergabe und Interpretation von Ergebnissen korrekt?
- Idee
  - Überdeckung der Spezifikation des Dienstnutzers und Dienstanbieters
- Erzeugung von Testfällen
  - die die unterschiedlichen vom Dienstanbieter erzeugbaren Rückgabewerte überdecken.
  - die die unterschiedlichen vom Dienstnutzer verarbeitbaren Rückgabewerte überdecken

⇒ Funktionale Äquivalenzklassenbildung der Schnittstelle zwischen Dienstbenutzer und Dienstanbieter

Objektorientierter Integrationstest
Integrationstest von Basisklassen

- Interaktion der Klassen *Münzprüfer* und *Getränkeautomaten-Steuerung* über die Botschaft *Geld_eingeben ()* des Getränkeautomaten



Objektorientierter Integrationstest
Integrationstest von Basisklassen

- Schnittstellenspezifikation der Operation Geld_eingeben ()
  - Die Operation Geld_eingeben () erwartet einen nicht-negativen Schnittstellenparameter, der maximal 500 sein kann. Er gibt den Geldbetrag in Cent an. Die Operation besitzt keinen Rückgabewert
- Schnittstellenspezifikation der Operation Prüfen () in Bezug auf die Botschaft Geld_eingeben ()
  - Prüfen belegt den Schnittstellenparameter der Botschaft Geld_eingeben mit einem der folgenden Werte: 10, 20, 50, 100, 200

Objektorientierter Integrationstest
Integrationstest von Basisklassen

- Folgen für den Integrationstest:
  - Es ist sicherzustellen, dass der Betrag, der der Operation "Geld_eingeben (Betrag)" übergeben wird, die folgende Bedingung erfüllt (sog. Zusicherung): *(Betrag >= 0) AND (Betrag <= 500)*
  - Äquivalenzklassen und gleichzeitig Testfälle für den Aufruf (Schnittstellen-Äquivalenzklassen des Dienstbenutzers in Aufrufrichtung):
    - Betrag = 10
    - Betrag = 20
    - Betrag = 50
    - Betrag = 100
    - Betrag = 200
  - Test der zurückgelieferten Ergebnisse nicht erforderlich, da keine Rückgabewerte vorhanden

Objektorientierter Integrationstest
Integrationstest in Gegenwart von Vererbung

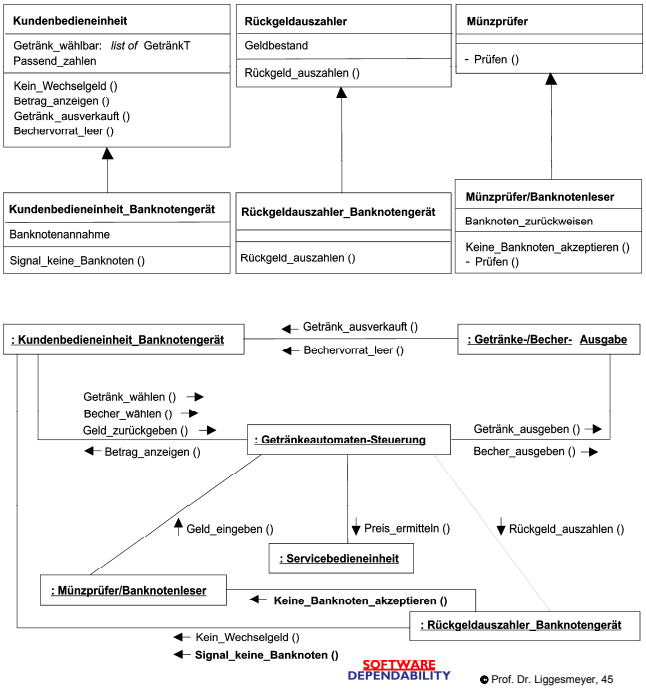
- Unterschiedliche Situationen
  - Integrationstest von dienstanbietenden abgeleiteten Klassen
  - Integrationstest von Dienstaufrufen aus abgeleiteten Klassen
  - Integrationstest dienstanbietender und dienstnutzender Unterklassen

Objektorientierter Integrationstest
Integrationstest in Gegenwart von Vererbung

- In der neuen Version des Getränkeautomaten ist es möglich, mit Banknoten (5 € und 10 €) zu zahlen. Um diese Funktionalität zu realisieren, werden folgende Änderungen vorgenommen
  - Zu der Klasse Münzprüfer wird durch Vererbung eine neue Klasse Münzprüfer/Banknotenleser erzeugt, deren Operation Prüfen () um das Prüfen der Banknoten erweitert ist. Diese Operation überschreibt die ursprüngliche Operation
  - Ferner wird eine neue Operation Keine_Banknoten_akzeptieren () vereinbart, deren Ausführung je nach Parameter die Akzeptanz von Banknoten sperrt oder freigibt
  - Zu der Klasse Rückgeldauszahler wird eine Unterklasse Rückgeldauszahler_Banknotengerät vereinbart, da eine die alte Operation überschreibende Operation Rückgeld_auszahlen () erforderlich ist, die den Klassen Münzprüfer/Banknotenleser und Kundenbedieneinheit_Banknotengerät signalisiert, ob mit Banknoten gezahlt werden kann
  - Das Zahlen mit Banknoten wird gesperrt, falls der Geldbestand in Münzen 15 Euro unterschreitet. Banknoten werden als Wechselgeld nicht zurückgegeben
  - Zu der Klasse Kundenbedieneinheit wird eine Unterklasse Kundenbedieneinheit_Banknotengerät erzeugt, die um die Operation Signal_keine_Banknoten () erweitert ist. Diese Operation setzt ein Signal, das dem Kunden die Freigabe bzw. Sperrung der Zahlung mit Banknoten signalisiert

Objektorientierter Integrationstest Vererbung

- Der Getränkeautomat mit Banknotenleser



Objektorientierter Integrationstest: Vererbung Integrationstest von dienstanbietenden abgeleiteten Klassen

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

- Situation
  - Der Dienstbenutzer benutzt Dienste einer abgeleiteten Klasse
- Voraussetzung
  - Dienstnutzer, Dienstanbieter und Oberklasse des Dienstanbieters sind sinnvoll klassengetestet
  - Integrationstest des Dienstnutzers und der Oberklasse des Dienstanbieters entsprechend der Vorgehensweise zum Integrationstest von Basisklassen durchgeführt
- Problem
  - Operationen des Dienstanbieters können von der Oberklasse (dem alten Dienstanbieter) unverändert geerbt sein, müssen aber nicht. Es können sowohl Methoden des neuen Dienstanbieters als auch Methoden des alten Dienstanbieters (der Oberklasse) ausgeführt werden
- Fragen
  - Sind die Aufrufe von Diensten seitens des Dienstbenutzers korrekt?
  - Funktioniert die Übergabe und Interpretation von Ergebnissen korrekt?

Integrationstest von dienstanbietenden abgeleiteten Klassen
Korrektheit der Aufrufe von Methoden des Dienstanbieter

- Vorgehensweise
  - Keine zusätzlichen Testfälle für ererbte Methoden, da dieser Fall bereits durch den Integrationstest der Basisklassen abgedeckt ist => Testfälle wiederholen
  - Keine zusätzlichen Testfälle bei überschriebenen Methoden, für die sich lediglich die Implementation geändert hat, da die Schnittstellenspezifikation identisch geblieben ist und dieser Fall ebenfalls bereits abgedeckt ist => Testfälle wiederholen
  - Falls sich die Schnittstellenspezifikation der überschreibenden Methode geändert hat, so ist eine Fallunterscheidung nötig:
    - Die Schnittstelle der überschreibenden Methode ist spezieller (d. h. akzeptiert weniger Daten) als die Schnittstelle der überschriebenen Methode.
      - Definition einer neuen Zusicherung

Integrationstest von dienstanbietenden abgeleiteten Klassen
Korrektheit der Aufrufe von Methoden des Dienstanbieters

      - Wiederholung sämtlicher Testfälle aus dem Integrationstest der Basisklassen; Reaktionsmöglichkeiten im Fehlerfall:
        - Modifikation des Dienstbenutzers, so dass nur korrekte Aufrufe abgesetzt werden
        - Modifikation des Dienstbenutzers, so dass keine Aufrufe zur überschreibenden Methode abgesetzt werden
        - Modifikation der überschreibenden Methode, so dass alle Aufrufe des Dienstbenutzers korrekt verarbeitet werden können
    - Falls die Schnittstelle durch das Überschreiben der Methode allgemeiner wird, so sind keine zusätzlichen Testfälle erforderlich, da dieser Fall bereits durch den Test der Basisklassen abgedeckt wird => Testfälle wiederholen

Integrationstest von dienstanbietenden abgeleiteten Klassen
Korrekte Übergabe und Interpretation der Ergebnisse

- Ggf. zusätzliche Testfälle für die Überdeckung einer breiteren Schnittstellenspezifikation der überschreibenden Methode, die beim Test der Basisklassen nicht hinreichend abgedeckt worden sind
- Beispiel: Getränkeautomat mit Banknotenleser
  - Der durch Vererbung entstandene neue Rückgeldauszahler ist ein Dienstanbieter ("Rückgeld_auszahlen ()") gegenüber der Getränkeautomatensteuerung
  - Die überschreibende Methode "Rückgeld_auszahlen ()" besitzt im Vergleich zur überschriebenen Methode jedoch nur eine geänderte Implementation (Versand zusätzlicher Botschaften). Die Schnittstellenspezifikation ist unverändert
  - Für den Integrationstest von "Getränkeautomatensteuerung" und "Rückgeldauszahler_Banknotengerät" ist es ausreichend, die alten Testfälle zu wiederholen. Die Zusicherung ist unverändert

Integrationstest von Dienstaufrufen aus abgeleiteten Klassen Überprüfung der Korrektheit der Aufrufe

- Keine zusätzlichen Testfälle für nicht überschreibende Methoden des Dienstnutzers => Testfälle wiederholen
- Keine zusätzlichen Testfälle, falls die Schnittstelle der überschreibenden Methode in Aufrufrichtung spezieller ist als die Schnittstelle der überschriebenen Methode (d. h. Aufrufe, die vorher möglich waren, sind nicht mehr möglich)

  ⇒ Testfälle wiederholen
- Falls die Schnittstelle durch das Überschreiben der Methode allgemeiner wird (d. h. Aufrufe, die vorher nicht möglich waren, sind möglich), so sind die alten Testfälle entsprechend zu ergänzen

  ⇒ alte Testfälle wiederholen und neue ergänzend durchführen
- Anmerkung: Die Zusicherung ist unverändert

Integrationstest von Dienstaufrufen aus abgeleiteten Klassen
Test der korrekten Interpretation von Übergabeparametern

- Testfälle wiederholen. Falls ein Fehler aufgrund einer spezielleren Schnittstelle in Rückgaberichtung auftritt, so ist eine entsprechende Korrektur erforderlich

Integrationstest dienstanbietender und dienstnutzender Unterklassen

- Vorgehensweise
  - Integrationstest der dienstanbietenden Unterklasse
  - Integrationstest der dienstnutzenden Unterklasse
  - Zusätzliche Testfälle für die Wechselwirkung in dienstanbietender und dienstnutzender Unterklasse

Integrationstest dienstanbietender und dienstnutzender Unterklassen

Beispiel: Getränkeautomat mit Banknotenleser

- Zwischen den abgeleiteten Klassen "Rückgeldauszahler_Banknoten-gerät" und "Münzprüfer/Banknotenleser" besteht eine Dienstanbieter-Dienstnutzer-Beziehung. Zusätzlich zu den beschriebenen Tests ist diese durch Überprüfung der Interaktion durch die Botschaft "Keine_ Banknoten_akzeptieren ()" zu testen
- Testfälle
  - Keine_Banknoten_akzeptieren (ja)
  - Keine_Banknoten_akzeptieren (nein)
- Eine analoge Situation existiert zwischen "Rückgeldauszahler_ Banknotengerät" und "Kundenbedieneinheit_Banknotengerät"

Objektorientierter Integrationstest
Vererbung und Integrationstest: Zusammenfassung

- Basistabelle für die Beachtung von Vererbung

Dienstnutzer	Dienstanbieter	Aktion
unverändert	unverändert	Testfälle wiederholen
unverändert	durch Vererbung erzeugt	Tabellen 1.1 und 1.2 auswerten
durch Vererbung erzeugt	unverändert	Tabelle 2 auswerten
durch Vererbung erzeugt	durch Vererbung erzeugt	Tabellen 1.1, 1.2 und 2 auswerten; Testfälle für Interaktion der Subklassen ergänzen

Objektorientierter Integrationstest
Vererbung und Integrationstest: Zusammenfassung

- Tabelle 1.1: Test der Aufrufschnittstelle

Dienstanbietende Operation	**Aufrufschnittstelle der dienstanbietenden Operation**	**Aktion**
geerbt	-	Testfälle wiederholen
überschreibend	identisch	Testfälle wiederholen
überschreibend	spezieller	Neue Zusicherung; Testfälle wiederholen
überschreibend	allgemeiner	Testfälle wiederholen

Objektorientierter Integrationstest
Vererbung und Integrationstest: Zusammenfassung

- Tabelle 1.2: Test der Rückgabeschnittstelle

Dienstanbietende Operation	**Rückgabeschnittstelle der dienstanbietenden Operation**	**Aktion**
geerbt	-	Testfälle wiederholen
überschreibend	identisch	Testfälle wiederholen
überschreibend	spezieller	Testfälle wiederholen
überschreibend	allgemeiner	Testfälle wiederholen; zusätzliche Testfälle erzeugen

Objektorientierter Integrationstest
Vererbung und Integrationstest: Zusammenfassung

- Tabelle 2: Test der Aufruf- und der Rückgabeschnittstelle

Dienstnutzende Operation	**Aufrufschnittstelle der dienstnutzenden Operation**	**Aktion**
geerbt	-	Testfälle wiederholen
überschreibend	identisch	Testfälle wiederholen
überschreibend	spezieller	Testfälle wiederholen
überschreibend	allgemeiner	Testfälle wiederholen; zusätzliche Testfälle erzeugen

Prüfen objektorientierter Software

Objektorientierter Systemtest

Objektorientierter Systemtest

- Eigenschaften
- Funktionstest
- Regressionstest
- Leistungstests
- Stresstest und Beta-Test

Objektorientierter Systemtest
Eigenschaften

- Mit Ausnahme des Funktionstests kaum Unterschiede gegenüber dem Test von klassischer Software
  - Das System ist eine Black Box
    ⇒ Ob es objektorientiert oder klassisch entwickelt ist, spielt keine Rolle
- Erzeugung funktionsorientierter Testfälle aus OOA-Diagrammen
  - DFDs
  - Zustandsautomaten (Rumbaugh)
  - Use Cases nach Jacobson:
    - Szenarien aus der Sicht eines Systembenutzers (Mensch oder anderes System)
    - Nicht sehr systematisch
    - Keine Vollständigkeit bei komplizierten Systemen
    - Können mit Zeitbedingungen annotiert werden
      ⇒ Leistungstest!

Objektorientierter Systemtest
Funktionstest

- Überprüft, ob alle definierten Funktionen vorhanden und wie vorgesehen implementiert sind. Als Referenz dient die Anforderungsdefinition
- Als Basis für die Erzeugung funktionsorientierter Testfälle aus objektorientierten Analyse-Diagrammen bietet sich je nach verwendeter OOA-Technik die Nutzung der folgenden Techniken an
  - Datenflussdiagramme
  - Zustandsautomaten
  - Sequence Charts
  - Use Cases
    - Lineare Sequenzen in zeitlicher Abfolge (Timethreads) oder
    - Baumförmige Darstellung der Szenarien (im Sinne eines Entscheidungsbaums)

Objektorientierter Systemtest
Regressionstest

- Im Falle von Versionsentwicklungen eines Software-Systems, aber auch im Falle neuer Versionen als Folge notwendiger Fehlerkorrekturen ist ein Regressionstest erforderlich
- Regressionstests sind möglichst zu automatisieren und zwar einschließlich des Soll-/Ist-Ergebnisvergleichs

Objektorientierter Systemtest
Leistungstests

- Leistungstests dienen zur Überprüfung des in der Anforderungsdefinition festgelegten Leistungsverhaltens. Dies betrifft sowohl die verarbeitbaren Mengen – z. B. Anzahl angeschlossener Sensoren – als auch das Antwortzeitverhalten. Leistungstests betreiben die Software an der Grenze der verkraftbaren Last, ohne diese Grenze zu überschreiten. Die Software muss bei Leistungstests folglich ein normales Verhalten zeigen. Der Test des Zeitverhaltens kann z. B. gegen entsprechend annotierte Use Cases durchgeführt werden. Eine Besonderheit objektorientiert programmierter Software ist das so genannte dynamische Binden, das die garantierte Einhaltung oberer Zeitschranken erschwert. In der Regel wird dynamisches Binden bei zeitkritischen Abläufen verboten. Dies kann sinnvoll durch ein Review der betroffenen Klassen geprüft werden. Das gehört allerdings in den Bereich des objektorientierten Komponententests.

Objektorientierter Systemtest
Stresstest und Beta-Test

- Beim Stresstest werden die Grenzen der verkraftbaren Last gezielt überschritten, während der Software bzw. dem System gleichzeitig die erforderlichen Ressourcen entzogen werden. Eine typische Testsituation ist die Messung des Antwortzeitverhaltens einer sicherheitskritischen als Mehrprozessorsystem aufgebauten Steuerung bei Überlast und Ausfall eines Prozessors. Hier sollen Fragen beantwortet werden, wie z. B.: Wie verhält sich ein System nach einer aufgetretenen Überlast, wenn die Last in den normalen Bereich zurückgeht? Stresstests erfordern in der Regel eine geeignete Werkzeugunterstützung, um Last zu simulieren. Regressionstestwerkzeuge enthalten oft derartige Stresstestkomponenten.
- Der Beta-Test besteht in der Installation der Software bei einigen speziell ausgewählten Pilotkunden, mit dem Ziel noch vorhandene Fehler zu erkennen und abzustellen, bevor die Software in größerer Stückzahl in den Markt gebracht wird.