Kapitel 14

Traits und Mixins

14.1 Grundbegriffe der Komponententechnologie

Eine *Komponente* ist ein Programmteil, der mit anderen Teilen in größeren Applikationen kombiniert wird. Komponenten müssen wiederverwendbar sein.

Grundidee: Komponente wird „angeflanscht“ (plug in).

Komponente hat *keinen eigenen Zustand*, sondern benutzt den Zustand des „Hauptsystems“ über eine „Versorgungsschnittstelle“ mit. Ergebnisse werden über eine eigene Schittstelle ans Hauptsystem geliefert

Deshalb: keine statischen Variablen, kein Zugriff auf statische Variablen anderswo, keine hartverdrahteten Beziehungen zum Hauptsystem/anderen Komponenten!

⇒ Komponente benötigt Interfaces, welche deren *angebotene* und *benötigte* Dienste beschreibt, um in neuen Umgebungen wiederverwendbar zu sein.

Alle Referenzen einer Komponente zu anderen müssen über deren Members oder Paramenter erfolgen.

Die meisten gängigen Sprachen können nur die angebotenen Dienste beschreiben, nicht aber die benötigten.

$$Komponente \neq API!$$

14.2 Komponenten in Scala

Scala von M. Odersky ist eine experimentelle OO-Sprache, die OO-Konzepte mit funktionalen Sprachkonzepten integriert und innovative Konzepte für Komponenten anbietet

Es gibt zB Funktionen höherer Ordnung, Pattern Matching (wie in ML/Haskell)

Scala bietet verschiedene Konstrukte für Komponenten:

Abstract type members Typvariablen (analog Generizität)

Self-type annotations erlauben das Abstrahieren des Type "self"

Modular mixin composition flexible Möglichkeit zur Erstellung von Komponenten und Komponententypen

Alle 3 Abstraktionen haben ihre theoretische Grundlage im νObj Kalkül [Odersky et al., ECOOP'03].

Abstraktion von Komponenten

Es gibt zwei grundsätzliche Formen der Generizität in Programmiersprachen:

> *Parametrisierung* (funktional, vgl generische Klassen)
> *Abstrakte Members* (objekt-orientiert, vgl abstrakte Methoden)

Scala bietet beide Arten der Abstraktion sowohl für Typen wie auch für Werte.

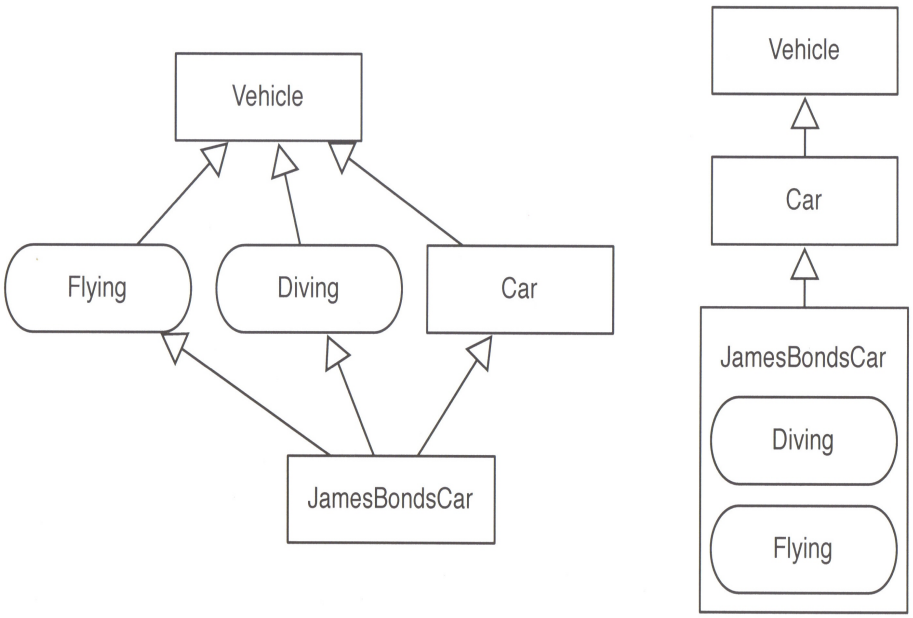
14.3 Mixins

Einfachvererbung ist manchmal nicht ausreichend, Mehrfachvererbung ist sehr komplex (vgl vtables etc)

Scala bietet *Traits und Mixins* als Kompromiss an.

- Ein Trait ist eine spezielle Art von Klasse, die irgendein Verhalten implementiert
- Eine Klasse kann von *mehreren* Traits erben
- Es muss eine gemeinsame Elternklasse mit dem geerbten Trait geben

Beispiel



Traits sind oval, normale Klassen rechteckig

Traits haben requires/provides Schnittstelle, aber keine eigenen Instanzvariablen (kein eigener Zustand)

Zugriff auf fremden Zustand über requires-Schnittstelle

Zugriff auf ererbte Members möglich

⇒ keine Probleme mit Adressierung/Mehrdeutigkeit (vgl Rossie-Friedmann, vtables)

Implementierung von Traits wie inner classes

Syntax:

```
trait Flying extends Vehicle {
  def takeOff = ... //concrete
  def land: Unit // abstract
}
```

Alle von Vehicle geerbten Member können verwendet werden: Dieser Trait wird später mit einer um Vehicle erweiterten Klasse gemischt (mixed-in).

Ein Trait kann geerbt werden,

- wenn eine Klasse definiert wird:
  `class JamesBondsCar extends Car with Flying with Diving`
- oder wenn eine Instanz erzeugt wird:
  `val jbsCar = new Car with Flying with Diving`

Falls mehrere Traits geerbt werden,

- können sie auf Member ihrer gemeinsamen Superklasse zugreifen
- aber nicht auf Member anderer mixed-in Traits (keine Hard Links!)

Zugriff nur über requires Schnittstelle! Syntax:

```
trait Reading extends Person requires Seeing
```

14.4 Beispiel 1: Symboltabellen

Compiler müssen Typen und Symbole modellieren. Beide Aspekte hängt voneinander ab.

1. Versuch

zwei globale Objekte, eines für jeden Aspekt:

```
object Symbols { //globale Objekte in Scala können innere
   Klassen haben
  class Symbol {
    def tpe: Types.Type;
    ...
    }
    // static data for symbols
}

object Types {
  class Type {
    def sym: Symbols.Symbol
    ...
    }
    // static data for types
}
```

Probleme:

1. Symbole und Typen besitzen gegenseitige harte Verweise ⇒ nicht möglich eines ohne das Andere zu verändern.
2. Globale Objekte sind statisch ⇒ Compiler nicht ablaufinvariant. Multiple Kopien können nicht im selben Prozess laufen.

2. Versuch: Verschachtelung

Statische Daten können dadurch verhindert werden, indem die Symbol und Type Objekte in einer gemeinsamen Kapselungsklasse verschachtelt werden.

```
class SymbolTable {
  object Symbols {
    class Symbol { def tpe: Types.Type; ... }
  }
  object Types {
    class Type { def sym: Symbols.Symbol; ... }
  }
}

... new SymbolTable() ...
```

Das löst das Ablaufinvarianz Problem.

Aber es löst nicht das Komponenten-Wiederverwendungsproblem:

- Symbole und Typen enthalten immer noch harte gegenseitige Verweise
- beide können (anders als vorher) nicht separat geschrieben und compiliert werden, da sie in einem umschließenden Objekt verschachtelt sind.

3. Versuch: Mixins

Wie lässt sich die gegenseitige Abhängigkeit auflösen?
durch Abstraktion!

„Wenn du es nicht kennst, mach es zum Parameter“

Zwei Formen der Abstraktion: *Parametrisierung* und *abstrakte Member*.

Nur abstrakte Member können rekursive Abhängigkeiten (wie im Bsp) darstellen:

```
abstract class Symbols {
  type Type; // abstrakter Typ, analog abstrakter Methode
  class Symbol { def tpe: Type }
}

abstract class Types {
  type Symbol;
  class Type { def sym: Symbol }
}
```

Zusammensetzen mit Mixins:

```
class SymbolTable extends Symbols with Types
```

Instanzen von `SymbolTable` enthalten alle Symbol-Member wie auch Types-Member. Erstere durch Ererbung, zweitere werden reinkopiert („reingemischt“, mixed-in)

Konkrete Definitionen in einer Oberklasse überschreiben dabei abstrakte Definitionen in einer Unterklasse.

⇒ konkretes `Symbol` in `Symbols` überschreibt abstraktes `Symbol` in `Type`

Mixins generalisieren Einfachvererbung und Interfaceverererbung. Im Gegensatz zum Originalkonzept der „Traits“ können Oberklassen (zB Symbols) durchaus Instanzvariablen haben

4. Versuch: Mixins und Self-Types

Nun: requires-Interface

Grund: abstrakte Typen kann man nicht vererben oder instanziieren (restriktiv)

Ein allgemeinerer Ansatz benutzt zudem *self-types*:

```
class Symbols requires Symbols with Types {
  class Symbol { def tpe: Type }
}

class Types requires Types with Symbols {
  class Type { def sym : Symbol }
}

class SymbolTable extends Symbols with Types
```

Hier hat jede Komponente einen *self-type* (Typ von `this`), der alle benötigten Members enthält.

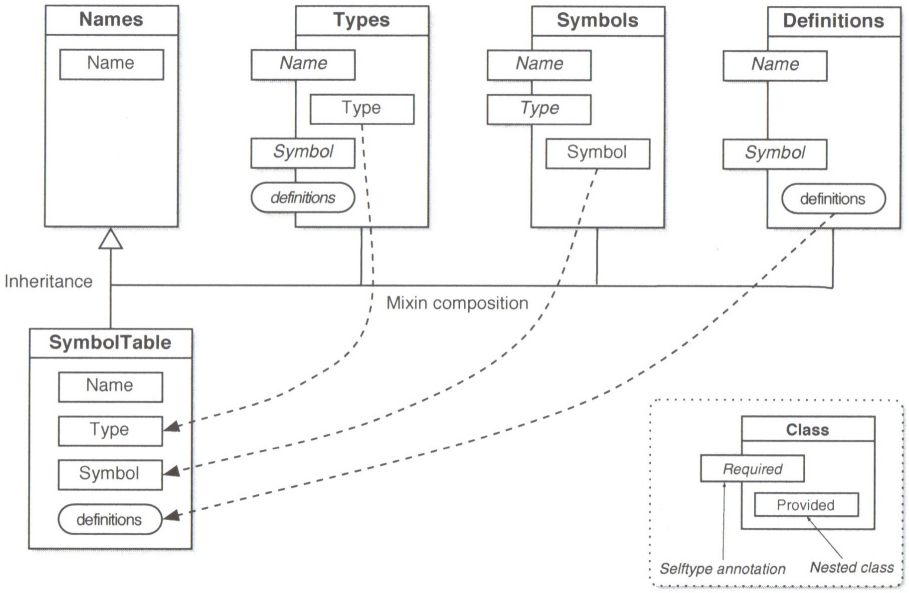
Im Rumpf von `Symbols` hat `this` Pointer auch Zugriff auf die (mixed-in) `Types` Members!

Self-Types

- in einer Klassedeklaration `class C requires T ...` wird `T` *self-type* der Klasse C genannt.
- Falls ein self-type gegeben ist, wird er als Typ von *this* innerhalb der Klasse verwendet.
- Ohne eine explizite Typangabe ist der self-type die Klasse ohne Mixins (wie gewöhnlich)
- Der self-type einer Klasse muss ein Untertyp der self-types aller Basis Klassen sein.
- Wenn eine Klasse mit dem *new* Ausdruck instanziiert wird, wird überprüft, ob der self-type der Klasse ein Supertyp des Typs des erzeugten Objekts ist.

Gesamte Symboltabelle

enthält weitere voneinander rekursiv abhängige Klassen:



Vorteile

1. *Allgemeinheit* – jede Kombination von statischen Modulen kann zu einer Gruppe von Komponenten werden.
2. Komponenten besitzen *dokumentierte Interfaces* für benötigte und angebotene Services.
3. Komponenten können *mehrfach instanziiert* werden ⇒ *Ablaufinvarianz* ist kein Problem.
4. Komponenten können flexibel *erweitert* und *adaptiert* werden.

14.5 Beispiel 2: Logging

Als Beispiel für eine Komponenten-Adaption fügen wir dem Compiler Logging hinzu (vgl AOP!)

Es soll ein Log für jede Symbol- und Typerzeugung auf die Standardausgabe geschrieben werden.

Es stellt sich das Problem, wie man in einen existierenden Compiler Aufrufe für die Logging Methoden einfügen kann

- ohne den Quellcode zu ändern,
- mit Berücksichtigung von Lokalität und Kohäsion,
- ohne Verwendung von AOP.

Logging Klassen

Idee: neue Unterklassen der Komponenten, die Logging Code enthalten.

```
abstract class LogSymbols extends Symbols {
  override def newTermSymbol(name: Name): TermSymbol = {
    val x = super.newTermSymbol(name) //alter code
    // +Logging
    System.out.println("creating term symbol" + name)
    x
  }
  ...
}
```

(analog für *LogTypes*)

Komposition mit Mixins

Originalcompiler:

```
class ScalaCompiler extends SymbolTable with ... ...
```

mit Logging:

```
class TestCompiler extends ScalaCompiler with LogSymbols with
 LogTypes
```

⇒ jeder Aufruf von `newTermSymbol` wird als Aufruf von `LogSymbols.newTermSymbol` interpretiert, da die ursprüngliche Methode beim Mixin überschrieben wird

NB 1: das ist nicht dynamische Bindung!

NB 2: auch ererbte Methoden werden durch entsprechende Mixins überschrieben!

Vergleich mit AOP:

+ kein Join Point ($\rightsquigarrow$ evtl Kontrollfluss-Chaos!) notwendig

− nicht alle Log-Methoden sind in *einer* Klasse zusammengefasst

14.6 Beispiel 3: Visitor Pattern

am Beispiel Datei-System: es gibt Dateien und Directories (Composite-Pattern), sowie einen Print-Visitor

Dann werden sowohl ein remove-Visitor als auch Dateilinks hinzugefügt, dh beide Hierarchien werden erweitert

Mixins ermöglichen für *beide* Erweiterungen Beachtung des Lokalitätsprinzips!

Basisklasse

```
trait Base {

  trait Node { // Basiselement der Hierarchie
    // abstrakte Methode, unit = void
    def accept(v: visitor): unit;
  }

  // konkrete Knotenart
  class File(name: String) extends Node {
    // name = Instanzvariable und
    // Parameter d. Konstruktors
    def accept(v: visitor): unit = v.visitFile(name);
  }
  class Directory(name: String) extends Node {
    def accept(v: visitor): unit =
        { v.visitDirectory(name);
          nodes.elements foreach { x => x.accept(v);}
  }
```

```
  type visitor <: Visitor;
  // tatsächliches Visitor-Interface
  // = abstrakter Typ mit Typschranke
  // beim Mixin durch vollständiges Interface ersetzt
  trait Visitor { // Visitor base of the hierarchy
    def visitFile(name: String): unit;
    def visitDirectory(name:String): unit;
  }

  // Printvisitor
  trait PrintingVisitor requires visitor extends Visitor
    {
    // braucht visitor-Interface
    // visitor wird beim Mixin durch das
    // vollständige Interface ersetzt
    def visitFile(name: String): unit =
      Console.println("printing file: " + name);
    def visitDirectory(name: String): unit =
      Console.println("printing directory: " + name);
  }
}
```

Datei-Links hinzufügen

Die Node-Hierarchie wird erweitert, entsprechend muss das Visitor-Interface erweitert werden.

Dies ist im klassischen Visitor nicht lokalitätswahrend möglich, da die gesamte Visitor-Hierarchie betroffen ist!

In Scala bleibt das Lokalitätsprinzip gewahrt

```
trait BaseLink extends Base {

  type visitor <: Visitor;

  trait Visitor extends super.Visitor { // Super = Base !
      // führt beim Mixin zu Überschreiben des alten
      // Visitor-Interfaces durch das erweiterte
    def visitLink(subject: Node): unit;
  }

  class Link(subject: Node) extends Node {
    def accept(v: visitor): unit = v.visitLink(subject);
  }

  trait PrintingVisitor requires visitor
  // braucht das volle Interface
      extends super.PrintingVisitor with Visitor {
      // erweitertes Interface wird mixed in
    def visitLink(subject: Node): unit =
    // Implementierung der neuen Interface-Methode
      subject.accept(this);
  }
}
```

Man braucht sowohl „requires visitor“ als auch „extends Visi-

tor“, weil „visitor“ beim Mixin mit dem vollen Interface überschrieben wird, und das wird in den Implementierungen der visit... Methoden auch benötigt. Andererseits wird in BaseLink nur „Link“ und „visitLink“ hinzugefügt, also „Visitor“ nur häppchenweise erweitert

Erweiterungen zusammenfassen

So wie Datei-Links kann man weitere neue Objektarten als unabhängige Traits definieren, zB BaseBla mit dem speziellen Dateityp Bla

Diese werden zusammengefasst:

```
trait BaseAll extends BaseBla with BaseLink {

  type visitor <: Visitor;
  trait Visitor extends super[BaseBla].Visitor
    with super[BaseLink].Visitor; // [ ] = von

  trait PrintingVisitor requires visitor
    extends super[BaseBla].PrintingVisitor
    with super[BaseLink].PrintingVisitor
    with Visitor;
```

Remove-Visitor hinzufügen

Dieser Fall ist auch im klassischen Visitor unproblematisch, da nur eine neue Unterklasse hinzukommt

```
// Funktionserweiterung
trait BaseExt extends BaseAll {
  class RemoveVisitor requires visitor extends Visitor {
    def visitFile(name: String): unit =
      Console.println("removing file: " + name);

    def visitDirectory(name: String): unit =
      Console.println("cannot remove directory: "
                      + name);

    def visitLink(subject: Node): unit =
      subject.accept(this);
  }
}
```

Komplettes System

alles wird zusammengesetzt

erst jetzt wird der abstrakte `visitor` instantiiert, mit dem vollständigen `Visitor`-Interface aus `BaseAll`

```
object VisitorTest extends BaseExt with Application {
  type visitor = Visitor; //ererbt aus BaseAll!

  val f = new File("foo.txt");
  val nodes = List(f, new Directory("bar"), new Link(f));

  class PrintingVisitor extends super.PrintingVisitor;
  nodes.elements foreach {
    x => x.accept(new PrintingVisitor()); }

  class RemoveVisitor extends super.RemoveVisitor;
  nodes.elements foreach {
    x => x.accept(new RemoveVisitor()); }
}
```