Spec# Einführung

Formale Software-Entwicklung
Seminar SS '07
Universität Karlsruhe
Hilal Akbaba

Inhalt

- Einführung in C#
- Das Spec# System
- Die Konstrukte
- Vererben der Verträge
- System Architektur

Einführung in C#

- Weiterentwicklung der Konzepte von C++, Java, Visual Basic, sowie Delphi
- Eine objekorientierte prozedurale Microsoft .NET Programmiersprache

Was ist das Spec# System?

- Die Erweiterung von der objektorientierten Sprache C#
- Der Spec# Compiler, der in Microsoft Visual Studio integriert ist
- Der statische Programm Verifizier (Boogie), der Spec# Programm auf Korrektheit prüft

Spec# Programmiersprache

- Erweiterung von C# mit:
  - Nicht-Null-Typen
  - Checked Exception
  - Methodenverträge
  - Objektinvarianten
- Abwärtskompatibilität
  - Jedes Spec# Programm ohne Spezikationen ist äquivalent zu einem C# Programm
- Sprachkompatibilität
  - Andere .NET Sprachen können Spec# Programme aufrufen

Spec# Konstrukte

- **Nicht-Null-Typen**
- **Exceptions**
- **Methodenverträge**
  - **Vorbedingung**
  - **Nachbedingung**
  - **Ausnahmebehaftete Nachbedingung**
  - **Rahmenbedingung**
- **Klassenverträge**
  - **Klasseninvariante**
  - **Objektinvariante**

Nicht-Null-Typen

- Jede referenzierte Typ T beinhaltet den Wert ***null***
- Spec# Type T**!** kann nicht den Wert ***null*** annehmen
- Eingeführt:
  - Um Null-Referenz zu beseitigen
- können benutzt werden für
  - Instanzen: kein Zugriff vor dem Initialisieren (Beispiel folgt)
  - Lokale Variablen
  - Rückgaben
- Können nicht benutzt werden für Arrayelemente
- Sprachkompatibilität
  - Virtuelle Maschine

Beispiel(1)

```
class Student : Person {
    Transcript! t ;
    public Student (string name, EnrollmentInfo! ei)
    : base(name) {
        t = new Transcript(ei); ...
    } ...
}
```

Beispiel(1)

```
class Student : Person {
    Transcript! t ;
    public Student (string name, EnrollmentInfo! ei)
    : t(new Transcript(ei)), base(name) {
        ...
    } ...
}
```

Beispiel(2)

```
class MyLibrary {
  public static void Clear(int[]! xs) {
    for (int i = 0; i < xs.Length; i++) {
      xs[i] = 0;
    }
  }
}
class ClientCode {
  static void Main() {
    int[] xs = null;
    MyLibrary.Clear(xs); // Error: null is not a valid argument
  }
}
```

Umgang mit Exceptions in C#

- Sie sind alle **unchecked** Exceptions
- Keine explizite Angabe an der Methodensignatur
- Unklarer Zustand von Aufrufer und seiner Datenstruktur

Fehlerursache

- **Client Fehler:**
  - tritt auf, wenn die Methode auf unerlaubte Weise aufgerufen wird, z.B.: RequiresViolationExc.
- **Provider Fehler:**
  - unterteilt in
    - Admissible Fehler: tritt auf, wenn die Methode auf ein Ereignis (z.B.: Eingabe) wartet
    - Observed Program Fehler: tritt unabhängig von dem Programm auf, z.B.: *ArrayIndexOutOfBounds* oder *StackOverflow*

Exceptions in Spec

- Gibt es zwei Arten von Exceptions:
  - Checked Exception: z.B.: admissible Fehler
    - muss in Methodensignatur deklariert werden
  - Unchecked Exception: z.B.: observed Programm Fehler oder Client Fehler
    - muss nicht in Methodensignatur deklariert werden
- Jede checked Exception ist ein Objekt von der abgeleiteten Laufzeitexception-Klasse
- Implementiert in Schnittstelle *ICheckedException*
- Wird in der Methode durch *throws* Menge angegeben

Checked Exception in Spec#(1)

public String ReadMessage()

throws SocketClosedException;

- Methode darf mit einer Exception *SocketClosedException* oder einer Exception eine Unterklasse von *SocketClosedException* terminieren
- Sicherer Zustand des Programms nach dem Terminieren

Checked Exception in Spec#(2)

- Auswerten des Typs von der *Exception e* kann in dem Fall fehlschlagen.

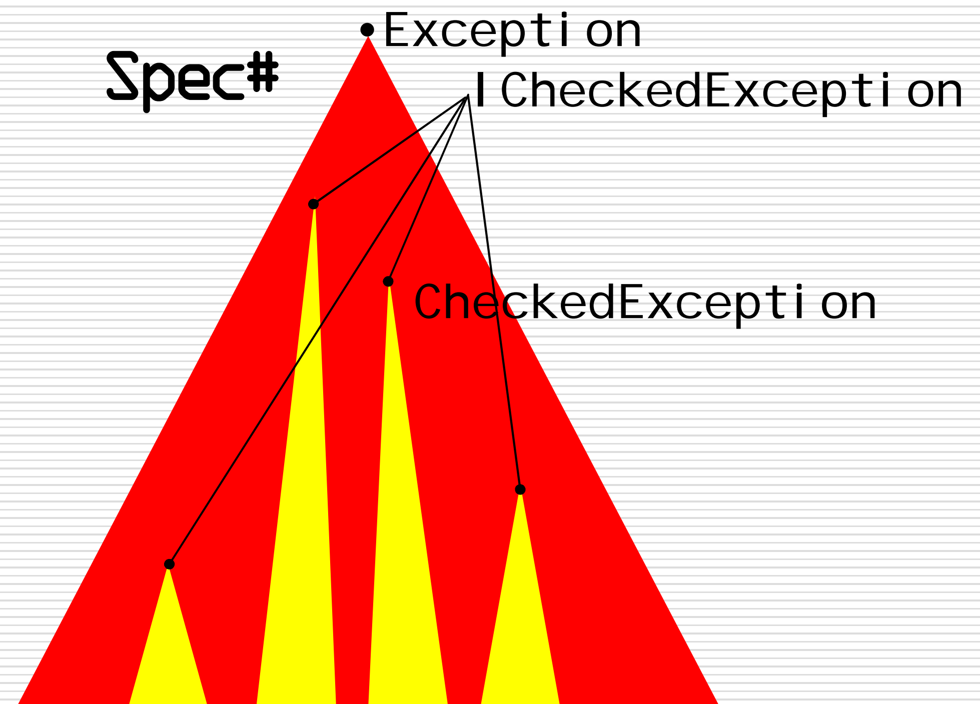
```
public void MyExcept(){
    Exception e = new CheckedException();
    throw e;    //checked Exception
}
```

- Wenn der Compiler nicht garantieren kann, dass die throws-Deklaration eingehalten wird, findet eine Laufzeit Prüfung statt.

```
public void Cheater(){
 Exception e = new CheckedException();
    if (!(e is ICheckedException)) throw e;
    else { /* irgendein Fehler*/ }
}
```

Checked- und Unchecked Exception

"Jede checked Exception ist ein Unchecked Exception aber nicht umgekehrt."



Methodenverträge

- **Vorbedingung:**
  - definiert Bedingung, welche vor dem Methodenaufruf gelten muss.
- **Nachbedingung:**
  - definiert Bedingung, welche nach dem Methodenaufruf gelten muss.
- **Ausnahmebehaftete Nachbedingung:**
  - definiert Bedingung, welche nach einer checked Exception gelten muss
- **Rahmenbedingung:**
  - definiert Einschränkung, welche den Schreibzugriff auf Speicherstellen begrenzt

Vorbedingung(1)

- wird durch *requires* gekennzeichnet

```
class ArrayList{
    int count; // Anzahl der Elemente
    bool IsReadOnly, IsFixedSize;
    ...
    public void Insert(int index, object value)
    requires 0<=index && index<=count;
    requires  !IsReadOnly && !IsFixedSize;
    {...}
}
```

Vorbedingung mit Exception(2)

```
class ArrayList{
    ...
    public void Insert(int index, object value)
    requires 0<=index && index<=count;
        otherwise ArgumentOutOfRangeException;
    requires  !IsReadOnly && !IsFixedSize;
        otherwise NotSupportException;
    {...}
}
```

- Syntax:
  - ***requires*** Bedingung [***otherwise*** unchecked Exception];

Nachbedingung

- Wird durch ensures gekennzeichnet

```
ensures count == old(count) + 1;
ensures value == this[index ];
ensures Forall{int i in 0 : index ; old(this[i]) == this[i]};
ensures Forall{int i in index : old(count); old(this[i]) == this[i + 1]};
```

- **Syntax**:
  - Forall Quantor
    - Index i bei unterer Grenze eingeschlossen, oberer ausgeschlossen
  - old(x) ruft den Wert beim Methodeneintritt von x ab

Ausnahmebehaftete Nachbedingung

- versichert den Zustand des Objekts (in dem Fall *a*), falls *EndOfFileException* geworfen wird

```
void ReadToken(ArrayList! a)
   throws EndOfFileException ensures a.Count ==old(a.Count);
```

Rahmenbedingung

```
class Konto{
        int kontostand;
        int kontoNr;
        ...
        public void einzahlen(int betrag)
        modifies kontostand;
        {...}
}
```

- **Syntax:**
  - ***modifies*** Liste der Variable mit Schreibrecht;

Rahmenbedingung

```
class Konto{
        int kontostand;
        int kontoNr;
        ...
        public void einzahlen(int betrag)
        modifies kontostand;
        {...}
}
```


- **Syntax:**
  - ***modifies*** Liste der Variable mit Schreibrecht;

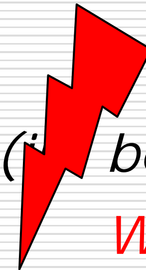
Rahmenbedingung

```
class Konto{
        private int kontostand;
        private int kontoNr;
        ...
        public void einzahlen(int betrag)
        modifies kontostand;
        {...}
}
```

- **Syntax:**
  - ***modifies*** Liste der Variable mit Schreibrecht

Rahmenbedingung

```
class Konto{
        private int kontostand;
        private int kontoNr;
        ...
        public void einzahlen(int betrag)
        modifies kontostand;
        {...}
}
```



Widerspruch zur Datenkapselung(Geheimnisprinzip)

- Syntax:
  - ***modifies*** Liste der Variable mit Schreibrecht

Rahmenbedingung

```
class Konto{
        private int kontostand;
        private int kontoNr;
        ...
        public void einzahlen(int betrag)
        modifies this^ Konto;
        {...}
}
```

- **Syntax:**
  - ***modifies*** Liste der Variable mit Schreibrecht;

Klassenverträge

- Objektinvariante
  - Bedingung, die in jedem Zustand **aller** Objekte gelten muss
- Klasseninvariante
  - Bedingung, die für **Klassenvariablen** gelten muss

Objektinvariante

```
class StudentenAusweisVerwaltung{
    Student[]! Students;
    bool[]! Verschollen;
    invariant Students.Length==Verschollen.Length;
    ...
}
```

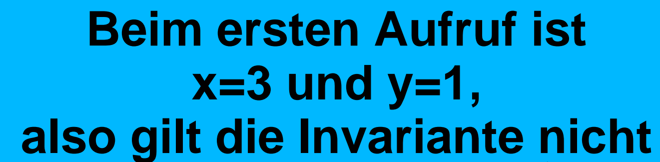
- **Syntax**:
  - ***invariant* Bedingung;**

Objektinvariante(2)

```
class T {
  private int  x , y;
  invariant 0 <= x  &&  x< y ;
  public T( ){
     x = 0 ; y = 1 ;
  }
  private void M( ){
     x = x + 3 ;
     y = 4 * y ;
  } ...
}
```

Objektinvariante(2)

```
class T {
  private int x , y;
  invariant 0 <=x &&  x< y ;
  public T( ){
    x= 0 ; y= 1 ;
  }
  private void M( ){
    x= x + 3 ;
    y= 4 * y ;
  }
}
```



Objektinvariante(2)

```
class T {
  private int x , y;
  invariant 0 <=x && x< y ;
  public T( ){
       x= 0 ; y= 1 ;
  }
  private void M( ){
       expose(this){
         x= x + 3 ;
         y= 4  y ;
       }
  }
}
```

- erzielt:
  - Exklusiven Zugriff auf das Objekt
  - Keine Invariantenprüfung innerhalb des *expose* Blocks
- Nebenbedingung:
  - expose(this) implizit bei public Methoden immer da
  - Mehrfaches *expose* auf das selbe Objekt verboten

⇒ Eine public Methode darf eine andere public Methode von der selben Klasse nicht aufrufen, wenn auf das selbe Objekt zugegriffen wird

Vererben der Verträge

- Methodenverträge werden vererbt
  - Zusätzliche Nachbedingungen können hinzugefügt werden
  - Eine Veränderung der Vorbedingung ist nicht erlaubt
- Methoden eines Interfaces können auch Verträge besitzen
- Mehrfachvererbung können Probleme verursachen
  - Vorbedingungen müssen gleich sein

System Architektur

- Es gibt drei Prüstufen:
  - **<u>1.Stufe</u>**: wird durch statischen Typprüfer angeboten, was als ein Teil der Compiler läuft
  - **<u>2.Stufe</u>**: Laufzeitprüfung, die durch den Compiler unterstützt wird. Der Compiler:
    - Produziert nicht nur ausführbaren Code, sondern bewahrt auch alle Verträge im compilerten Code
    - Werden manche Verträge (z.B.: Vor und Nachbedingungen) geprüft
    - (auf Wusch) können die Laufzeittests abgeschaltet werden (aus Effizienzgründen)
  - **<u>3.Stufe</u>**: wird durch den statische Verifizierer angeboten,
    - wandelt das Programm in eine Zwischensprache BoogiePL um
    - beweist anhand der Formel, die aus BoogiePL ensteht, die Korrektheit, falls es für Boogie möglich ist

Zusammenfassung

- Spec# Sprache:
  - ist in Entwicklungsphase
  - typsicher
  - abwärtskompatibel
  - werden Fehler leichter gefunden

Vielen Dank für die Aufmerksamkeit …