Reflection

Entwurfsmuster Reflection

Ein reflexives System besteht aus zwei Teilen.

Eine Metaebene stellt Informationen über ausgewählte Systemeigenschaften zur Verfügung und macht damit die Software sich ihrer „selbst bewusst".

Eine Basisebene enthält die Anwendungslogik. Ihre Implementierung baut auf der Metaebene auf

Zitat: Patternorientierte Softwarearchitektur,
Buschmann et. al.

Beispielanwendung: Softwarekomponente, die Persistierung von Objekten bel. Klassen ermöglicht.

Reflection

Aufbau von Programmen

- Assembly hat
  - Modul(e) hat
    - Datei(en) hat
      - Klassen hat
        - Methoden, Felder, ...
- All dies sind Informationen über das Programm, so genannte Metadaten (Daten über Daten)
- Reflection = Mechanismus, der es erlaubt, sich selbst zu betrachten (untersuchen, reflektieren)
- Reflection = Frage an den Code:
  - Wer bist du?
  - Was enthältst du?
- ermöglicht: Untersuchung von Klassen
  - Zugreifen auf Elemente
  - Ausführen von Methoden

Reflection

Grundprinzip, Vor- Nachteile

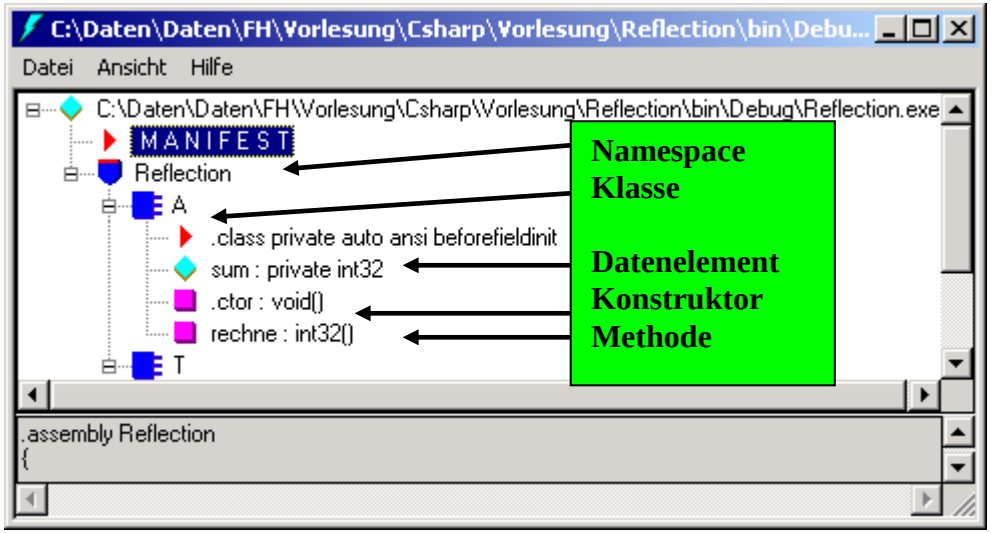
- Reflektion wertet die Metadaten aus, die neben den eigentlichen Daten in der Assembly abgespeichert werden
- Metadaten werden automatisch vom Compiler generiert
- Die Metadaten sind prinzipiell vorhanden
- compilierter Code (IL-Code, Maschinencode) kann somit prinzipiell dekompiliert werden in
  - C# Code
  - z. B. Anakrino Dekompiler (IL -> C#)
- Nachteil: Problem für Firmen die Wissen z. B. über spezielle Algorithmen verbergen müssen
  - Lösungsansatz: Obfuskatoren, machen Code unlesbar

Namensraum System.Reflection stellt Reflection-API zur Verfügung

Reflection

Quellcode und Metadaten

```
namespace Reflection {
    class A
    {   private int sum=0;
        public int rechne()
            {return ++sum;}
    }
    class T{. . .}
}
```



Auswertung der Metadaten zur Laufzeit (Grundprinzip)

```
...
Assembly a = Assembly.LoadFrom("Reflection.exe");    //Lade Assembly
Type[] types = a.GetTypes();                         // Finde Typen (Klassen)
foreach(Type t in types){
    MemberInfo[] mbrarr = t.GetMembers();    // Finde Typelemente
...
```

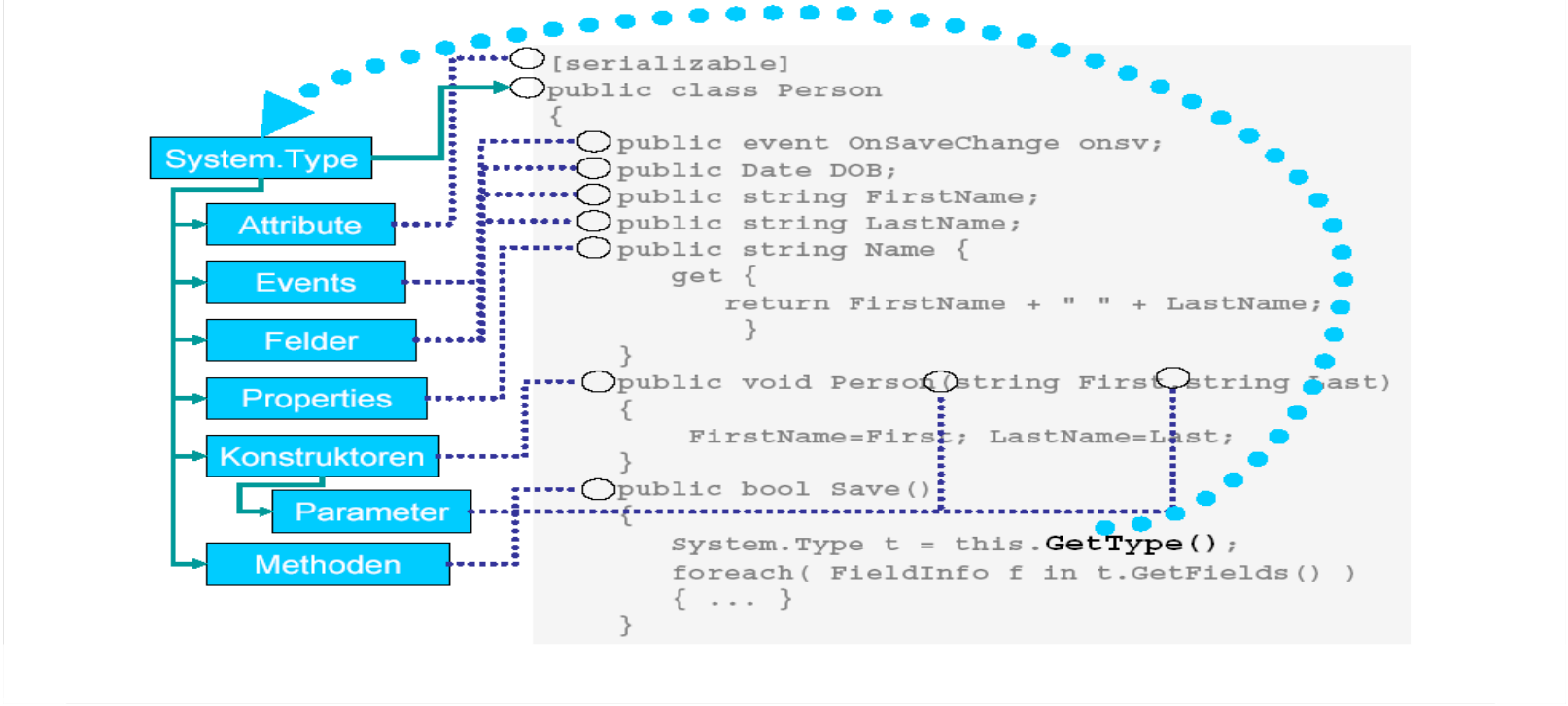
Reflection

Aufgaben des Reflectioneinsatzes

- Lesen von Metainformationen (Assembly, Modul, Typen, Elemente)
- Erzeugen von Instanzen zur Laufzeit
- dynamischer Methodenaufruf sowie Zugriff auf Werte von Datenelementen (Felder, Properties,...)
- Erstellen von neuen Typen erst zur Laufzeit und Verwendung dieser Typen (Reflection-Emit)

Reflection

Ermitteln von Code-Informationen zur Laufzeit



Reflection

Die Klasse System.Type

- Mittelpunkt der Reflectionoperationen
- repräsentiert jeden Typ den es im System gibt
- ist abstrakte Basisklasse
- enthält genau eine Instanz pro Typ im System (quasi Singleton)
- Instanziierung mit Type bedeutet Instanziierung einer von Type abgeleiteten Klasse

  Bsp.: Instanz a der Klasse A

```
Type t1 = a.GetType();          // von object geerbt
Type t2 = Type.GetType("A");    // stat. Methode
Type t3 = typeof(A);            // typeof
```

Mit den Methoden von Type können Informationen über den Typ gelesen aber nicht verändert werden.

Reflection

Anfragen

- **Was bist du?**
  - **Wert (IsValue), Interface (IsInterface), Klasse (IsClass)**
  - **Zugriffsrechte: IsPublic, IsPrivate, IsSealed**

- **Was bist du?**
  - **Memberinfo: GetMembers(), FindMembers()**
  - **Fieldinfo: GetFields()**
  - **PropertyInfo: GetProperties()**
  - **Konstruktor, Methoden, Events**
    GetConstructors(), GetMethods(), GetEvents()

Reflection

Anwendungssituationen

- zur Entwicklungszeit
  Bsp.: Intellisense
  Visual Studio .Net zeigt bei der Entwicklung auf Basis von Reflection alle Elemente an, die verwendet werden können
- zur Compilierzeit
  Bsp.: Serialisierung (siehe: Kapitel Attribute)
  Zur Speicherung von Objekten werden mit Reflection alle zu speichernden Elemente ermittelt (Syntax: Attribut [Serializable])
- zur Laufzeit
  Bsp.: spätes Binden d.h. Aufruf von Methoden oder Objekten, die zur Compilezeit noch nicht zur Verfügung stehen (Bsp.: Plugin DLLs)

Reflection

Spätes Binden

- Zur Compilezeit ist nicht bekannt welche Funktion zur Laufzeit aufgerufen wird, so dass Methodenaufruf nicht zur Compilezeit gebunden werden kann
- Vorgehensweise

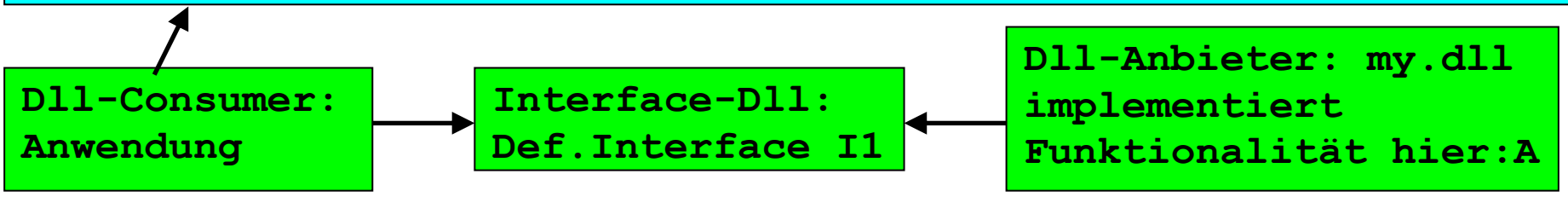
```
// Assembly laden
Assembly a = Assembly.LoadFrom("my.dll");
// Typ bestimmen oder auswählen
Type type = a.GetType("Reflektion.A");
// Instanz des Typs mit Activator erstellen
object ob = Activator.CreateInstance(type);
// Methode des Typs zur Laufzeit suchen (auswählen)
MethodInfo fnInfo = type.GetMethod("fkt");
// Methode aufrufen (evtl. Parameter bestimmen)
object r = fnInfo.Invoke(ob,new object[1]{4});
```

Reflection

Verwendung eines Interfaces

- Interface in Interface-Dll definiert
- DLL-Anbieter verwendet Interface-Dll (Verweis)
- DLL-Consumer programmiert auf Interface (Verweis Interface-Dll) hin, lädt zur Laufzeit Assemblies der DLL-Anbieter und verwendet deren Funktionalität

```
Assembly a = Assembly.LoadFrom("my.dll");  // Assembly laden
Type typ = a.GetType("Reflection.A");      // Klasse auswählen
                                // Test ob Interface implementiert ist
if (typ.GetInterface("I1") != null)
{   I1 i1 = Activator.CreateInstance(typ) as I1;
    Console.WriteLine(i1.fn(17)); }  // Aufruf Interfacemethode
```



Reflection

Reflection in C# vs Reflection in Java

- In beiden Umgebungen werden die Metadaten analysiert

- Die Möglichkeiten der Analyse sowie des Aufrufs zur Compilezeit unbekannten Codes sind in beiden Sprachen vergleichbar.

- Hauptunterschied:

  - Reflection in C# erfolgt auf Assembly-Ebene
    Klasse wird über Assembly gefunden

  - Reflection in Java erfolgt auf Klassen-Ebene