Materialsammlung - Programming by Contract

Prof. Dr. Hans-Jürgen Buhl

Sommersemester 2007

Bergische Universität Wuppertal
Fachbereich C — Mathematik und Informatik

Inhaltsverzeichnis

Vorbemerkungen – Softwarequalität heute . . . 3
Haftung . . . 3
Beispiele für Softwaredisfunktionalitäten . . . 5
Deep Impact . . . 5
USV-Software legt Server lahm . . . 5
Chaos an Hannovers Geldautomaten . . . 6
Therac 25 . . . 6
Berliner Magnetbahn . . . 7
Elektronik-Fehler führt zu Überhitzung bei Volvo-PKW . . . 7
The Patriot Missile . . . 8
Kontenabrufverfahren startet wegen Softwareproblemen als Provisorium 9
Buffer Overflow im Linux-Kernel . . . 9
Auch Superhirne können irren - das Risiko Computer . . . 10
Explosion der Ariane 5 . . . 11
Neueste Risikoinformationen/Softwareprobleme . . . 11
Programming by Contract im Umfeld der Algorithmenverifikation . . . 13
Programmverifikation . . . 13
Programming by Contract . . . 14

1 Qualitätsanforderungen an SW-Produkte 17
1.1 Prinzipien der ordnungsgemäßen Programmerstellung . . . 18
1.2 Spezifikation einer abstrakten Datenkapsel . . . 19
1.2.1 Axiomatische Spezifikation . . . 19
1.2.2 Beschreibende (denotationale) Spezifikation . . . 19
1.3 Prinzipien der Moduarisierung: . . . 20
1.4 Typen der Modularisierung . . . 20

2 Wiederverwendbarkeit 21
2.1 Begriffshierarchien . . . 23
2.2 Objekthierarchien als strukturierte Modulsammlungen: Beispiele aus Eiffel 24
2.2.1 Vererbung und Erweiterung . . . 24
2.2.2 Vererbung und Abänderung . . . 25
2.2.3 Generizität . . . 26
2.2.4 Eingeschränkte Generizität . . . 26
2.2.5 Polymorphie und „late binding“ . . . 26
2.2.6 Aufgeschobene Feature-Implementierungen . . . 27
2.3 Resümee . . . 28

3 Programming by Contract **31**
3.1 Spezifikation durch Verträge 31
3.2 Invarianten 33
3.3 Nachbedingungen 33
3.4 Vorbedingungen 34

4 Qualitätssicherung mit normalen C++ Sprachmitteln **37**
4.1 Umgangssprarchliche Spezifikation? 37
4.2 Unbeachtet - Integer-Overflows in C/C++ (Vermeidung von Qverflows) . 39
4.3 Vergessene Problematik: unordered floats — IEEE NaNs 44
4.4 assert in C/C++ 45
4.5 Vermeidung von `enum` 46
4.6 Compiletime Assertions 50
4.7 Ausnahmebedingungen: Exceptions/Traps 52

5 Spezifikation mit sprachexternen Hilfsmitteln: VDM, NANA, OC **57**
5.1 Invarianten in der Spezifikationssprache VDM 57
5.2 Ausnahmen (errs-Klausel) in VDM 58
5.3 Module in VDM 58
5.4 Klassen in VDM++ 59
5.5 Object Constraint Language (OCL) 59
5.5.1 Vor- und Nachbedingungen: 59
5.5.2 Invarianten 59
5.6 ANNA (annotation language for ADA) 60
5.7 Contracting and Subcontracting 65
5.7.1 Contracting 65
5.7.2 Subcontracting (is-a-Vererbung) 66

6 Sprachabstraktion füt gut lesbare Spezifikationen: newmat10, STL, ... **73**
6.1 newmat10 73
6.1.1 Matrizen/Vektoren in newmat10 73
6.1.2 Generelle Beschreibung/General description 73
6.1.3 Matrizenmanipulation 74
6.1.4 Ausdrucksauswertung – lazy evaluation 75
6.1.5 Eine Beispielanwendung 77
6.2 STL 80
6.3 CXSC 80

7 Degign by Contract, by Example, in C++ with nana **81**
7.1 A first Taste of Design by Contract 82
7.2 Elementary Principles of Design by Contract 87
7.2.1 First Trial 87
7.2.2 Redesign 91
7.2.3 Destruktor, Kopierkonstruktor und Wertzuweisung 96

7.3 Applying the Six Principles 101
7.3.1 Design und Contracts 101
7.3.2 Implementierung und Tests 105
7.3.3 konstante Referenzparameter/private Hilfsmethoden für die Spezifikation/old-Wert durch I
7.3.4 old-Wert durch den Kopierkonstruktor 116
7.3.5 Redesign 117
7.4 Immutable Lists 118
7.5 Using Immutable Lists 118
7.6 Subcontracting in Design by Contract in Nana 120
7.6.1 name_list-Design (Subcontracting) 120
7.6.2 Implementierung und Tests 124
7.6.3 Mit Frameregel 127
7.6.4 Mit Iterator-Methode (Design) 129
7.6.5 Implementierung der Iterator-Methode 131
7.6.6 Test des Iterators in display_contents() und main() 132
7.6.7 Qstl.h bei Contracts und Klassen mit eigenen Iteratoren: Framebedingungen mit Hilfe eine
7.6.8 Hilfsoperatoren für die STL 135
7.7 Neuformulierung: Regeln und Leitlinien für PbC in C++ 136

Abbildungsverzeichnis

0.1 Design by Contract, by Example von Richard Mitchell und Jim McKim . . . 4
0.2 Bilder von Deep Impact . . . 5
0.3 http://catless.ncl.ac.uk/Risks/22.92.html . . . 11

2.1 Begriffshierarchien . . . 23

3.1 Kunden-Lieferanten-Modell . . . 31

6.1 (A+B)*C - Baum . . . 76

Tabellenverzeichnis

0.1 Divergence in the Range Gate of a PATRIOT MISSILE 8
3.1 Pflichten - Nutzen von Kunden und Lieferanten 32
5.1 Verpflichtungen/Vorteile von Verträgen zwischen Komponentenanbieter und -benutzer 65
6.1 newmath10 - Eigenschaften 75

Programming by Contract

2 V Do 10 - 12 in D13.08

Einordnung: Diplom Mathematik/Nebenfach Informatik: Hauptstudium - Praktische und Technische Informatik; Bachelor IT: Praktische Informatik A - Programmiersprachen und Sprachkonzepte; Master Wirtschaftsmathematik: Wahlpflichtbereich Informatik; Wirtschaftswissenschaften: Modul I - Software- und Programmiertechnik; Studienschwerpunkte und Nebenfächer Informatik anderer Studiengänge

Vorkenntnisse: Einführung in die Informatik; Programmierkenntnisse in C++; erfolgreiche Teilnahme an „Einführung in die Benutzung der Ausbildungsrechner"

Inhalt: Die Programmiermethodik „Programming/Design by Contract" klärt die Verantwortlichkeit von Diensteanbieter (Funktion/Methode) und Dienstenehmer (Aufrufer einer Funktion) durch genaue Vereinbarungen. Durch das Sprachmittel der Zusicherung werden Voraussetzungen, Diensteerfüllung und Ausnahmebedingungen zur Laufzeit eines Programms (automatisch) überprüft und führen zu Code besserer Qualität.

Literatur: wird in der Veranstaltung bekannt gegeben.

Vorbemerkungen – Softwarequalität heute

Haftungsausschluß

Die Überlassung dieser Baupläne erfolgt ohne Gewähr. Der Plan gibt keine Garantie, Gewährleistung oder Zusicherung, daß diese Pläne für einen bestimmten Zweck geeignet sind, daß sie richtig sind oder daß ein Gebäude, das nach diesen Plänen gebaut wird, den Ansprüchen des jeweiligen Erwerbers genügt. Der Planer erklärt sich bereit, Ersatzkopien derjenigen Teile der Pläne zu liefern, die zum Zeitpunkt des Kaufs unleserlich sind. Darüber hinaus wird keinerlei Haftung übernommen. Der Erwerber dieser Pläne sollte beachten, daß in den entscheidenden Phasen des Baus und nach der Fertigstellung geeignete Tests durchzuführen sind und daß die üblichen Vorsichtsmaßnahmen zum Schutz des Lebens der Bauarbeiter zu treffen sind.

(Zitat: Robert L. Baber: Softwarereflexionen, Springer-Verlag)

und in der Praxis:

...
2. Haftung
Wir werden immer bemüht sein, ihnen einwandfreie Software zu liefern. Wir können aber keine Gewähr dafür übernehmen, daß die Software unterbrechungs- und fehlerfrei läuft und daß die in der Software enthaltenen Funktionen in allen von Ihnen gewählten Kombinationen ausführbar sind. Für die Erreichung eines bestimmten Verwendungszweckes können wir ebenfalls keine Gewähr übernehmen. Die Haftung für unmittelbare Schäden, mittelbare Schäden, Folgeschäden und Drittschäden ist, soweit gesetzlich zulässig, ausgeschlossen. Die Haftung bei grober Fahrlässigkeit und Vorsatz bleibt hiervon unberührt, in jedem Fall ist jedoch die Haftung beschränkt auf den Kaufpreis.

Hauptgegenstand dieser Veranstaltung ist die konstruktive Methode

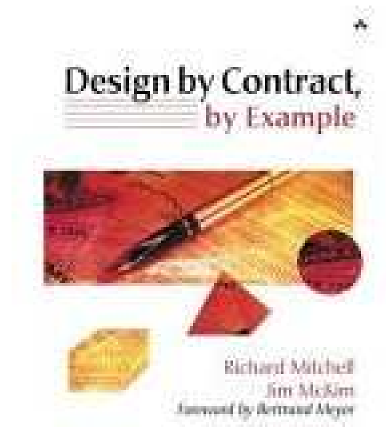


Abbildung 0.1: Design by Contract, by Example von Richard Mitchell und Jim McKim

zur Sicherung grundlegender Softwaregüte. In den ersten Kapiteln wird noch einmal Grundwissen zur Softwarequalität und -qualitätssicherung repetiert.

Beispiele für Softwaredisfunktionalitäten

Ein sahniger Brocken

(aus: Die Zeit vom 15.09.2005)

Begleitet von großem Werberummel hat die NASA den Kometen Tempel1 beschossen. Nun zeigen die Daten: Getroffen hat sie gut, gelernt hat sie wenig.

Auch wenn in den offiziellen Mitteilungen der NASA keine Rede davon ist - unter den versammelten Astronomen hat sich längst herumgesprochen, dass der Erfolg von *Deep Impact* nicht nur von aufgewirbeltem Feinstaub verdunkelt wurde. Ein Softwarefehler hat dazu geführt, dass die ersten - und besten - Bilder des Zusammenpralls im Datenspeicher des Begleitsateliten von späteren Aufnahmen überschrieben wurden.

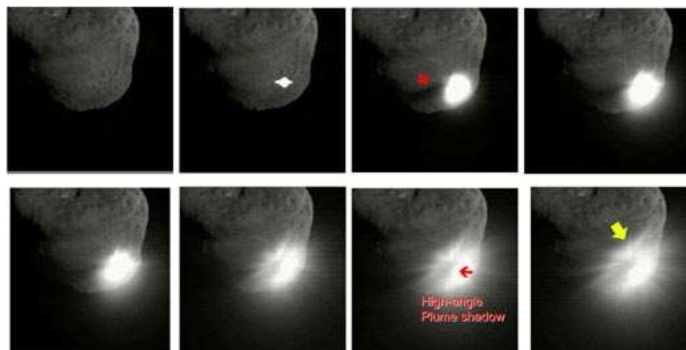


Abbildung 0.2: Bilder von Deep Impact

Der vollständige Artikel: http://www.zeit.de/2005/38/komet

USV-Software legt Server lahm

APC, Hersteller von unterbrechungsfreien Stromversorgungssystemen (USV), rät in einem Knowledgebase-Artikel dazu, alte Versionen der **PowerChute Business Edition-Software 6.X** umgehend durch die Version 7.X zu ersetzen.

Die Software zur Steuerung unterbrechungsfreier Stromversorgungen und zum sicheren Server-Shutdown hat Probleme mit einem auslaufenden Java-Runtime-Zertifikat. Dies führt dazu, dass die Windows-Server, auf denen die alte Version läuft, zum Teil mehrere Stunden für eine Ab- beziehungsweise Anmeldung benötigen. Die Dienste des Servers wie zum Beispiel Netzwerkfreigaben funktionieren allerdings trotz der Anmeldeprobleme weiterhin.

(aus http://www.heise.de/newsticker/meldung/62344)

Chaos an Hannovers Geldautomaten (05.10.2003 13:00 Uhr)

Computerprobleme haben am Samstag alle 240 Geldautomaten der Sparkasse in der Stadt und Region Hannover lahm gelegt. Die Fusion der Stadt- und Kreissparkasse sollte am Wochenende auch technisch umgesetzt werden, sagte der Sprecher des Geldinstituts, Stefan Becker. Beim Hochfahren eines Server habe sich ein Fehler eingeschlichen, so dass die Geldautomaten nicht mehr funktionierten. Die Sparkasse öffnete stadtdessen fünf Filialen, damit Kunden etwa in Einkaufszonen Bargeld abheben können.

(aus: http://www.heise.de/newsticker/meldung/40834)

THERAC 25

Selten sind solch schädliche Vorfälle so gut dokumentiert worden wie im Fall des „THERAC 25“, eines computergestützten Bestrahlungsgerätes. Dabei handelt es sich um ein Bestrahlungsgerät, welches in zwei „Modi“ arbeitet: im „X-Modus“ wird ein Elektronenstrahl von 25 Millionen Elektronen-Volt durch Beschuß einer Wolframscheibe in Röntgenstrahlen verwandelt; im „E-Modus“ werden die Elektronen selbst, allerdings „weicher“ mit erheblich reduzierter Energie als Korpuskelstrahlung erzeugt. Je nach therapeutischer Indikation wird die geeignete Strahlungsart eingestellt; in beiden Fällen kann der Bestrahlungsverlauf, nach Modus, Intensität und Bewegungskurve der Strahlungsquelle, mit einem Bildschirm-„Menü“ eingegeben werden.

Als mehrere Patienten berichteten, sie hätten bei Behandlungsbeginn das Gefühl gehabt, „ein heißer Strahl“ durchdringe sie, wurde dies vom Hersteller als „unmöglich“ zurückgewiesen. Erst nach dem Tod zweier Patienten sowie massiven Verbrennungen bei weiteren Personen kam heraus, daß neben dem X- sowie E-Modus mit niedriger Elektronenintensität infolge Programmierfehler ein unzulässiger dritter Zustand auftrat, nämlich direkt wirkende, 25 Millionen Elektronen-Volt „heiße“ Elektronen.

Dies geschah immer dann, wenn ein vorgegebenes „Behandlungsmenü“ mittels Curser-Taste modifiziert wurde. Um aufwendige Umprogrammierung zu vermeiden, wollte der kanadische Hersteller die Benutzung der Curser-Taste verbieten bzw. diese ausbauen und die Tastenlücke mit Klebeband abdichten lassen! Es ist zu befürchten, daß der Fall „THERAC 25“ kein Einzelfall ist. Zumeist ist es mangels entsprechender Vorsorge in computergesteuerten Medizingeräten schwerlich möglich, schädliches Systemverhalten später aufzuklären.

Berliner Magnetbahn

Computer spielen in allen gesellschaftlichen Bereichen eine immer größere Rolle. Angesichts der von fehlerhafter Software ausgehenden Gefahr wird versucht, die Sicherheit von computergesteuerten Systemen so weit wie möglich zu garantieren.

Softwarefehler: Kleine Ursache, große Wirkung

Fünf - Null, tippt der Operator in die Tastatur und erwartet, daß die Magnetschwebebahn auf 50 Stundenkilometer beschleunigen würde. Doch nichts geschah. Wieder tippt er fünf - null und vergaß diesmal nicht die „Enter"-Taste zu betätigen, mit der die Daten erst in den Rechner abgeschickt werden. Die insgesammt eingegebene Tastenfolge „fünf - null - fünf - null" interpretiert der Rechner als Anweisung, auf unsinnige 5050 Stundenkilometer zu beschleunigen. Dies konnte die Bahn zwar nicht, aber immerhin wurde sie so schnell, daß sie nicht mehr rechzeitig vor der Station gebremst werden konnte. Es kam zum Crasch mit Personenschaden – so geschehen vor zwei Jahren bei einer Probefahrt der Berliner M-Bahn.

Vernünftigerweise hätte die den Computer steuernde Software die Fehlerhaftigkeit der Eingabe „5050" erkennen müssen. Schon dieses Beispiel mangelnder Software zeigt, von welcher Bedeutung das richtige Verhalten von Computerprogrammen sein kann. Nicht nur bei Astronauten, die mit softwaregesteuerten Raumfähren ins All starten, hängt heute Leben und Gesundheit von Software ab. Computerprogramme erfüllen mittlerweile in vielen Bereichen sicherheitsrelevante Aufgaben.

Elektronik-Fehler führt zu Überhitzung bei Volvo-PKW

Kaum ein KFZ-Hersteller, der nicht mit Elektronik, Software und Hightech-Ausstattung das Autofahren komfortabler und die Wartung in der Werkstatt einfacher machen will. Doch die Tücken der Technik lassen für manchen Kunden den PKW zum IT-Sicherheitsrisiko werden. Nachdem vor kurzem erst Softwarefehler bei Mercedes-Dieseln für Aufsehen sorgten, können nun Defekte in der elektronischen Steuerung der Motorkühlung bei Volvo-Personenwagen zur Überhitzung führen.

Der Fehler tritt bei den Modellen S60, S80, V70 und XC70 aus den Baujahren 2000 und 2001 auf, erklärte Volvo, einzelne Modelle aus dem Jahr 1999 seien ebenfalls betroffen. Die fehlerhaft arbeitende Elektronik hat Bosch an Volvo geliefert – wer für den Fehler, der vor allem bei langsamer Fahrt bei hohen Außentemperaturen zur Überhitzung führen kann, verantwortlich ist, steht laut Volvo noch nicht fest. Insgesamt 460.000 Fahrzeuge weltweit ruft der schwedische Hersteller daher in die Werkstätten zurück. Laut dpa erhalten in Deutschland rund 40.000 Besitzer eines Volvo-PKW eine Aufforderung zum Werkstattbesuch – der für die Halter zumindest kostenlos bleibt.

(aus: http://www.heise.de/newsticker/meldung/51019)

The Patriot Missile

The Patriot missile defense battery uses a 24 bit arithmetic which causes the representation of real time and velocities to incur roundoff errors; these errors became substantial when the patriot battery ran for 8 or more consecutive hours.

As part of the search and targeting procedure, the Patriot radar system computes a "Range Gate" that is used to track and attack the target. As the calculations of real time and velocities incur roundoff errors, the range gate shifts by substantial margins, especially after 8 or more hours of continous run.

The following data on the effect of extended run time on patriot operations from Appendix II of the report would be of interest to numerical analysists anywhere.

Hours	Real Time (seconds)	Calculated Time (seconds)	Inaccuracy (seconds)	Approximate Shift In Range Gate (meters)
0	0	0	0	0
1	3600	3599.9966	.0034	7
8	28800	28799.9725	.0275	55
20a	72000	71999.9313	.0687	137
48	172800	172799.8352	.1648	330
72	259200	259199.7528	.2472	494
100b	360000	359999.6667	.3333*	687

Tabelle 0.1: Divergence in the Range Gate of a PATRIOT MISSILE

a: continuous operation exceeding 20 hours-target outside range gate
b: Alpha battery [at Dhahran] ran continuously for about 100 hours
* corrected value [GAO report lists .3433]

On Februrary 21, 1991 the Partiot Project Office send a message to all patriot sites stating that very long run times "could cause a shift in the range gate, resulting in the target being offset". However the message did not specify "what constitutes very long run times". According to the Army officials, they presumed that the users would not run the batteries for such extended periods of time that the Patriot would fail to track targets. "Therefore, they did not think that more detailed guidance was required".

The air fields and seaports of Dhahran were protected by six Patriot batteries. Alpha battery was to protect the Dhahran air base.
On February 25, 1991, Alpha battery had been in operation for over 100 consecutive hours. That was the day an incomming Scud struck an Army barracks and killed 28 American soldiers.

On February 26, the next day, the modified software, which compensated for the inaccurated time calculation, arrived in Dhahran.

Kontenabrufverfahren startet wegen Softwareproblemen als Provisorium

Das automatische Kontenabrufverfahren nach dem „Gesetz zur Förderung der Steuerehrlichkeit“, das ab dem 1. April die Abfrage der Kontostammdaten für einige Behörden möglich macht, startet mit Anlaufproblemen. Sie liegen vor allem darin begründet, dass die entsprechende Abfragesoftware der Stammdaten, die ab November 2003 zum Zwecke der Terroristenfahndung entwickelt wurde, nicht richtig skaliert. Diese Software wurde auf ca. 2000 Abfragen pro Tag durch die Polizeifahnder ausgelegt. Mit mehr als täglichen 50.000 Abfragen, die von Finanzämtern, Bafög- oder Sozialämtern ab dem 1. April erwartet werden, ist die Software hoffnungslos überfordert. Für die 18 bis 20 Millionen Konten, die jährlich nach dem Willen des Gesetzgebers gesucht werden sollen, wird derzeit eine völlig neue Schnittstellenspezifikation entwickelt und ein komplett neues Programm geschrieben. Bis dieses Programm für die automatische Abfrage durch die Sachbearbeiter fertig ist, muss die Abfrage wie bisher manuell erfolgen.

Bei dieser manuellen Abfrage reichen Polizeibehörden und Strafverfolger ihre Anfragen auf Papier oder per Fax oder E-Mail bei der Bundesanstalt für Finanzdienstleistungsaufsicht (BaFin) ein und bekommen die gewünschten Kontodaten auf demselben Wege zurück. Dieses Verfahren soll durch eine Suchmaske ersetzt werden, die jede Behörde aufrufen kann – wenn die dahinter liegende Abfragesoftware die Datenmengen bewältigen kann.

(aus: http://www.heise.de/newsticker/meldung/58096)

Buffer Overflow im Linux-Kernel

Paul Starzetz von isec hat Details zu einer neuen Lücke im Linux-Kernel veröffentlicht, mit der ein Angreifer Programme mit Root-Rechten ausführen kann. Anders als bei vergangenen Veröffentlichungen von Starzetz, wurden die Hersteller aber offenbar nicht vorab informiert, etwa über die geschlossene Mailing-Liste Vendor-Sec. Nach seinen Angaben würde die Linux-Community Veröffentlichungen ohne Embargos von Distributoren bevorzugen. Um aber die Regeln der so genannten Responsible Disclosure einzuhalten, veröffentlicht er diesmal keinen Exploit-Code.

Der Fehler findet sich wieder einmal im Linux ELF-Binary-Loader, in dem Starzetz in der Vergangenheit bereits mehrere Lücken aufdeckte. Diesmal ist ein Buffer Overflow in der Funktion elf_core_dump schuld, der beim Aufruf einer weiteren Funktion (copy_from_user) mit einer negativen Längenangabe auftritt. Starzetz hat nach eigenen Angaben die Lücke bereits durch ein präpariertes

ELF-Binary demonstrieren können, das mit Kernel-Privilegien lief. Ein Proof-of-Concept-Programm ist seinem Advisory beigefügt, das aber nur den Kern des Problems demonstriert.

(aus:http://www.heise.de/newsticker/meldung/59498)

Auch Superhirne können irren - das Risiko Computer

Lenkwaffen, Flugsteuerungen, Diagnosegeräte, Verkehrsleitsysteme, Dateien, Produktions-Steuerung – überall hat der Computer das Kommando übernommen. Doch nicht überall gibt er die richtigen Befehle. Mancher Irrtum schon hatte tödliche Folgen. Das Vertrauen in das elektronische Superhirn ist angeschlagen.

Sollten US-Kriegsschiffe, die mit dem computergestützten Waffensystem „Aegis“ ausgerüstet sind, in Zukunft wieder in Spannungsgebieten kreuzen, werden die verantwortlichen Offiziere dort mit der Angst leben, daß sich die Ereignisse des 3. Juli 1988 wiederholen könnten: Damals folgte der Kapitän des Kreuzers „Vincennes“, von elektronischen Befehlen unter Entscheidungsdruck gesetzt, der Logik des Computers, dessen Abtastsystem ein Verkehrsflugzeug mit einer Kampfmaschine verwechselte. Er gab den verhängnisvollen Befehl zum Abfeuern der Raketen. Alle 290 Insassen des iranischen Airbus kamen dabei ums Leben. ...

Aus anderer Quelle:
Auch der erste KI-Unfall, bei dem das „künstlich intelligente“ AEGIS-System des US-Kreuzers „Vincennes“ im Sommer 1988 einen zivilen Airbus mit einem MIG-Militärjet verwechselte, durfte bei heutigem Kenntnisstand durch einen Konzeptfehler mitverursacht worden sein. Aus der „Sicht“ des einzelnen AEGIS-Systems werden alle Signale, die auf einem Richtstrahl innerhalb einer 300 Meilen umfassenden Überwachungszone entdeckt werden, einem einzelnen Objekt zugeordnet. So können ein Militar- und ein Zivil-Jet nur durch ein räumlich getrenntes System unterschieden werden. Offenbar hat das AEGIS-System aber weder Inkonsistenzen der Daten (militärische und zivile Transponder-Kennung) noch die unvollständige räumliche Auflösung dem verantwortlichen Kommandeur übermittelt, der im Vertrauen auf die Datenqualität den Befehl zum Abschuß von fast 300 Zivilisten gab. Offensichtlich ist in Streßsituationen eine menschliche Plausibilitätskontrolle nicht nur bei derart komplexen Systemen erschwert. Aus einem bis dahin fehlerfreien Funktionieren wird induktiv auf korrektes Verhalten im Ernstfall geschlossen. Daher sind besondere Hinweise auf inkonsistente und unvollständige „Datenlagen“ und gegebenenfalls Sperren gegen automatische Prozeduren zwingend erforderlich.

Explosion der Ariane 5

http://www.ima.umn.edu/ arnold/disasters/ariane5rep.html

Neueste Risikoinformationen/Softwareprobleme

... findet man unter: http://catless.ncl.ac.uk/Risks:

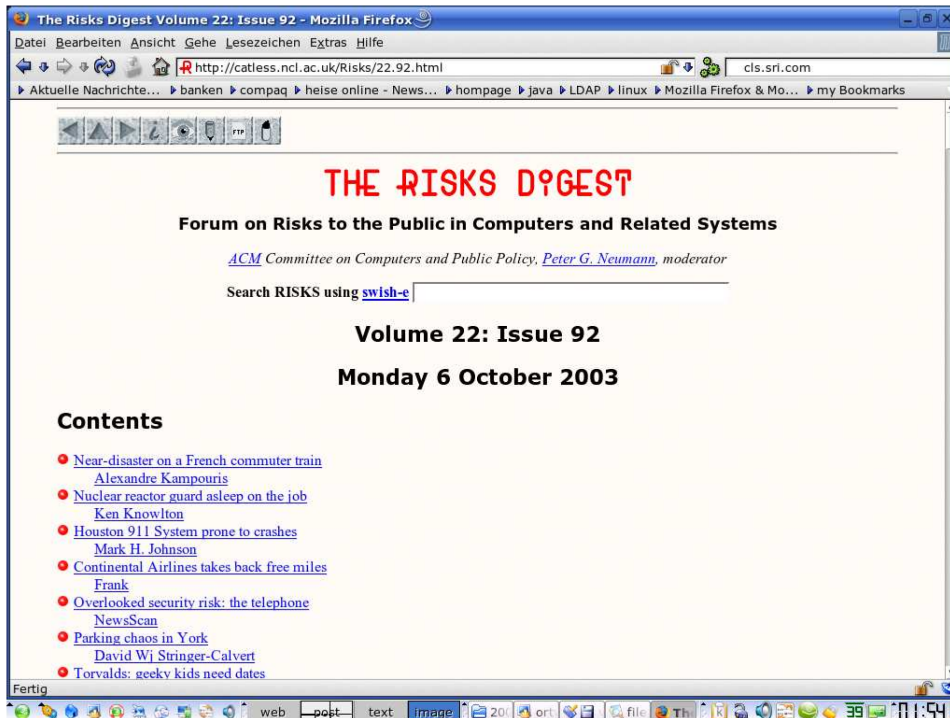


Abbildung 0.3: http://catless.ncl.ac.uk/Risks/22.92.html

Programming by Contract im Umfeld der Formalen Methoden

Programmverifikation

Erläuterung

Programm-Verifikation ist ein systematischer Ansatz zum Nachweis der Fehlerfreiheit von Programmen. Dabei wird bewiesen, dass ein vorgegebenes Programm bestimmte wünschenswerte Eigenschaften besitzt. Bei sequentiellen Programmen geht es vor allem um die Ablieferung korrekter Ergebnisse und die Terminierung. Bei Programmen mit parallel ablaufenden Komponenten sind zusätzliche Eigenschaften wie Interferenz-Freiheit, Deadlock-Freiheit und faires Ablaufverhalten wichtig.
(*vergleiche* http://www.software-kompetenz.de/servlet/is/22224/)

Beschreibung

Die Voraussetzung für die Programm-Verifikation ist eine Spezifikation der Vor- und Nachbedingungen (pre- und post conditions) des Programms bzw. von Teilen davon. Solche Bedingungen stellen logische Aussagen dar, die bei jedem möglichen Programmdurchlauf an den betreffenden Stellen den Wahrheitswert *True* liefern müssen. Aus den Vor- und Nachbedingungen und dem vorliegenden Quellcode lassen sich mit Hilfe eines Verifikationswerkzeugs sog. Verifikationsbedingungen (verification conditions) ableiten, von deren Erfüllung die Korrektheit des Programms abhängt.
Die Verifikation lässt sich aus Aufwandsgründen zumeist nicht *flächendeckend* auf mittlere bis große Programme anwenden, dennoch kann die Sicherheit eines Programms beträchtlich erhöht werden, wenn Kernalgorithmen formal verifiziert werden. ... (*Zitat aus*: http://www.software-kompetenz.de/?22225)

Ein klassisches Beispiel:

begin $\{a > 0, b \geq 0\}$
    $x := a; y := b;$
    while $y \neq 0$ **do** {gcd(a,b) = gcd(x,y) }
    begin $r := x$ **mod** y;
        $x := y;$
        $y := r$
    end

$\{x = gcd(a, b)\}$
end

(*entnommen:* Suad Alagić/Michael A. Arbib: THE DESIGN OF WELL-STRUCTURED AND CORRECT PROGRAMS, Springer-Verlag, New York, 1978)

Programming by Contract

- **Invarianten** einer Komponente sind allgemeine, unveränderliche Konsistenzbedingungen an den Zustand einer Komponente, die vor und nach jedem Aufruf eines Dienstes gelten. Formal sind Invarianten boolsche Ausdrücke über den Abfragen der Komponente; inhaltlich können sie z.B. Geschäftsregeln (business rules) ausdrücken.

- **Vorbedingungen** (preconditions) eines Dienstes sind Bedingungen, die vor dem Aufruf eines Dienstes erfüllt sein müssen, damit er ausführbar ist. Vorbedingungen sind boolsche Ausdrücke über den Abfragen der Komponente und den Parametern des Dienstes.

- **Nachbedingungen**(postconditions) eines Dienstes sind Bedingungen, die nach dem Aufruf eines Dienstes erfüllt sind; sie beschreiben, welches Ergebnis ein Dienstaufruf liefert oder welchen Effekt er erzielt. Nachbedingungen sind boolsche Ausdrücke über den Abfragen der Komponente und den Parametern des Dienstes, erweitert um ein Gedächniskonstrukt, das die Werte von Ausdrücken vor dem Dienstaufruf liefert.

(*vergleiche:* http://userserv.fh-reutlingen.de/~hug/artikel/ForumWI01%20SdV.pdf)

Ein Beispiel in C++ mit Hilfe von Nana:

```
#define EIFFEL_CHECK CHECK_ALL
#include <set>
#include <vector>
#include <eiffel.h>
#include <nana.h>
...
void quicksort(double v[], int l, int h)
{
  REQUIRE(l <= h+1);
  ...
  ENSURE(A(int k=l, k<h, k++, v[k]<=v[k+1]));
};
```

```
void quicksort(double v[], int n)
{
  REQUIRE(n>=1);
  ID(multiset<double> v_old_contents(&v[0],&v[n]););
  ...
  ENSURE(A(int k=0, k<n-1, k++, v[k]<=v[k+1]));
  ID(multiset<double> v_contents(&v[0],&v[n]));
  ENSURE(v_old_contents == v_contents);
};
```

```
class name_list{
     ...
void name_list::put(const string& a_name)    // Push a_name into list
DO
  REQUIRE(/* name not in list */    !has(a_name));
  ID(set<string> contents_old(begin(),end()));
  ID(int count_old = get_count());
  ID(bool not_in_list = !has(a_name));
  ...
  ENSURE(has(a_name));
  ENSURE( (!not_in_list) || (get_count() == count_old + 1));
  ID(set<string> contents(begin(),end()));
  ENSURE( (!not_in_list) || (contents == contents_old + a_name));
END;
...
}
```

1 Qualitätsanforderungen an SW-Produkte

A. Produktorientiert:

1. funktionale Korrektheit
2. funktionale Vollständigkeit
3. Robustheit gegenüber dem Benutzer
4. Benutzerfreundlichkeit
5. Effizienz in Laufzeit
6. Effizienz im Arbeitsspeicherbedarf
7. Effizienz im Plattenspeicherbedarf
8. Integrität (gegenüber unauthorisierten Änderungen)
9. Kompatibilität, Integrationsfähigkeit, Standards

B. Projektorientiert:

1. Überprüfbarkeit
2. Verständlichkeit
3. Wartbarkeit
4. Änderbarkeit, Erweiterbarkeit
5. Portierbarkeit
6. Wiederverwertbarkeit

1.1 Prinzipien der ordnungsgemäßen Programmerstellung

1. Konstruktive Voraussicht und methodische Restriktion
2. Strukturierung
3. Modularisierung
4. Lokalität
5. Integrierte Dokumentation
6. Standardisierung
7. Funktionale und informelle Bindung
8. Schmale Datenkopplung
9. Vollständige Schnittstellenspezifikation
10. Lineare Kontrollstrukturen
11. Verbalisierung

1.2 Spezifikation einer abstrakten Datenkapsel

1.2.1 Axiomatische Spezifikation

TYPES

  STACK[X]

FUNCTIONS

empty:	STACK[X]	$\nrightarrow$ BOOLEAN
new:		$\rightarrow$ STACK[X]
push:	X x STACK[X]	$\nrightarrow$ STACK[X]
pop:	STACK[X]	$\nrightarrow$ STACK[X]
top:	STACK[X]	$\nrightarrow$ X

PRECONDITIONS

  pre pop (s: STACK[X]) = (not empty(s))
  pre top (s: STACK[X]) = (not empty(s))

AXIOMS

  for all x:X, S : STACK[X]:
  empty(new())
  not empty (push(x,S))
  top (push(x,S))=x
  pop (push(x,S))=S

Vollständigkeit + Widerspruchsfreiheit (+ Unabhängigkeit)

1.2.2 Beschreibende (denotationale) Spezifikation

$Queue = Qelem^*$
$q_0 = [\ ]$

ENQUEUE ($e : Qelem$)
ext wr $q : Queue$
post $q = \overleftarrow{q} \frown [e]$

DEQUEUE() $e : Qelem$
ext wr $q : Queue$
pre $q \neq [\ ]$
post $\overleftarrow{q} = [e] \frown q$

ISEMPTY() $r : \mathbb{B}$
ext rd $q : Queue$
post $r \Leftrightarrow (len\ q = 0)$

„mathematische“ Modellierung mit Hilfe von Folgen, Mengen, ...

1.3 Prinzipien der Moduarisierung:

1. Module sollten **syntaktischen Einheiten** der Programmiersprache entsprechen.

2. Module sollten **mit möglichst wenigen anderen Modulen „kommunizieren“.**

3. „Kommunizierende“ Module sollten so **wenig** wie möglich **Informationen (Daten) austauschen**.

4. Jeder **Datentausch** zweier „kommunizierender“ Module muß **offensichtlich** in der Modulspezifikation (und nicht indirekt) kenntlich gemacht werden.

5. Alle **Daten** eines Moduls sollten **nur diesem bekannt** sein (außer im Falle einer gezielten Exportierung an möglichst wenige Nachbarmodule).

6. Ein Modul sollte **abgeschlossen und offen** sein.

1.4 Typen der Modularisierung

1. modulare **Zerlegbarkeit** (z.B. Top-Down-Design)

2. modulare **Zusammenfügbarkeit** (z.B. UNIX-Filter)

3. modulare **Verständlichkeit** (d.h. jede Modulbeschreibung selbsterklärend)

4. modulare **„Stetigkeit“**

   Kleine Spezifikationsänderungen wirken sich nur in **wenigen** Modulen aus. (Z.B. dyn. Felder, symbolische Konstanten, ...)

5. modularer **„Schutz“**

   Fehler/Ausnahmebedingungen bleiben in ihrer Auswirkung auf nur **wenige** Module beschränkt. (Z.B. direkte Konsistenzüberprüfung von Tastatureingaben, ...)

2 Wiederverwendbarkeit

Vermeide es, das Rad immer wieder neu zu erfinden!

1. **Algorithmen (Programme)** lösen i. allg. eine Klasse von Problemen, die durch Eingabewerte parametrisiert sind.
2. **Unterprogramme** (Funktionen, Prozeduren, Operatoren) lösen eine Klasse von Problemen: Gemäß dem Prinzip der methodischen Restriktion sind dabei die einzelnen Parameter jeweils Werte des Wertebereiches eines festen Typs.
3. **Unterprogramme mit konformen Feldparametern** (in Pascal bzw. open-array-Parameter in Modula2) erlauben es Parametern, einer Klasse von Feldern anzugehören (variable Dimension);

   ```
   PROCEDURE EuklNorm (v:ARRAY OF REAL): REAL;
   ```

4. **Dynamische Felder / Teilfeld-Selektoren** erlauben einen in der Dimension noch nicht festgelegten Feldtyp:

   ```
   TYPE vector = ARRAY[*] OF REAL;
   a := t[*,2];
   ... t[min, k:l] ...
   ```

5. **Polymorphismen** (d.h. Überladen) **von Funktionen/Operatoren** erlauben die Benutzung einer mit demselben Namen versehenen Klasse von Funktionen, in denen jeder Parameter aus einer (disjunkten) Vereinigung von Typen stammen darf:

   $writeln(x : real);$ $k := i * j;$
   $writeln(i : integer);$ $z := x * y;$
   $\dots$

6. **Unterprogramme als Parameter** anderer Unterprogramme erlauben Algorithmen für eine Klasse von Unterprogrammen gleicher Signatur:

```
function Bisection (function f(x : real) : real;
                    xLeft, xRight        : real;
                    success              : boolean
                    ) : real;
```

7. **Generizität** ermöglicht Parametrisierung nach Typen:

```
generic
      type T is private;
procedure swap (x, y : in out T) is t : T
begin
      t := x; x := y; y := t
end swap
      ⋮
procedure int_swap is new swap (INTEGER);
```

Eingeschränkte Generizität schränkt die aktuellen Typ-Parameter ein:

```
generic
    type T is private;
    with funktion "<=" (a, b : T) return BOOLEAN is <>;
 funktion minimum (x, y : T) return T is
 begin
    if x <= y then return x;
    else           return y
    end if
 end minimum
```

(Ähnliches kann durch Textproprozessoren oder die typunsichere Benutzung des typungebundenen Zeigers `ADDRESS` erreicht werden.)

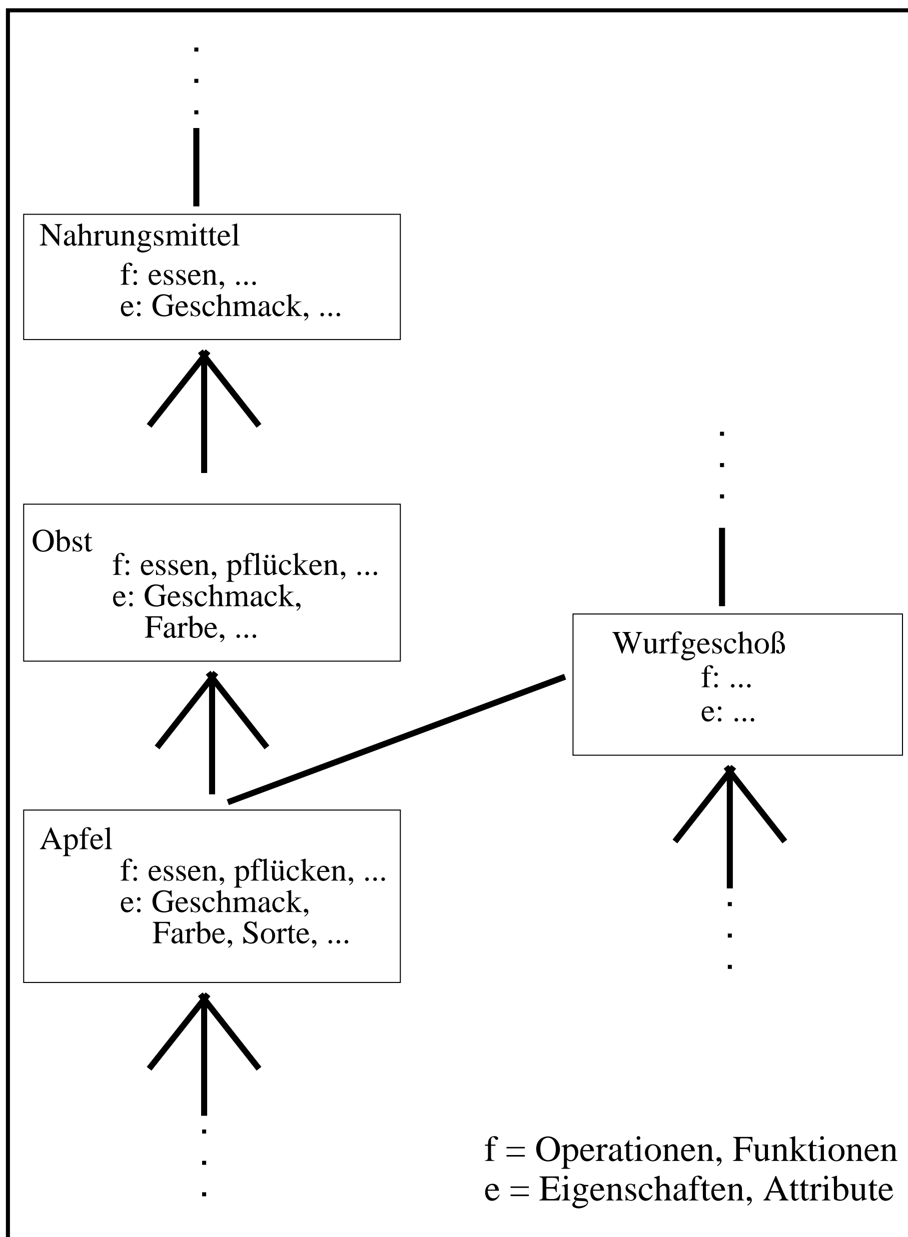


Abbildung 2.1: Begriffshierarchien

2.2 Objekthierarchien als strukturierte Modulsammlungen: Beispiele aus Eiffel

Objektorientierte Programmiersprachen ermöglichen die **Datenkapselung** und eine **evolutionäre** Programmerstellung:

> Nutze vorhandene Objektklassen (Typen) oder erzeuge neue Objektklassen, wobei **bei Teilstrukturgleichheit** möglichst viel durch **Vererbung** existierender Klassen realisiert wird.

2.2.1 Vererbung und Erweiterung

Namensänderung

Neues *feature*

```
class Multiindex inherit
 ARRAY[CARDINAL] rename
                   count as Dimension,
                   clear_all as Null
                  end;
feature
   abs: CARDINAL is
     require not empty
     do ...
     ensure
          - - abs = For all i:lower..upper:
                    SUM item(i)
     end - - abs
end - - class Multiindex
```

2.2.2 Vererbung und Abänderung

	class Multiindex **inherit** ARRAY[CARDINAL]**rename** count **as** Dimension, clear_all **as** Null
alternative Implementierung ⟶	**redefine** abs, put, make, **end**;
alternative Implementierung ⟶	**feature** abs: CARDINAL;
auch die Vorbedingung wird geerbt ⟶	put(v:**like** item; i:INTEGER) - - replace i-th entry, if in index interval, by v ⋮
zusätzliche Nachbedingung ⟶	**ensure then** abs = **old** abs - **old** item(i) + v **end** - - put ⋮
alte Invariante wird geerbt ⟶	**invariant** - - abs = For all i:lower..upper:Sum item(i) end - - class Multiindex

Wenn aus Effektivitätsgründen **redundante Daten** angelegt werden, so sollten diese Redundanzen spezifiziert werden!

Es gelten folgende Regeln bei der Vererbung (von is-a-Methoden):

a) Vorbedingungen können in einer Kindklasse abgeschwächt werden.

b) Nachbedingungen in einer Kindklasse müssen stärker sein als diejenigen der Elterklasse.

c) Invarianten in der Kindklasse müssen ebenfalls stärker als in der Elterklasse sein.

Dann ist ein echtes *Subcontracting* realisiert.

Bemerkung: Es reicht die Kindnachbedingung im Falle des Eintreffens der Eltervorbedingung stärker als die Elternachbedingung zu realisieren. Im Falle „Kindvorbedingung **and not** Eltervorbedingung“ darf die Kindnachbedingung frei gewählt werden.

2.2.3 Generizität

```
class STACK[T]
feature
    ⋮
end - - class STACK[T]
```

2.2.4 Eingeschränkte Generizität

```
class VECTOR[T -> ADDABLE]
feature
    ⋮
end - - class VECTOR
```

2.2.5 Polymorphie und „late binding“

```
class Rectangle inherit
      POLYGON redefine perimeter
                    end
   feature {NONE}
        side1: REAL;
        side2: REAL;
   feature {ANY}
        perimeter: REAL is
        do
            Result := 2 * (side1 + side2)
        end - - perimeter
        ⋮
end - - class Rectangle
```

Wegen des Zusammenhangs Rectangle $\subset$ POLYGON und MethodenVon(Rectangle) $\supset$ MethodenVon(POLYGON) gilt in der Anwendung:

perimeter für die Menge aller Erben von POLYGON verfügbar unter gleichem Namen.

```
⋮
p : POLYGON;
r : Rectangle;
⋮
!!p; !!r;
⋮
print (p.perimeter);        - - perimeter aus
⋮                           - - POLYGON
p := r
print (p.perimeter);        - - perimeter aus
⋮                           - - Rectangle
```

2.2.6 Aufgeschobene Feature-Implementierungen

```
deferred class Stack[T]
feature
        nb_elements : INTEGER is
           defered
        end - - nb_elements
        empty : BOOLEAN is
        do
           Result := (nb_elements = 0)
        ensure Result = (nb_elements = 0)
        end - - empty
      ⋮
end - - class STACK[T]
```

... dienen der partiellen Implementierung einer Gruppe möglicher Implementierungen (Schablone). Sie stehen somit in Konkurrenz und ergänzen generische Klassen.

– ”Objektorientiertes” Programmieren (als Alternative zum funktionalen Top-Down-Entwurf und zum datengesteuerten Entwurf nach Jackson) ist die Softwarekonstruktion mit Hilfe der **Adaption** von Sammlungen abstrakter Datentyp-Implementierungen.

– Unterklassen können sich von ihren Basisklassen unterscheiden durch:
  1) mehr Operationen
  2) mehr Daten (Attribute)
  3) eingeschränkte Wertebereiche der Daten
  4) alternative Implementierungen

2.3 Resümee

1. Objektorientiertes Programmieren setzt eine gute Kenntnis der vorhandenen Klassenhierarchien voraus! Diese sind heute jedoch häufig nicht ausreichend dokumentiert (fehlende Spezifikation, fehlende Fixierung der Design-Ideen, ...). Häufig steht nur ein Browser zur Betrachtung der Quellen der Klassen zur Verfügung, und der Programmierer muß sich selbst den Durchblick durch die Konzeption der Klassenbibliotheken erkämpfen.

2. Einige **objektorientierte** Sprachen bieten gar keine mitgelieferten Klassenbibliotheken an. Andere haben sprachspezifisch bzw. sogar herstellerspezifisch eigene — zwar häufig an Smalltalk angelehnte, aber dennoch in wichtigen Details abweichende — Klassenhierarchien. Für viele Gebiete in der Informatik/Mathematik/Anwendungswissenschaft fehlen geeignete Klassenbibliotheken gänzlich.

3. Geordnete **evolutionäre** objektorientierte Entwicklung im Team erfordert ein richtiges Management (open-close-Phasen, Versions-Management, ...)

4. Objektorientierte Programmiersprachen sollten syntaktische Sprachmittel für **Zusicherungen** besitzen (mindestens Aussagenlogik, besser **Prädikatenlogik**). Diese sollten **in** den geforderten **Klassenhierarchien** (zumindest in Kommentarform) **intensiv genutzt** werden. Eine etwa VDM ähnliche Syntax wäre gewiß interessant.

Wir müssen anspruchsvoller werden im Hinblick auf die Verläßlichkeit und die Qualität unserer Software. Die Benutzer müssen kritischer werden und weniger bereit, Softwareerzeugnisse geringer Qualität zu akzieptieren.

(Zitat: Robert L. Baber: Softwarereflexionen, Springer-Verlag)

Forschungsministerium fördert Standard für IT-Sicherheit

Trotz des flächendeckenden Einsatzes von Computersystemen in sicherheitsrelevanten Bereichen fehlt bislang eine standardisierte Methode, die das fehlerfreie Funktionieren solcher Systeme garantiert. Das Bundesministerium für Bildung und Forschung (BMBF) will nun Arbeiten fördern, bei denen mit Methoden der Verifikation der so genannte geschlossene integrierte Korrektheitsbeweis erbracht werden kann. Damit sollen sich Fehler bereits im Entwurf von autonomen oder integrierten Computersystemen erkennen und korrigieren lassen – eine sorgfältige Spezifikation vorausgesetzt. Alle möglichen Fehlersituationen könnten aber nur dann abgefangen werden, wenn bereits in der Planung die entsprechenden Einsatzszenarien definiert wurden, betonte Projektleiter Prof. Dr. Wolfgang Paul gegenüber heise Security.

Für die erste zweijährige Forschungsphase werde das BMBF 7,2 Millionen Euro zur Verfügung stellen, teilte das Ministerium am heutigen Mittwoch in Berlin mit. An dem Projekt beteiligen sich neben der Universität Saarland unter anderen auch die TUs Darmstadt, Karlsruhe, München sowie Infineon, T-Systems und BMW.

Die Entwicklung eines integrierten Korrektheitsbeweises gilt zurzeit als eine der größten Herausforderungen der Informatik. Er soll die Funktionen bei der Entwicklung von Hard- und Systemsoftware bis zur Netzwerk- und Anwendungsebene laufend überprüfen. Zunächst sollen die mathematischen Grundlagen entwickelt, vollständig formalisiert und für Informatikanwendungen in den Bereichen Embedded Systems, Kommunikation und Anwendungssoftware erschlossen werden. Darauf aufbauend sollen die Projektpartner Demonstratoren entwickeln und mit ihnen Computersysteme für Chipkarten, Telekommunikation und Automobilelektronik von der Hardware bis zur Anwendungssoftware überprüfen. Im Rahmen des Projektes werden auch Softwaretools entwickelt, die den Verifikationsprozess unterstützen. (dab/c't)

Link: http://www.heise.de/newsticker/data/dab-01.10.03-002/

Siehe auch (Thema Produkthaftung):
http://www.heise.de/newsticker/result.xhtml?url=/newsticker/meldung/86839

3 Programming by Contract

3.1 Spezifikation durch Verträge

(**SdV**, *Design by Contract* [1], *Programming by Contract*) ist eine Methode zur Spezifikation der dynamischen Semantik von Softwarekomponenten mit Hilfe von Verträgen aus erweiterten boolschen Ausdrücken. **SdV** basiert auf der Theorie der abstrakten Datentypen und formalen Spezifikationsmethoden. Spezifizierte Komponenten können Module, Klassen oder Komponenten im Sinne von Komponententechnologien (wie Microsofts COM, .NET oder Suns EJB) sein. Verträge ergänzen das Kunden-Lieferanten-Modell:

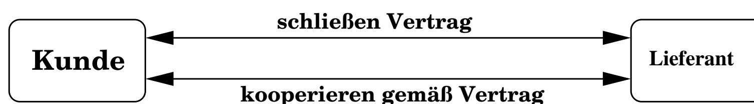


Abbildung 3.1: Kunden-Lieferanten-Modell

Grundlegend für die Vertragsmethode ist das **Prinzip der Trennung von Diensten in Abfragen und Aktionen** (*command-query separation)*:

- **Abfragen** geben Auskunft über den Zustand einer Komponente, verändern ihn aber nicht. Sie liefern als Ergebnis einen Wert. Die Abfragen einer Komponente beschreiben ihren abstrakten Zustand.
- **Aktionen** verändern den Zustand einer Komponente, liefern aber kein Ergebnis. Die Aktionen einer Komponente bewirken ihre Zustandsveränderungen.

Diesem Prinzip folgend sind seiteneffektbehaftete Funktionen als Dienste zu vermeiden[2].

[1] "Design by Contrakt"ist ein Warenzeichen von Interactive Software Engeneering.

[2] In bestimmten Fällen, z.B. bei Fabrikfunktionen, können Seiteneffekte sinnvoll sein. Solche Funktionen sind nicht als Spezifikatoren verwendbar und sollten entsprechend gekennzeichnet sein.

Ein Grund dafür ist, dass Abfragen als **Spezifikatoren** dienen, d.h. als Elemente von Verträgen. **Verträge** setzen sich aus Bedingungen folgender Art zusammen:

- **Invarianten** einer Komponente sind allgemeine unveränderliche Konsistenzbedingungen an den Zustand einer Komponente, die vor und nach jedem Aufruf eines Dienstes gelten. Formal sind Invarianten boolsche Ausdrücke über den Abfragen der Komponente; inhaltlich können sie z.B. Geschäftsregeln (busisness rules) ausdrücken.

- **Vorbedingungen** (preconditions) eines Dienstes sind Bedingungen, die vor dem Aufruf eines Dienstes erfüllt sein müssen, damit er ausführbar ist. Invarianten sind boolsche Ausdrücke über den Abfragen der Komponente und den Parametern des Dienstes.

- **Nachbedingungen** (postconditions) eines Dienstes sind Bedingungen, die nach dem Aufruf eines Dienstes erfüllt sind; sie beschreiben, welches Ergebnis ein Dienstaufruf liefert oder welchen Effekt er erzielt. Nachbedingungen sind boolsche Ausdrücke über den Abfragen der Komponente und den Parametern des Dienstes, erweitert um ein Gedächniskonstruckt, das die Werte von Ausdrücken vor dem Dienstaufruf liefert.

Verträge legen Pflichten und Nutzen für Kunden und Lieferanten fest. Die Verantwortlichkeiten sind klar verteilt:
Der Lieferant garantiert die Nachbedingung jedes Dienstes, den der Kunde aufruft, falls der Kunde die Vorbedingung erfüllt. Eine verletzte Vorbedingung ist ein Fehler des Kunden, eine verletzte Nachbedingung oder Invariante (bei erfüllter Vorbedingung) ist ein Fehler des Lieferanten.

	KUNDE	LIEFERANT
PFLICHT	Die Vorbedingung einhalten.	Anweisungen ausführen, die die Nachbedingungen herstellen und die Invarianten erhalten
NUTZEN	Ergebnisse/Wirkungen nicht prüfen, da sie durch die Nachbedingungen garantiert sind.	Aufrufe, die die Vorbedingung verletzen, ignorieren. (Die Vorbedingungen nicht prüfen.)

Tabelle 3.1: Pflichten - Nutzen von Kunden und Lieferanten

Schwache Vorbedingungen erleichtern den Kunden die Arbeit, starke Vorbedingungen dem Lieferanten. Je schwächer die Nachbedingungen sind, umso freier ist der Lieferant und umso ungewisser sind die Kunden über das Ergebnis/den Effekt. Je stärker die Nachbedingungen sind, umso mehr muß der Lieferant leisten.

Siehe auch:
Spezifikation durch Vertrag — eine Basistechnologie für eBusiness

3.2 Invarianten

```
INTERFACE Natural

    QUERIES
       N:INTEGER
    INVARIANTS
       N > 0
    ACTIONS
       Set (IN newN:INTEGER)
          PRE
             newN > 0
          POST
             N = newN
END Natural
```

Cleo unterscheidet Abfragen und Aktionen durch die QUERIES- und ACTIONS- Abschnitte. Die Invariante N>0 im INVARIANTS- Abschnitt schränkt den Wertebereich der Abfrage N ein. Das **Prinzip des einheitlichen Zugriffs** (uniform access) verlangt, von der Implementierung parameterlose Abfragen zu abstrahieren: N ist als Attribut oder parameterlose Funktion implementierbar, der Zugriff auf N wird davon unabhängig notiert.
Die Aktion Set (Konstruktor) bewirkt, dass N nach einem Aufruf von Set den als Parameter newN übergebenen Wert liefert. Die PRE- und POST- Abschnitte für die Vor- und Nachbedingungen spezifizieren diese Semantik. Da Set die Invariante nicht verletzen darf, schränkt es durch die Vorbedingung den akzeptablen Wertebereich von newN ein.

3.3 Nachbedingungen

Als zweites Beispiel dient eine parametrisierte Abfrage. Die Fakultätsfunktion Factorial könnte Teil einer Komponente mit mathematischen Funktionen sein:

```
QUERIES
    Factorial (IN n : INTEGER) : INTEGER
       PRE
          N >= 0
       POST
          (n>=0) IMPLIES (result=1)
          (n>1) IMPLIES (result=n*Factorial (n - 1))
```

Zum Formulieren der Nachbedingung braucht man einen Namen für das Ergebnis des Funktionsaufrufs (result).

3.4 Vorbedingungen

Das dritte Beispiel modelliert eine (mathematische) Menge als generische Komponente Set: Element ist der generische Parameter, der als Elementtyp fungiert.

```
INTERFCE Set[Element]

    QUERIES
        Count:INTEGER
            - - Number of elements in the set.

        Has(IN x:Element):BOOLEAN
            - - Does the set contain x?
            POST
                result IMPLIES(Count>0)

        IsEmpty:BOOLEAN
            - - Does the set contain no element?

    INVARIANTS
        Count>=0
        IsEmpty=(Count=0)
    ACTIONS
        Put(IN x:Element)
            - - include x into the set.
            POST
              Has(x)
              OLD (Has(x)) IMPLIES (Count=OLD(Count))
              NOT OLD (Has(x)) IMPLIES (Count=OLD(Count)+1)

      Remove(IN x:Element)
            - - Exclude x from the set.
            POST
              NOT Has(x)
              OLD (Has(x)) IMPLIES (Count=OLD(Count)-1)
              NOT OLD (Has(x)) IMPLIES (Count=OLD(Count))
            WipeOut
              - - Exclude all elements from the set.
              POST
                Count=0
    END Set
```

Um das Verhalten der Aktionen **Put** und **Remove**, des Hinzufügens eines Elementes zur Menge und des Entfernens eines Elementes aus der Menge, zu spezifizieren, ist der

Zustand der Menge vor einem Aktionsaufruf mit ihrem Zustand nach dem Aktionsaufruf zu vergleichen. Das Vorzustandskonstruckt ermöglicht dies: Mit OLD (...) geklammerte Ausdrücke liefern den Wert des geklammerten Ausdrucks vor einem Dienstaufruf. OLD-Ausdrücke dürfen nur in Nachbedingungen auftreten. der Ausdruck

Count=OLD(Count)+1

bedeutet, das sich der Wert von Count durch die Ausführung des Dienstes um 1 erhöht. Dabei muss es sich um eine Aktion handeln, denn da eine Abfrage q den Zustand ihrer Komponente unverändert lässt, gelten für sie Nachbedingungen der Art

q=OLD(q)

mit beliebigem Attribut q. Nachbedingungen, die ausdrücken, was sich nicht ändert, lässt man meist weg.

4 Qualitätssicherung mit normalen C++ Sprachmitteln

4.1 Umgangssprarchliche Spezifikation?

„Informelle Beschreibung“: Auf einem Parkplatz stehen PKW's und Motoräder. Zusammen seien es n Fahrzeuge mit insgesamt m Rädern. Bestimme die Anzahl P der PKW's.

„Lösung“: Sei

P := Anzahl der PKW's
M := Anzahl der Motoräder

$$\left\{ \begin{array}{r} P+M=n \\ 4P+2M=m \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{r} M=n-P \\ P=\frac{m-2n}{2} \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{r} M=\frac{4n-m}{2} \\ P=\frac{m-2n}{2} \end{array} \right\}$$

„Algorithmus“:

```
M := (4 * n - m) / 2;
P := (m - 2 * n) / 2;
write (M,P);
```

Problem: ****Null-Euro-Rechnung, Null-Euro-Mahnung,...

$(m,n)=(9,3) \Rightarrow P=1\frac{1}{2}$
$(m,n)=(2,5) \Rightarrow P=-4$

Vor der Entwicklung eines Algorithmus ist zunächst für das Problem eine *Spezifikation* bestehend aus

1. Definitionsbereich,
2. Wertebereich *und*
3. für die Lösung wichtigen Eigenschaften (insbesondere funktionaler Zusammenhang zwischen Eingabe- und Ausgabegrößen)

anzufertigen.

Besser ist also:

Eingabe: $m, n \in \{0, 1, ..., INT_MAX\}$

Vorbedingungen: m gerade, $2n \leq m \leq 4n$

Ausgabe: $P \in \{0, 1, ..., INT_MAX\}$, falls die Nachbedingung erfüllt ist (sonst „keine Lösung“)

Nachbedingung: Ein $(P, M) \in \{0, 1, ..., INT_MAX\}$ mit

$$P + M = n$$
$$4P + 2M = m$$

4.2 Unbeachtet - Integer-Overflows in C/C++ (Vermeidung von Qverflows)

„Spezifikation" von Variablen so genau wie in der benutzten Programmiersprache möglich:

```
const
   ErrorResult = -1;

function fakultaet(n : integer {n >= 0}): integer;
var
     zaehl : 2..maxint;
     teilres : 1..maxint; {Teilresultat}
begin
     teilres := 1;
     for zaehl := 2 to n do begin
           teilres := teilres * zaehl
     end;
     if (n >= 0) then fakultaet := teilres
          else fakultaet := ErrorResult;
end;
```

Verifikation:

Endlichkeit: Die Schleife wird genau (n-1)-mal durchlaufen, falls $n \geq 2$. Ansonsten wird sie kein mal durchlaufen.
Schleifenvariante: Zerlegung des gewünschten Ergebnisses $\boxed{n!}$ im ein schon berechnetes Teilergebnis $\boxed{\text{teilres}}$ und einen noch zu berechnenden Rest:

$$\begin{array}{c} \| \qquad | \\ n! = teilres * \prod_{i=zaehl}^{n} i \\ \| \qquad | \end{array}$$

(bei Schleifenbeginn: teilres=1, zaehl=2)

Korrektheit bei Abbruch:

```
begin
              {n in integer; zaehl, telres undef}
      telres := 1
              {n! = teilres * n!}

      for zaehl := 2 to n do begin       {n >= 2}
              { n! = teilres · ∏_{i=zaehl}^{n} i, n ≥ zaehl }
          teilres := teilres * zaehl
              {n! = teilres · ∏_{i=zaehl+1}^{n} i, n ≥ zaehl}
      end;
              {n! ≥ 2: n!=teilres*1, zaehl undef}

      {n! ≥ 2: n!=teilres; n in [0,1]: teilres = 1 = n!}
      {n < 0: n! undef}
        ⋮
```

Informelle Spezifikation:

Zweck: „fakultaet" berechnet die Fakultät von n:

$$n! = 1 * 2 * 3 * ... * (n-1) * n,$$
$$0! = 1$$

für alle n in [0,1,...,maxint], für die $n!$ auch in INTEGER liegt.

Vorbedingungen:

$n \in$ integer

Nachbedingungen:

fakultaet liefert:
$n!$, falls $n \geq 0$ **und** $n! \leq$ maxint;
ErrorResult, falls $n < 0$
undefiniertes Verhalten bei $n! >$ maxint.

Beachte:
Diese Spezifikation ist für die Praxis ungeeignet!
„Verhalten undefiniert" hängt von Pascal und der Laufzeitumgebung ab (Overflow-Exception ja/nein).

Mögliche Varianten/Verbesserungen

Ziel: Berechne `teilres := Teilres * zaehl` nur dann, wenn das innerhalb [1,...,maxint] möglich ist.

⋮

```
          {n! = teilres · ∏_{i=zaehl}^{n} i, n ≥ zaehl}
          {teilres <= maxint}
if (maxint div teilres >= zaehl) then
          {teilres * zaehl <= maxint}
      teilres := teilres * zaehl
          {n! = teilres · ∏_{i=zaehl+1}^{n} i, n ≥ zaehl}
else begin
          {n! ≥ teilres * zaehl > maxint}
   ⋮
```

Ähnlich: Etwa bei einer verbesserten Version von „Power“ unter Zugrundelegung von:

$$x^{2i} = (x^2)^i, x^{2i+1} = x^{2i} * x$$

statt von:

$$x^i = x^{i-1} * x$$

```
   ⋮
begin
      Teilerg := 1.0;
      while (n > 0) do
         if odd(n) then begin
            Teilerg := Teilerg * x; n := n-1;
         end else begin
            x := sqr(x); n:= n div 2;
      end;
   ⋮
```

Im Algorithmus nach **O.J. Dahl/ E.W. Dijkstra/C.A.R. Hoare: Structured Programming auf Seite 14** wird eine Vermeidung von Fallunterscheidungen auf folgende Weise angestrebt:

```
  ⋮
begin
    Teilerg := 1.0;
    while (n > 0) do begin
      if odd(n) then begin
        Teilerg := Teilerg * x; n := n-1;
      end;                                {n gerade}
      x := sqr(x); n:= n div 2;
    end;
  ⋮
```

Obwohl diese drei Varianten in $(\mathbb{R}, \mathbb{Z}_{\not{0}+})$ alle das richtige Ergebnis x^n liefert, wird in der dritten Variante am Schluß eine unnötige Quadierung von x vorgenommen(while-Statement mit $n = 1$). Das kann zu einem Overflow führen, obwohl der durch das entsprechende x := sqr(x) berechnete x-Wert für die gesuchte Potenz gar nicht mehr benötigt wird!
Es wird also für Variante 3 eine strengere Vorbedingung als nötig und wünschenswert gefordert(d.h. der Bereich der Argumente, für die x^n berechnet wird, ist unnötigerweise verkleinert worden):

- **Vorbedingung zu Variante 1/2:** x^n ist [im Rahmen der begrenzten Rechnergenauigkeit in IEEE_real) kleiner oder gleich FLT_MAX.
- **Vorbedingung zu Variante 3:** x^n und $x^{2[log_2(n)]+1}$ ist [im Rahmen der begrenzten Rechnergenauigkeit in IEEE_real) kleiner oder gleich FLT_MAX.

Es muß also für die Variante 3 nicht nur x^n, sondern auch die Potenz von x zum Exponenten der zu n nächsthöheren Zweierpotenz kleiner oder gleich FLT_MAX sein.

1. „Vermeide unnötigen Overflow in der Zwischenrechnung:“

```
n := n div 2; if (n <> 0) then x := sqr(x)
```

2. „Fange unvermeidlichen Overflow bei der Berechnung ab:“

   a) ... bei der Berechnung von sqr(x):

```
n := n div 2;
if (n <> 0) then
   if (x < SQR_FLT_MAX) then
                    ↑ selbst zu definierende Konstante aus
                      FLT_MAX (float.h,ect.)
      x := sqr(x)
   else ...;
```

b) ... bei der Berechnung von Teilerg * x:

```
if ((MAX_FLT/Teilerg) >= x) then
    Teilerg := Teilerg * x;
```

4.3 Vergessene Problematik: unordered floats — IEEE NaNs

In

```
y1 = 0;
y2 = 0;
for (i=0; i<ITERATIONS; i++)
{
        x1 = y1;
        x2 = y2;

        y1 = x1 * x1 - x2 * x2 + c1;
        y2 = 2.0 * x1 * x2 + c2;

        n2 = y1 * y1 + y2 * y2;
        if (n2 > 4.0) then
                /* skip */
        else
                plot(y1,y2);
}
```

werden leider auch viele Punkte, an denen die Reihe divergiert, gezeichnet. Warum? (Konvergente Iterationspunkte sollen geplottet werden. Ein Wert größer als 4.0 kann als Divergenzkriterium gewertet werden. Im Falle $\infty - \infty = NaN$ ist jedoch der Fall der Divergenz ohne Erfüllung von $n2 > 4.0$ gegeben.)

Eine Modifikation zu

```
else if unordered (n2,4.0) then
        /* skip */
else
        plot(y1,y2);
```

beachtet „unordered“ Argumente!

4.4 assert in C/C++

```
# include <assert.h>
      ⋮
int fakultät(int i)
{
        assert (i >= 0);        /* Vorbedingung */
              ⋮
        assert(result>0);        /* Nachbedingung */
}
```

liefert „Assertion failed: file ass.c, line 15“ bei **fakultaet(17)** statt dem falschen Ergebnis -28522240.

Nach dem Austesten eventuelles Abstellen der Überprüfungen durch

```
# define NDEBUG
# include <assert.h>
      ⋮
```

und erneute Compilation oder Übersetzung mittels `g++ -NDEBUG ...`

Ein Beispiel in C++:

```
# include <cassert>
// ...
      ...
       assert(i <= DIM);
// ...
```

4.5 Vermeidung von enum

Der enum-Typ ist ein voller Interger-Datentyp und deshalb nur bedingt zur Anwendung zu empfehlen.

Zwar erlaubt er aussagekräftige Namen für die zugehörigen Konstanten, ja sogar das Überladen von Operatoren (hier zum Beispiel des Inkrementoperators),

```
#include        <iostream>
#include        <string>

using namespace std;

enum Day {Montag, Dienstag, Mittwoch, Donnerstag,
          Freitag, Samstag, Sonntag};

Day& operator++(Day& d)
{
        return d = (Sonntag == d) ? Montag : Day(d + 1);
}

string DayTable[] = {"Montag", "Dienstag", "Mittwoch", "Donnerstag",
          "Freitag", "Samstag", "Sonntag"};

int main()
{
        Day Tag(Montag);

        for (int i = 0; i < 15; i++)
                cout << ++Tag << "  " << DayTable[int(Tag)] << endl;

}
```

er verhindert aber nicht die Benutung von unsinnigen Integer-Operationen (hier zum Beispiel der Multiplikationen von Wochentagen).

Ausweg bietet nur die Einführung einer eigenen Klasse nach dem folgenden Muster:

```
#include        <iostream>
#include        <sstream>
#include        <string>

using namespace std;
```

```
namespace DayNS {

class Day{
private:
        enum DayType {_Montag, _Dienstag, _Mittwoch, _Donnerstag,
                      _Freitag, _Samstag, _Sonntag};

        static const string DayTable[7];

        DayType t;

        Day(const DayType& dt): t(dt) {};

public:

        static const Day Montag;
        static const Day Dienstag;
        static const Day Mittwoch;
        static const Day Donnerstag;
        static const Day Freitag;
        static const Day Samstag;
        static const Day Sonntag;

        Day(const Day& d = Montag): t(d.t) {};

        Day& operator++();
        const Day operator++(int);

        friend istream& operator>>(istream&, Day&);
        friend ostream& operator<<(ostream&, const Day&);

};

const Day Day::Montag(_Montag);
const Day Day::Dienstag(_Dienstag);
const Day Day::Mittwoch(_Mittwoch);
const Day Day::Donnerstag(_Donnerstag);
const Day Day::Freitag(_Freitag);
const Day Day::Samstag(_Samstag);
const Day Day::Sonntag(_Sonntag);

const string Day::DayTable[] = {"Montag", "Dienstag", "Mittwoch", "Donnerstag",
                                "Freitag", "Samstag", "Sonntag"};
```

```
Day& Day::operator++()
{
        (*this).t = (_Sonntag == t) ? _Montag : DayType((*this).t + 1);
        return *this;
}

const Day Day::operator++(int)
{
        Day old_value(*this);
        ++(*this);
        return old_value;
}

istream& operator>>(istream& is, Day& d)
{
        string s;
        is >> s;
        for (int i = 0; i < 7; i++)
                if (s == Day::DayTable[i]) {
                        d.t =  Day::DayType(i);
                        return is;
                }


        // falscher Eingabestring:

        is.putback(' ');
        for (int i = s.length()-1; i >= 0; i--)
                is.putback(s[i]);

        is.clear(ios_base::badbit);
        return is;
}

ostream& operator<<(ostream& os, const Day& d)
{
        os << Day::DayTable[int(d.t)];
        return os;
}

}

int main()
{
```

```
        DayNS::Day d1;

        namespace d = DayNS;
        d::Day d3;


        using DayNS::Day;
        Day d2(DayNS::Day::Sonntag);


        using namespace DayNS;

        cout << d1 << endl;
        cout << d2 << endl;


        d1 = Day::Montag;
        for (int i = 0; i < 15; i++)
                cout << ++d1 << endl;
        cout << endl;

        d1 = Day::Montag;
        for (int i = 0; i < 15; i++)
                cout << d1++ << endl;
        cout << endl;

}
```

Zum Vermeiden eines eventuell ungewünschten Kopierkonstruktors beziehungsweise Zuweisungsoperators vergleiche:
http://www.mozilla.org/hacking/portable-cpp.html#copy_constructors

4.6 Compiletime Assertions

Um ein Programm nur auf solchen Compilern übersetzbar zu machen, auf dem der Typ `int` mindestens 16 Bit Genauigkeit bietet, kann man folgenden Code in sein Programm einbauen:

CT_ASSERT(sizeof(int) * CHAR_BIT >= 16, INT_TO_SMALL)

Die Compilation bricht im Fehlerfalle ab mit einer Meldung der Art:

"example.cc", line 21 : Error:
    Cannot cast from *ERROR_INT_TO_SMALL* to CompileTimeChecker < 0 >.
1 Error(s) detected.

Insbesondere sind solche CT_ASSERTs in der Deklaration von Templates nützlich, um sinnvolle Fehlermeldungen zu erhalten.

`CT_ASSERT` ist dabei etwa wie in

```
#include        <iostream>
#include        <iomanip>

using namespace std;

template<bool> struct CompileTimeChecker
{
        CompileTimeChecker(...);
};

template<> struct CompileTimeChecker<false> {};

#define CT_ASSERT(expr, msg)\
 {\
    class ERROR_##msg{};\
    (void)sizeof(CompileTimeChecker< (expr) != 0  > ((ERROR_##msg())));\
 }

int main()
{
   CT_ASSERT(LDBL_DIG > DBL_DIG, No_Higher_Accuracy);
   return 0;
}
```

```
/*************************************************

"ct_asert_example.cc", line 21: Error:
    Cannot cast from ERROR_No_Higher_Accuracy to CompileTimeChecker<0>.
1 Error(s) detected.

*************************************************/
```

zu definieren.

Siehe auch:
http://blogs.geekdojo.net/pdbartlett/archive/2004/05/19/TemplateMetaProgramming.aspx

4.7 Ausnahmebedingungen: Exceptions/Traps

Das Codestück:

```
...
double power1(double x, int exp)
{
    double erg(1.0);
    if (exp < 0 ) throw(exp);
...
}
```

erzeugt bei Nichterfüllen der Vorbedingung eine (abfangbare) Ausnahmebedingung (Exception) des Typs `int`.

Exceptions sollten in Handlern (catch-Anweisungen) abgefangen werden:

```
try{
        //...
} catch(const char* err){
        cerr << endl << "### Fehler: "<< err << endl;
        exit(1);
} catch(const string& err){
        //...
} catch(const int& i_err){
        //...
} catch (...){
        cerr << endl << "### Fehler: unbekannte Exception"<< endl;
        exit(2);
}
```

Dabei müssen speziellere vor allgemeineneren Exceptions abgefangen werden:

```
//...
#include <exception>
//...
try{
    //...
    if (exp < 0) throw range_error("exp invalid, should be >= 0");
    //...
} catch(const std::range_error& re){
    cerr << endl << "### Fehler: "<< re.what() << endl;
    exit(1);
} catch(const std::bad_alloc& bae){
    //...
} catch(const std::exception& e){
    //...
} catch (...){
    cerr << endl << "### Fehler: unbekannte Exception"<< endl;
    exit(2);
}
```

wobei folgende standartmäßig vorhandene `exception`-Hierarchie benutzt wurde:

```
- exception
- - bad_alloc
- - bad_exception
- - bad_cast
- - bad_typeid
- - ios::failure
- - runtime_error
- - - - range_error
- - - - overflow
- - - - underflow
- - logic_error
- - - - length_error
- - - - domain_error
- - - - out_of_range
- - - - invalid_argument
```

Beachten Sie, dass in catch-Anweisungen keine Typkonversion statttfindet, eine `const char*`-Exception also nicht von einer `const string&`-Catch-Anweisung abgefngen wird.

Wollen Sie eine eigene Exception-Hierarchie, etwa

```
- exception
- - MathError
- - - Overflow
- - - ZeroDivide
```

aufbauen, so kann das folgendermaßen geschehen:

```
//...
class Matherr;
class Overflow : public Matherr();
class ZeroDivide : publc Matherr();
//...
try{
        //...
        if (Bed1) throw ZeroDivide();
        //...
} catch(const ZeroDivide& e){
        //...
} catch(const Matherr&){
        //...
}
```

Richtet man MathError als Unterklasse von exception ein, ist (wie im Beispiel http://www.cplusplus.com/doc/tutorial/exceptions.html) die virtuelle Methode `what()` zu implementieren.

Zusammenfassung:

Wird in einer Funktion eine erzeugte Exception nicht abgefangen, so wird diese Funktion abgebrochen und ein Handler für die Exception in der sie aufrufenden Programmeinheit gesucht ...

Ist schließlich auch in `main()` kein Handler (zutreffende `catch`-Anweisung) aufzufinden, wird das gesamte Programm abgebrochen.

Vergleiche auch:

http://java.sun.com/docs/books/tutorial/essential/exceptions/
http://www.ica1.uni-stuttgart.de/Courses_and_Lectures/C++/script/node29.html
http://www.codeproject.com/csharp/exceptions.asp

Zur Spezifikation von Exceptions siehe Abschnitt 5.2 (errs-Klausel) dieser Materialsammlung beziehungsweise
ftp://reports.stanford.edu/pub/cstr/reports/csl/tr/84/265/CSL-TR-84-265.pdf. (Abschnitt 3.5).

5 Spezifikation mit sprachexternen Hilfsmitteln: VDM, NANA, OCL

5.1 Invarianten in der Spezifikationssprache VDM

```
Date::
    DAY:ℕ₁
    MONTH:ℕ₁
    YEAR:ℕ₁
inv mk-Date(d,m,y)≜
    (1901 ≦ y) ∧
    (m ≦ 12) ∧
    ( d ≦
    cases m:
       4,6,9,11 → 30,
       2        → if((y rem 4 = 0)∧ (y rem 100 ≠ 0)) ∨ (y rem 400 = 0))
                  then 29 else 28,
       others → 31
end
)
```

5.2 Ausnahmen (errs-Klausel) in VDM

DEQUEUE() e: [Qelement]
ext wr q:Queue
pre p $\neq$ []
post $\overleftarrow{q}$ = [e]$\frown$q
errs QueueEnty:
$q = [] \rightarrow q = \overleftarrow{q} \wedge e = nil$

5.3 Module in VDM

```
module Multiset
  parameters
    types elem
    functions
      empty_bag: () -> bag;
      num_bag: elem * bag -> nat;
      plus_bag: elem * bag -> bag;
      mems_bag: bag -> set of elem;
      merge_bag,
      diff_bag: bag * bag -> bag
  definitions
    types bag = map elem to nat1
    functions
      empty_bag: () -> bag
      empty_bag() == {};
      num_bag: elem * bag -> nat
      num_bag(e,m) == if e in set dom m then m(e) else 0;
      plus_bag: elem * bag -> bag
      plus_bag(e,m) == m ++ {e |-> num_bag(e,m)+1};
      mems_bag: bag -> set of elem
      mems_bag(m) == dom m;
      merge_bag: bag * bag -> bag
      merge_bag(m_1,m_2) == {e |-> num_bag(e,m_1)+
        num_bag(e,m_2) | e in set dom m_1 union dom m_2};
      diff_bag: bag * bag -> bag
      diff_bag(m_1,m_2) == {e |-> num_bag(e,m_1)-
        num_bag(e,m_1) | ((e in set dom m_1) and
        (num_bag(e,m_1)>num_bag(e,m_2))):int}
end Multiset
```

Vergleiche: http://www.vdmtools.jp/uploads/manuals/langmansl_a4E.pdf beziehungsweise http://www.vdmtools.jp/en/modules/tinyd2/index.php?id=2

5.4 Klassen in VDM++

In VDM++ sind übliche Klassen als Modulkonzept vorgesehen.
http://fm06.mcmaster.ca/VDM++%20tutorial%20FM%202006%20handouts.pdf

5.5 Object Constraint Language (OCL)

Siehe auch: http://www.klasse.nl/ocl/ocl-introduction.html

5.5.1 Vor- und Nachbedingungen:

```
context Seminar::fuegeStudentHinzu(s: Student)
pre: Student::erfuelltBelegungsVoraussetzungen(s)
pre: s = NachrueckStudent->first()
pre: NachrueckStudent->first().Belegung.Seminar->excludes(self)
post: NachrueckStudent->first().Belegung.Seminar->includes(self)
post: NachrueckStudent->first().Belegung->select(Seminar=self).
           bisherigePunkte->sum() = 0
```

5.5.2 Invarianten

```
context Student
inv: let anzAbgeschlossenerSeminare : Integer =
         Belegung->select(gueltigerEndPunktstand = true)->
         size()
     in
        NotenDurchschnitt = Belegung->
         select(gueltigerEndPunktstand = true).
         getEndPunktstand()->sum() / anzAbgeschlossenerSeminare

context Student::getBelegteSeminare(): Set(Seminar)
body: Seminar
```

5.6 ANNA (annotation language for ADA)

http://sunset.usc.edu/classes/cs599_2000/October5a.ppt
ftp://reports.stanford.edu/pub/cstr/reports/csl/tr/84/265/CSL-TR-84-265.pdf

Fortgeschrittene Spracheigenschaften von ANNA/ADA:

— Unterbereichstypen auch von Gleitkommazahlen:

```
   type TARGET is
       record
           BULLS_EYE : REAL range 1.0 .. 10.0;
           INNER_CIRCLE : REAL range 2.0 .. 20.0;
           OUTER_CIRCLE : REAL range 4.0 .. 40.0;
       end record;
```

Frage: Durch welche Invariante müßte das in C++ realisiert werden?

— Abhängige Record-Felder:

```
   type TARGET is
       record
           BULLS_EYE : REAL range 1.0 .. 10.0;
           INNER_CIRCLE : REAL range 2.0 .. 20.0;
           OUTER_CIRCLE : REAL range 4.0 .. 40.0;
       end record;
--|    where T : TARGET =>
--|        T.INNER_CIRCLE = 2*T.BULLS_EYE and
--|        T.OUTER_CIRCLE = 2*T.INNER_CIRCLE;
```

— Nachbedingung einer virtuellen (Spezifikations-)Funktion:

```
--: function SCORE(X, Y : REAL; T : TARGET)
--: return INTEGER
--|    where return
--|        if X*X+Y*Y <= T.BULLS_EYE ** 2 then 10
--|        elseif X*X+Y*Y <= T.INNER_CIRCLE ** 2 then 5
--|        elseif X*X+Y*Y <= T.OUTER_CIRCLE ** 2 then 1
--|        else
--|            0
--|        end if;
```

— Nachbedingung nutzt virtuelle Funktionen:

```
     procedure QUICKSORT(A : in out VECTOR);
--|     where out (A) = SORTED(in A);
```

— Ein Modul mit virtuellen Funktionen:

```
--: package SORTING_CONCEPTS is
--:    subtype INDEX_TYPE is INTEGER;
--:    type VECTOR is array(INDEX_TYPE range <>) of INTEGER;

--:    function ORDERED (A : VECTOR) return BOOLEAN;
--|         where
--|         return for all I, J : A'RANGE => I <= J -> A(I) <= A(J);

--:    function PERMUTATION(A,B : VECTOR)
--:    return BOOLEAN;
--|         where
--|         in (A'LENGTH = B'LENGTH),
--|         return
--|              A'LENGTH = 0
--|                  or else
--|              (exist I : B'RANGE => A(A'FIRST) = B(I)
--|                  and
--|                 PERMUTATION(A(A'FIRST+1 .. A'LAST)),
--|                             B(B'FIRST+1 .. I-1) &
--|                               B(I+1 .. B'LAST)));

--:    function SORTED(A : VECTOR) return VECTOR;
--|          where return B : VECTOR =>
--|            PERMUTATION(A, B) and ORDERED(B);

--:    function IS_IN_INTERVAL(X : ITEM; A : VECTOR)
--:    return BOOLEAN;
--|          where
--|            return (exist I : A'RANGE => X = A(I)

--:    function PARTITIONRD(A :VECTOR; i : INDEX_TYPE)
--:    return BOOLEAN;
--|          where return
--|            if I is in A'RANGE then
--|                 for all J :A'FIRST .. I => A(J) <= A(I) and
--|                 for all J : I .. A'LAST => A(I) <= A(J)
--|            else
--|                  FALSE
--|            endif

--: end SORTING_CONCEPTSTS;
```

und seine Benutzung:

```
--:  with SORTING_CONCEPTS
--:  use SORTING_CONCEPTS;

--     procedure to sort a vector A, a one dimensional array of integers.
       procedure SORT(A : in out VECTOR);
--|         where
--|             out (PERMUTATION(A, in A)),
--|             out (ORDERED(A));
```

— Rahmenbedingung in einer Nachbedingung („was bleibt unverändert?“):

```
    procedure
         TRANSFER(AMOUNT : POSITIVE_DOLLARS_SUBTYPE;
                  OUT_ACCOUNT: in out  WITHDRAWAL_SUBTYPE
                  IN_ACCOUNT : in out DEPOSIT_SUBTYPE);
--|          where
--|               in (AMOUNT <= BALANCE(OUT_ACCOUNT)),
--|               out (BLANCE(OUT_ACCOUNT) =
--|                    in (BLANCE(OUT_ACCOUNT)-AMOUNT),
--|               out (BLANCE(OUT_ACCOUNT) =
--|                    in (BLANCE(IN_ACCOUNT)+AMOUNT),
--|               out (for all A : ACCOUNT_TYPE =>
--|                 A /= OUT_ACCOUNT and A /= IN_ACCOUNT ->
--|                     BALANCE(A) = in BALANCE(A));
```

— Variablen mit Einschränkungen/Quantoren in Spezifikationen:

```
     declare
           X : INTEGER range 1..10000;
--|        X mod 2=0;

    type DAY is (SUN,MON,TUE,WED,THU,FRI,SAT);
--       Every value of DAY has length 3.
--|      for all X : DAY => DAY'IMAGE(X)'LENGTH = 3;
--       Names of Weekdays do not contain 'S'.
--|      for all X : DAY range MON .. FRI =>
--|         for all I : 1 .. DAY'WIDTH =>
--|                DAY'IMAGE(X)(I) /= 'S';
```

oder kürzer:

```
--| ...
--|   for all X : DAY range MON .. FRI,
--|        I : 1 .. DAY'WIDTH => DAY'IMAGE(X)(I) /= 'S';
```

— Einschränkungen in Typdeklarationen:

```
    subtype LETTERS is CHARACTER;
--|      where S : LETTER =>!\\
--|          S isin CHARACTER range 'A' .. 'Z' or
--|          S isin CHARACTER range 'a' .. 'z';

    type INTERVALL is;
         record
             LEFT_POINT, RIGHT_POINT : REAL;
         end record;
--|   where X : INTERVAL =>
--|           X.LEFT_POINT <= x.RIGHT_POINT;
```

— Ein abstrakter Datentyp:

```
    package STACK_MANAGER is
          type STACK is private;
--:       function LENGTH(S : STACK) return NATURAL;
--:       function TOP(S : STACK) return ITEM;

          procedure PUSH(X : in ITEM; S : in out STACK);
--|          where out (LENGTH(S) = LENGTH(in S)+1),
--|                out (TOP(S) = X);

          procedure POP(Y : out; ITEM; S : in out STACK);
--|          where out (LENGTH(S) = LENGTH(in S)-1),
--|                out (Y = TOP(in S));

          private ...

    end STACK_MANAGER;
```

— Versuch einer numerischen Spezifikation

```
  function SIN(X : RADIANS_TYPE) return AMPLITUDE_TYPE;
--|      where return A : AMPLITUDE_TYPE =>
--|           abs (FLOAT(A)-SINE_SERIES(X, CUT)) <=
--|                          MAX_DELTA;
```

mit Hilfe einer virtuellen Spezifikationsfunktion:

```
--:  function SINE_SERIES(x : FLOAT; CUT_OFF : INTEGER)
--:  return FLOAT is
--:      Y : FLOAT := 0.0;
--:  begin
--:      for I in 0 .. CUT_OFF loop
--:           Y := Y+FLOAT((-1)**I)/
--:                FLOAT(FACTORIAL(2*I+1))*X** (2*I+1);
--:      end loop;
--:      return Y;
--:  end SINE_SERIES;
```

Links zu ADA/ANNA:
http://de.wikipedia.org/wiki/Ada_(Programmiersprache)
http://www.huber-net.de/adagag.htm
http://www.wackerart.de/ada.html
http://www.ada-deutschland.de/AdaTourCD2004/index_dokumente.html
http://www.ada-deutschland.de/AdaTourCD2004/ada_dokumentation/einfuehrung/

5.7 Contracting and Subcontracting

5.7.1 Contracting

Erinnerung an Kapitel 3:

PbC	VERPFLICHTUNGEN	VORTEILE
Benutzer der Klasse	delegiert nur bei erfüllter Vorbedingung	kommt in den Genuß der garantierten Nachbedingung und Invarianten
Anbieter der Klasse	(nur bei gültiger Vorbedingung:) muß die Nachbedingung erfüllen	braucht Vorbedingung nicht überprüfen

Tabelle 5.1: Verpflichtungen/Vorteile von Verträgen zwischen Komponentenanbieter und -benutzer

Klassen-Invarianten (Gültige Attributwertkombinationen)

- schränken Werte von Attributen ein, trennen gültige von ungültigen Exemplaren einer Klasse
- spezifizieren Redundanzen (vgl. Day/Month/Year, Count/IsEmpty, ...)

Methoden-Vorbedingungen (an Attribute und Parameter)

- schränken den Bereich ein, in dem die Methode erfolgreich sein muß

Methoden-Nachbedingungen (an Attribute und Parameter)

- spezifizieren (formal) das Ergebnis der Methode (das **was**, nicht das **wie**)

Was vor und nach jeder Methode gelten muß:

Konstruktor:

$\{Vorbed_an_Parameter\}\ Konstruktor\ \{\text{Inv}\}$

Jede andere Methode M:

$\{Vorbed_M \wedge Inv\}\ M\ \{Nachbed_M \wedge Inv\}$

5.7.2 Subcontracting (is-a-Vererbung)

Ein erstes Beispiel:

Ursprüngliche Definition:
invert(epsilon:REAL) is - - Invert martix with precision epsilon
    require epsilon>= 10^(−6)
    ...
    ensure abs ((Current * inverse) - Identy) <= epsilon
end
 	
Redefinition:
invert(epsilon:REAL)is - - Invert matrix with precision epsilon
    require else epsilon>= 10^(−20)
...
    ensure abs ((Current * inverse) - Identy) <= (epsilon/2)
end

aus: http://www.cse.yorku.ca/course_archive/2004-05/F/3311/sectionA/22-InheritDBCgen.pdf

Unternehmer und Subunternehmer:

Ein Vertrag zwischen Kunde und Unternehmer laute:

```
                     Interface Directory
.
.
.
ACTIONS
   Put(IN k:Keys, IN v:Values)
      PRE
         NOT Has(k)
      POST
         Has(k)
         ValueFor(k)= v
         Count = OLD(Count)+1
.
.
.
```

Er kann durch den Unternehmer allein nicht zeitgerecht erfüllt werden. Jedoch bietet ein anderer Unternehmer den folgenden Vertrag an:

```
                    Interface DirectoryB
.
.
.
ACTIONS
   Put(IN k:Keys, IN v:Values)
      PRE
         TRUE
      POST
         Has(k)
         ValueFor(k)= v
         NOT OLD(Has(k)) IMPLIES Count = OLD(Count)+1
         OLD(Has(k))     IMPLIES Cout = OLD(Count)
.
.
.
```

Er kann als Subunternehmer des Unternehmers in die Pflicht genommen werden, da DirectoryB den Vertrag Directory vollständig erfüllt. Der Kunde kann, da an die Vorbedingungen von Directory gebunden, keinen Unterschied zwischen der Benutzung von Directory und DirectoryB feststellen. Lediglich wenn er sich nicht an die Vorbedingungen des Vertrags Directory hielte, wäre das Verhalten seiner Software anders, aber das darf er ja nicht!
Subcontracting (is-a-Vererbung) verlangt also folgende Beziehungen zwischen den Verträgen der public Methoden einer Kindklasse und denjenigen ihrer Elterklasse:

- Invarianten des Kindes dürfen nicht schwächer sein als diejenigen der Elterklasse,
- Vorbedingungen einer Kindmethode dürfen nicht stärker sein als die Vorbedingungen der Eltermethode,
- Nachbedingungen einer Kindmethode dürfen beim Eintreten der Vorbedingung der Eltermethode nicht schwächer sein als die Nachbedingung der Eltermethode, andernfalls dürfen sie beliebig sein. (Diese Beliebigkeit wird natürlich im allgemeinen vom Subunternehmer dazu genutzt, einen möglichst großen Marktwert für seine Mehrfunktionalität — es gibt mehr Fälle, in denen die Vorbedingung der Kindmethode erfüllt ist, als die Fälle in denen lediglich die Vorbedingung der Eltermethode erfüllt sind — zu erreichen.)

Kurz:

$$Invariante_{Kindklasse} = Invarinte_{Elterklasse} \wedge ...$$
$$Vorbedingung_{Kindmethode} = Vorbedingung_{Eltermethode} \vee ...$$
$$Nachbedingung_{Kindmethode} = \begin{cases} Nachbedingung_{Eltermethode} \wedge ... & \text{, falls } Vorbedingung_{Eltermethode} \\ \text{beliebig} & \text{, sonst} \end{cases}$$

Diese formalen Subcontracting-Regeln können in der Anwendung häufig logisch zusammengefaßt werden. Im obigen Beispiel also:

```
-------------------------------- Directory
Pre-Put_Directory(k,v)  = NOT Has(k)

Post-Put-Directory(k,v) = Has(k) AND ValueFor(k) = v AND
                          Count = OLD(Count)+1
```

liefert nach den obigen Regeln:

```
-------------------------------- DirectoryB
Pre-Put_DirectoryB(k,v)  = TRUE

Post-Put_DirectoryB(k,v) =
  (Pre-Put_Directory(k,v) IMPLIES Post-Put-Directory(k,v))
 AND
  ((NOT Pre-Put_Directory(k,v) AND Pre-Put_DirectoryB(k,v)) IMPLIES "Beliebiges"

  = (NOT OLD(Has(k))) IMPLIES (Has(k) AND ValueFor(k) = v AND
                                         Count = OLD(Count)+1))
    AND ((OLD(Has(k)) AND TRUE) IMPLIES "Beliebiges")
```

Nach dem Namen der Methode `Put()` ist "Beliebiges" natürlich einzig sinnvoll durch

```
Has(k) AND Count = OLD(Count) AND ValueFor(k) = ?
```

zu ersetzen. Dabei hat man für den letzten Anteil (Wert beim Schlüssel `k`) noch eine gewisse Entscheidungsfreiheit. Mögliche, sinnvoll erscheinende Spezifikationen:

- Der alte Wert bleibt erhalten.
- Der alte Wert wird überschrieben.
- Der Wert wird als „unbestimmt“ gekennzeichnet, da er in der Historie mit unterschiedlichen Wertbindungen versehen werden sollte.

Entscheidet man sich für die vorraussichtlich marktrelevanteste Spezifikation (Wertüberschreibung), so erhält man nach ein paar Zusammenfassungen:

```
-------------------------------- DirectoryB
Pre-Put_DirectoryB(k,v)  = TRUE
Post-Put_DirectoryB(k,v) =
 (NOT OLD(Has(k))) IMPLIES (Has(k) AND ValueFor(k) = v AND
                                      Count = OLD(Count)+1)) AND
 ((OLD(Has(k)) AND TRUE) IMPLIES (Has(k) AND Count = OLD(Count) AND
                                                ValueFor(k) = v))
```

```
  = Has(k) AND ValueFor(k) = v AND
    (NOT OLD(Has(k)) IMPLIES Count = OLD(Count)+1) AND
    ((OLD(Has(k)) IMPLIES Count = OLD(Count))
```

Ein weiteres Subcontracting-Beispiel:
„Löse ein LGS“

```
-------------------------------- LoeseLGS-Elter
ACTIONS
   LoeseLGS( IN A : Matrix,
             IN b : Vektor,
            OUT x : Vektor )
      PRE
         NOT Det(A) = 0
      POST
         || A * x - b || <= EPSILON
```

sowie ein Subcontract:

```
------------------------------- LoeseLGS-Kind
ACTIONS
   LoeseLGS( IN A : Matrix,
             IN b : Vektor,
            OUT x : Vektor )
      PRE
         TRUE
      POST
         NOT Det(A) = 0 IMPLIES || A * x - b || <= EPSILON
         Det(A) = 0     IMPLIES "x ist eine Minimalstelle von || A * x - b ||"
```

Beipiel: Bruecke

```
------------------------------ Fussgaengerbruecke
   QUERIES
      MaxLast : REAL
      AktLast : REAL
   INVARIANTS
      MaxLast >= 7500
      AktLast <= MaxLast
   ACTIONS
      ueberquereBruecke( IN gew : REAL,
                         OUT Guthaben : INTEGER )
         PRE
            gew  + AktLast <= MaxLast
            gew <= 200
            Guthaben >= 2
         POST
            AktLast = OLD(AktLast) + gew
            Guthaben = OLD(Guthaben) - 2
      verlasseBruecke( IN gew : REAL )
         ...
```

sowie ein Subcontract:

```
------------------------------ Autobruecke
   QUERIES
      MaxLast : REAL
      AktLast : REAL
   INVARIANTS
      MaxLast >= 800000
      AktLast <= MaxLast
   ACTIONS
      ueberquereBruecke( IN gew : REAL,
                         OUT Guthaben : INTEGER )
         PRE
            gew  + AktLast <= MaxLast
            gew <= 20000
            Guthaben >= 20
         POST
            AktLast = OLD(AktLast) + gew
            OLD(gew) <= 200     IMPLIES Guthaben = OLD(Guthaben) - 2
            NOT OLD(gew) <= 200 IMPLIES Guthaben = OLD(Guthaben) - 20
      verlasseBruecke( IN gew : REAL )
         ...
```

Aufgabe:

Überlege Contracts und Subcontracts im Umfeld:
— Kunde/Stammkunde
— Firmenkonto/Privatkundenkonto
— Vereinsmitglied /Vorstandsmitglied
— ...

6 Sprachabstraktion füt gut lesbare Spezifikationen: newmat10, STL, ...

6.1 newmat10

6.1.1 Matrizen/Vektoren in newmat10

Voraussetzungen an den Benutzer:

1. Verstehen Sie `A * B` als Matritzenmultiplikation und nicht als Multiplikation von einem Skalar und einem weiteren Skalar.
2. Benötigen Sie Matritzenoperatoren wie `*` und `+` , so daß sie folgende Formel schreiben können.

$$X = A * (B + C)$$

3. Benötigen Sie eine Auswahl von (nicht wenigen) Matrizentypen.
4. Benötigen Sie nur einen Skalartyp (`double` oder `float`) für die Matrix-Komponenten.
5. Möchten Sie mit Matritzen in einer Größenordnung von 10×10 bis zur maximalen Arbeitsspeichergröße arbeiten.
6. Können Sie ein großes Paket zulassen oder dulden.

6.1.2 Generelle Beschreibung/General description

Das Paket ist für Wissenschaftler und Ingenieure geschrieben, die unterschiedlichste Typen von Matritzen manipulieren müssen, und dazu die Standardmatrixoperationen benutzen. Der Schwerpunkt liegt bei Operationen die in statistischen Berechnungen erforderlich sind, wie etwa die Berechnung kleinster Quadrate, das Lösen von linearen Gleichungssystemen, die Berechnung von Eigenwerten, ...

`newmat10` unterstützt die folgenden Matrizen-Typen:

Listing 6.1: Matrizen-Typen der *newmat10*-Bibliothek

```
Matrix                      (rectangular matrix)
nricMatrix                  (for use with Numerical Recipes
                             in C programs)
UpperTriangularMatrix
LowerTriangularMatrix
DiagonalMatrix
SymmetricMatrix
BandMatrix
UpperBandMatrix             (upper triangular band matrix)
LowerBandMatrix             (lower triangular band matrix)
SymmetricBandMatrix
RowVector                   (derived from Matrix)
ColumnVector                (derived from Matrix)
```

Nur ein Elementen-Typ (float oder double) wird unterstützt.

Die Bibliothek schließt die Operationen *, +,-, Verkettung, Inverse, Transponierte, Umwandlung zwischen Typen, Submatrix, Determinante, Cholesky-Zerlegung, QR-Zerlegung, Singulärwert-Zerlegung, Eigenwerte einer symmetrischen Matrix, Sortieren und die schnelle-Fourier-Transformation ein. Ebenso ist das Ausdrucken möglich.

Sie ist für Matrizen der Größenordnung 10 x 10 bis zur maximalen Größe des Speicherbereiches, die Ihre Maschiene zuläßt, geeignet. Die Anzahl von Elementen in einem Feld ist nur durch maximale Größe eines `int`s beschränkt. Das Paket kann auch für sehr kleine Matrizen verwendet werden, arbeitet dann aber ziemlich ineffizient. Einige der Faktorisierungs-Funktionen sind für „paged memory" (noch) nicht optimiert und deshalb für sehr großen Matrizen ineffizient.

Es wird *lazy evaluation* (hinausgezögerte Auswertung) von Matrixausdrücken verwendet, um die Effizienz zu verbessern und die Benutzung temprärer (Hilfs-)Variablen klein zu halten.

6.1.3 Matrizenmanipulation

Die newmat-Bibliothek dient der Manipulation von Matrizen inklusive der Standardoperationen wie (`*`, `+`, ...) in der gewohnten mathematischen Schreibweise.

Eine Matrix ist ein zwei dimensionales Feld von Zahlen. Eine Matrizenbibliothek unterscheidet sich von einer Array-Bibliothek:

Kennzeichen	Matrixbibliothek	Array-Bibliothek
Ausdrücke	Matrixausdrücke in mathematiknaher Schreibweise	elementweise wirkende Operatoren
Element-Zugriff	Element, Slice, View	Element, Iterator
Elementare Funktionen	Determinante, Spur	elementweise wirkende Matrixfunktionen
Fortgeschrittene Funktionen	Eigenwert, Singulärwert	—
Element-Typen	double, complex	double, float, int, string, etc.
Typen	Rechteckig, symmetrisch, diagonal, etc.	Eins, zwei und dreidimensionale Felder

Tabelle 6.1: newmath10 - Eigenschaften

6.1.4 Ausdrucksauswertung – lazy evaluation

Betrachten sie die folgende Vorschrift:

$$X = B - X;$$

Ein einfaches Programm subtrahiert X von B, speichert das Ergebnis in einer temporären Variable T1 und kopiert dann die temporäre Variable T1 in die Variable X. Dieser Vorgang könnte beschleunigt werden, wenn das Programm erkennen könnte, das das Ergebnis direkt in die Variable X gespeichert werden kann. Dies würde automatisch geschehen, wenn das Programm sich zuerst die Rechenanweisung ansehen und dann X als temporär markieren könnte.

C-Programmierer können das Problem

$$X = X - B;$$

umgehen, indem sie den Operator **-=** benutzen:

$$X -= B;$$

Wie auch immer, dies ist eine unnatürliche (unmathematische) schreibweise für Naturwissenschaftler und Ingenieure. Es wäre schöner, wenn man $X = X - B$ schreiben und das Programm die Vereinfachung vornehmen lassen könnte.

Ein weiteres Beispiel, wo diese automatische Analyse einer Anweisung hilfreich wäre, ist beispielsweise:

```
X = A.i() * B;
```

Hier bezeichnet $i()$ die Umkehrfunktion oder das Inverse.

Numeriker wissen, dass es uneffizient ist, in diesem Ausdruck erst die Inverse und dann die Multiplikation auszuwerten. Auch hier ist es sehr nützlich, den Weg der Anweisungen aufzuschreiben.

In dieser Hinsicht ist die Interpretation von $A.i() * B$ als eine unterstützende und geeignete Schreibweise zu sehen.

Es gibt einen dritten Grund für diese zweistufige Auswertung eines Ausdrucks und dies ist wahrscheinlich der wichtigere: In C++ ist es sehr schwierig, das Ergebnis einer Funktion zurückzugeben, (Beispielsweise *, + usw.), ohne eine Kopie zu verwenden. Dies ist insbesondere der Fall, wenn eine Wertzuweisung (=) erfolgt. Der hier beschriebene Mechanismus zeigt einen Weg, um diese Problematik in Matrizenausdrücken zu vermeiden.

Der C++ Standart (Abschnitt 12.8/15) erlaubt es dem Compiler, das Programm zu optimieren, indem er die Kopien eines zurückgegebenen Objektes einer Funktion wegfallen läßt, aber es gibt immer eine Kopie, wenn eine Zuweisung (=) ausgeführt wird. Das bedeutet, dass eine spezielle Handhabung von Rückgabewerten (return) einer Funktion, weniger wichtig ist, wenn ein moderner Compiler benutzt wird.

Um eine vernünftige Analyse von Matrizenausdrücken durchzuführen wird eine Baumstruktur des Ausdrucks erstellt. Hier als Beispiel $(A + B) * C$:

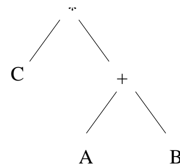


Abbildung 6.1: (A+B)*C - Baum

Um nicht einfach nur A und B zu addieren, ergibt der + Operator ein Objekt der Klasse **AddedMatrix**, welches nur die Zeiger(Pointer) zu A und B enthält. Dann ergibt der * Operator ein Objekt der Klasse **MultipledMatrix**, welches die Zeiger zu der **AddedMatrix** und C enthält. Der Baum wird auf Vereinfachungen geprüft und dann rekursiv ausgewertet.

Weitere noch nicht eingeschlossene Möglichkeiten sind, A.t() *A und A.t() +A zu erkennen und wie symmetrisch diese sind oder die Effizienz(Leistungsfähigkeit) der

Auswertung von Ausdrücken wie A+B+C, A*B*C, A*B.t() zu verbessern.

Einer der Nachteile des zweistufigen Ansatzes ist es , dass die Typen von Matrixausdrücken zur Laufzeit bestimmt werde, so dass der Compiler Fehler der folgenden Codezeilen nicht entdecken wird:

```
Matrix M;
DiagonalMatrix D;
....;
D = M;
```

Informationen und Einleitung zur Bibsubliothek *newmat10*
http://www.robertnz.net/nm_intro.htm
http://www.robertnz.net/nm10.htm
Download:
http://www.robertnz.net/download.html
Weitere Links:
http://objcryst.sourceforge.net/ObjCryst/newmat.htm
http://freshmeat.net/projects/newmat/?branch_id=7126&release_id=23424

6.1.5 Eine Beispielanwendung

```
// LU-Zerlegung, A = L * U nach Gauss ohne Pivotisierung, uebersetze mit:
// g++ -O2 -Wall -o lu-gauss lu-gauss.cc -L. -lm -lnewmat


#include <iostream>
#include <iomanip>
#include <cmath>

#include "newmat.h"
#include "newmatio.h"
#include "newmatnl.h"

using namespace std;

int main()
{
 int m;                  // mXm-Matrix
 int i,j;                // Zeilen-/Spaltenindices

                                        // Matrix - Eingabe

        cout << "Gauss-Elimination ohne Pivotisierung:" << endl << endl;
```

```
cout << "Geben Sie bitte m ein (m x m-Matrix) : ";
cin >> m;

Matrix A(m,m);
cout <<"\nGeben Sie die Matrix-Elemente zeilenweise ein :";

for(i=1;i<=m;i++)                  // Marixeingabe (zeilenweise)
        for(j=1;j<=m;j++)
        {
                cin >> A(i,j);
        }

cout <<"\nDie eingegebene Matrix lautet:" << endl << endl <<
        setprecision(7) << setw(14) << A << endl;

int maxLSpalte = (m - 1);

                                // LU-Zerlegung

UpperTriangularMatrix U(m);

DiagonalMatrix Id(m); Id = 1.0;
LowerTriangularMatrix L;
L << Id;

// L * U + Rest = A

{

Matrix Rest = A;        // Rest: noch lu zu zerlegende Rextmatrix

//
//      Dimensionsreduktion um 1 bei der LU-Zerlegung:
//
// ( 1 | 0 )    ( uii | u )    (  uii  |   u    )!
// ( --+-- ) *  ( ----+-- ) =  ( ------+------- )=Rest.SubMatrix(i,m,i,m)
// ( l | L )    (  0  | U )    ( l*uii | l*u+LU )
//
//
//                      uii := Rest(i,i)
//                      u   := Rest.SubMatrix(i,i,i+1,m)
//                      l   := (1.0/uii)*Rest.SubMatrix(i+1,m,i,i)
//                      LU  := Rest.SubMatrix(i+1,m,i+1,m)-l*u
//
```

```
      for(i = 1; i <= maxLSpalte; i++)
      {
        if (fabs(Rest(i,i)) <= 1.0E-15)
        {
              cerr << "\nHauptunterdeterminantenkriterium nicht erfuellt!\n";
              exit(1);
        }

      //

      U.SubMatrix(i,i,i,m) = Rest.SubMatrix(i,i,i,m);
      L.SubMatrix(i+1,m,i,i) =
                      (1.0 / Rest(i,i)) * Rest.SubMatrix(i+1,m,i,i);
      Rest.SubMatrix(i+1,m,i+1,m) = Rest.SubMatrix(i+1,m,i+1,m) -
                      L.SubMatrix(i+1,m,i,i) * Rest.SubMatrix(i,i,i+1,m);

      //              ( 0 |              0              )
      //      L * U + ( --+--------------------------- ) = A, Invariante
      //              ( 0 | Rest.SubMatrix(i+1,m,i+1,m) )

      }

      U(m,m) = Rest(m,m);

    }

      //      L * U = A

      cout << "\nLU-Zerlegung\nL = \n" << setw(14) << L << endl <<
                "\nU = \n" << setw(14) << U << endl;

                                      // Probe
      Matrix resid;
      resid = L * U - A;

      cout.setf(ios::scientific, ios::floatfield);
      cout << "\n1-Norm der Abweichung: " << resid.Norm1() << "\n";
//    cout.setf(ios::fixed,ios::floatfield);

//    exit(0);                        // unnoetig, da normales Prg.-Ende

}
```

6.2 STL

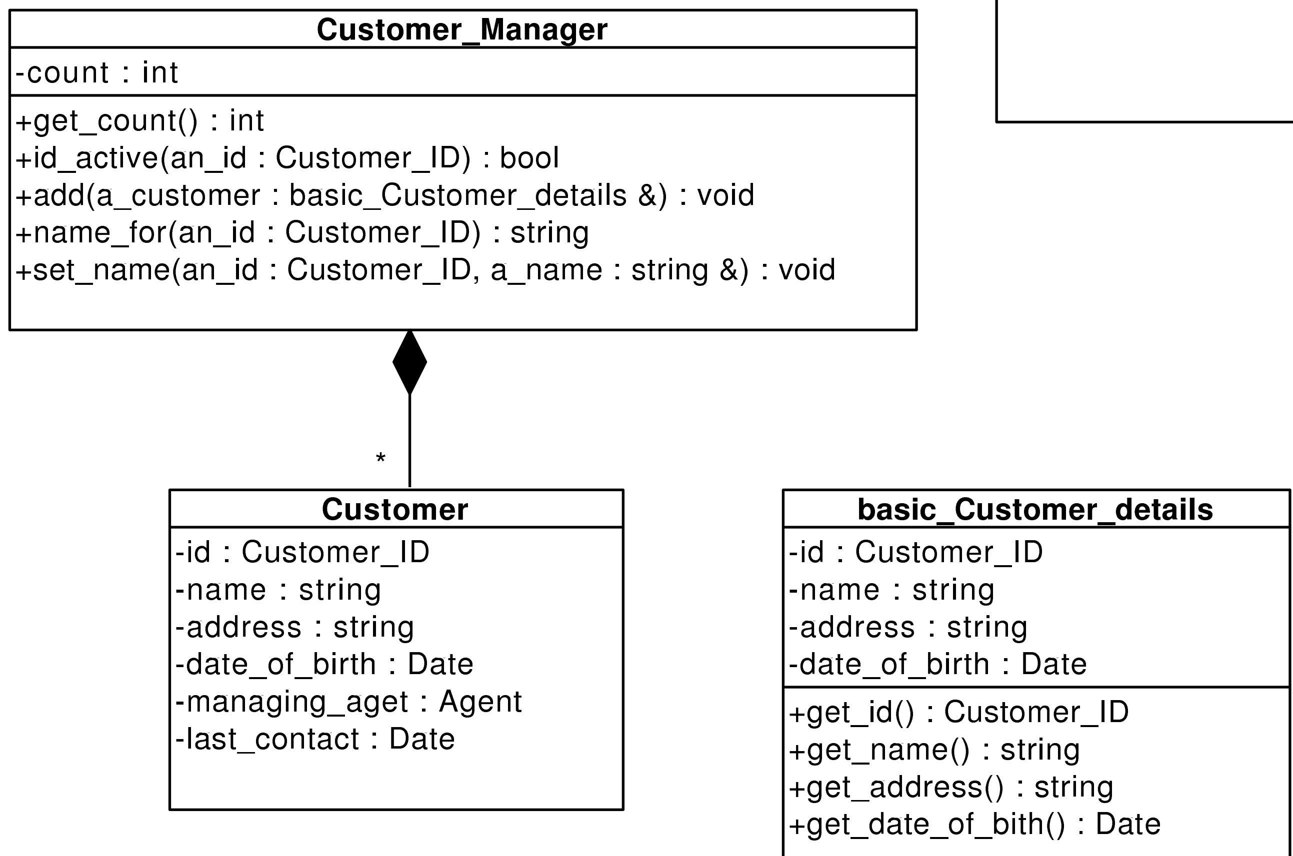
Siehe http://www.sgi.com/tech/stl

6.3 CXSC

Siehe http://www.math.uni-wuppertal.de/∼xsc

7 Degign by Contract, by Example, in C++ with nana

7.1 A first Taste of Design by Contract



```
// Kapitel 7.1.2
//
// c++ -c  DbCrev.cc -I$HOME/include -L$HOME/lib -lnana
//
// eventuell mit
//                  -DWITHOUT_NANA
// und/oder
//                  -DNDEBUG
//
// und/oder
//                  -DEIFELL_CHECK=CHECK_....
//
// --------------------------------------------------------------------
// evtl. myexcept.o mit angeben

#define EIFFEL_DOEND

#ifndef EIFFEL_CHECK
#define EIFFEL_CHECK CHECK_ALL
//                                          Makros CHECK() und folgende
//                    CHECK_LOOP            Makros INVARIANT() und folgende
//                     CHECK_INVARIANT      Methode invariant() und folgende
//                      CHECK_ENSURE        Nachbedingungen und folgende
//                       CHECK_REQUIRE      Vorgedingungen
//                        CHECK_NO
#endif
#include "eiffel.h"
#include "nana.h"



#include <string>
#include <vector>
#include <exception>

using namespace std;

// --------------------------------------------------------------------

class Date
{
private: int day;
private: int month;
private: int year;
```

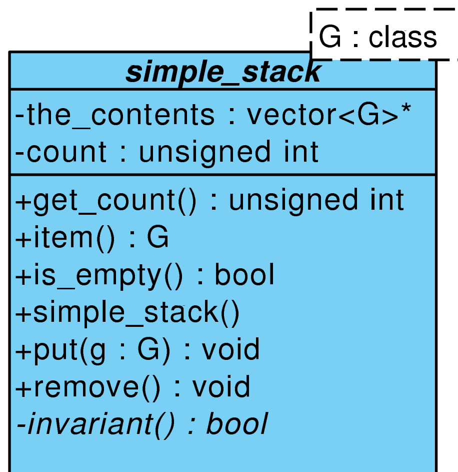
```
public:
  // ...
};

class Customer_ID
{
  // ...
};

class Agent
{
  // ...
};

class Customer_Manager;

// ------------------------------------------------------------------

        class Customer
        {
                private: Customer_ID id;
                private: string name;
                private: string address;
                private: Date date_of_birth;
                private: Agent managing_aget;
                private: Date last_contact;

                private: Customer_Manager* customer_manager;
        };


// -------------------------------------------------------------------

       class basic_Customer_details
        {
                private: Customer_ID id;
                private: string name;
                private: string address;
                private: Date date_of_birth;

                public: Customer_ID get_id() const;
                public: string get_name() const;
                public: string get_address() const;
```

```
                public: Date get_date_of_bith() const;
        };

// --------------------------------------------------------------

        class Customer_Manager
        {
                private: int count;
                private: virtual bool invariant() const;

                private: vector<Customer*> customer;

                public: int get_count() const;
                public: bool id_active(Customer_ID an_id) const;
                public: void add(const basic_Customer_details& a_customer);
                public: string name_for(Customer_ID an_id) const;
                public: void set_name(Customer_ID an_id, const string& a_name);
        };

bool Customer_Manager::invariant() const
{
  return get_count() >= 0;
};


// Anzahl der Kunden, die vom Custom_Manager verwaltet werden
//
int Customer_Manager::get_count() const
DO
        throw "Not yet implemented";
END;

// Existiert ein Kunde mit der Kennung an_id?
//
bool Customer_Manager::id_active(Customer_ID an_id) const
DO
        throw "Not yet implemented";
END;

// Fuege a_customer der Customer_Manager-Datenbasis hinzu
//
void Customer_Manager::add(const basic_Customer_details& a_customer)
DO
    REQUIRE(!id_active(a_customer.get_id()));
```

```
    ID(int old_count = get_count());
        throw "Not yet implemented";
    ENSURE(get_count() == old_count + 1 );
    ENSURE(id_active(a_customer.get_id()));
END;

// was ist der Name des mit an_id gekennzeichneter Kunden?
//
string Customer_Manager::name_for(Customer_ID an_id) const
DO
    REQUIRE(id_active(an_id));
        throw "Not yet implemented";
END;

// Setze/Ändere den Namen des mit an_id gekennzeichneten Kunden auf a_name
//
void Customer_Manager::set_name(Customer_ID an_id, const string& a_name)
DO
    REQUIRE(id_active(an_id));
        throw "Not yet implemented";
    ENSURE(name_for(an_id) == a_name);
END;



// --------------------------------------------------------------------

int main(){

  exit (0);
}
```

7.2 Elementary Principles of Design by Contract

7.2.1 First Trial




```
//
// simple_stack0.cc
//
// g++ -g -o simple_stack0 simple_stack0.cc -I$HOME/include -L$HOME/lib -lnana
//
//
// eventuell mit
//                  -DWITHOUT_NANA
// und/oder
//                  -DNDEBUG
//
// und/oder
//                  -DEIFFEL_CHECK=CHECK_....
//
// --------------------------------------------------------------------
// evtl. myexcept.o mit angeben
// --------------------------------------------------------------------
//
// oder: nana-c++lg simple_stack0.cc
//


#define EIFFEL_DOEND
#ifndef EIFFEL_CHECK
#define EIFFEL_CHECK  CHECK_ALL
//                                          Makros CHECK() und folgende
```

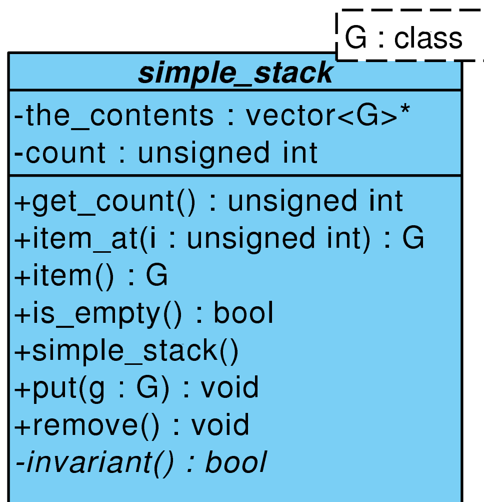
```
//                      CHECK_LOOP              Makros INVARIANT() und folgende
//                       CHECK_INVARIANT        Methode invariant() und folgende
//                        CHECK_ENSURE          Nachbedingungen und folgende
//                         CHECK_REQUIRE        Vorgedingungen
//                          CHECK_NO
#endif


#include <iostream>
#include <vector>

#include <eiffel.h>
#include <nana.h>


using namespace std;


///////////////// class declaration ///////////////////////

template<class G>
class simple_stack{

private:

  vector<G>* the_contents;
  unsigned int count;

public:

  /////////// basic queries:

  unsigned int get_count() const;    // number of items in stack

  G item() const;                    // get top item

  /////////// class invariant:

private:
  virtual bool invariant() const;
public:

  /////////// derived queries:
```

```
  bool is_empty() const;

  /////////// constructors:

  simple_stack();

  /////////// (pure) modificators:

  void put(G g);                  // Push 'g' onto the stack

  void remove();                  // delete the top item

};

///////////////// class definition ///////////////////////

/////////// basic queries:

template<class G>
unsigned int simple_stack<G>::get_count() const{
    return count;
};

template<class G>
G simple_stack<G>::item() const{         // the item on the top of stack
  REQUIRE( /* stack not empty */ get_count() > 0 );
  return the_contents->at(count-1);
};

/////////// class invariant:

template<class G>
bool simple_stack<G>::invariant() const {
  return (count >= 0) && (the_contents != 0);
};
/////////// derived queries:



template<class G>
bool simple_stack<G>::is_empty() const{  // does the stack contain no items?
  bool Result = (get_count() == 0);
  ENSURE( /* consistend with count */       Result == (get_count() == 0) );
  return Result;
```

```
};

/////////// constructors:

template<class G>
simple_stack<G>::simple_stack(){
  count = 0;
  the_contents = new vector<G>(100);

  ENSURE(/* stack is empty */       (get_count() == 0));
  ENSURE(invariant());
};

/////////// (pure) modificators:

template<class G>
void simple_stack<G>::put(G g)     // Push 'g' onto the stack
DO
  ID(int count_old = get_count());
  count++;
  the_contents->at(count-1) = g;
  ENSURE(/* count incremented */     get_count() == count_old + 1 );
  ENSURE(/* g on top */              item() == g );
END;

template<class G>
void simple_stack<G>::remove()     // delete the top item;
DO
  ID(int count_old = get_count());
  REQUIRE(/* stack not empty */      get_count() > 0 );
  count--;
  ENSURE(/* count decremented */     get_count() == count_old - 1 );
END;


/////////////// class test main() program //////////////

int main(){

  cout << "\nTest der Klasse simple_stack: ---------------" << endl;

  simple_stack<long> s;

  s.put(10);
```

```
  cout << s.item() << endl;
  s.put(20);
  cout << s.item() << endl;
  s.put(30);
  std::cout << s.item() << endl;

  std::cout << endl;

  // Exercise 'remove'
  cout << s.item() << endl;
  s.remove();
  cout << s.item() << endl;
  s.remove();
  cout << s.item() << endl;

  // Exercise contract violation
  s.remove();
  cout << s.item() << endl;

  cout << "------------------ Testende ------------------" << endl << endl;

}
```

7.2.2 Redesign




```
//
// simple_stack.cc
//
```

```
// g++ -g -o simple_stack simple_stack.cc -I$HOME/include -L$HOME/lib -lnana
//
//
// eventuell mit
//                  -DWITHOUT_NANA
// und/oder
//                  -DNDEBUG
//
// und/oder
//                  -DEIFFEL_CHECK=CHECK_....
//
// ----------------------------------------------------------------------
// evtl. myexcept.o mit angeben
// ----------------------------------------------------------------------
//
// oder: nana-c++lg simple_stack0.cc
//


#define EIFFEL_DOEND
#ifndef EIFFEL_CHECK
#define EIFFEL_CHECK  CHECK_ALL
//                                                 Makros CHECK() und folgende
//                    CHECK_LOOP                   Makros INVARIANT() und folgende
//                     CHECK_INVARIANT             Methode invariant() und folgende
//                      CHECK_ENSURE               Nachbedingungen und folgende
//                       CHECK_REQUIRE             Vorgedingungen
//                        CHECK_NO
#endif

#include <iostream>
#include <vector>

#include <eiffel.h>
#include <nana.h>


using namespace std;

///////////////// class declaration ///////////////////////

template<class G>
class simple_stack{
```

```
private:

  vector<G>* the_contents;
  unsigned int count;

public:

  /////////// basic queries:

  unsigned int get_count() const;    // number of items in stack

  G item_at(unsigned int i) const;

  /////////// class invariant:

private:
  virtual bool invariant() const;

public:

  /////////// derived queries:

  G item() const;                        // the item on the top of stack

  bool is_empty() const;                 // does the stack contain no items?

  /////////// constructors:

  simple_stack();

  /////////// (pure) modificators:

  void put(G g);                   // Push 'g' onto the stack

  void remove();                  // delete the top item;

};

///////////////// class definition ///////////////////////

/////////// basic queries:

template<class G>
G simple_stack<G>::item_at(unsigned int i) const {
```

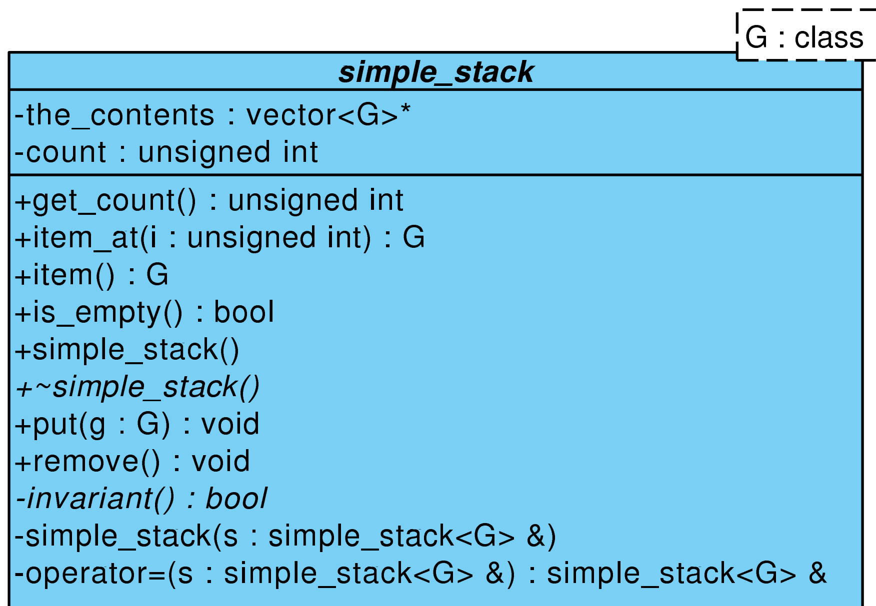
```
  REQUIRE( /* big enough */    i >= 1);
  REQUIRE( /* small enough */  i <= get_count() );

  return the_contents->at(i-1);
};

template<class G>
unsigned int simple_stack<G>::get_count() const{
    return count;
};

/////////// class invariant:

template<class G>
bool simple_stack<G>::invariant() const {
  return (get_count() >= 0) && (the_contents != 0);
};
/////////// derived queries:

template<class G>
G simple_stack<G>::item() const{         // the item on the top of stack
  REQUIRE( /* stack not empty */ get_count() > 0 );

  G result = the_contents->at(count-1);
  ENSURE( /* consistend with item_at() */   result == item_at(get_count()) );
  return result;
};

template<class G>
bool simple_stack<G>::is_empty() const{  // does the stack contain no items?

  bool result = (count == 0);
  ENSURE( /* consistend with count */       result == (get_count() == 0) );
  return result;

};

/////////// constructors:

template<class G>
simple_stack<G>::simple_stack(){

  count = 0;
  the_contents = new vector<G>(100);
```

```
  ENSURE(/* stack is empty */       (get_count() == 0));
  ENSURE(invariant());
};

/////////// (pure) modificators:

template<class G>
void simple_stack<G>::put(G g)     // Push 'g' onto the stack
DO
  ID(unsigned int count_old = get_count());
  count++;
  the_contents->at(count-1) = g;
  ENSURE(/* count incremented */     get_count() == count_old + 1 );
  ENSURE(/* g on top */              item_at(get_count()) == g );
END;

template<class G>
void simple_stack<G>::remove()     // delete the top item;
DO
  ID(unsigned int count_old = get_count());
  REQUIRE(/* stack not empty */      get_count() > 0 );
  count--;
  ENSURE( /* count decremented */        get_count() == count_old - 1 );
END;

//////////////// class test main() program ///////////////

int main(){

  cout << "\nTest der Klasse simple_stack: ----------------" << endl;

  simple_stack<long> s;

  s.put(10);
  cout << s.item() << endl;
  s.put(20);
  cout << s.item() << endl;
  s.put(30);
  cout << s.item() << endl;

  cout << endl;

  // Exercise 'remove'
```

```
  cout << s.item() << endl;
  s.remove();
  cout << s.item() << endl;
  s.remove();
  cout << s.item() << endl;

  // Exercise contract violation
  s.remove();
  cout << s.item() << endl;

  cout << "------------------ Testende ------------------" << endl << endl;

}
```

7.2.3 Destruktor, Kopierkonstruktor und Wertzuweisung

Visual Paradigm for UML Standard Edition(University of Wuppertal)




```
...
#define EIFFEL_DOEND
#define EIFFEL_CHECK  CHECK_ALL
...
#include <iostream>
#include <vector>
#include <eiffel.h>
#include <nana.h>
...
class simple_stack{
```

```
private:

  vector<G>* the_contents;
  unsigned int count;

public:

  /////////// basic queries:

  unsigned int get_count() const;  // number of items in stack

  G item_at(unsigned int i) const;

  /////////// class invariant:

private:
  virtual bool invariant() const;
public:

  /////////// derived queries:

  G item() const;                        // the item on the top of stack

  bool is_empty() const;                 // does the stack contain no items?

  /////////// constructors:

  simple_stack();

  virtual ~simple_stack();               // notwendig wegen new in Konstruktor

private:                                // default copy-Konstruktor
                                        // und default operator=
                                        // fragwuerdig

  simple_stack<G>(const simple_stack<G>& s);
  simple_stack<G>& operator=(const simple_stack<G>& s);

public:

/////////// (pure) modificators:

void put(G g);                   // Push 'g' onto the stack
```

```
void remove();                  // delete the top item;

};
...
template<class G>
G simple_stack<G>::item_at(unsigned int i) const {
  REQUIRE( /* big enough */    i >= 1);
  REQUIRE( /* small enough */  i <= get_count() );
  ...
};
...
template<class G>
unsigned int simple_stack<G>::get_count() const{
    ...
};
...
template<class G>
bool simple_stack<G>::invariant() const {
  ...
};
...
template<class G>
G simple_stack<G>::item() const{         // the item on the top of stack
  REQUIRE( /* stack not empty */ get_count() > 0 );
  ...
  ENSURE( /* consistend with item_at() */   result == item_at(get_count()) );
  ...
};
...
template<class G>
bool simple_stack<G>::is_empty() const{  // does the stack contain no items?
  ...
  ENSURE( /* consistend with count */       result == (get_count() == 0) );
  ...
};
...
template<class G>
simple_stack<G>::simple_stack(){
  ...
  ENSURE(/* stack is empty */       (get_count() == 0));
  ENSURE(invariant());
};
...
```

```
template<class G>
simple_stack<G>::~simple_stack<G>(){
  REQUIRE (/* pointer not null */ the_contents != 0);
  ...
};
...
template<class G>
void simple_stack<G>::put(G g)     // Push 'g' onto the stack
DO
  ID(int count_old = get_count());
  ID(vector<G> old(*the_contents));
  ...
  ENSURE(/* count incremented */     get_count() == count_old + 1 );
  ENSURE(/* g on top */              item_at(get_count()) == g );
  ENSURE(/* old contents unchanged */
                A(int k=1, k<get_count(), k++, item_at(k)== old.at(k-1) ));
  ...
END;
...
template<class G>
void simple_stack<G>::remove()     // delete the top item;
DO
  REQUIRE(/* stack not empty */      get_count() > 0 );
  ID(int count_old = get_count());
  ID(vector<G> old(*the_contents));
  ...
  ENSURE( /* count decremented */        get_count() == count_old - 1 );
  ENSURE( /* consistancy with item() */  item_at(get_count()) == old.at(count_old-2) );
  ENSURE( /* old contents unchanged */
                A(int k=1, k<get_count(), k++, item_at(k)== old.at(k-1) ));
END;
...
int main(){
...
s.put(10);
cout << s.item() << endl;
s.put(20);
cout << s.item() << endl;
s.put(30);
cout << s.item() << endl;

cout << endl;

// Exercise 'remove'
```

```
cout << s.item() << endl;
s.remove();
cout << s.item() << endl;
s.remove();
cout << s.item() << endl;

// Test copy-Konstruktor ...
//
// simple_stack<long> s2(s);
simple_stack<long> s3;
// s3 = s;

cout << "------------------ Testende ------------------" << endl << endl;

}
```

Voller Code: simple_stack4.cc

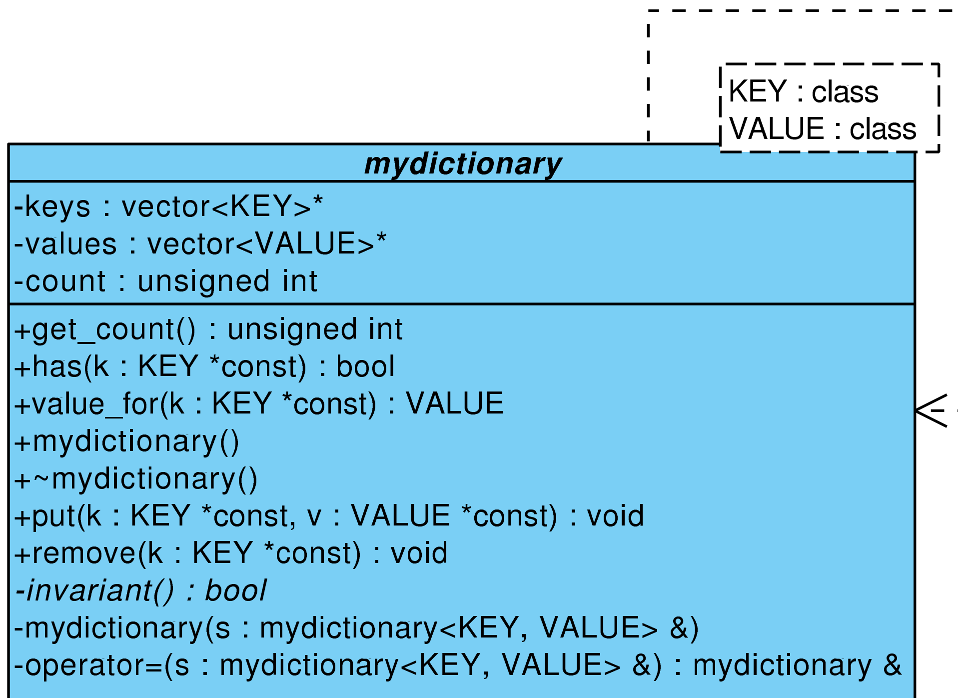
7.3 Applying the Six Principles

PbC-Regeln:

- Vermeide statusändernde Methoden, die einen Wert liefern! (Observator **oder** Modifikator)
- Unterscheide grundlegende von abgeleiteten (redundanten) Observatoren.
- Schreibe für jeden abgeleiteten Observator eine Nachbedingung mit Hilfe der (aller) grundlegenden Observatoren.
- Schreibe für jeden Modifikator Nachbedingungen, die mit Hilfe der (aller) grundlegenden Observatoren den Inhalt des Klassenexemplars nach Methodenende in seiner Relation zum Exemplarinhalt bei Methodenbeginn **exakt** beschreiben. Nutze implizite (als Kommentar) oder explizite Frame-Bedingungen.
- Schreibe für alle Methoden Vorbedingungen (an Parameter bzw. Exemplarinhalt).
- Schreibe und benutze Invarianten, die gültige von ungültigen Exemplaren trennen.

7.3.1 Design und Contracts

Visual Paradigm for UML Standard Edition(University of Wuppertal)



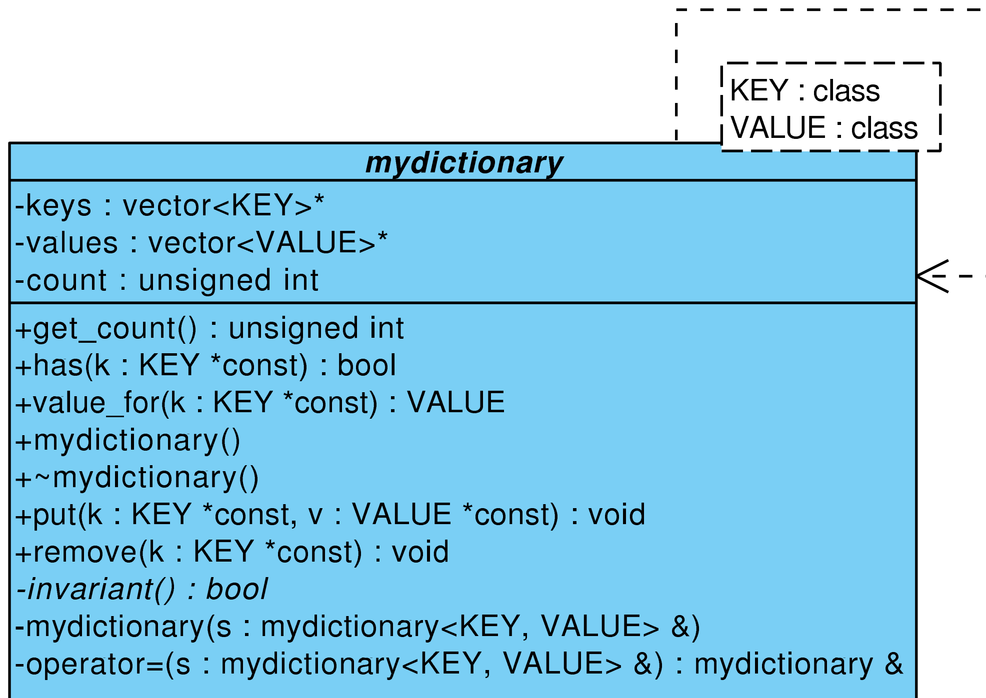

```
...
#define EIFFEL_DOEND
#define EIFFEL_CHECK  CHECK_ALL
```

```
...
#include <iostream>
#include <vector>
#include <eiffel.h>
#include <nana.h>
...
class mydictionary{

  vector<KEY>* keys;
  vector<VALUE>* values;
  unsigned int count;

public:
  //////////////// basic queries:

  unsigned int get_count() const;     // number of key/value-pairs in dict.

  bool has(const KEY *const k) const;              // key in dictionary?

  VALUE value_for(const KEY *const k) const;      // lookup value for key

  /////////////// class invariant:
private:
  virtual bool invariant() const;
public:
  /////////////// derived queries:

  // not yet necessary

  /////////////// constructors and destructors:

  mydictionary();

  ~mydictionary();

  /////////////// deactivate default copy constructor and operator=

private:
  mydictionary(const mydictionary<KEY, VALUE>& s);
  mydictionary& operator=(const mydictionary<KEY, VALUE>& s);
public:

  /////////////// (pure) modificators
```

```
  void put(const KEY *const k, const VALUE *const v);
                                            // put key/value-pair in dict.

  void remove(const KEY *const k);          // remove key/value-pair

};
...
template<class KEY, class VALUE>
unsigned int mydictionary<KEY, VALUE>::get_count() const{
  ...
};
...
template<class KEY, class VALUE>
bool mydictionary<KEY, VALUE>::has(const KEY *const k) const {
  REQUIRE( /* key exists */      k != 0);
  ...
  ENSURE( /* consistent with count */  (get_count() != 0) || ! result);
  ...
};
...
template<class KEY, class VALUE>
VALUE mydictionary<KEY, VALUE>::value_for(const KEY *const k) const{
  REQUIRE(/* key exists */       k != 0);
  REQUIRE(/* key in dict. */     has(k));
  ...
};
...
template <class KEY, class VALUE>
bool mydictionary<KEY, VALUE>::invariant() const{
  ...
};
...
template<class KEY, class VALUE>
mydictionary<KEY, VALUE>::mydictionary(){
  ...
  ENSURE(invariant());
};
...
template<class KEY, class VALUE>
mydictionary<KEY, VALUE>::~mydictionary(){
  REQUIRE(/* keys exist */      keys != 0);
  REQUIRE(/* values exist */    values != 0);
  ...
};
```

```
...
template<class KEY, class VALUE>
void mydictionary<KEY, VALUE>::put(const KEY *const k, const VALUE *const v)
DO
  ID(unsigned int count_old=get_count());
  REQUIRE(/* key exists */         k != 0);
  REQUIRE(/* key not in dict. */   ! has(k));
  ...
  ENSURE(/* count incremented */   get_count() == count_old + 1);
  ENSURE(/* key in dict. */        has(k) );
  ENSURE(/* correct value */       value_for(k) == *v);
END;
...
template<class KEY, class VALUE>
void mydictionary<KEY, VALUE>::remove(const KEY *const k)
DO
  ID(unsigned int count_old = get_count());
  REQUIRE(/* key exists */      k != 0);
  REQUIRE(/* key in dict. */    has(k));
  ...
  ENSURE(/* count decremented */ get_count() == count_old - 1);
  ENSURE(/* key not in dict. */ ! has(k));
  ...
END;
...
int main(){
  ...
}
```

7.3.2 Implementierung und Tests




```
// dictionary2.cc
//
// g++ -g -o dictionary2 dictionary2.cc -I$HOME/include -L$HOME/lib -lnana
//
//
// eventuell mit
//                  -DWITHOUT_NANA
// und/oder
//                  -DNDEBUG
//
// und/oder
//                  -DEIFFEL_CHECK=CHECK_....
//
// -----------------------------------------------------------------
// evtl. myexcept.o mit angeben
// -----------------------------------------------------------------
//
// oder: nana-c++lg simple_stack4.cc
//


#define EIFFEL_DOEND
```

```
#ifndef EIFFEL_CHECK
#define EIFFEL_CHECK  CHECK_ALL
//                                                Makros CHECK() und folgende
//                    CHECK_LOOP                  Makros INVARIANT() und folgende
//                     CHECK_INVARIANT            Methode invariant() und folgende
//                      CHECK_ENSURE              Nachbedingungen und folgende
//                       CHECK_REQUIRE            Vorgedingungen
//                        CHECK_NO
#endif

#include <iostream>
#include <vector>

#include <eiffel.h>
#include <nana.h>


using namespace std;

///////////////// class declaration ///////////////////////

template<class KEY, class VALUE>
class mydictionary{

  vector<KEY>* keys;
  vector<VALUE>* values;
  unsigned int count;

public:
  //////////////// basic queries:

  unsigned int get_count() const;     // number of key/value-pairs in dict.

  bool has(const KEY *const k) const;              // key in dictionary?

  VALUE value_for(const KEY *const k) const;      // lookup value for key

  /////////////// class invariant:
private:
  virtual bool invariant() const;
public:
  /////////////// derived queries:

  // not yet necessary
```

```
  /////////////// constructors and destructors:

  mydictionary();

  ~mydictionary();

  /////////////// deactivate default copy constructor and operator=

private:
  mydictionary(const mydictionary<KEY, VALUE>& s);
  mydictionary& operator=(const mydictionary<KEY, VALUE>& s);
public:

  /////////////// (pure) modificators

  void put(const KEY *const k, const VALUE *const v);
                                            // put key/value-pair in dict.

  void remove(const KEY *const k);          // remove key/value-pair

};

///////////////// class definition ///////////////////////

//////////////// basic queries:

template<class KEY, class VALUE>
unsigned int mydictionary<KEY, VALUE>::get_count() const{
  return count;
};

template<class KEY, class VALUE>
bool mydictionary<KEY, VALUE>::has(const KEY *const k) const {

  REQUIRE( /* key exists */      k != 0);

  unsigned int i = 0;
  do {
    i++;
  }while((i <= count) && (keys->at(i-1) != *k) );

  bool result = (i <= count) && (keys->at(i-1) == *k);
  ENSURE( /* consistent with count */  (get_count() != 0) || ! result);
```

```
  return result;
};

template<class KEY, class VALUE>
VALUE mydictionary<KEY, VALUE>::value_for(const KEY *const k) const{
  REQUIRE(/* key exists */       k != 0);
  REQUIRE(/* key in dict. */     has(k));

  unsigned int i = 0;
  do {
    i++;
  }while(keys->at(i-1) != *k);
  return values->at(i-1);

};

////////////// class invariant:

template <class KEY, class VALUE>
bool mydictionary<KEY, VALUE>::invariant() const{

  return  (get_count() >= 0) && (keys != 0) && (values != 0);

};

/////////////// constructors and destructors:

template<class KEY, class VALUE>
mydictionary<KEY, VALUE>::mydictionary(){

  count = 0;
  keys = new vector<KEY>(100);
  values = new vector<VALUE>(100);
  ENSURE(invariant());
};

template<class KEY, class VALUE>
mydictionary<KEY, VALUE>::~mydictionary(){

  REQUIRE(/* keys exist */      keys != 0);
  REQUIRE(/* values exist */    values != 0);
  delete keys;
  delete values;
};
```

```
/////////////// (pure) modificators

template<class KEY, class VALUE>
void mydictionary<KEY, VALUE>::put(const KEY *const k, const VALUE *const v)
DO
  ID( unsigned int count_old=get_count());
  REQUIRE(/* key exists */        k != 0);
  REQUIRE(/* key not in dict. */   ! has(k));

  count++;
  keys->at(count-1) = *k;
  values->at(count-1) = *v;

  ENSURE(/* count incremented */   get_count() == count_old + 1);
  ENSURE(/* key in dict. */        has(k) );
  ENSURE(/* correct value */       value_for(k) == *v);
END;

template<class KEY, class VALUE>
void mydictionary<KEY, VALUE>::remove(const KEY *const k)
DO
  ID(unsigned int count_old = get_count());
  REQUIRE(/* key exists */      k != 0);
  REQUIRE(/* key in dict. */    has(k));

  unsigned int i = 0;
  do {
    i++;
  }while(keys->at(i-1) != *k);
  CHECK(i <= count);
  if (i < count){
    keys->at(i-1) = keys->at(count-1);
    values->at(i-1) = keys->at(count-1);
  };
  count--;

  ENSURE(/* count decremented */  get_count() == count_old - 1);
  ENSURE(/* key not in dict. */ ! has(k));
  // REQUIRE precondition for value_for() is false for k
END;

//////////////// class test main() program //////////////
```

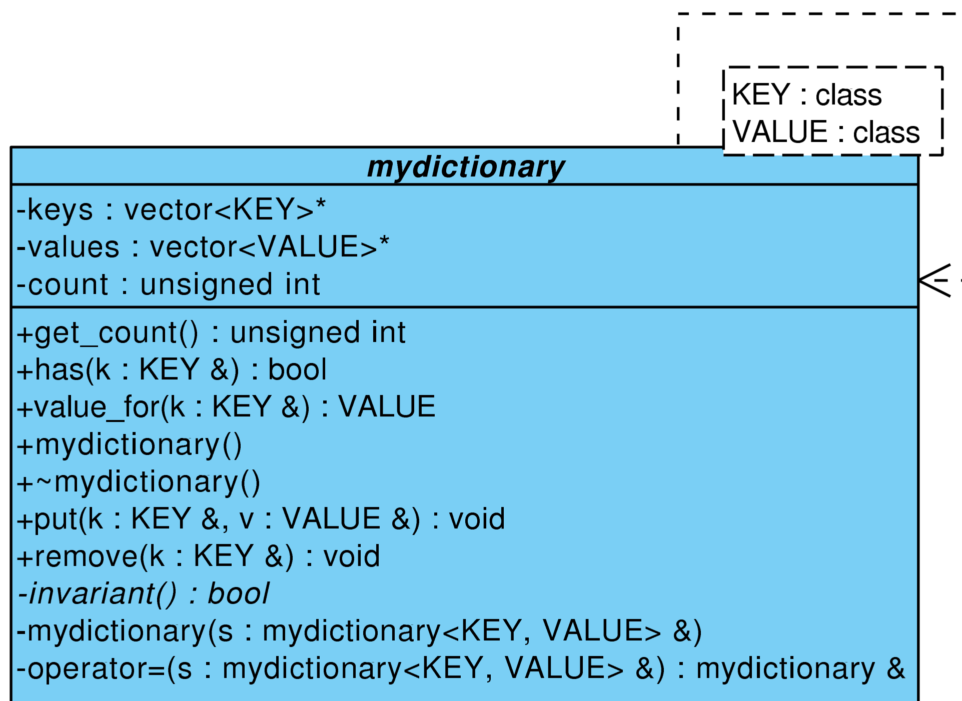
```
int main(){

  cout << "Test der Klasse dictionary: ----------------" << endl;

  mydictionary<string, string> d;

  string k("Denver"); string v("Colorado");
  d.put(&k, &v);
  k = "London"; v = "Ontario";
  d.put(&k, &v);
  k = "Austin"; v = "Texas";
  d.put(&k, &v);
  k = "Boston"; v = "Massachusetts";
  d.put(&k, &v);
  k = "Mobile"; v = "Alabama";
  d.put(&k, &v);

  cout << endl << "List of 5 pairs:" << endl;
  {
   string k1("Denver");
   if (d.has(&k1)) cout << k1 << " " << d.value_for(&k1) << endl;
   string k2("London");
   if (d.has(&k2)) cout << k2 << " " << d.value_for(&k2) << endl;
   string k3("Austin");
   if (d.has(&k3)) cout << k3 << " " << d.value_for(&k3) << endl;
   string k4("Boston");
   if (d.has(&k4)) cout << k4 << " " << d.value_for(&k4) << endl;
   string k5("Mobile");
   if (d.has(&k5)) cout << k5 << " " << d.value_for(&k5) << endl;
  };

  string l2("Mobile");
  d.remove(&l2);
  cout << endl << "List of " << d.get_count() << " pairs, last removed:" << endl;
   {
   string k1("Denver");
   if (d.has(&k1)) cout << k1 << " " << d.value_for(&k1) << endl;
   string k2("London");
   if (d.has(&k2)) cout << k2 << " " << d.value_for(&k2) << endl;
   string k3("Austin");
   if (d.has(&k3)) cout << k3 << " " << d.value_for(&k3) << endl;
   string k4("Boston");
   if (d.has(&k4)) cout << k4 << " " << d.value_for(&k4) << endl;
   string k5("Mobile");
```

```
 if (d.has(&k5)) cout << k5 << " " << d.value_for(&k5) << endl;
};

string l1("Denver");
d.remove(&l1);
cout << endl << "List of " << d.get_count() << " pairs, 1st removed:" << endl;
 {
 string k1("Denver");
 if (d.has(&k1)) cout << k1 << " " << d.value_for(&k1) << endl;
 string k2("London");
 if (d.has(&k2)) cout << k2 << " " << d.value_for(&k2) << endl;
 string k3("Austin");
 if (d.has(&k3)) cout << k3 << " " << d.value_for(&k3) << endl;
 string k4("Boston");
 if (d.has(&k4)) cout << k4 << " " << d.value_for(&k4) << endl;
 string k5("Mobile");
 if (d.has(&k5)) cout << k5 << " " << d.value_for(&k5) << endl;
};

string l3("Austin");
d.remove(&l3);
cout << endl << "List of " << d.get_count() << " pairs, middle removed:" << endl;
{
 string k1("Denver");
 if (d.has(&k1)) cout << k1 << " " << d.value_for(&k1) << endl;
 string k2("London");
 if (d.has(&k2)) cout << k2 << " " << d.value_for(&k2) << endl;
 string k3("Austin");
 if (d.has(&k3)) cout << k3 << " " << d.value_for(&k3) << endl;
 string k4("Boston");
 if (d.has(&k4)) cout << k4 << " " << d.value_for(&k4) << endl;
 string k5("Mobile");
 if (d.has(&k5)) cout << k5 << " " << d.value_for(&k5) << endl;
};

// d.remove(&string("Austin"));

d.put(&k, &v);
cout << endl << "List of " << d.get_count() << " pairs after put:" << endl;
{
 string k1("Denver");
 if (d.has(&k1)) cout << k1 << " " << d.value_for(&k1) << endl;
 string k2("London");
 if (d.has(&k2)) cout << k2 << " " << d.value_for(&k2) << endl;
```

```
    string k3("Austin");
    if (d.has(&k3)) cout << k3 << " " << d.value_for(&k3) << endl;
    string k4("Boston");
    if (d.has(&k4)) cout << k4 << " " << d.value_for(&k4) << endl;
    string k5("Mobile");
    if (d.has(&k5)) cout << k5 << " " << d.value_for(&k5) << endl;
  };

  //  d.put(&k, &v);

  cout << "------------------ Testende ------------------" << endl;

}
```

Zum Download: dictionary2.cc

7.3.3 konstante Referenzparameter/private Hilfsmethoden für die Spezifikation/old-Wert durch Kopie in Form eines geeigneten STL-Container-Exemplars




```
...
#define EIFFEL_DOEND
#define EIFFEL_CHECK  CHECK_ALL
```

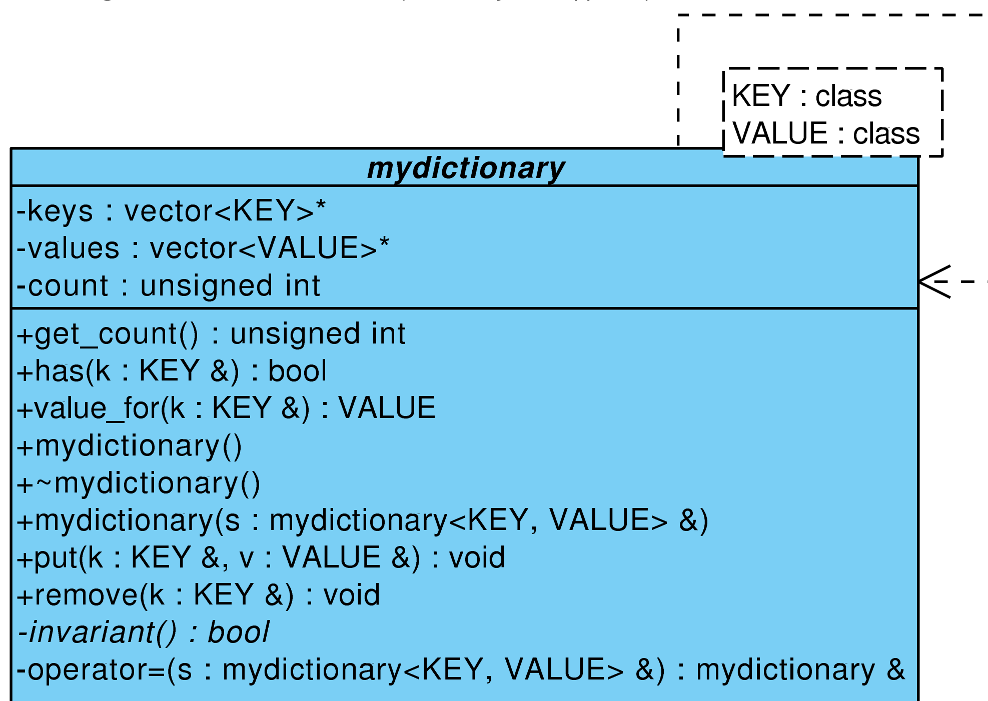
```
...
#include <iostream>
#include <vector>
#include <set>
#include <eiffel.h>
#include <nana.h>
...
// private Spezifikations-Hilfsmethoden: fuer STL-Container
template <class T>
set<T> operator+(const set<T>& s, const T& e){
    ...
};
...
template <class T>
set<T> operator-(const set<T>& s, const T& e){
    ...
};
...
class mydictionary{

  vector<KEY>* keys;
  vector<VALUE>* values;
  unsigned int count;

public:
  //////////////// basic queries:

  unsigned int get_count() const;     // number of key/value-pairs in dict.

  bool has(const KEY& k) const;              // key in dictionary?

  VALUE value_for(const KEY& k) const;      // lookup value for key

  /////////////// class invariant:
private:
  virtual bool invariant() const;
public:

  /////////////// derived queries:
  // not yet necessary

  /////////////// constructors and destructors:

  mydictionary();
```

```
  ~mydictionary();

  /////////////// deactivate default copy constructor and operator=

private:
  mydictionary(const mydictionary<KEY, VALUE>& s);
  mydictionary& operator=(const mydictionary<KEY, VALUE>& s);
public:

  /////////////// (pure) modificators

  void put(const KEY& k, const VALUE& v);
                                           // put key/value-pair in dict.

  void remove(const KEY& k);               // remove key/value-pair

};
...

/// basic queries:

template<class KEY, class VALUE>
unsigned int mydictionary<KEY, VALUE>::get_count() const{
  ...
};
...
template<class KEY, class VALUE>
bool mydictionary<KEY, VALUE>::has(const KEY& k) const {
  ...
  ENSURE( /* consistent with count */  (get_count() != 0) || ! result);
  ...
};
...
template<class KEY, class VALUE>
VALUE mydictionary<KEY, VALUE>::value_for(const KEY& k) const{
  REQUIRE(/* key in dict. */     has(k));
  ...
};
...

/// Invariante:

template <class KEY, class VALUE>
```

```
bool mydictionary<KEY, VALUE>::invariant() const{
  ...
};
...
/// Konstruktor/Destruktor:

template<class KEY, class VALUE>
mydictionary<KEY, VALUE>::mydictionary(){
  ...
  ENSURE(invariant());
  ENSURE(count == 0);
};
...
template<class KEY, class VALUE>
mydictionary<KEY, VALUE>::~mydictionary(){
  REQUIRE(/* keys exist */      keys != 0);
  REQUIRE(/* values exist */    values != 0);
  ...
};
...
/// Modifikatoren:

template<class KEY, class VALUE>
void mydictionary<KEY, VALUE>::put(const KEY& k, const VALUE& v)
DO
  REQUIRE(/* key not in dict. */   ! has(k));
  ID(unsigned int count_old=get_count());
  ID(set<KEY> old_keys(keys->begin(), keys->begin()+count_old));
  ...
  ENSURE(/* count incremented */    get_count() == count_old + 1);
  ENSURE(/* key in dict. */         has(k) );
  ENSURE(/* correct value */        value_for(k) == v);
  ID(set<KEY> new_keys(keys->begin(),keys->begin()+count));
  ENSURE(old_keys + k == new_keys);
  ...
END;
...
template<class KEY, class VALUE>
void mydictionary<KEY, VALUE>::remove(const KEY& k)
DO
  REQUIRE(/* key in dict. */    has(k));
  ID(unsigned int count_old = get_count());
  ID(set<KEY> old_keys(keys->begin(), keys->begin()+count_old));
  ...
```

```
  ENSURE(/* count decremented */  get_count() == count_old - 1);
  ENSURE(/* key not in dict. */   ! has(k));
  ...
  ID(set<KEY> new_keys(keys->begin(),keys->begin()+count));
  ENSURE(old_keys - k == new_keys);
  ...
END;
...
int main(){
  ...
}
```

7.3.4 old-Wert durch den Kopierkonstruktor



Mit Hilfe des Kopierkonstruktors

```
...
template<class KEY, class VALUE>
mydictionary<KEY, VALUE>::mydictionary(const mydictionary<KEY, VALUE>& s){
    count = s.count;
    keys = new vector<KEY>(100);
    values = new vector<VALUE>(100);
    for (int i=1; i<=count; i++){
        keys->at(i-1) = s.keys->at(i-1);
        values->at(i-1) = s.values->at(i-1);
```

```
    };
    ENSURE(count == s.count);
    ENSURE(A(int i=1, i<=count, i++, keys->at(i-1) == s.keys->at(i-1)));
    ENSURE(A(int i=1, i<=count, i++, values->at(i-1) == s.values->at(i-1)));
    ENSURE(invariant());
};
...
```

können Sie den alten Exemplarwert vollständig in einem Stück als konstante Variable memorieren und in den Nachbedingungen aller Modifikatoren benutzen.

AUFGABE: Ändern Sie dictionary4.cc so ab, dass put() und remove() so spezifiziert wird!

Es ist noch unschön, dass die Nachbedingungen des Kopierkonstruktors auf die Implemantierungsdetails Bezug nehmen. Deshalb:

7.3.5 Redesign

Ein nächster Schritt sollte die Einführung eines neuen grundlegenden Observators `set<KEY> keys()` sein. Führen Sie die notwendigen Änderungen durch!

Alternativ könnte man einen Iterator in der neuen Containerklasse mydictionary implementieren (siehe folgender Abschnitt).

7.4 Immutable Lists

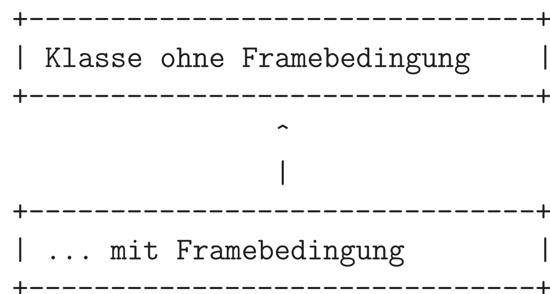
... entfällt in C++ wegen der Existenz der STL

7.5 Using Immutable Lists

... entfällt in C++ wegen der Existenz der STL

Leitlinien:

- Nutze technische Einschränkungen, wo immer erforderlich: z.B. Zeiger != 0, nichtleere Container, nichtidentische Exemplare, ...
- In Vorbedingungen genutzte Observatoren sollten effizient berechnet werden. Notfalls führe neue effiziente abgeleitete Observatoren ein und benutze sie in den Vorbedingungen. Die neuen effizienten Observatoren benötigen Nachbedingungen, die ihre Konsistenz zu den grundlegenden Observatoren sicherstellen.
- Attribute haben keine Nachbedingungen. Benutze deshalb die Klasseninvariante für Contracts (oder Nachbedingungen von get-Methoden).
- Nachbedingungen von virtuellen Methoden sollten die Form
  `ENSURE (!Vorbedingung || Nachbedingung)` haben;
  Invarianten sollten `protected` als `virtual bool invariant() const` deklariert werden.
- Nutze die Vererbung:



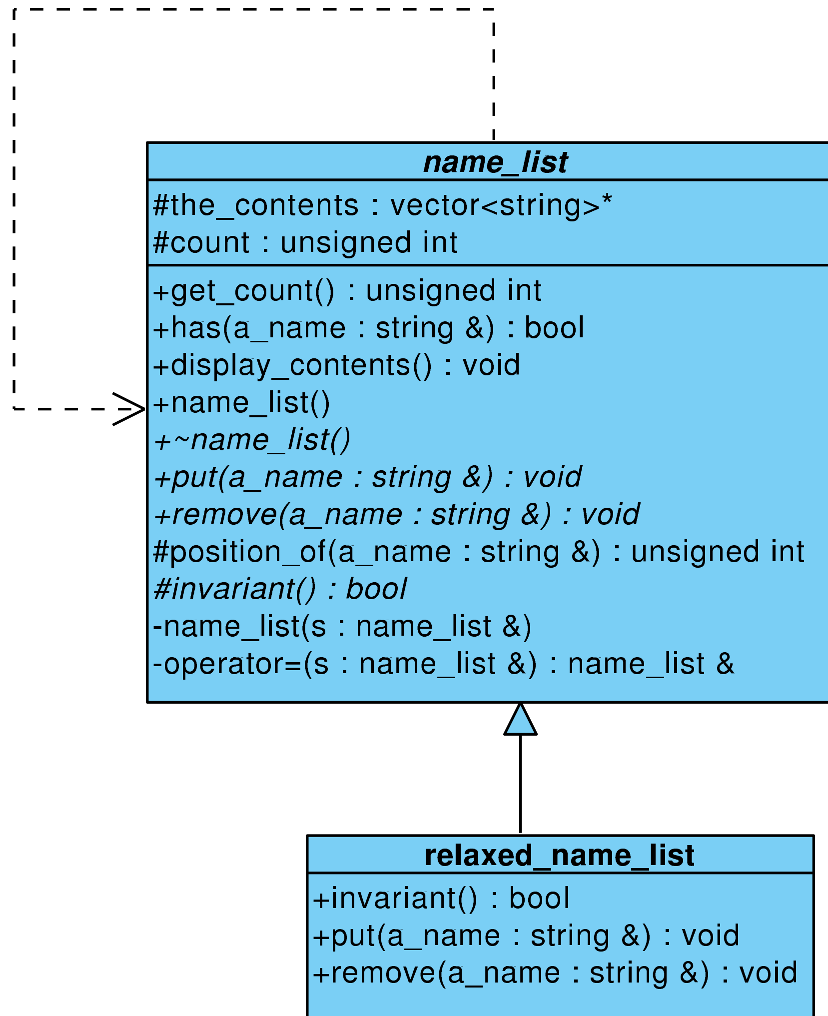
  um dem wiederverwendenden Nutzer die Wahl zwischen der Verwendung der effizienten oberen oder sicheren unteren Klasse zu überlassen.
- Nutze die Vererbung analog z.B. für:

```
Klasse ohne Framebedingungen

Klasse mit Framebedingungen

... mit unzugänglichen (private) Observatoren,
    die weiter oben lediglich für die Contracts
    genutzt werden
```

```
... mit benutzerfreundlichen Methodenvarianten
    ohne Vorbedingungen

...
```

7.6 Subcontracting in Design by Contract in Nana

7.6.1 name_list-Design (Subcontracting)




```
...
#define EIFFEL_DOEND
#define EIFFEL_CHECK  CHECK_ALL
...
#include <iostream>
#include <vector>
```

```
#include <eiffel.h>
#include <nana.h>
...
class name_list{

protected:

  vector<string>* the_contents;
  unsigned int count;

  // private support function

  unsigned int position_of(const string& a_name) const;

public:

  /////////// basic queries:

  unsigned int get_count() const;        // number of items in stack

  bool has(const string& a_name) const;

  /////////// class invariant:

protected:
  virtual bool invariant() const;
public:

  /////////// derived queries:

  void display_contents() const;         // print contents of list to cout

  /////////// constructors:

  name_list();

  virtual ~name_list();              // notwendig wegen new in Konstruktor

private:                             // diable default methods

  name_list(const name_list& s);
  name_list& operator=(const name_list& s);

public:
```

```
/////////// (pure) modificators:

  virtual void put(const string& a_name);       // Push a_name into list

  virtual void remove(const string& a_name);    // delete a_name in list

};
...
unsigned int name_list::position_of(const string& a_name) const{
    ...
    ENSURE(/* a_name in list */
                                ((1<=result)&&(result<=get_count())&&
                                 (the_contents->at(result-1)==a_name)) ||
           /* otherwise: */
                                (0 == result)                           );
    ...
};
...
unsigned int name_list::get_count() const{    // number of items in stack
    ...
    ENSURE(result == count);
    ...
};
bool name_list::has(const string& a_name) const{
    ...
    ENSURE((get_count()>0) || !result);
    ...
};
...
bool name_list::invariant() const{
    ...
};
...
void name_list::display_contents() const{     // print contents of list to cout
    ...
};
...
name_list::name_list(): count(0), the_contents(new vector<string>(100)){
    ENSURE(invariant());
};
name_list::~name_list(){                  // notwendig wegen new in Konstruktor
    ...
};
```

```
...
void name_list::put(const string& a_name)    // Push a_name into list
DO
   ID(bool pre = !has(a_name));
   ID( unsigned int count_old = get_count());
   REQUIRE(/* name not in list */    pre );
   ...
   ENSURE(has(a_name));
   ENSURE((!pre) || (get_count() == count_old + 1));
   ...
END;
void name_list::remove(const string& a_name) // delete a_name in list
DO
   ID(bool pre(has(a_name)));
   ID( unsigned int count_old = get_count());
   REQUIRE(/* name in list */     has(a_name));
   ...
   ENSURE(! has(a_name));
   ENSURE((!pre) || (get_count() == count_old -1));
   ...
END;
...
class relaxed_name_list : public name_list{

public:

    ///////// invariant:

    virtual bool invariant() const;

    ///////// (pure) modificators: (redefined)

    virtual void put(const string& a_name);       // Push a_name into list

    virtual void remove(const string& a_name);    // delete a_name in list

};
...
bool relaxed_name_list::invariant() const{
    ...
};
...
void relaxed_name_list::put(const string& a_name)    // Push a_name into list
DO
```

```
   ID(bool pre_parent(!has(a_name)));
   ID(unsigned int count_old = get_count());
   REQUIRE(/* nothing */  true);       // pre_parent || has(a_name)
   ...
   ENSURE(has(a_name));
   ENSURE((!pre_parent) || (get_count() == count_old + 1));  // &&
   ENSURE(  pre_parent  || (get_count() == count_old));
   ...
END;
void relaxed_name_list::remove(const string& a_name) // delete a_name in list
DO
   ID(bool pre_parent(has(a_name)));
   ID(unsigned int count_old = get_count());
   REQUIRE(/* nothing */ true);        // pre_parent || !has(a_name)
   ...
   ENSURE(! has(a_name));
   ENSURE((!pre_parent) || (get_count() == count_old -1));  // &&
   ENSURE(  pre_parent  || (get_count() == count_old));
   ...
END;
...
int main(){
    ...
}
```

Zum Download: name_list.cc

7.6.2 Implementierung und Tests

... der Contracts und der Prototypinstallation:

```
...
unsigned int name_list::position_of(const string& a_name) const{

    unsigned int index(1);
    unsigned int result;
    for(; (index<=count) && (the_contents->at(index-1)!=a_name); index++);
    if (index <= count)
        result = index;
    else
        result = 0;

    ENSURE(/* a_name in list */
```

```
                                ((1<=result)&&(result<=get_count())&&
                                 (the_contents->at(result-1)==a_name)) ||
           /* otherwise: */
                                (0 == result)                         );
    return result;
};
...
/////////// basic queries:

unsigned int name_list::get_count() const{    // number of items in stack

    unsigned int result = count;
    return result;
};

bool name_list::has(const string& a_name) const{

    bool result = (position_of(a_name) > 0);
    ENSURE((get_count()>0) || !result);   // consistency
    return result;
};
...

/////////// class invariant:

bool name_list::invariant() const{
    return (count >= 0) && (the_contents != 0);
};

...
/////////// derived queries:

void name_list::display_contents() const{     // print contents of list to cout

    cout << endl << "Anzahl der Listenelemente: " << count << endl;
    for (unsigned int i=1; i<=count; i++)
        cout << "  " << the_contents->at(i-1);
    cout << endl << endl;
    // ENSURE('Drucke alle Namen in Name_list');
};

/////////// constructors:

name_list::name_list(): count(0), the_contents(new vector<string>(100)){
```

```
    ENSURE(invariant());
};

name_list::~name_list(){                     // notwendig wegen new in Konstruktor
    delete the_contents;
};

/////////// (pure) modificators:

void name_list::put(const string& a_name)    // Push a_name into list
DO
   ID(bool pre(!has(a_name)));
   ID(unsigned int count_old = get_count());

   REQUIRE(/* name not in list */    pre);


   count++;
   the_contents->at(count-1) = a_name;

   ENSURE(has(a_name));
   ENSURE( (!pre) || (get_count() == count_old + 1));
   // ENSURE: Fuer alle s in list_old: has(s)
   // ENSURE: Fuer alle s in list:     (s == a_name) || s in list_old
END;
...
//////////////// class test main() program //////////////

int main(){

    cout << "\nTest der Klasse name_list: ----------------" << endl;

    { name_list s;
    s.display_contents();

    s.put("Richard"); s.display_contents();
    //s.put("Richard"); s.display_contents(); // Test fuer pre-Verletzung
    s.put("Helen"); s.display_contents();
    s.put("Yu"); s.display_contents();
    s.put("Jim"); s.display_contents();
    s.put("Chen"); s.display_contents();
...
```

Zum Download: name_list2.cc

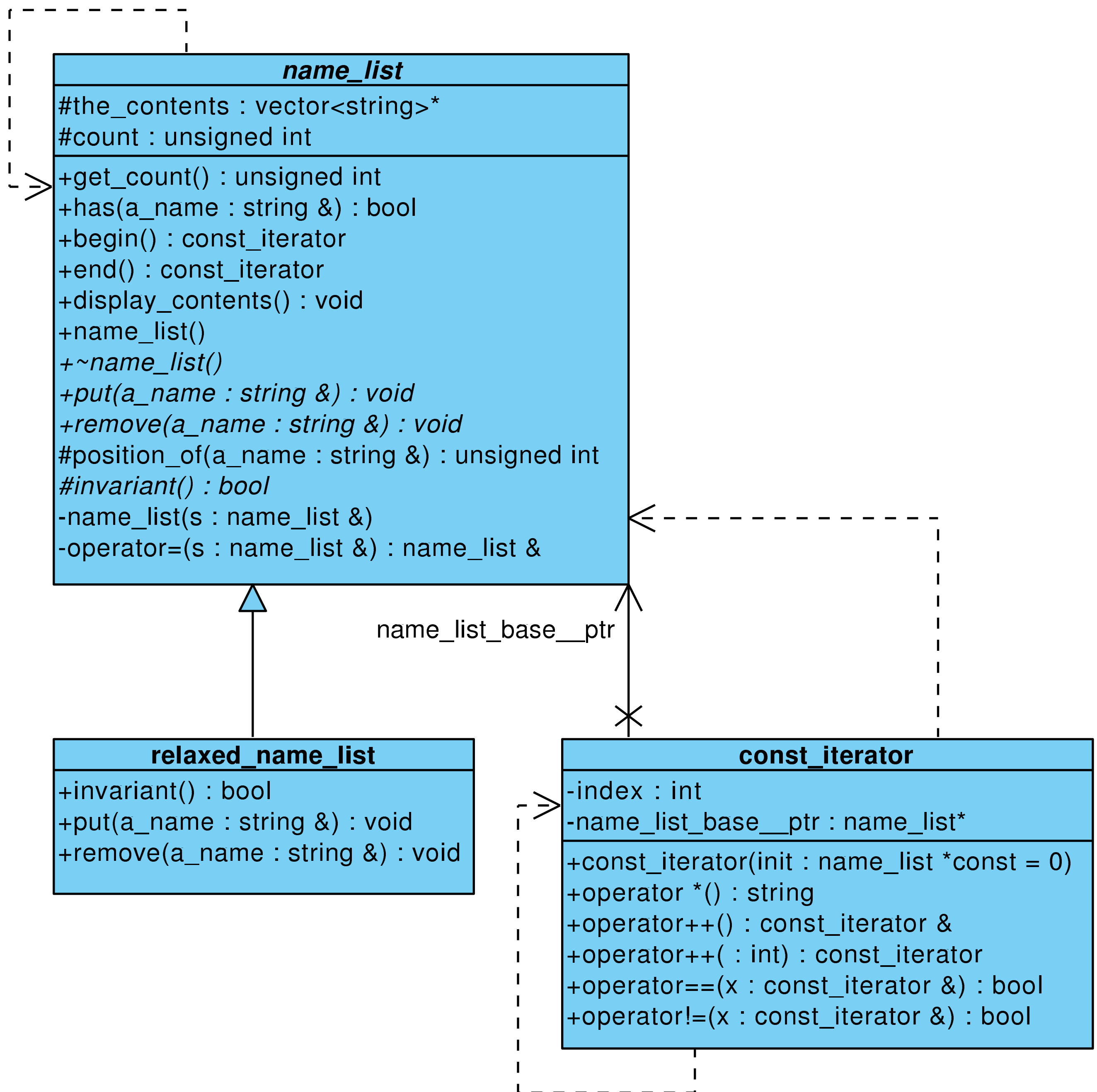
7.6.3 Mit Frameregel

```
unsigned int name_list::position_of(const string& a_name) const{

    unsigned int index(1);
    unsigned int result;
    for(; (index<=count) && (the_contents->at(index-1)!=a_name); index++);
    if (index <= count)
        result = index;
    else
        result = 0;
    ENSURE(!(result < 0));
    ENSURE(!(result >get_count()));
    ID(set<string> values(the_contents->begin(),the_contents->begin()+get_count()));
    ENSURE(/* a_name in list */
                                ((1<=result)&&(result<=get_count())&&
                                 (the_contents->at(result-1)==a_name)) ||
           /* otherwise: */
               ( ((0 == result) && (values.find(a_name) == values.end())))
          );
    return result;
};
...
unsigned int name_list::get_count() const{    // number of items in stack

    unsigned int result = count;
    CHECK(result == count);
    return result;
};

bool name_list::has(const string& a_name) const{

    bool result = (position_of(a_name) > 0);
    ENSURE((get_count()>0) || !result);
    ID(set<string> values(the_contents->begin(),
                   the_contents->begin()+get_count()));
    CHECK(result == (values.find(a_name) != values.end( )));
    return result;
};
...
void name_list::put(const string& a_name)    // Push a_name into list
DO
   ID(bool pre);
   IS(pre = !has(a_name));
```

```
   // DS($pre_false = has(a_name));    // funktioniert nicht
   ID(set<string> values_old(the_contents->begin(),
                             the_contents->begin()+get_count()));
   REQUIRE(/* name not in list */    pre);
   DS($count_old = get_count());

   count++;
   the_contents->at(count-1) = a_name;

   ENSURE(has(a_name));
   DI( (!pre) || (this->get_count() == $count_old + 1));
   ID(set<string> values(the_contents->begin(),
                         the_contents->begin()+get_count()));
   IS(values_old.insert(a_name));    // stilistisch unschoen!
   ENSURE(values_old == values);
END;
...
```

Zum Download: name_list3.cc

7.6.4 Mit Iterator-Methode (Design)




```
...
class name_list{
...
  /////////// Iteratoren:

  class const_iterator;
```

```
  friend class const_iterator;

  const_iterator begin() const;
  const_iterator end() const;
...
}
...
//////////////// class name_list::const_iterator /////////

class name_list::const_iterator{

    int index;
    const name_list* name_list_base__ptr;

public:

    const_iterator(const name_list* const init = 0);

    string operator*() const;        // Zugriff auf Elemet am Iterator-Ort
    const_iterator& operator++();    // Praefix-Inkrement
    const_iterator  operator++(int); // Postfix-Inkrement

    bool operator==(const const_iterator& x) const; // Vergleich von Iteratoren
    bool operator!=(const const_iterator& x) const;
};

/// contract for Iterator:  (as a basic query)

name_list::const_iterator::const_iterator(const name_list* const init){

    // ...
    // (*this)=="first" element of name_list (*init) if (init != 0),
    // this is an iterator not in any name_list if (init == 0)
};

name_list::const_iterator name_list::begin() const {
    // return ...
    // returns const_iterator pointing to "first" element of name_list
};

name_list::const_iterator name_list::end() const {
    // return ...
    // returns const_iterator denoting to be not any more in name_list
};
```

```
string name_list::const_iterator::operator*() const {
    // return ...
    // return element const_iterator is pointing to
};

name_list::const_iterator& name_list::const_iterator::operator++(){ // Praefix
   // return ...
   // increment position of const_iterator and return reference to this
   //       incremented const_iterator afterwards
};

name_list::const_iterator  name_list::const_iterator::operator++(int){//Postfix
    // return ...
    // return copy of const_iterator and as a side effect increment position
    // of the actual const_iterator
};

bool name_list::const_iterator::operator==(const name_list::const_iterator& x) const{
   // return ...
   // return if const_itrator and x point to the same element in the
   // same name_list
};

bool name_list::const_iterator::operator!=(const name_list::const_iterator& x) const{
    bool result; // = ...
    ENSURE(!((*this)==x));          // Konsistenzbedingung
};
...
```

Zum Download: name_list4.cc

7.6.5 Implementierung der Iterator-Methode

```
...
/// contract for Iterator:  (as a basic query)

name_list::const_iterator::const_iterator(const name_list* const init){

    name_list_base_ptr = init;
    if ((name_list_base_ptr != 0)&&(name_list_base_ptr->count > 0)){
        index = 0;
    } else
        index = -1;
```

```
    if (index == -1) name_list_base_ptr = 0;
    // (*this)=="first" element of name_list (*init) if (init != 0),
    // this is an unique iterator not in any name_list if (init == 0)
};

name_list::const_iterator name_list::begin() const{
    return const_iterator(this);
    // returns const_iterator pointing to "first" element of name_list
};


name_list::const_iterator name_list::end() const{
    return const_iterator();
    // returns const_iterator denoting to be not any more in name_list
};

string name_list::const_iterator::operator*() const {
    REQUIRE(index != -1);
    return name_list_base_ptr->the_contents->at(index);
    // return element const_iterator is pointing to
    // name_list of const_iterators not changed
};

...
```

Zum Download: name_list5.cc

7.6.6 Test des Iterators in display_contents() und main()

```
...
////////// derived queries:

void name_list::display_contents() const{     // print contents of list to cout

    cout << endl << "Anzahl der Listenelemente: " << count << endl;
    for (name_list::const_iterator i = begin(); i != end(); i++)
        cout << "  " << *i;
    cout << endl << endl;
    // ENSURE('Drucke alle Namen in Name_list');
};
...
/////////////// class test main() program //////////////

int main(){
```

```
    cout << "\nTest der Klasse name_list: ----------------" << endl;

    { name_list s;
    s.display_contents();

    s.put("Richard"); s.display_contents();
    // s.put("Richard"); s.display_contents(); // Test fuer pre-Verletzung
    s.put("Helen"); s.display_contents();
    s.put("Yu"); s.display_contents();
    s.put("Jim"); s.display_contents();
    s.put("Chen"); s.display_contents();
    s.put("Moira"); s.display_contents();
...
```

Zum Download: name_list6.cc

7.6.7 Qstl.h bei Contracts und Klassen mit eigenen Iteratoren: Framebedingungen mit Hilfe eines Iterators

```
...

#include <eiffel.h>
#include <nana.h>
#include <Qstl.h>     //<<</////////////////////////////////////////// NEU!

using namespace std;
...
unsigned int name_list::position_of(const string& a_name) const{

    unsigned int index(1);
    unsigned int result;
    for(; (index<=count) && (the_contents->at(index-1)!=a_name); index++);
    if (index <= count)
        result = index;
    else
        result = 0;

    ENSURE(/* a_name in list */
                            ((1<=result)&&(result<=get_count())&&
                             (the_contents->at(result-1)==a_name)) ||
           /* otherwise: */
                          ((0 == result)&&(!EO(i,(*this),(*i)==a_name))));
    return result;
```

```
};
...
bool name_list::has(const string& a_name) const{

    bool result = (position_of(a_name) > 0);
    ENSURE((get_count()>0) || !result);              // Konsistenz
    ENSURE(result == EO(i, (*this), (*i)==a_name));  // Konsistenz
    return result;
};
...
/////////// (pure) modificators:

void name_list::put(const string& a_name)    // Push a_name into list
DO
   REQUIRE(/* name not in list */    !has(a_name));
   ID(set<string> contents_old(begin(),end()));
   ID(int count_old = get_count());
   ID(bool not_in_list = !has(a_name));

   count++;
   the_contents->at(count-1) = a_name;

   ENSURE(has(a_name));
   ENSURE( (!not_in_list) || (get_count() == count_old + 1));
   ID(set<string> contents(begin(),end()));
   IS(contents_old.insert(a_name));
   ENSURE(contents == contents_old);
END;
...
```

Zum Download: name_list7.cc

7.6.8 Hilfsoperatoren für die STL

```
...
///////////////// Hilfsfunktionen /////////////////////////

template <class T>
set<T> operator+(const set<T>& s, const T& e){
    set<T> result(s);
    result.insert(e);
    return result;
};

template <class T>
set<T> operator-(const set<T>& s, const T& e){
    set<T> result(s);
    result.erase(e);
    return result;
};
...
void relaxed_name_list::remove(const string& a_name) // delete a_name in list
DO
   REQUIRE(/* nothing */ true);        // pre_parent || !has(a_name)
   ID(set<string> contents_old(begin(),end()));
   ID(int count_old = get_count());
   ID(bool pre_parent = has(a_name));

   if (has(a_name)){
      the_contents->at(position_of(a_name)-1) = the_contents->at(count-1);
      count--;
   };

   ENSURE(!has(a_name));
   ENSURE((!pre_parent) || (get_count() == count_old -1));  // &&
   ENSURE(  pre_parent  || (get_count() == count_old));
   // Menge der Eintraege in list == Menge der Eintraege in list_old ohne a_name
   ID(set<string> contents(begin(),end()));
   ENSURE(  pre_parent  || (contents == contents_old));;
   ENSURE((!pre_parent) || (contents == contents_old - a_name));
END;
...
```

Zum Download: name_list9.cc

7.7 Neuformulierung: Regeln und Leitlinien für PbC in C++

1. Formuliere *grundlegende Observatoren*, die den Zustand eines Exemplars vollständig beschreiben können und eine *Klasseninvariante*, die gültige von ungültigen Exemplaren trennt. Falls grundlegende Observatoren mit Parametern existieren, so gib den zur vollständigen Exemplarbeschreibung nötigen Parameter-Wertebereich an. Im Contract sollte kein Bezug auf Implementierungsdetails sondern lediglich auf die grundlegenden Observatoren genommen werden! Nachbedingungen von grundlegenden Observatoren spezifizieren deshalb lediglich Konsistenzbedingungen zwischen den Methoden. Observatoren sind const-Methoden.
2. *Abgeleitete Observatoren* sind i.a. besser lesbar als eine (komplizierte) Kombination grundlegender Observatoren, können evtl. effizienter implementiert sein und sollten dann in Vorbedingungen unbedingt statt der grundlegenden Observatoren benutzt werden. Sie sind const-Methoden. Ihre Nachbedingungen sollten die Return-Werte mit Hilfe der grundlegenden Observatoren vollständig spezifizieren.

3. Konstruktoren (default, Kopier-): ...
4. Zuweisungsoperator: ...
5. `operator==`, `operator!=`: ...
6. Destruktor: ...
7: Modifikatoren: ...
8. `friend`-Methoden und Operatoren: ...
9. Iteratoren: ...