22. Generic Programming with Generic Components

Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de

Version 13-1.0, June 4, 2013

1. Full Genericity in BETA
2. Universal Genericity with Slot Markup Languages
3. Semantic Macros
4. Template Metaprogramming
5. Evaluation

Obligatory Reading

- Invasive Software Composition, Chapter 6
- [BETA-DEF] The BETA language. Free book. http://www.daimi.au.dk/~beta/Books/. Please, select appropriate parts.
- Bent Bruun Kristensen, Ole Lehrmann Madsen, and Birger Møller-Pedersen. 2007. The when, why and why not of the BETA programming language. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages* (HOPL III). ACM, New York, NY, USA, 10-1-10-57. DOI=10.1145/1238844.1238854 http://doi.acm.org/10.1145/1238844.1238854

Literature

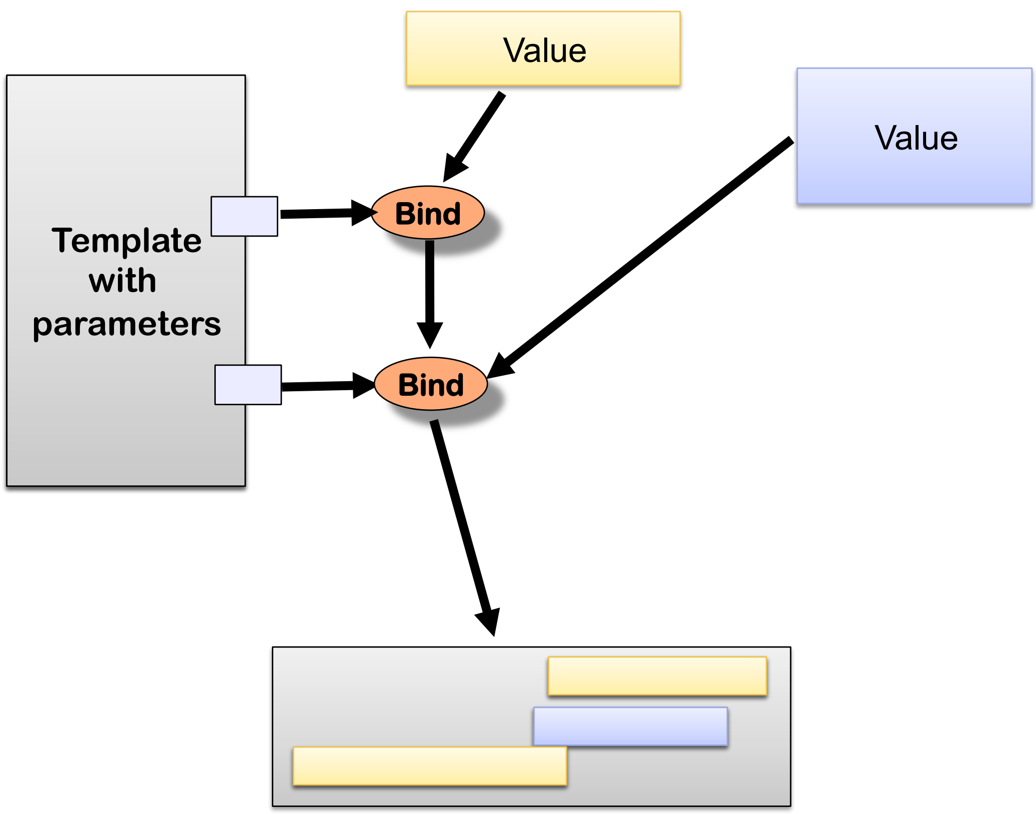
- BETA home page http://www.daimi.au.dk/~beta/
- [BETA-ENV] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, B. Magnusson. Object-Oriented Environments. The Mjölner Approach. Prenctice-Hall, 1994. Great book on BETA and its environment. Unfortunately not available on the internet.
- Ole Lehrmann Madsen. The Mjölner BETA fragment system. In [BETA-ENV]. See also http://www.daimi.au.dk/~beta/Manuals/latest/yggdrasil
- GenVoca: Batory, Don. Subjectivity and GenVoca Generators. In Sitaraman, M. (ed.). proceedings of the Fourth Int. Conference on Software Reuse, April 23-26, 1996, Orlando Florida. IEEE Computer Society Press, pages 166-175
- [CE00] K. Czarnecki, U. Eisenecker. Generative Programming. Addison-Wesley, 2000.
- J. Goguen. Principles of Parameterized Programming. In Software Reusability, Vol. I: Concepts and Models, ed. T. Biggerstaff, A. Perlis. pp. 159-225, Addison-Wesley, 1989.
- [Hartmann] Falk Hartmann. Falk Hartmann. Safe Template Processing of XML Documents. PhD thesis. Juli 2011, Technische Universität Dresden, Fakulät Informatik. http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-75342
- [Arnoldus] Jeroen Arnoldus, Jeanot Bijpost, and Mark van den Brand. 2007. Repleo: a syntax-safe template engine. In Proceedings of the 6th international conference on Generative programming and component engineering (GPCE '07). ACM, New York, NY, USA, 25-32. DOI=10.1145/1289971.1289977 http://doi.acm.org/10.1145/1289971.1289977
- The boost C++ library project http://www.boost.org/

22.1 Full Genericity in BETA

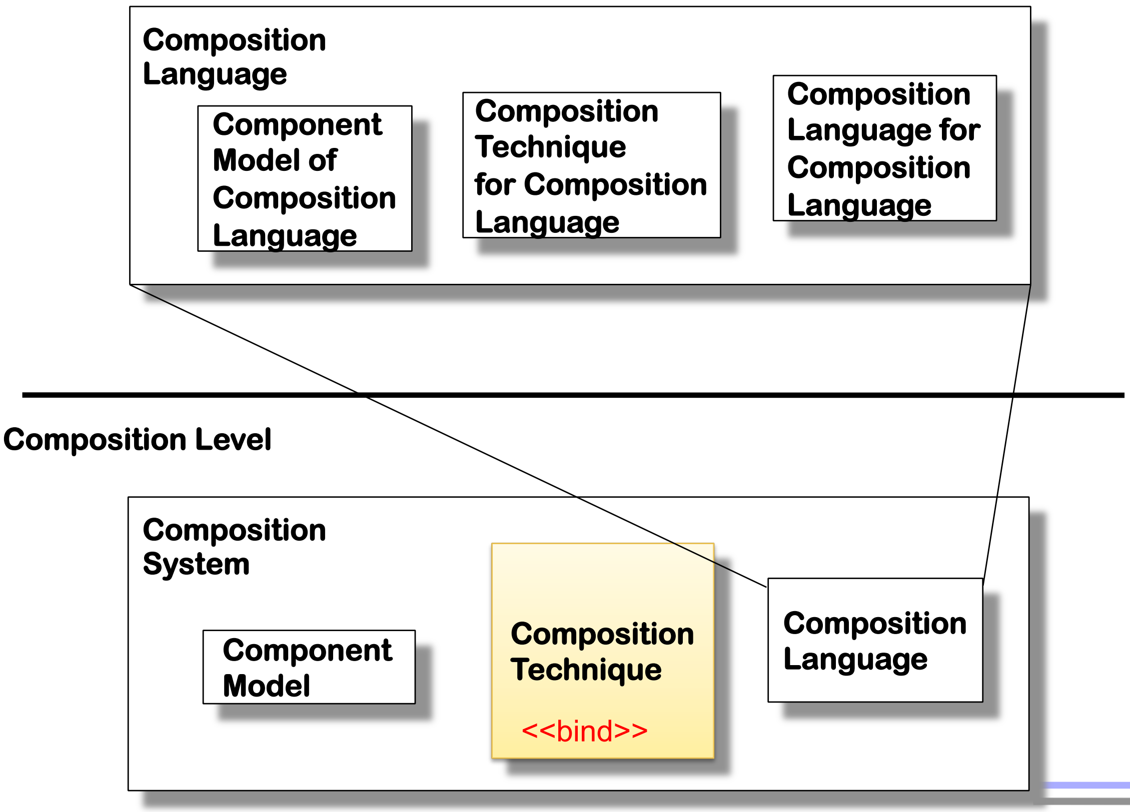
Generic Components

- A **generic component** is a *template* from which other components can be generated
  - Generic components rely on *bind* operations that bind the template parameter with a value (*parameterization*)
    - The result is called the *extent*
  - A *generic class* is a special case, in which types are parametric
- A **fully generic language** is a language, in which all language constructs can be generic
  - Then, the language need to have a *metamodel*, by which the parameters are typed

Binding Templates As Sequence of Compositions



Generic Programming is a Composition Technique Relying on the Bind Operator (Parameterization)

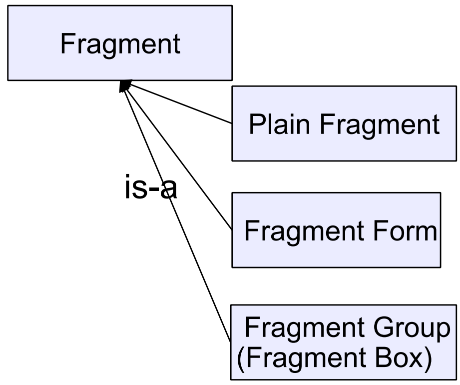


BETA Fragment Metaprogramming System

- BETA is a modern object-oriented language, developed in the North
  - BETA definition [BETA]
  - BETA programming environment Mjölner 1994 [BETA-ENV]
- Features
  - Single inheritance
  - Classes and methods are unified to *patterns (templates)*
    - Classes are instantiated statically, methods dynamically
  - Environment is controlled by BETA grammar
    - Extension of the grammar changes all tools
  - Fully generic language
  - BETA metaprogramming system *Yggdrasil*
    - Separate compilation for all sentential forms of the grammar (all fragments generatable by the grammar)
    - Essentially, a BETA module is a *generic fragment* of the language
- BETA is a better LISP, supports *typed metaprogramming*

The Component Model of BETA

- The basic module in the BETA system is a *fragment*
  - **Plain Fragment**: Sentential form, a partial sentence derived from a nonterminal
  - **Generic Fragment** (fragment form, template): Fragment that still contains nonterminals (***slots***)
  - **Fragment Group** (fragment box): Set of fragments




```
define fragment component PersonTemplate = {
  name '/home/assmann/PersonTemplate'
  Person : PatternDecl
  Person : begin
    PersonMembers : begin
      name : @String
      <<EmployerSlot : Attribute>>
    end
  end
}
```


BETA Fragments

- A **fragment** is a sequence of terminals, derived from a nonterminal in a grammar
- Example:
  - `Z ::=  Address Salary .`
  - `Address ::= FirstName SecondName Street StreetNr Town Country.`
  - `Salary ::= int.`
- Then, the following ones are fragments:
  - `Uwe Assmann Rudolfstrasse 31 Frankfurt Germany`
  - `34`
- But a complete sentence is
  - `Uwe Assmann Rudolfstrasse 31 Frankfurt Germany 34`
- A fragment can be given a *name*
  - `MyAddress: Uwe Assmann Rudolfstrasse 31 Frankfurt Germany`

Generic Fragments

- A **generic fragment** (*fragment form, sentential form*) is a sequence of terminals and nonterminals, derived from a nonterminal in a grammar
- Example:
  - `Uwe Assmann <Strasse> Frankfurt Germany`
  - `MyAddress: Uwe Assmann <Strasse> Frankfurt Germany`
- In BETA, the “left-in” nonterminals are called *slots*

Binding a Slot of a Generic Fragment in BETA

Done implicitly by name binding

```
define fragment component PersonFiller = {
  name `/home/assmann/PersonFiller'
  origin `/home/assmann/PersonTemplate'
  EmployerSlot: Attribute
  EmployerSlot: begin
    employer: @Employer;
    salary: Integer
  end
}
```

```
define fragment component PersonTemplate = {
  name '/home/assmann/PersonTemplate'
  Person : PatternDecl
  Person : begin
    PersonMembers : begin
      name : @String
      <<EmployerSlot : Attribute>>
    end
  end
}
```

```
Person : PatternDecl
Person : begin
  PersonMembers : begin
    name : @String
    employer: @Employer;
    salary: Integer
  end
end
```

Binding a Slot Seen as a Composition in BETA

- Binding a slot can be seen as a call to the bind composition operator

```
define fragment component PersonTemplate = {
  name '/home/assmann/PersonTemplate'
  Person : PatternDecl
  Person : begin
    PersonMembers : begin
      name : @String
      <<EmployerSlot : Attribute>>
    end
  end
}
```

```
define fragment component PersonFiller = {
  name `/home/assmann/PersonFiller'
  origin `/home/assmann/PersonTemplate'
  EmployerSlot: Attribute
  EmployerSlot: begin
    employer: @Employer;
    salary: Integer
  end
}
```

```
fragment Person = PersonTemplate.
EmployerSlot.bind(PersonFiller);
```

```
Person : PatternDecl
Person : begin
  PersonMembers : begin
    name : @String
    employer: @Employer;
    salary: Integer
  end
end
```


Generic Statements in BETA Syntax

```
Component methodComponent = cs.createGenericComponent();
  Hook statement = methodComponent.findSlot("MY");
if (StdoutVersion) {
  statement.bind("System.out.println("Hello World");");
} else {
  statement.bind("FileWriter.println("no way");");
```

```
public print() {
    <<MY:Statement>>;
}
```

```
public print () {
  System.out.println("Hello World");
}
```

```
public print () {
  FileWriter.println("no way");
}
```

BETA Fragment Groups

- A **fragment group** is a group of sentential forms, derived from the same nonterminal:

```
standardLoopIterators : {
   Upwards: for (int i = 0; i < array.<<len:Function>>; i++)
   Downwards: for (int i = array.<<len:Function>>-1; i >= 0; i--)
}
```

len:Funktion

```
standardLoopIterators : {
   Upwards: for (int i = 0; i < array.<<len:Function>>; i++)
   Downwards: for (int i = array.<<len:Function>>-1; i >= 0; i--)
}
```


Implicit Binding also works in BETA Fragment Groups

- Fragments can be combined with others by reference (*implicit* bind operation)
- Given the following fragments:

```
len : { size() }
standardLoopIterators : {
   Upwards: for (int i = 0; i < array.<<len:Function>>; i++)
   Downwards: for (int i = array.<<len:Function>>-1; i >= 0; i--)
}
LoopIterators : standardLoopIterators, len
```

- The reference binds all used slots to defined fragments. Result:

```
LoopIterators : {
   Upwards: for (int i = 0; i < array.size(); i++)
   Downwards: for (int i = array.size()-1; i >= 0; i--)
}
```

Advantages

- Fine-grained *fragment component model*
  - The slots of a beta fragment form its *parameterization interface*
  - The BETA compiler can compile all fragments separately
  - All language constructs can be reused
  - Type-safe composition with composition operation *bind-fragment*
  - Mjölner metaprogramming environment is one of the most powerful software IDE in the world (even after 15 years)

Full genericity: A language is called *fully generic*, if it provides genericity for every language construct.

Inclusion of Fragments into Fragment Groups

- Fragments can be inserted into others by the *include* operator
- Given the above fragments and a new one

```
whileloopbody : WHILE <<statements:statementList>> END;
```

- a while loop can be defined as follows:

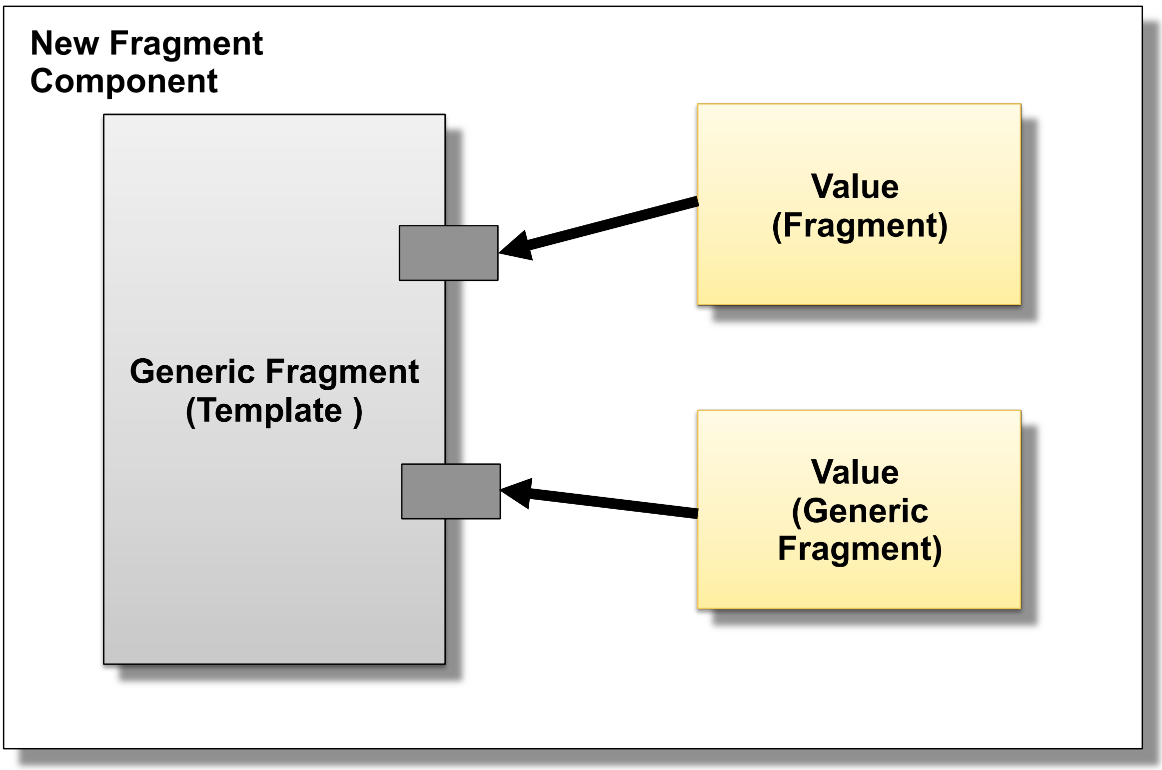
```
whileloop:
   include LoopIterators.Upwards
   whileloopbody
```

- BETA is a fully generic language:
  - Modular reuse of all language constructs
  - Separate compilation: The BETA compiler can compile every fragment separately
  - Much more flexible than ADA or C++ generics!

Evaluating BETA as a Composition System

- BETA's fragment combination facilities use as composition operations:
  - An *implicit bind* operation (fragment referencing by slots)
  - An inclusion operation (concatenation of fragments)
- Hence, BETAs composition language is rather simple, albeit powerful

Generic Components (Templates) Bind at Compile Time

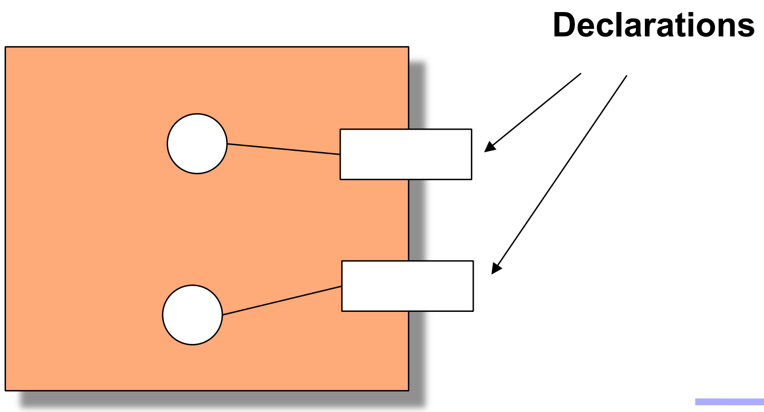


22.2 Universal Genericity with Slot Markup Languages

Slots (Declared Hooks)

Slots are declared variation points of fragments.

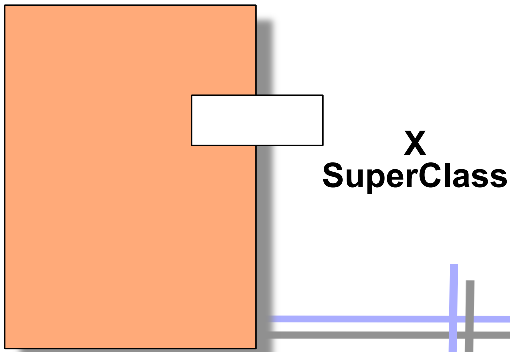
Slots (declared hooks) are declared by the component writer as code parameters



Different Ways to Declare Slots

Slots are denoted by metadata. There are different alternatives:

- Language extensions with **new keywords**
  - SlotDeclaration ::= 'slot' <Construct> <slotName> ';'
  - In BETA, angle brackets are used:
  - SlotDeclaration ::= '<<' SlotName ':' Construct '>>'
- **Markup Tags** in XML:
  - <superclasshook> X </superclasshook>
- Standardized Names (**Hungarian Notation**)
  - class Set extends *genericXSuperClass* { }
- **Comment Tags**
  - class Set /* @superClass */
- **Meta-Data Attributes**
  - Java: @superclass(X)
  - C#: [superclass(X)]

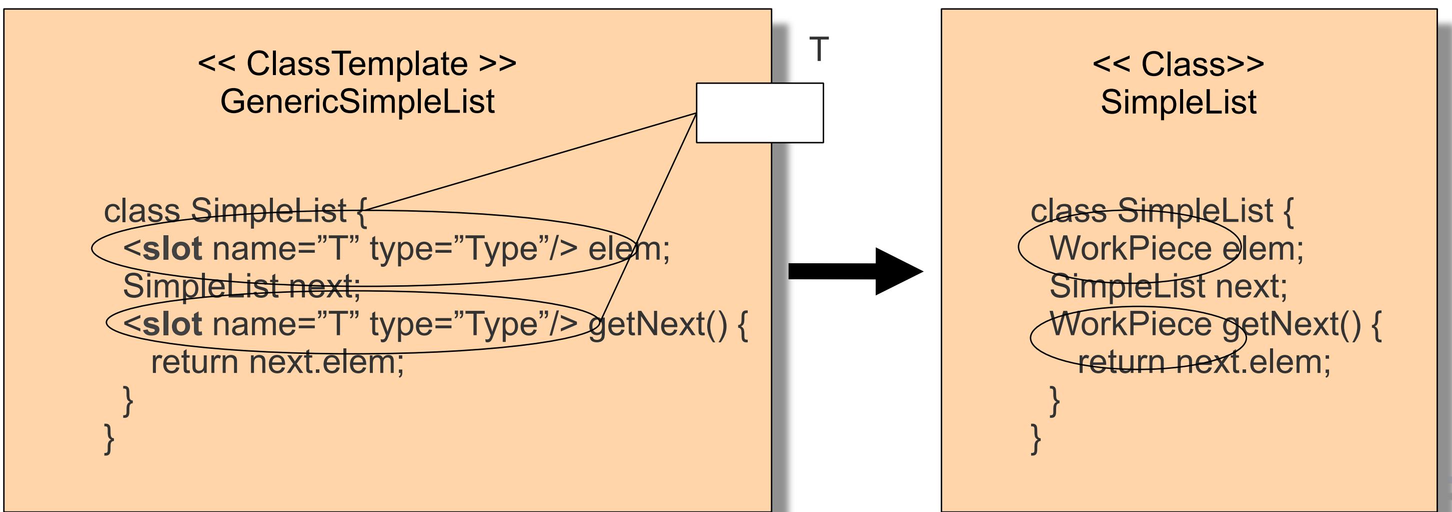


Defining Generic Types with XML Markup

[Hartmann] showed that any XML language can be enriched by a **slot markup language** to define slots

Slot markup languages use **hedge symbols** to demarcate template and slot (BETA: << >>, XML: < >)

[Arnoldus] did the same for textual languages



Generic Modifiers in XML Markup Syntax

Slot markup languages may contain elements of a composition language, io.g., control flow structures

A **slot program** expands the slot to a fragment [Hartmann]

```
Component methodComponent = cs.createGenericComponent();
  Hook modif = methodComponent.findSlot("M");
if (parallelVersion) {
  modif.bind("synchronized");
} else {
  modif.bind(" ");
```

```
<slot name="M" type="Modifier"/>
public print() {
    System.out.println("Hello World");
}
```

```
synchronized public print () {
   System.out.println("Hello World");
}
```

```
public print () {
System.out.println("Hello World");
}
```


Universal Genericity with Slot Markup Languages

- Do not use string template engines, they render development error-prone
- Use slot markup languages to exploit their typing
- With appropriate hedge symbols, a slot markup language can be combined with a base language [Hartmann]

Principle of universal genericity:
With slot markup separated by appropriate hedge symbols, any language may have typed generic components, as well as full genericity.

22.3 Semantic Macros

Semantic Macros (Hygenic Macros)

- Usually, macros are string-replacement functions (lambdas)
- Macro arguments can be typed by nonterminals (as in BETA; builds on the typed lambda calculus)

```
function makeExpression(Left:Expression, Op:Operator,
   Right:Expression):Expression {
    return Left ++ Op ++ Right;  // ++ is AST concatenation
}
function incr(a:Expression):Expression {
   return makeExpression(1,+,a); }
function sqr(a:Expression):Expression {
   return makeExpression(a,*,a); }
i:int = eval(incr(2));
// result: i == 3;
k:int = eval(sqr(10));
// result k == 100;
```

Comparing Semantic Macros and Slot Markup Languages

- Semantic Macros use the functional application symbols () as hedge symbols, i.e., are better integrated with the host language
  - Like slot programs they expand in-place
- Semantic Macros are better reusable, because they have a name
  - Slot programs are anonymous lambdas

22.4 Template Metaprogramming and Layered Template Metaprogramming

The poor man‘s generic programming

Template Metaprogramming

- Template Metaprogramming [CE00] is an attempt to realize the generic programming facilities of BETA in C++
  - C++ has templates, i.e., parameterized expressions over types, but is not a fully generic language
  - C++ template expressions are Turing-complete and are evaluated at compile time
  - C++ uses class parameterization for composition
- Disadvantage: leads to unreadable programs, since the template concept is being over-used
- Advantage: uses standard tools
- Widely used in the
  - C++ Standard Template Library STL
  - *boost* library www.boost.org
- Should be replaced by full genericity (generic fragments) or semantic macros

Template Metaprogramming in C++

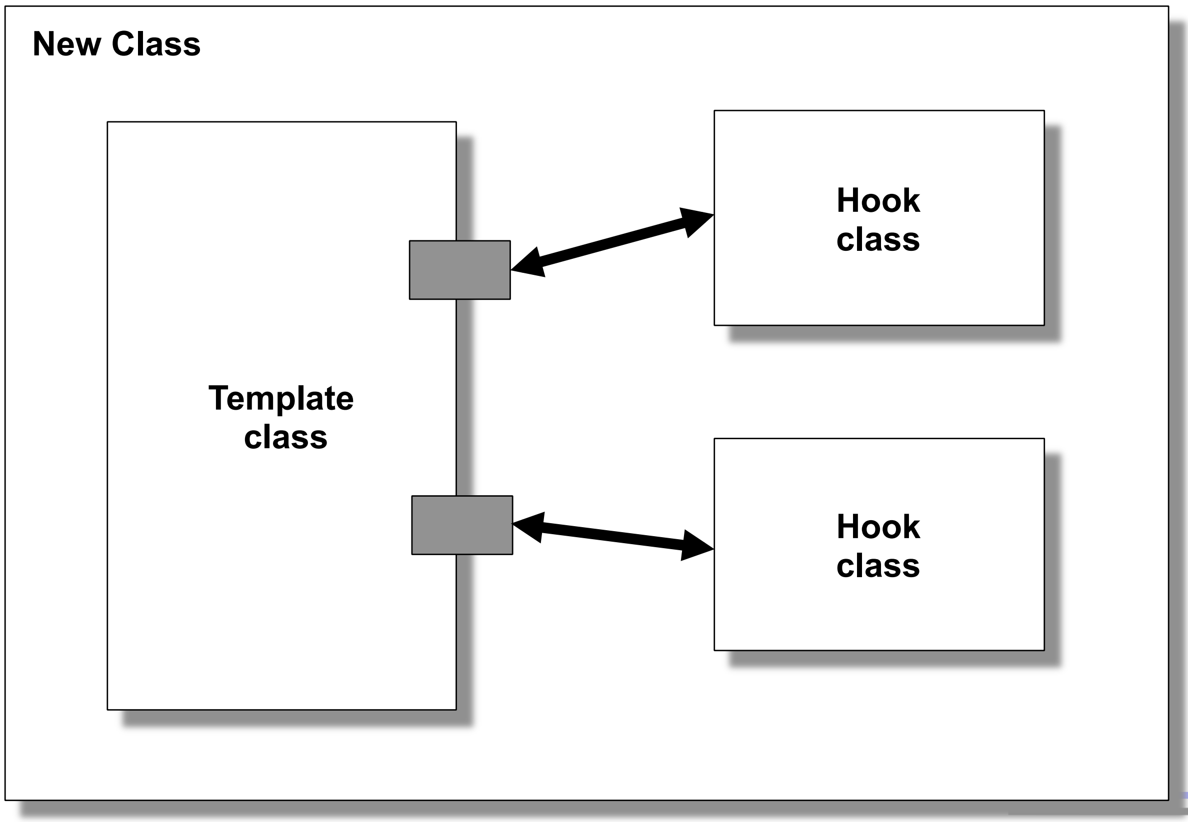
```
template <int N>
struct fact {
  enum { value = N * fact<N-1>::value };
};

template <>
struct fact<1> {
  enum { value = 1 };
};

std::cout << "5! = " << fact<5>::value << std::endl;
```

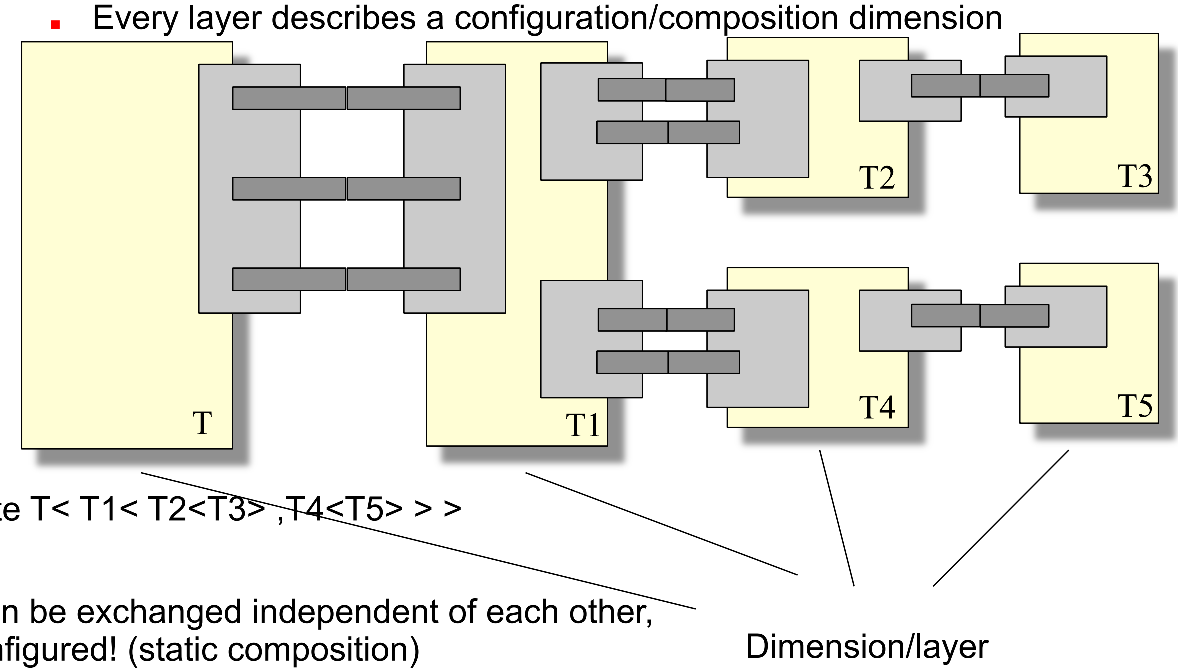
More advanced examples in [CE00]

Generic Classes (Class Templates) Bind At Compile Time



Layered Template Metaprogramming with GenVoca

- GenVoca: Composition by Nesting of Generic Classes [Batory]
- Use nesting of templates parameters to parameterise multiply
  - Every nesting level is called a *layer*
  - Every layer describes a configuration/composition dimension



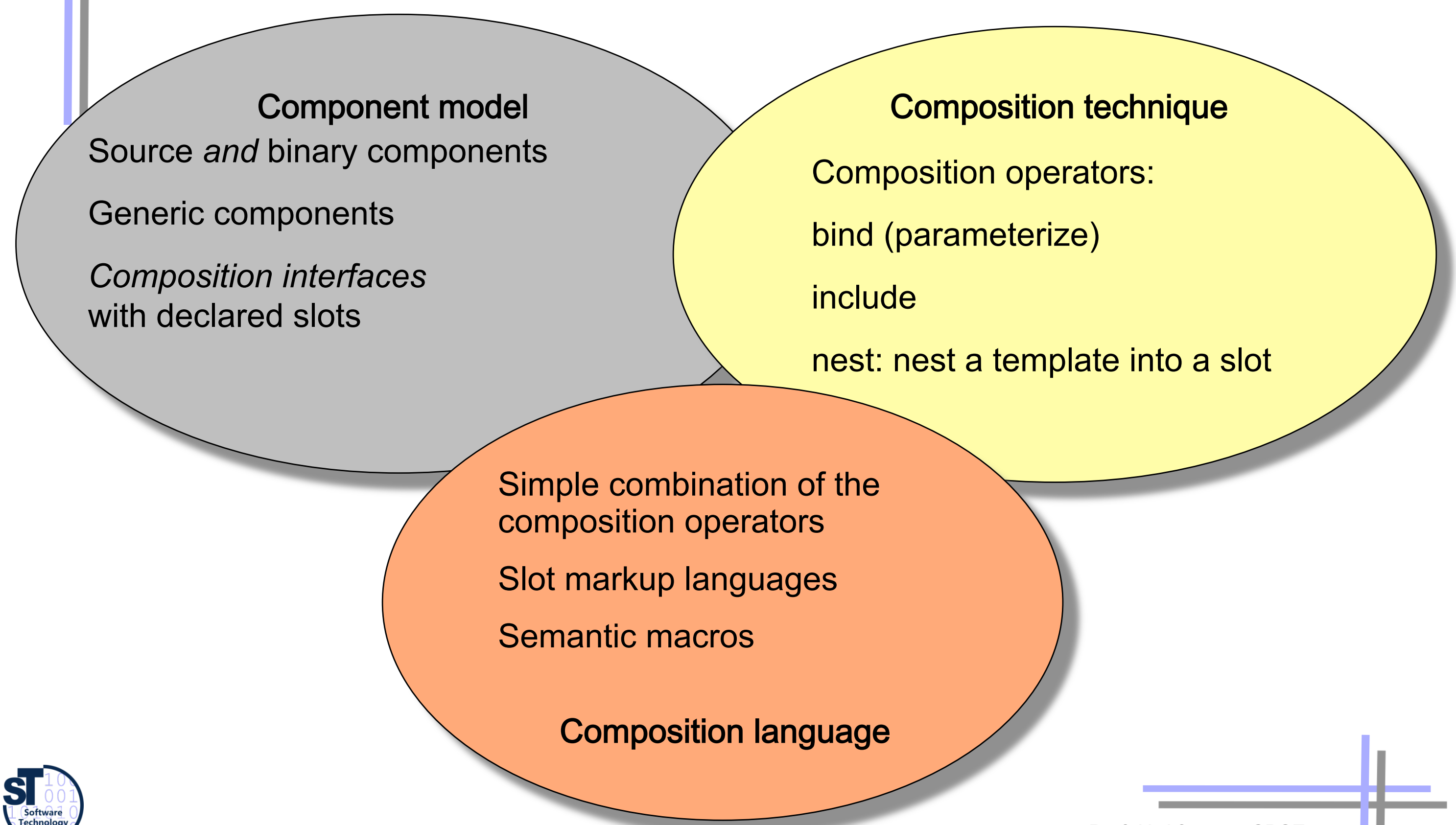
Template T< T1< T2<T3> ,T4<T5> > >

all Ti can be exchanged independent of each other, i.e., configured! (static composition)

GenVoca

- Applications
  - Parameterizing implementations of data structures
  - Synchronization code layers
- Interesting parameterization concept
  - Not that restricted as C++ templates: nested templates are a simpler form of GenVoca
  - Maps to context-free grammars. A single configuration is a word in a context-free language
  - Many tools around the technique
- However: parameterization is the only composition operator, there is no full composition language
- more in “Design Patterns and Frameworks”

22.5 Evaluating BETA Fragments, TMP, GenVoca as Composition Systems



The End

- Do not use string template engines, they render development error-prone
- Use slot markup languages and semantic macros to exploit their typing
- Look out for languages with full genericity