

Variabilities

Representing Variability In Software Systems

Markus Völter

voelter@acm.org
www.voelter.de



About me



Markus Völter

voelter@acm.org
www.voelter.de

- Independent Consultant
- Based out of Heidenheim, Germany
- Focus on
 - Model-Driven Software Development
 - Software Architecture
 - Middleware



CONTENTS

- Software System Families
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- Binding Times
- Techniques for Handling Variabilities
 - Source time
 - Compile time
 - Deployment/Configuration time
 - Link time
 - Run time
- Summary



CONTENTS

- **Software System Families**
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- Binding Times
- Techniques for Handling Variabilities
 - Source time
 - Compile time
 - Deployment/Configuration time
 - Link time
 - Run time
- Summary



Software System Families

- Typically, MDD makes most sense in the context of **software system families** because developing modeling environments, generators, translators, etc. can be a lot of work and it pays only if reused.
- What is a **software system family**?
We consider a **set of programs** to constitute a family whenever it is worthwhile to study programs from the set by **first studying the common properties** of the set and **then determining the special properties of the individual family members**.

Definition by Parnas



Examples for Software System Families

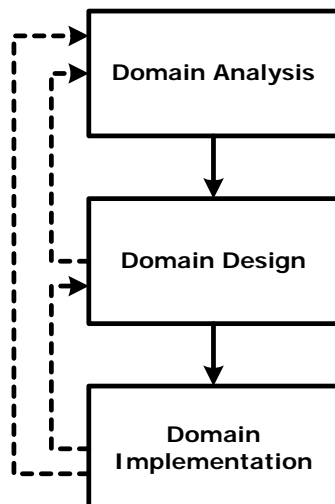
- A **set of projects in the same domain** (banking, telecom switching, automotive diagnosis).
 - You might be able to generate recurring business logic from models.
- A set of **artifacts based on the same infrastructure** (such as EJB) in one project.
 - Here, you might be able to generate all the infrastructure-specific code around manually implemented business logic.
- you have some **specific business logic that you want to run on different platforms**.
 - You might be able to generate platform-specific implementation code from the models (this is the focus of MDA)
- a set of **artifacts based on the same modelling paradigm**, such as state chart.
 - You might be able to generate the complete implementation based on the model and its predefined mapping to lower-level implementations.



CONTENTS

- Software System Families
- **Product-Line Engineering**
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- Binding Times
- Techniques for Handling Variabilities
 - Source time
 - Compile time
 - Deployment/Configuration time
 - Link time
 - Run time
- Summary

Product Line Engineering



- Domain Scoping
- Variability Analysis
- Domain Structuring

- Define common architecture
- Define Production Plan
- Define Building Blocks

- Components
- DSLs & Generators
- Production Process

Domain Analysis

- **Goal:** A domain model (aka metamodel)
- **Scoping** defines, **what is part of the domain, and what is not** (defines the range of possible products of the family)
- A **common vocabulary** defines the terms in which the domain can be described
- A result of this process is also the knowledge of **whether the domain is mature, or not** („are there more commonalities than differences?“)
- The **commonalities and the differences** between different products in the domain have to be defined
 - ➔ Variability Analysis

CONTENTS

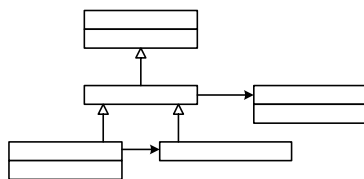
- Software System Families
- Product-Line Engineering
- **Variability Analysis**
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- Binding Times
- Techniques for Handling Variabilities
 - Source time
 - Compile time
 - Deployment/Configuration time
 - Link time
 - Run time
- Summary

Variability Analysis

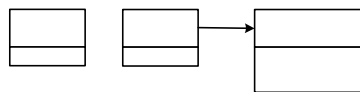
- **Variability analysis** discovers the variable and fixed parts of a product in a domain.
Parts can be
 - Structural or behavioral
 - Functional or non-functional (technical)
 - Modularized or aspectual
- To define variable parts, we need to **have a commonality base**: a base platform, a common architecture
- There are **two kinds of variability**:
 - positive variability: add something (optional)
 - negative variability: removes something (essential)
- Another classification: **structural** vs. **non-structural** var.

Structural vs. Non-Structural Variability

- **Structural Variations**
Example Metamodel



- Based on this sample metamodel, you can build a **wide variety of models**:

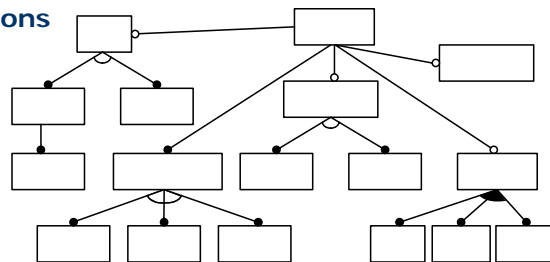


- **Non-Structural Variations**
Example Feature Models

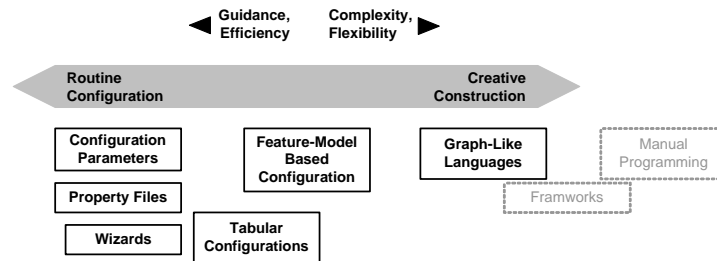
Dynamic Size, ElementType: int,
Counter, Threadsafe

Static Size (20),
ElementType: String

Dynamic Size, Speed-Optimized,
Bounds Check



Routine Configuration vs. Creative Construction



- This slide (adopted from K. Czarnecki) is **important for the selection of DSLs** in the context of MDS in general:
 - The more you can move your DSL „form“ to the configuration side, the simpler it typically gets.
 - We will see why this is especially important for behavior modelling.

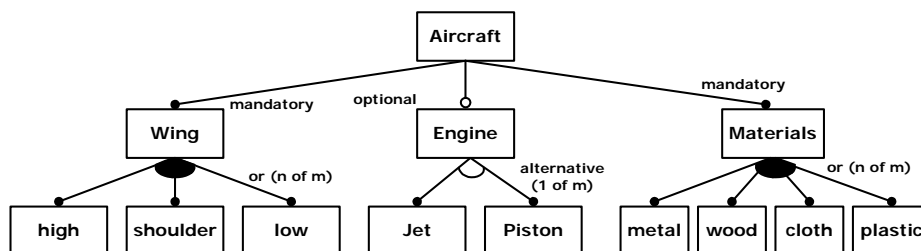
CONTENTS

- Software System Families
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - **Non-Structural Variability: Feature Modelling**
 - Combination of Both
- Binding Times
- Techniques for Handling Variabilities
 - Source time
 - Compile time
 - Deployment/Configuration time
 - Link time
 - Run time
- Summary

Feature Modeling

- As a consequence of the domain analysis and before defining a concrete metamodel, usually, a **feature modelling** phase makes sense to systematically define optional and mandatory features.
- A feature model **describes the (sub-)features of a concept**, subfeatures can be
 - Mandatory
 - Optional
 - Alternative
 - N of M
- A feature can represent some kind of **component** or an **aspect**.
- A feature model can be **multi-dimensional**
- A graphical notation exists: **feature diagrams**

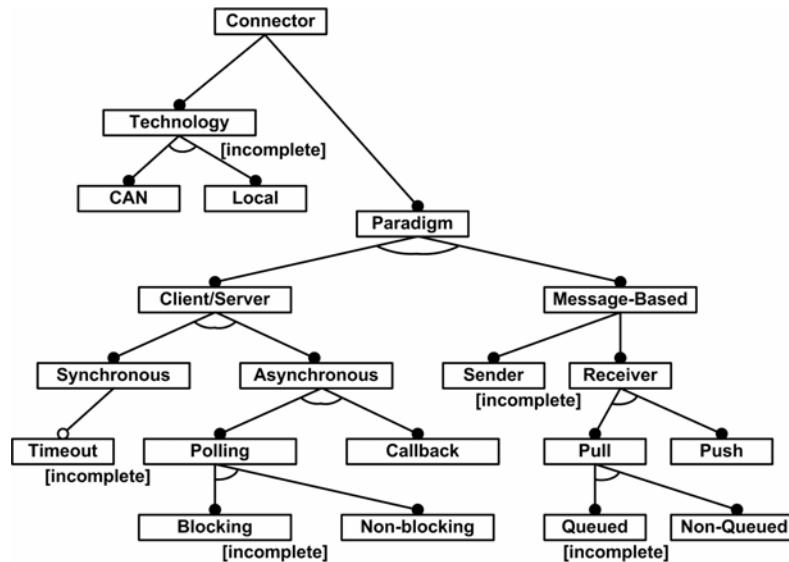
Example Feature Diagram



- Example products:
 - An aircraft with a low wing, piston engine and made of metal, wood and cloth: **Robin DR-400**
 - An aircraft with shoulder wing, no engine and made of plastic: **ASW-27**
 - An aircraft with low wing, jet engine(s) and made of metal: **Airbus A320**



More Feature Models: Communication



Feature Diagrams cont'd

- They can contain **constraints** on the combinations of features
- They can define „names“ for specific combinations of features; feature groups
- Features can be **open**: additional subfeatures can be added
- Features can be **incomplete**: the subfeatures are not yet defined
- **Multiplicity of subfeatures** can be added
- And more...

Feature Diagrams cont'd

- They can contain **constraints** on the combinations of features
- They can define „**names**“ for specific combinations of features; feature groups
- Features can be **open**: additional subfeatures can be added
- Features can be **incomplete**: the subfeatures are not yet defined
- **Multiplicity of subfeatures** can be added
- And more...

CONTENTS

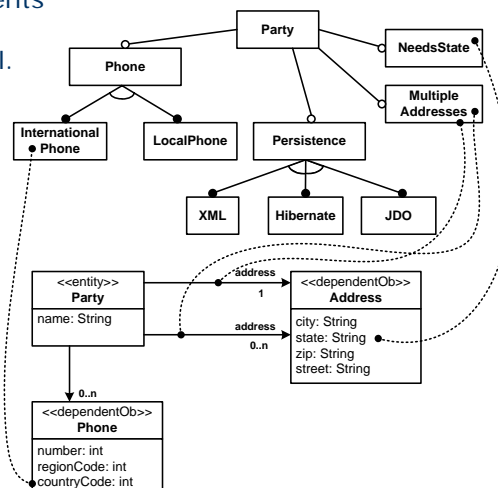
- Software System Families
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - **Combination of Both**
- Binding Times
- Techniques for Handling Variabilities
 - Source time
 - Compile time
 - Deployment/Configuration time
 - Link time
 - Run time
- Summary

Variants and Models II

- It is especially useful to **combine structural and non-structural** variations
 - specifically, you may want to „configure“ structural models with the help of feature models,
 - we want to describe variants of structural models (and use these variants as generator input)
- Examples:
 - A party may have one or more addresses
 - A party may store telecontacts or not
 - In case of telephone numbers, you may want to store the country code
 - Addresses may have the *state* field (USA)

Variants and Models III

- You can **assign** model elements of the structural model to features in the feature model.
 - The respective model elements are only there if the associated feature is selected,
 - And it's removed, if the feature is not there.



CONTENTS

- Software System Families
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- **Binding Times**
- Techniques for Handling Variabilities
 - Source time
 - Compile time
 - Deployment/Configuration time
 - Link time
 - Run time
- Summary



Binding Time Analysis

- A feature diagram defines the common and variable parts of a system. It has to be determined **when it needs to be decided** if a specific (sub-) feature will be present in a specific product in the family. This is called **binding time**.
- Binding time has **consequences** on
 - flexibility
 - performance
 - code size
 - type safety
 - and: on the **technique used to implement** the variable aspect



Typical Binding Times & Techniques

- **source time:** manual programming, generators
- **Compile time:** function overloading, precompiler, template evaluation, static aspect weaving
- **deployment/configuration time:** component deployment (impl. for an interface), environment variables
- **link time:** DLLs, class loading
- **run time:** virtual functions, inheritance & polymorphism, factory-based instance creation, delegation, meta programming

Binding Time Consequences

	flexibility	performance	code size	complexity
source time	-	+	+	-
compile time	+	+	+	-
link time	+	+	+	-
load time	++	+	+	+
run time	+++	-	-	+

CONTENTS

- Software System Families
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- Binding Times
- **Techniques for Handling Variabilities**
 - **Source time**
 - Compile time
 - Deployment/Configuration time
 - Link time
 - Run time
- Summary



Manual Programming

- Handling variabilities by manually programming is the **simplest way** of handling variabilities.
- However, it is obviously **very inflexible**, since, whenever something changes you have to go back to the code and change it manually.
- Actually, there's **no variability in the code**.
- We won't elaborate this any further.



Generators

- With **generators**, you can also create modifications on source level ... Simply by generating different code based on models
- Can be **very efficient**, since it happens at source time...
- ... And is nevertheless **very flexible**.
- **But that's another talk...**

CONTENTS

- Software System Families
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- Binding Times
- **Techniques for Handling Variabilities**
 - Source time
 - **Compile time**
 - Deployment/Configuration time
 - Link time
 - Run time
- Summary

Function/Method Overloading

- Assume the following piece of code:

```
public class Calculator {
    public int add( int x, int y ) {
        return x+y;
    }
    public double add( double x, double y ) {
        return x+y;
    }
}

// somewhere else..
Calculator c = new Calculator();
int res = c.add(3,5); // calls the first one
double res2 = c.add(3.1415, 1.142); // calls the second one
```

- Here we use **compile-time overloading**, not polymorphism!

DEMO:
wsaj/.../overloading

Preprocessors: C/C++ Examples

- The following statement **replaces the statement itself** with the **content of the file** specified as part of the statement:

```
#include „iostream.h“
```

- The next statement is a **conditional statement** that includes the part between the *#if* and the *#endif*.

```
#if defined (ACE_HAS_TLI)
    static ssize_t t_snd_n (ACE_HANDLE handle,
        const void *buf, size_t len, int flags,
        const ACE_Time_Value *timeout = 0,
        size_t *bytes_transferred = 0);
#endif /* ACE_HAS_TLI */
```

- The *ACE_HAS_TLI* can be considered a **boolean variable** that can be *defined* or *undefined*.

```
#define ACE_HAS_TLI // defines ACE_HAS_TLI, sets it true
```

Simple Macros: C/C++ Examples (II)

- A typical case is to make sure **include files are only included once** per compilation unit.

```
#if !defined(ComponentHelloWorldIncluded)
#define ComponentHelloWorldIncluded
#include "components\HelloWorld\HelloWorldSI.h"
#include "components\HelloWorld\HelloWorldRI.h"
#include "container\Diagnosable.h"

class HelloWorld: public LifecycleInterface, public ComponentBase,
                 public HelloWorldSI,public HelloWorldRI,public Diagnosable {
private:
public:
    HelloWorld();
    ~HelloWorld();

    static int PARAMETER_NOT_DEFINED;
    static int DP_inputvoltage;
    static int DP_clientcount;

    int getDiagnosticParameter( int name );
};
#endif
```

Simple Macros: C/C++ Examples (III)

- Macros can also be used to **define constants** (although specifically C++ provides better (and typesafe) means to achieve this):

```
#define MAX_ARRAY_SIZE 200
#define AUTHORNAME MarkusVoelter
```

- Processing is done via **strict text pattern matching**. Wherever the preprocessor finds the pattern, it replaces it. It has **no clue about language semantics**.
- Some **more complex expressions involving parameters** can also be preprocessed:

```
#define MAX(x,y) (x<y ? y : x)
#define square(x) x*x
```

Simple Macros: Summary

- Macros are a useful means **to achieve simple text replacement**.
- In the **context of programming languages**, the problem is that macros are **not syntax- or semantic-aware** (and not type safe).
- As with almost anything, it **can be abused by being used too heavily** or by constructing formally legal, but nearly incomprehensible macro definitions.
- However, it is a **proven tool** and has been **used successfully** in many systems.



Template Parameters (in C++)

- Unlike the generics implementation in Java, the templates in C++ are **completely static** – this is why this is a source time mechanism.
- For every instantiated template (i.e. Template parameter) a **completely new variant** of the respective generic class is created.
- Consequently, this approach is quite efficient – but potentially produces **large images**.



Template Metaprogramming (in C++)

- Also called **compile-time metaprogramming**, because metaprograms „run“ while the program is compiled
- Uses the features of C++ **template instantiation**
- Programming style is **functional** and operates on types
- Note that some **awkward constructs** are required,
 - because C++ templates were not originally intended for this purpose
 - and many generally unknown and non-trivial features of the standard are used.
- Error reporting is usually clumsy

Template Metaprogramming (in C++) II

- A static **IF** can be used to check boolean conditions on types at compile time.

```
#include <iostream>
using namespace std;

template<bool condition, class Then, class Else>
struct IF {
    typedef Then RET;
};

//specialization for condition==false
template<class Then, class Else>
struct IF<false, Then, Else> {
    typedef Else RET;
};

void main() {
    cout << "sizeof(short) = " << sizeof(short) << endl
         << "sizeof(int) = " << sizeof(int) << endl
         << "sizeof(IF<1+2>4), short, int>::RET) = "
         << sizeof(IF<1+2>4), short, int>::RET << endl;
    IF<1+2>4, short, int>::RET i; //the type of i is int!
}
```

Static Aspect Weaving

- AOP can be used for **various reasons**
 - „fixing“ broken code
 - Separate cross-cutting (often technical) concerns
 - Handling **variants**
- Depending on the features we want in our system, we **add** additional pieces of (AspectJ) source code.
 - In our example, we can optionally add error handling
 - ... and billing
- It happens **statically**, it's woven on byte code level
 - Can also be done at deployment time...
- Using advices, you can **attach additional behaviour** to existing code.

D E M O:
wsaj/aspectJExample

CONTENTS

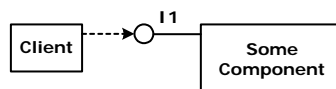
- Software System Families
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- Binding Times
- **Techniques for Handling Variabilities**
 - Source time
 - Compile time
 - **Deployment/Configuration time**
 - Link time
 - Runtime
- Summary

Component Deployment

- Consider a J2EE application server. When deploying EJBs you can
 - Pass in **configuration parameters**
 - „wire“ the **dependencies** to other components
 - Configure **security** and **transactions**,
 - ... and generally **address QoS issues** by deploying on different hardware

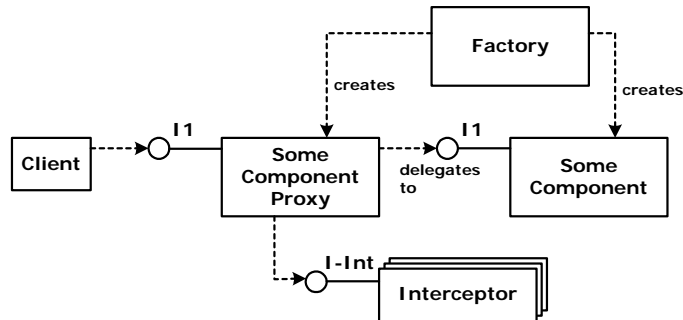
Component Deployment; Interceptors

- In general, whenever you can **add interceptors** to a system, this allows you to add/configure certain cross-cutting concerns:
 - Typically, this consists of generating **proxies** [GoF] for application components
 - that can hook-in **interceptors** [POSA2].
 - Use a **factory** to instantiate the proxies if necessary.
- Consider you face the following situation:



Component Deployment; Interceptors II

- You can replace this setup by the following:



- From a client's perspective, nothing has changed, the client still uses the interface I1. However, the client **actually talks to a proxy that handles CCC**, and then forwards to the real object.

Component Deployment; Interceptors III

- Make sure that the join points are method calls; then the following **interceptor interface** can be used:

```

public interface Interceptor {
    public void beforeInvoke( Object target,
        String methodName,
        Object[] params );
    public void afterInvoke( Object target,
        String methodName,
        Object[] params,
        Object retValue );
}
  
```

- The factory **determines which interceptors will be used** for a given object based on some kind of configuration (file).

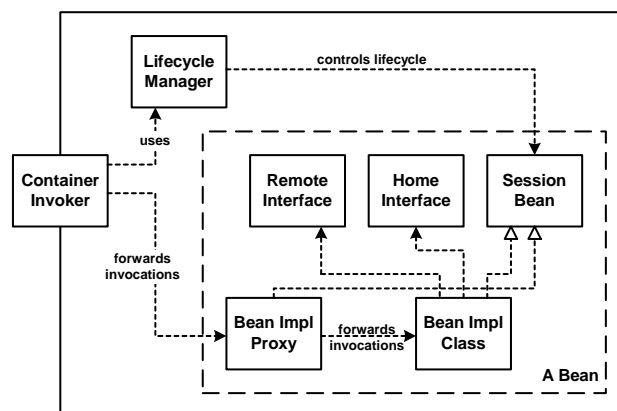
Component Deployment; Interceptors IV

- The following is the basic structure of the proxy:

```
public class SomeComponentProxy implements I1 {
    private SomeComponent delegate;
    private Interceptor interceptor; // can also be a list
                                   // of interceptors
    public String someOperation( String p1, int p2 ) {
        Object target = delegate;
        String opName = "someOperation";
        Object[] params = {p1, p2};
        Interceptor.beforeInvoke( target, opName, params );
        String res = delegate.someOperation( p1, p2 );
        Interceptor.afterInvoke( target, opName, params, res );
        return res;
    }
    // more operations of I1
}
```

Component Deployment; Interceptors V

- Example.** In the EJB scenario introduced above, the generated proxy would be the bean implementation class from the perspective of the application server, the real bean implementation would be an "implementation detail" of this class.



CONTENTS

- Software System Families
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- Binding Times
- **Techniques for Handling Variabilities**
 - Source time
 - Compile time
 - Deployment/Configuration time
 - **Link time**
 - Run time
- Summary



DLL loading and Classloading

- In static languages such as C/C++, you can load **different DLLs** that define the same entry points.
- In Java you can use **class loading** ... Although the variability mechanism in fact will be a runtime solution using polymorphism



Polymorphism

- This is well known. The method that is invoked depends on the **runtime (dynamic) type** of the object on which you invoke the operation.
 - A factory is often used in conjunction
 - Related to the strategy & bridge patterns

DEMO:
wsaj/.../polymorphism

CONTENTS

- Software System Families
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- Binding Times
- **Techniques for Handling Variabilities**
 - Source time
 - Compile time
 - Deployment/Configuration time
 - Link time
 - **Run time**
- Summary

Metaprogramming

In as much as a computational process can be constructed to **reason about an external world** in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world,

so, too, a computational process could be made to **reason about itself** in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures.

Smith, The Reflection Hypothesis



Metaprogramming in OO Languages

- There are several terms in use:
 - **Introspection/Reflection**: read/modify the program
 - **Reification**: change the semantics of existing code
- Often, the term **Meta Object Protocol** is used
- **Example Languages:**
 - **CLOS**: Reification, Reflection, Introspection, MOP, Lisp in Lisp
 - **Smalltalk**: Reflection, Dictionary, (Smalltalk := nil)...
 - **Java**: Introspection, teilweise Reflection, java.lang.Class, java.reflect
 - **Self**: Reification, Reflection, Introspection
 - **Ruby**: Reflection, Introspection



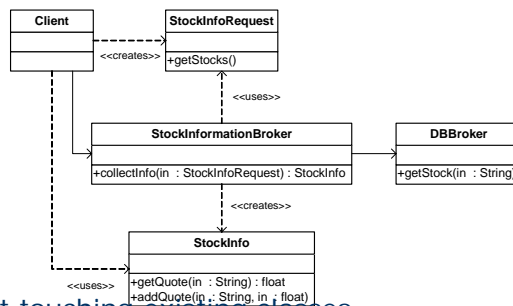
Metaprogramming in OO Languages II

- I assume you all know Java Reflection ... and it's also **not very interesting** (since it's not very powerful).
- Considering the current hype about **dynamic languages** such as Ruby, and the fact, that these languages
 - Will integrate with Java nicely not far in the future,
 - And that Java (at least, the VM) may even get native support for more dynamic languages (*invokedynamic* keyword)
- ... **I will show an Example in Ruby**
 - It shows how to handle structural variability using metaprogramming

D E M O:
wsruby

Dynamic Aspects, Virtual Classes & Collaboration Interfaces

- Example: Stock Broker
- Challenge: add **pricing** feature
 - Optionally, people can be charged
 - Different strategies should be usable
 - ... and of course without touching existing classes
- Implementation with **CaesarJ** and its composite/virtual classes
 - CaesarJ is developed by the **Software Technology Group** at TU Darmstadt, see www.caesarj.org
 - The example given here is based on an article by Iris Groher, Vaidas Gasiunas, Christa Schwanninger, Ostermann. Thanks for letting me use it!



Dynamic Aspects, Virtual Classes & Collaboration Interfaces II

- cclasses encapsulate **several „inner“ classes** which can be overridden in subclasses of the cclass.
- **provided methods** are provided by the cclass
expected methods are expected by the cclass from it's environment
- Note how this pricing feature is still **completely independent** of our stock broker application!

```

abstract cclass PricingCI {
  abstract public cclass Customer {
    /* provided */
    abstract public double getBalance();
    abstract public void charge(Item it);
    abstract public Bill createBill();
    /* expected */
    abstract public String getCustInfo();
  }
  abstract public cclass Item {
    /* provided */
    public double getPrice();
    public double getTax();
    public BillLine createBillLine();
    /* expected */
    public double getBasePrice();
    public String getItemDescr();
  }
}

```

Dynamic Aspects, Virtual Classes & Collaboration Interfaces III

- The following code shows a **possible implementation** of the PricingCI, implementing a simple pricing strategy.
- Note how the inner classes are still abstract, since their **expected methods are still not implemented**.

```

abstract cclass SimplePricing extends PricingCI {
  abstract public cclass Customer {
    private double balance;
    private List billLines;
    public double getBalance() { return balance; }
    public void charge(Item item) { balance -= item.getPrice();
      billLines.add(item.createBillLine()); }
    public Bill createBill() { String header = "Bill for " +
      getCustInfo(); ... }
  }
  abstract public cclass Item {
    public double getPrice() { return getBasePrice() + getTax(); }
    public BillLine createBillLine() {
      return new BillLine(getItemDescr(), getPrice(), getTax());
    }
    public double getTax() { ... }
  }
}

```

Dynamic Aspects, Virtual Classes & Collaboration Interfaces IV

- We now need to **bind** the pricing feature with the stock broker app. The binding code is a separate entity.
- Note how the binding **extends only the abstract CI**, not the specific pricing strategy! Here the **expected** methods are implemented.

```

abstract public cclass PerStockRequestBinding extends PricingCI {
    public cclass ClientCustomer extends Customer wraps Client {
        public String getCustInfo() {
            return wrappee.getClientName() + " " + wrappee.getClientId();
        }
    }
    public cclass StockItem extends Item wraps StockInfoRequest {
        public double getBasePrice() {return 5 +
            wrappee.getStocks().length * 0.2}
        public String getItemDescr() {return "Stock req. "+
            wrappee.getStocks().length;}
    }
    after(Client client, StockInfoRequest request) :
    (call(StockInfo StockInfoBroker.collectInfo(..)) && this(c)
    && args(request)) {
        ClientCustomer(client).charge(StockItem(request));
    }
}

```

Dynamic Aspects, Virtual Classes & Collaboration Interfaces V

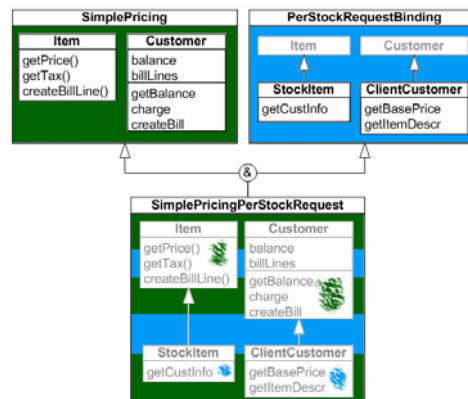
- You now need to **combine** the pricing strategy with the application binding:

```

cclass SimplePricingPerStockRequest
    extends SimplePricing & PerStockRequestBinding { }

```

- The illustration on the right shows the **result** of this combination.



Dynamic Aspects, Virtual Classes & Collaboration Interfaces VI

- You can now, at any place in your application code, **deploy the feature dynamically** and use it:

```
final PerStockRequest pricing = new SimplePricingPerStockRequest();
deploy (pricing) {
    StockInfoRequest request = new StockInfoRequest(stockList);
    StockInfo si =
        StockInformationBroker.getInstance().collectInfo(request);
    // advice is activated inside the deploy block
}
```

Dynamic Aspects, Virtual Classes & Collaboration Interfaces VII

- And of course you can **extend** the system with new strategies:

```
abstract cclass DiscountPricing extends SimplePricing {
    abstract public cclass Customer {
        protected int discountState = 4;
        public void charge(Item item) {
            if (discountState == 0) discountState = 4;
            else {
                super.charge(item); discountState--;
            }
        }
    }
}
```

```
cclass DiscountPricingPerStockRequest
    extends DiscountPricing & PerStockRequestBinding { }
```

```
final PerStockRequest pricing = new DiscountPricingPerStockRequest ();
deploy (pricing) {
    ...
}
```

CONTENTS

- Software System Families
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- Binding Times
- **Techniques for Handling Variabilities**
 - Source time
 - Compile time
 - Deployment/Configuration time
 - Link time
 - Run time
- **Summary**



Summary

- There are **many ways** how variabilities can be handled, a lot more than just the "usual suspects" strategy, factory and polymorphism
- Each of them have their particular **advantages and drawbacks** wrt. performance, complexity, flexibility and image size.
- As machines and interpreters get faster and faster, **more and more will be done dynamically**. Keep an eye on dynamic languages!



CONTENTS

- Software System Families
- Product-Line Engineering
- Variability Analysis
 - Structural Variability
 - Non-Structural Variability: Feature Modelling
 - Combination of Both
- Binding Times
- **Techniques for Handling Variabilities**
 - Source time
 - Compile time
 - Deployment/Configuration time
 - Link time
 - Runtime
- Summary

THE END.