



# Softwarearchitekturen der Zukunft

Prof. Dr. Uwe Aßmann  
Technische Universität Dresden  
Institut für Software- und Multimediatechnologie

# ■ A Staged Architecture from Nature

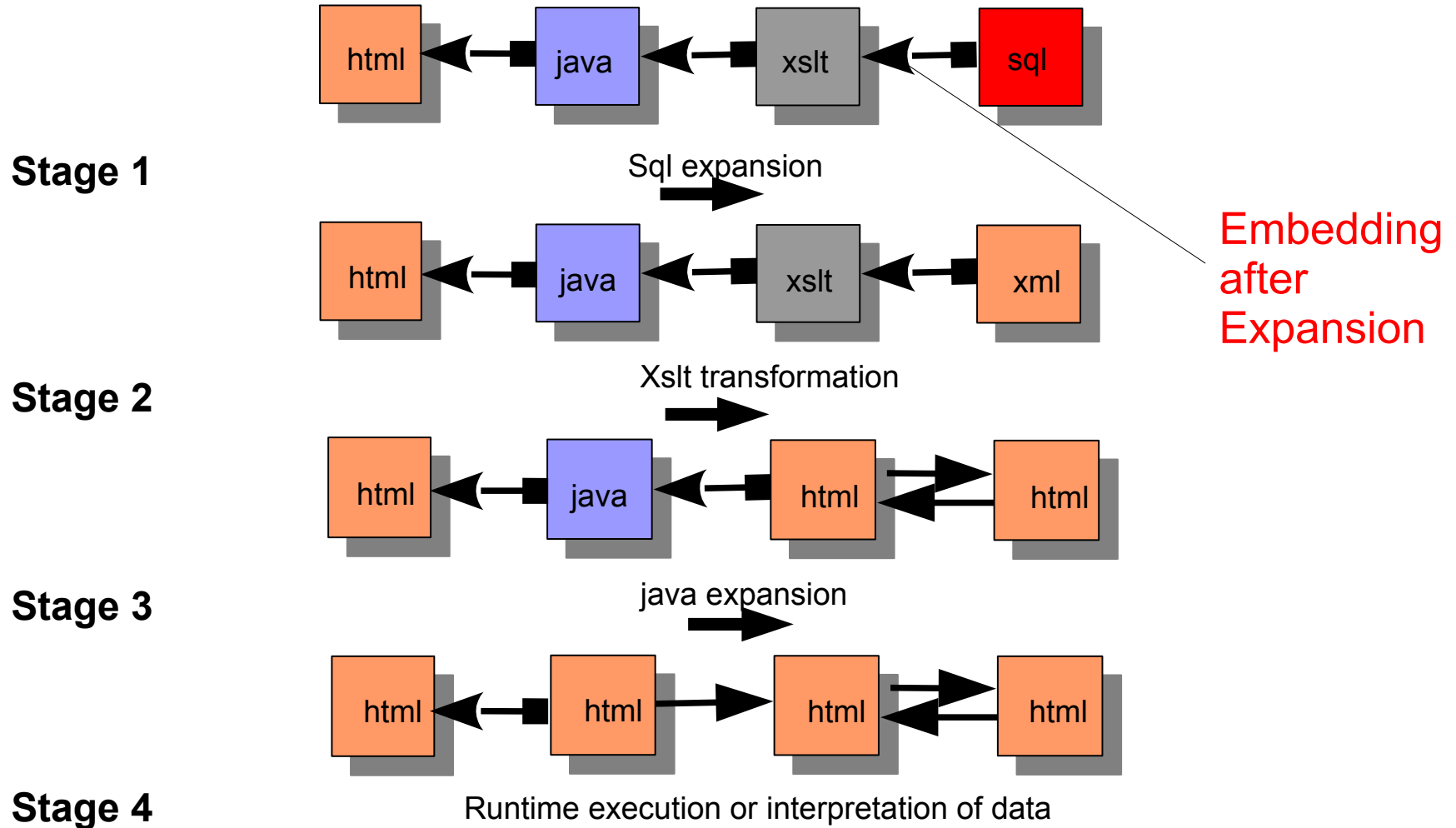




# Web Programming Considered Harmful



# Web Programming: Staged, Untyped Template Expansion



# ■ Problems of Template Expansion

Untyped

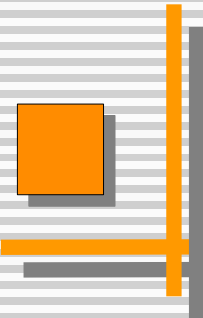
Error-prone

Comprehension very difficult, due to the different stages

Spaghetti-code-like

Scripts mixed with templates

Only valuable for programming-in-the-small




# Problem 1: Untyped Template Expansion

# ■ Elements of Software Composition

**Component Model**

**Composition Technique**

**Composition Language**



# The Composition Technique of Invasive Composition

**Invasive Composition  
adapts and extends  
fragment components  
at hooks**



# The Component Model of Invasive Composition

The component is a fragment component (template)

A subword of the language, with *holes*

Hooks are variation points of a component

Parameters

Positions, which are subject to change

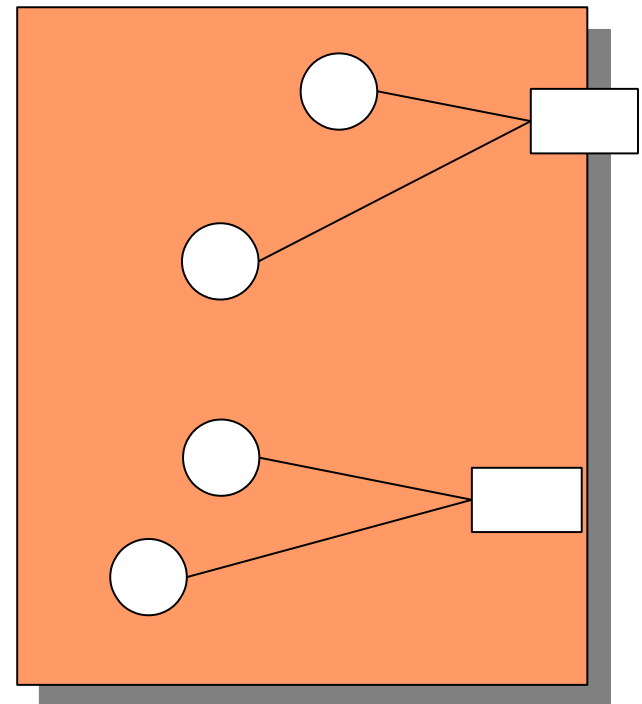
Example:

A generic class or package

A method with a extensible entry or exit

A generic XML tree

A XML list with extension points



# Generic Classes as Fragment Components

<< ClassBox >>

T

```
class SimpleList {  
    genericTType elem;  
    SimpleList next;  
    genericTType getNext() {  
        return next.elem;  
    }  
}
```

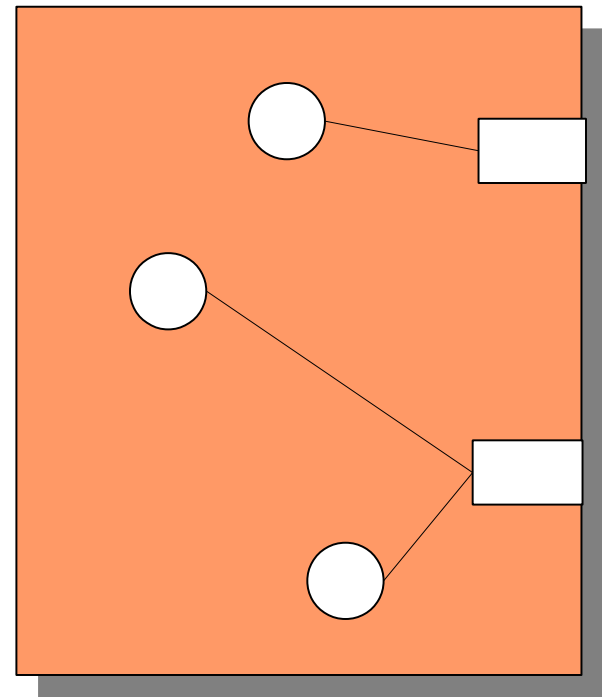
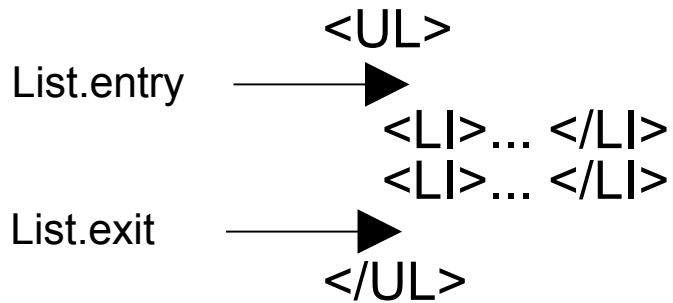


<< ClassBox >>

```
class SimpleList {  
    WorkPiece elem;  
    SimpleList next;  
    WorkPiece getNext() {  
        return next.elem;  
    }  
}
```

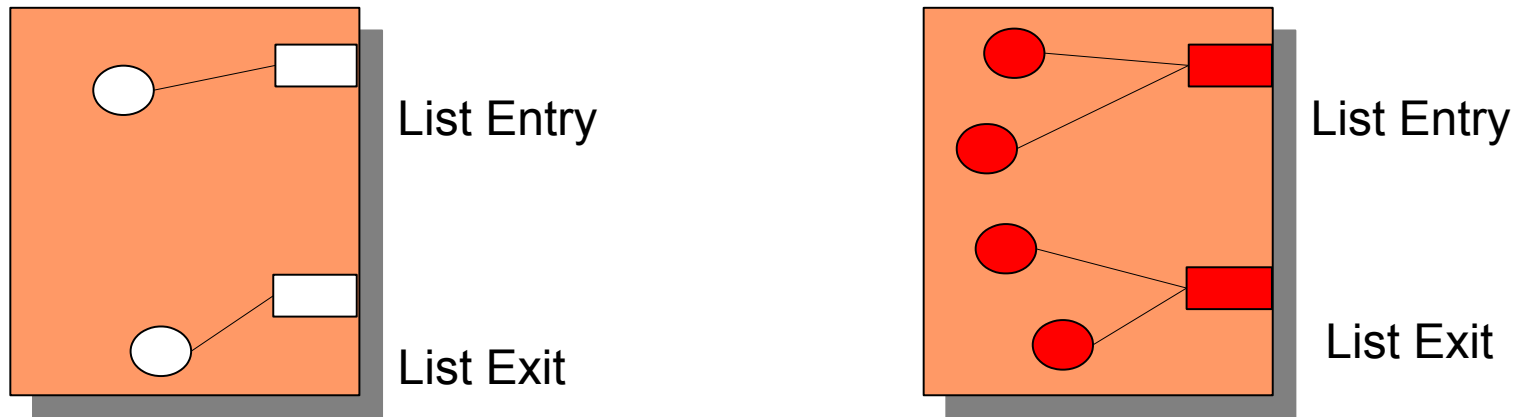
# XML Fragment Components

## Example List Entry/Exit



Hook types are given by the Xschema, the metamodel

# Typed Hook Expansion for XML Components



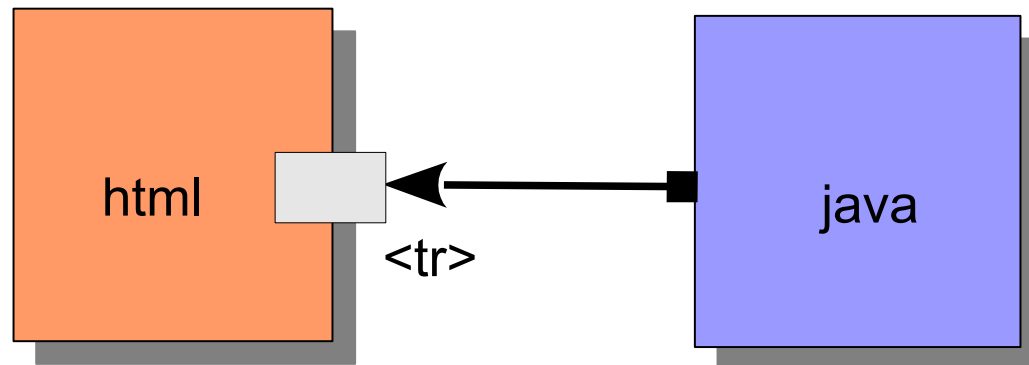
```
<UL>  
  <LI>... </LI>  
  <LI>... </LI>  
</UL>
```

```
<UL>  
  <LI>... </LI>  
  <LI>... </LI>  
  <LI>... </LI>  
  <LI>... </LI>  
</UL>
```

```
XMLcomponent.findHook(„ListEntry“).extend(„<LI>... </LI>“);  
XMLcomponent.findHook(„ListExit“).extend(„<LI>... </LI>“);
```

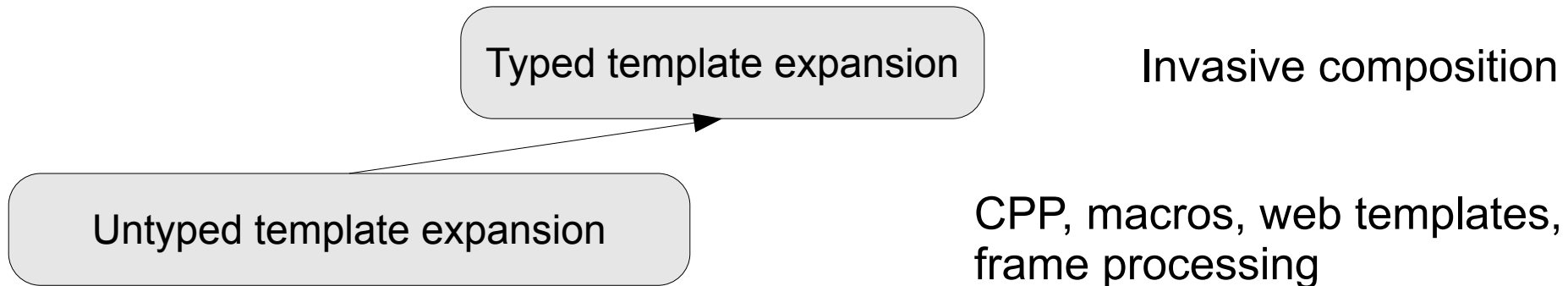
# ■ Type-Safe Template Expansion

How can you be sure that table rows are filled in?



Answer: the compiler of the invasive composition program will tell you.

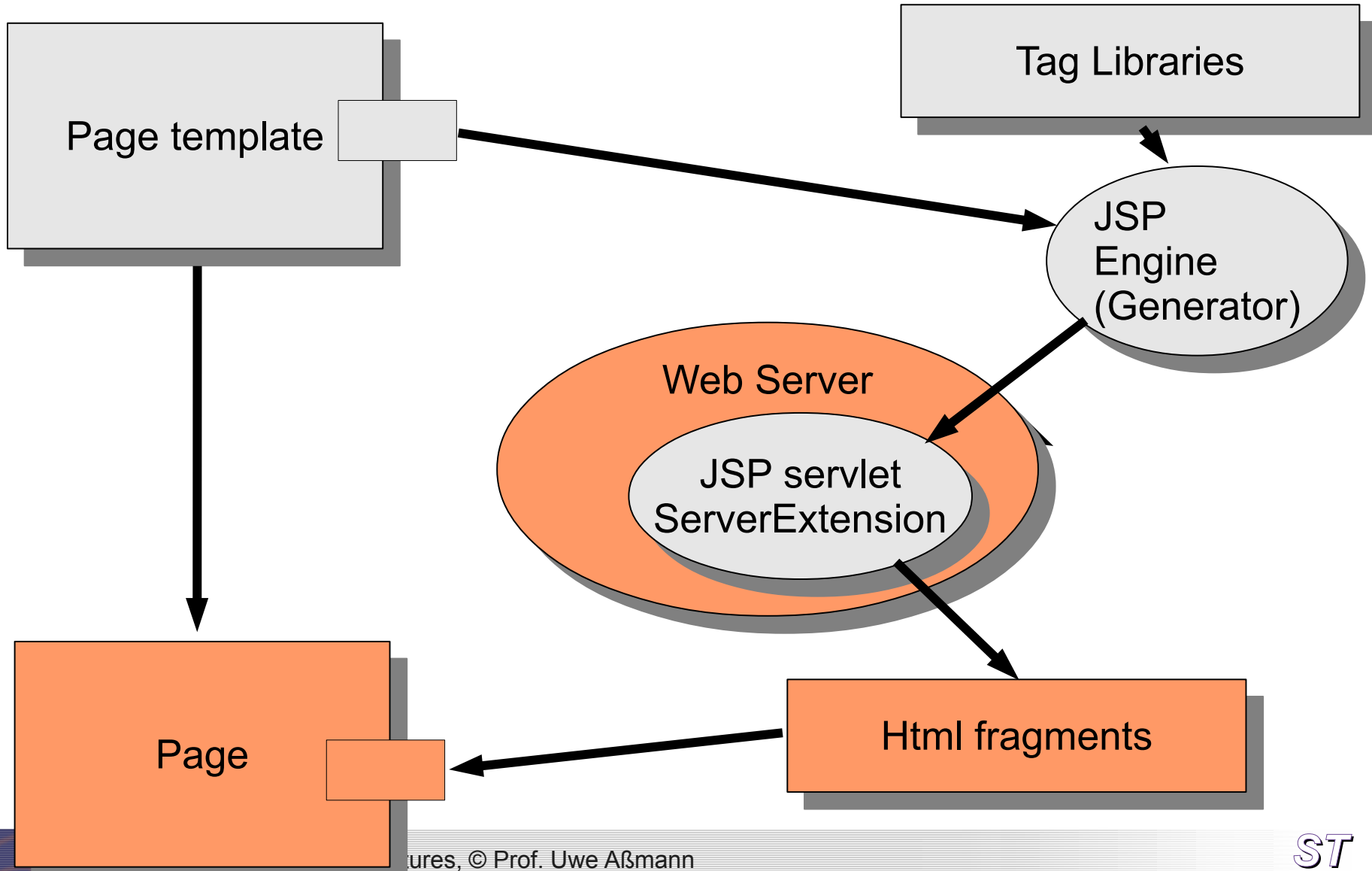
# ■ The Hierarchy of Staged Architectures





# Problem 2: Staging

# The JSP Mechanism





# Spagetti Code from JSP Tutorial

```
<html>
<%@page language="java" imports="java.util.*" %>
<h1> Welcome! </h1>
<jsp:useBean id="clock" class="jspCalendar" />
<p> Today is
<%=clock.getYear() %>-<%=clock.dayOfTheMonth() %>
</p>
<p>
<% if (Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM) %>
    Good Morning!
<% }else { %>
    Good afternoon...
<% } %>
</p>
<html>
```

# Spaghetti Code - Belongs to Different Execution Stages

```
<html>
<%@page language="java" imports="java.util.*" %>
<h1> Welcome! </h1>
<jsp:useBean id="clock" class="jspCalendar" />
<p> Today is
<%=clock.getYear() %>-<%=clock.dayOfTheMonth() %>
</p>
<p>
<% if (Calender.getInstance().get(Calendar.AM_PM) == Clalender.AM) %>
    Good Morning!
<% }else { %>
    Good afternoon...
<% } %>
</p>
<html>
```

# A Web Scripting Language with 5 Stages [www.xml4all.com](http://www.xml4all.com)

```
<xfa1:profession>
  <xfa2:ref pop-up>
    <sql>select arbitrary lastName from bakers</sql> baker
  <xfa2:ref pop-up>
</xfa1:profession>
<xfa:function hello>
  <body>
    <h1>This is My Personal Page with XFA</h1>
    <xfa:if Odd(environment^DATE)>
      <xfa:ref message>
    <xfa:else>
      Even day. No money for <xfa1:profession> :-(
    </xfa:if>
  </body>
</xfa:function>
<xfa:function message>
  Odd day today, dear student. You may visit the <xfa1:profession> shop.
</xfa:function>
```



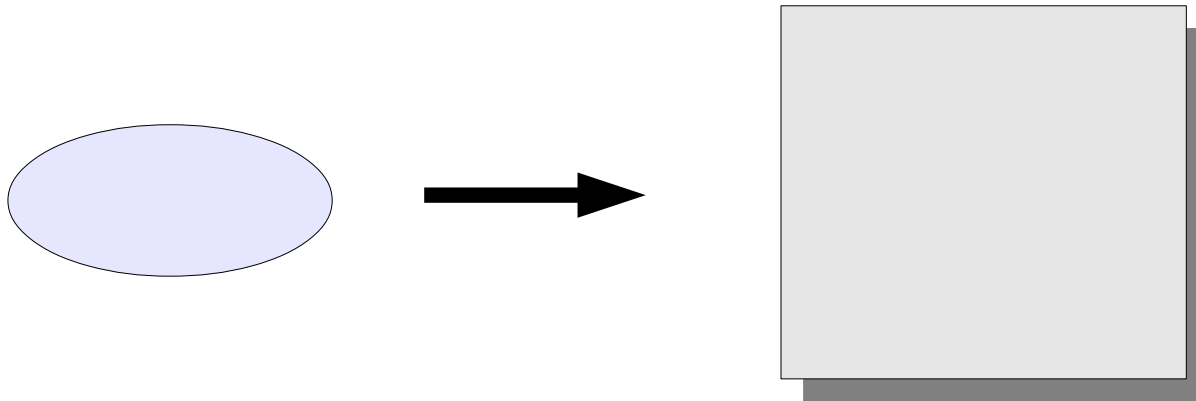
# A Possible Solution: Staged Programming

# ■ In the Beginning, there was the Data



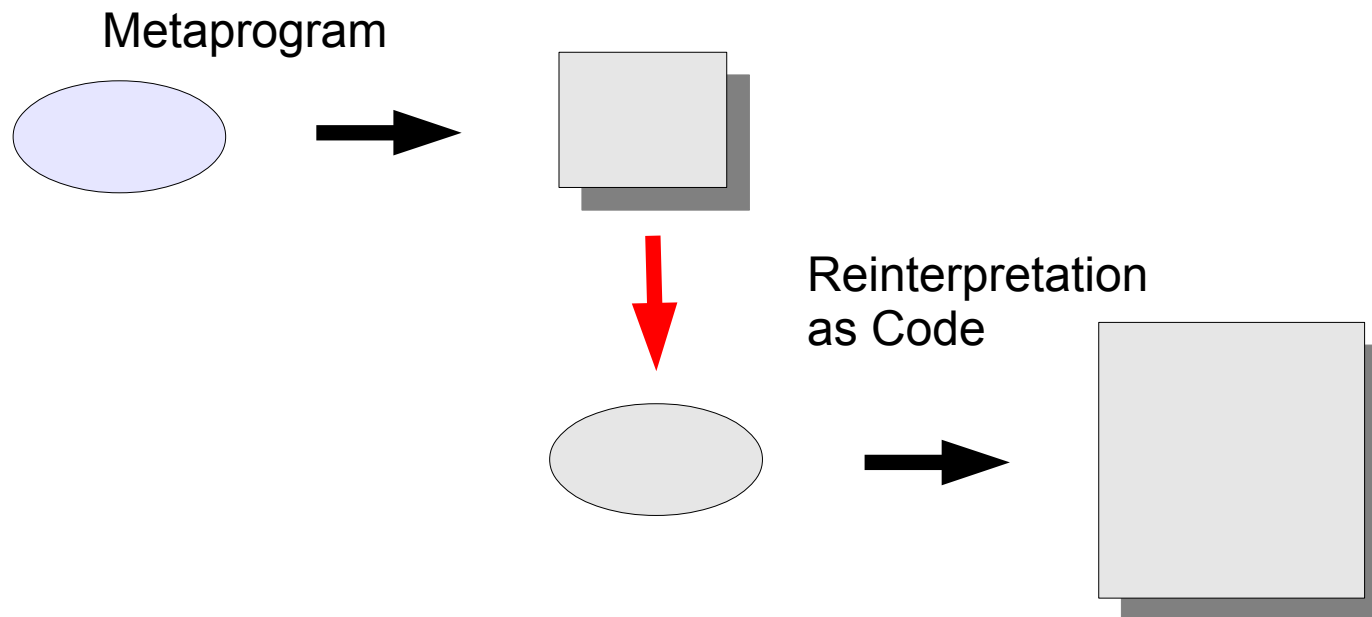
# Then Came the Programs

Producing lots of data out of little code



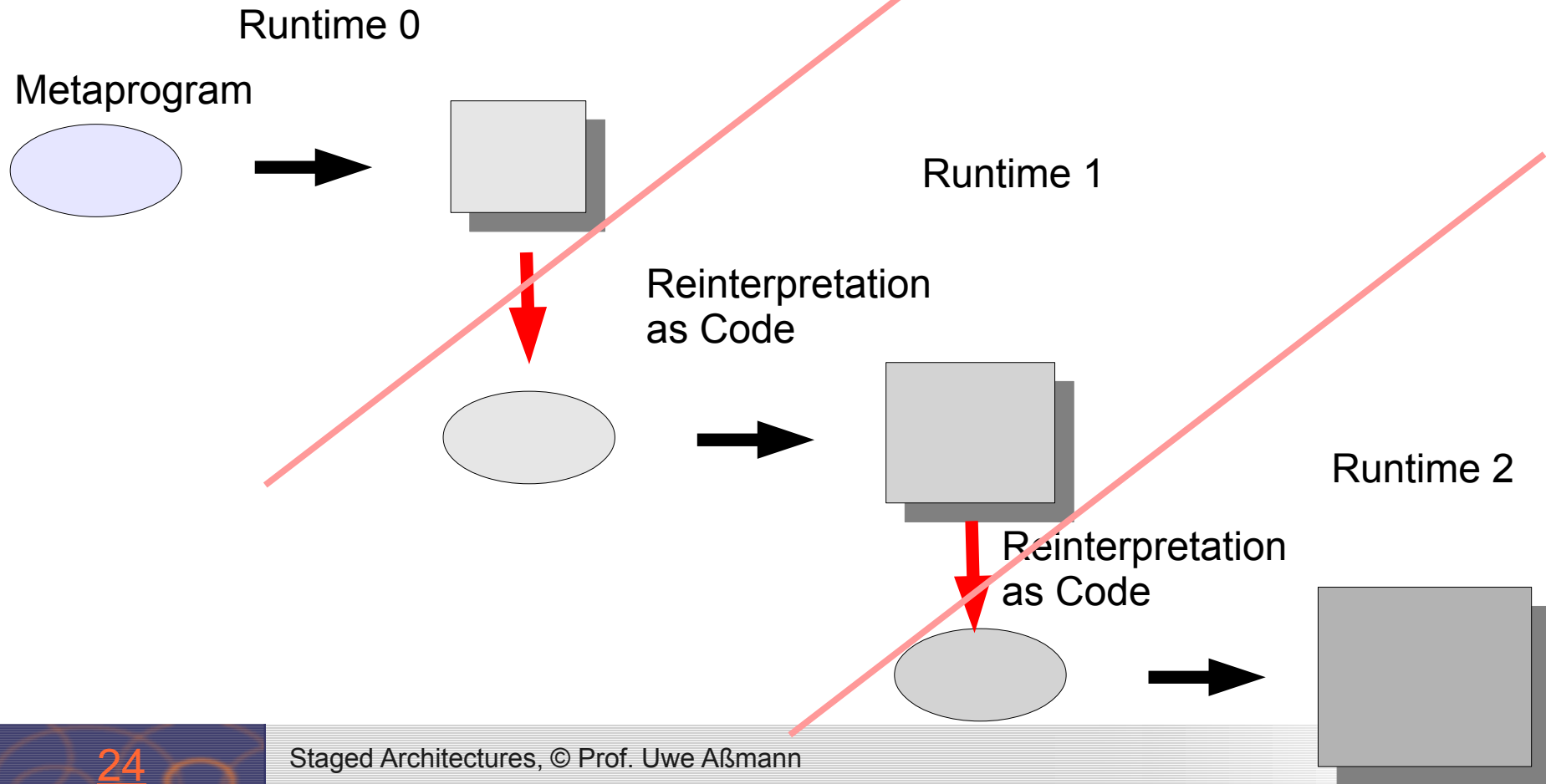
# Then Came the Metaprograms

Producing lots of programs from few metaprograms



# Then Came the Staged Metaprograms

Invented by Chiba, Sheard, Taha





# ■ Staged Programming

Staged programming (e.g., MetaML, MetaOCaML) has pioneered the mix of static metaprograms and programs

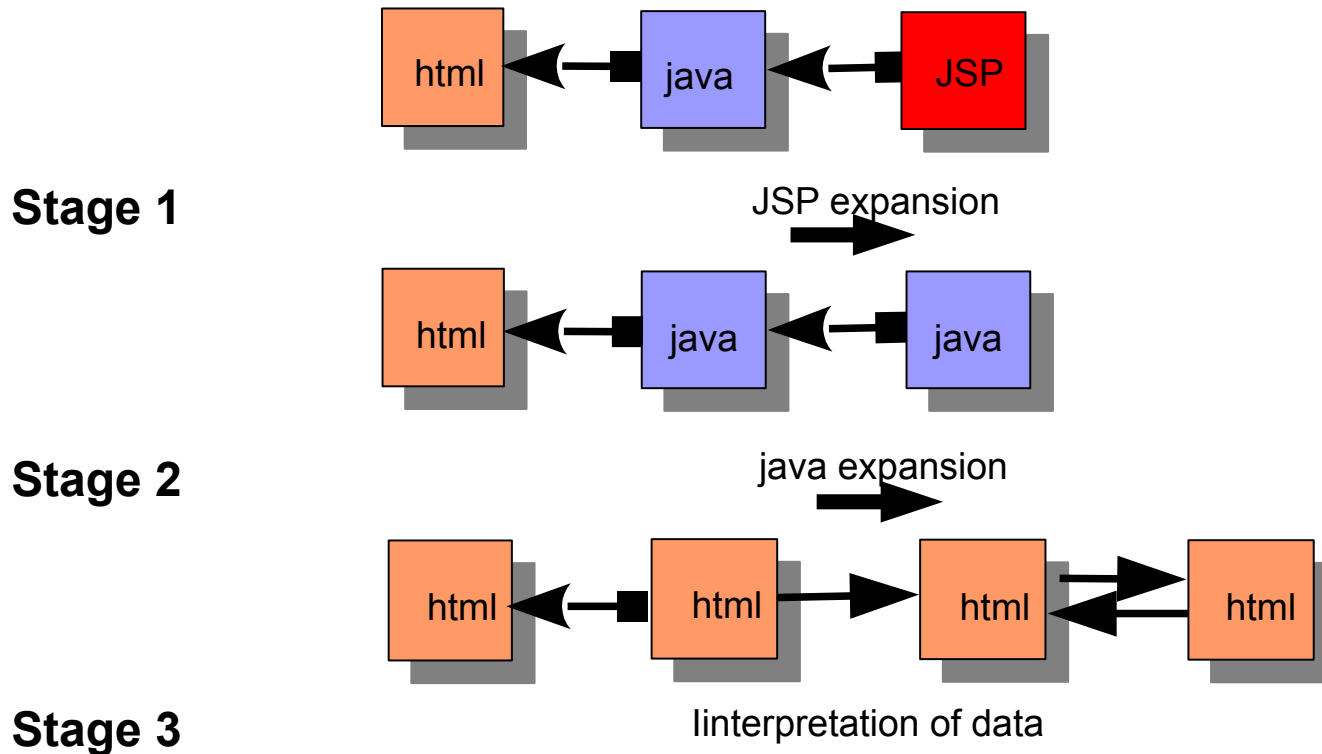
The metaprograms are expanded statically (stage 1) to produce the final program (stage 2)

Metaprograms are typed in the metamodel of the programs (type-safe expansion of metaprograms)

Example [Taha]:

```
# let a = 1+2;;  
val a: int = 3  
# let a = .<1+2>.;;  
val a: int code = .<1+2>.  
# let b = .! a;;  
val b = 3
```

# JSP Uses Staged Programming



# Spagetti Code Revisited

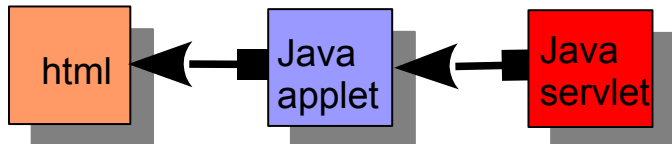
```
<html>
<%@page language="java" imports="java.util.*" %>
<h1> Welcome! </h1>
<jsp:useBean id="clock" class="jspCalendar" />
<p> Today is
<%=clock.getYear() %>-<%=clock.dayOfTheMonth() %>
</p>
<p>
<% if (Calender.getInstance().get(Calendar.AM_PM) == Clalender.AM) %>
    Good Morning!
<% }else { %>
    Good afternoon...
<% } %>
</p>
<html>
```

Servlet generator expands  
blue lines to Java code



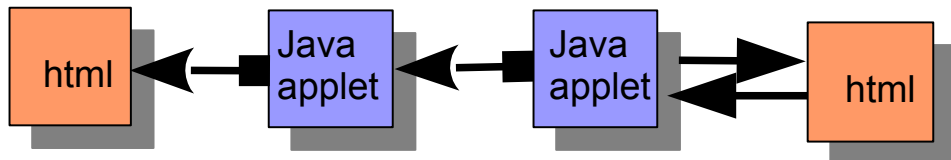
# Example 2: Staged Servlet/Applet Processing

**Stage 1**



JSP expansion  
→

**Stage 2**



java expansion  
interpretation of data

# Insight: Web Systems Need Staged Processing

Web programming is based on *staged untyped template expansion (frame processing)*

Because of the client-server stage separation

Because legacy tools must be encapsulated into a stage (e.g., databases)

And the dynamic server-based generation of web pages

And the applets

It should be based on typed template expansion (invasive composition)

# Insight 2: Web Systems Need Staged Programming

Web programming is sometimes based on  
*staged programming*

If code is generated

E.g., servlet or applet generation

Staged programming should be typed, otherwise chaotic


# N.B.: Variant Selection with Staged Programming

```
# fun f variant =  
  if variant = 1 then .<.fun q x = x*x.>.  
  else .<.fun q x = x/x.>.  
  
;;
```

```
# let variant = 1;;  
# fun g = (f variant) 2;;  
val g: int code = .<let q x =  
  x*x>.  
# let res = g 3;;  
val res = 9
```

Different behavior  
of second stage

```
# let variant = 2;;  
# let g = (f variant) 2;;  
val g: int code = .<let q x =  
  x/x>.  
# let res = g 3;;  
val res = 1
```



# Staging Is Used for Variant Management

On stage  $n-1$ , control-flow denotes variant selection for stage  $n$

*Platforms* are often selected by evaluating control-flow in previous stages



# Spagetti Code Revisited

```
#ifdef HTML
<html>
#else
<wap>
#endif
<%@page language="java" imports="java.util.*" %>
#ifdef HTML
<h1> Welcome! </h1>
#else
<bold>Welcome!</bold>
#endif
<jsp:useBean id="clock" class="jspCalendar" />
#ifdef HTML
<p>
#endif
.....
```

CPP stage selects  
HTML or WAP

Evaluating the CPP script  
chooses the platform



# The C Preprocessor as Staged Programming System

C with `#ifdef` language is a real staged programming system  
That's why it's being used...

# ■ Advantages of Staged Programming

## Typed

Type-safe development, less error-prone

## Concise representation of system

Representation is expanded through every stage

## Easy to code variants

Control flow on a build stage does variant selection

## Problems:

Still, lots of spaghetti code.



# Example: The START Conference Management System

START is a review management system

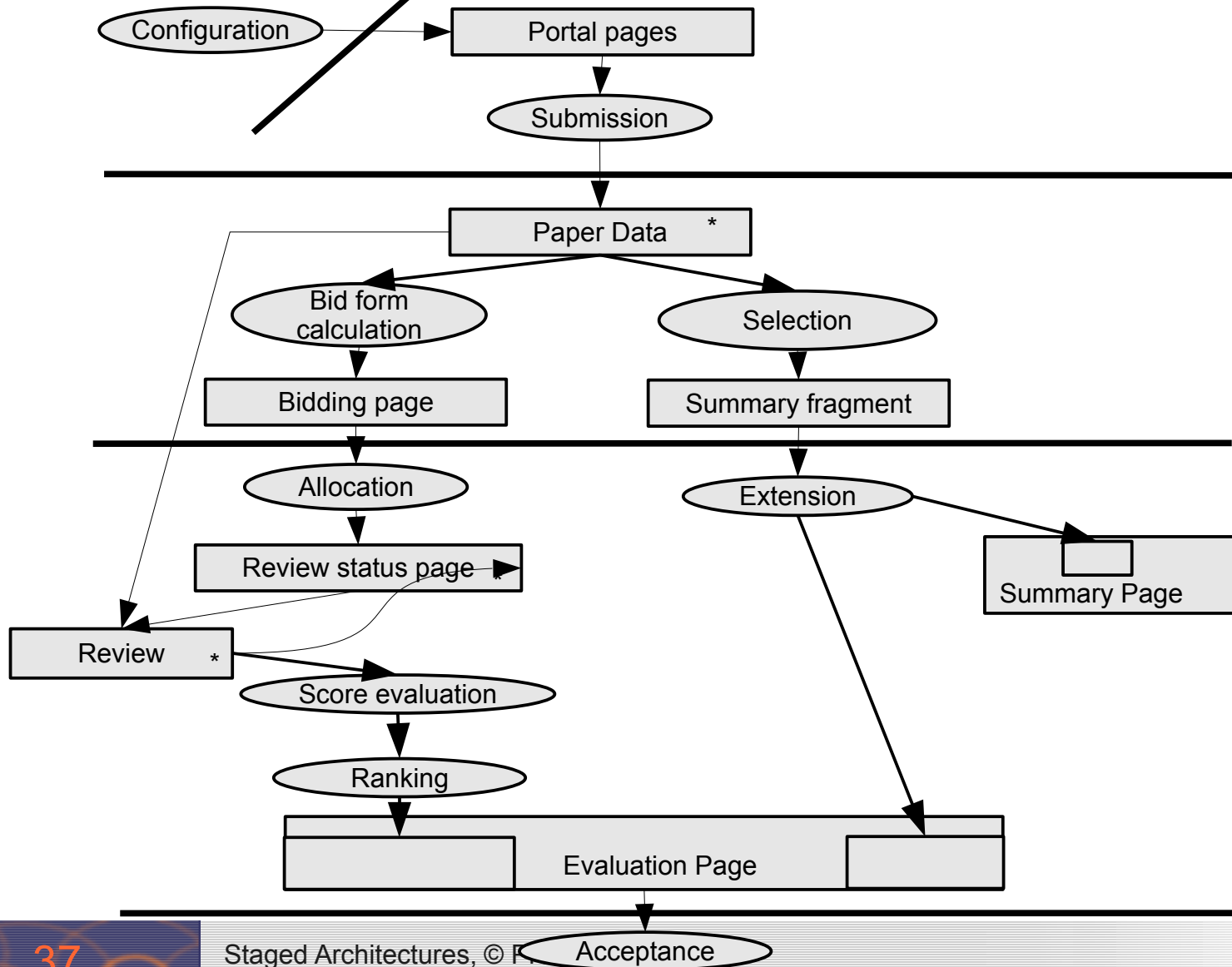
It has a 5-phase staged template expansion architecture

START servlets are composition scriptlets that compose (parameterize, extend) html-templates

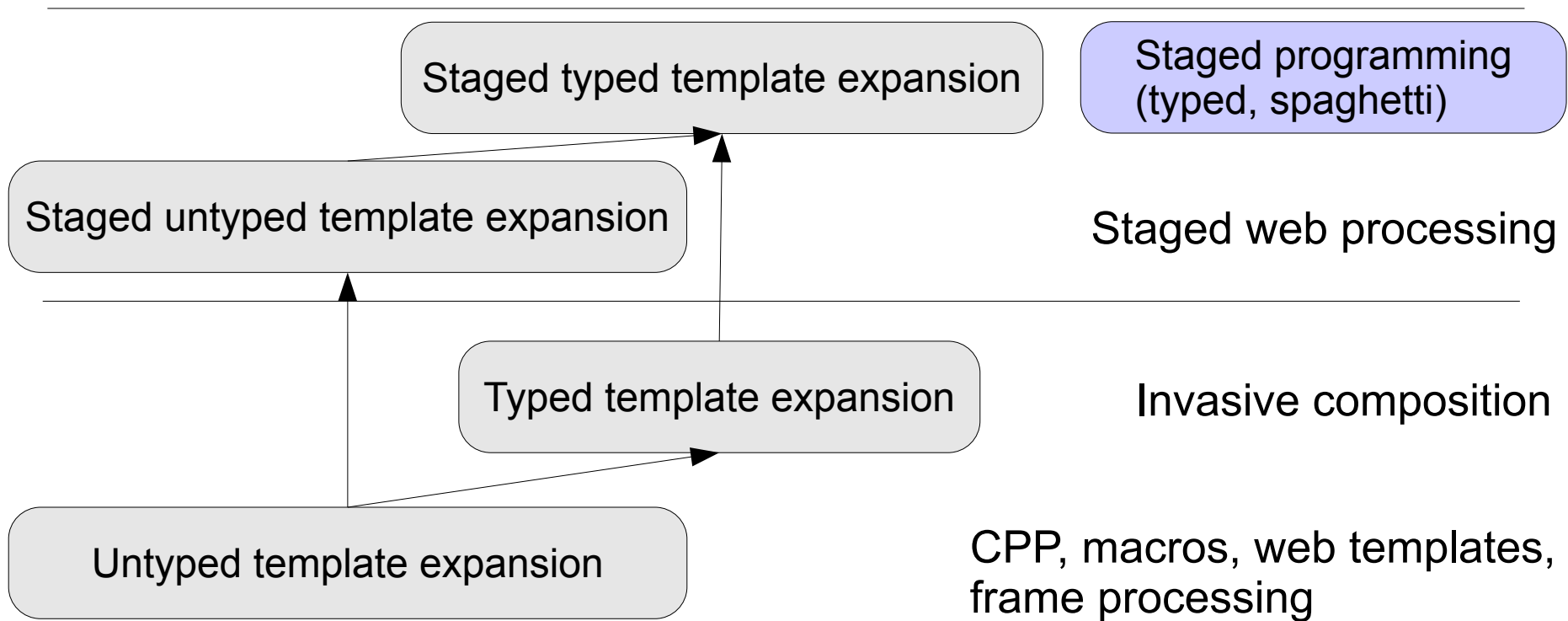
Using invasive composition, we developed a *staged typed template expansion* system

It is no problem to generate servlets, too. Then we have real staged programming

# The Staged Template Expansion Architecture of START



# ■ The Hierarchy of Staged Architectures





## Problem 3: Spaghetti Code

and a possible remedy:  
staged architectures

# Architecture and Composition

Two of the central insights of the software engineering in the 1990s are:

Separate architecture from the components

Compose components by a *composition language*

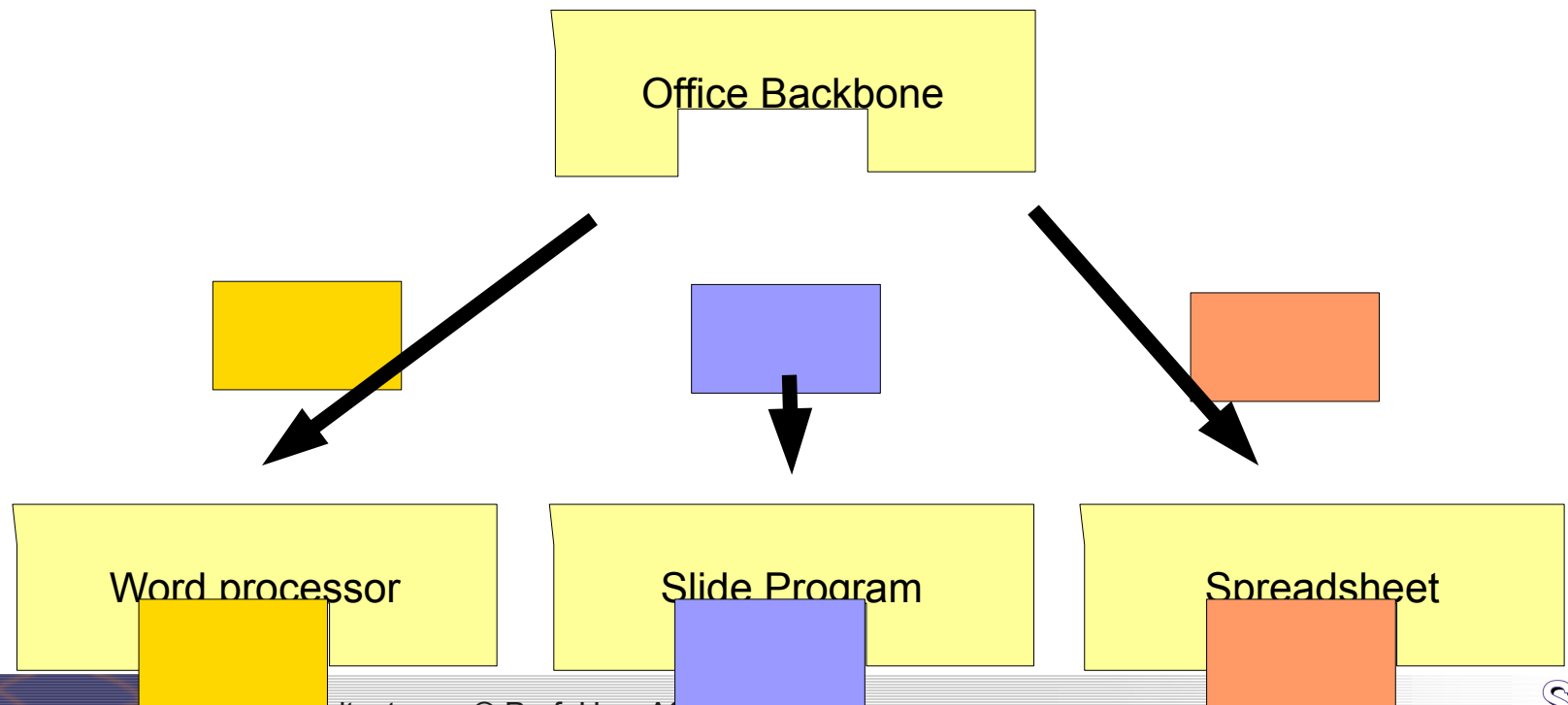


# Benefit of Architectures

Comprehensibility

Commonalities into the architectural level, variabilities into the application-specific components

Does this also hold for web programming?



# Less Spaghetti Code: A Fragment-Based Template and its Architecture

## Component

```
<html>
    <hook id="imports">
<h1> Welcome! </h1>
    <hook id="use">
<p> Today is
    <hook id="year"/>
    -<hook id="day"/>
</p>
<p>
    <hook id="greeting"/>
</p>
<html>
```

## Composition Program (Architecture)

```
public class composeTemplate {
    String use = „jspCalendar“
    String imports=„java.util.*“;

    compose() {
        Template template = read();
        Bean clock = new jspCalendar();
        String year = clock.getYear();
        String day = clock.dayOfTheMonth();
        if (Calender.getInstance().get(Calendar.AM_PM) ==
            Calender.AM)
            greeting = “Good Morning!”;
        else
            greeting = “Good afternoon...”;
        this.merge(template);
    }
}
```

# Separation of Components and Architecture Allows for Variants

```
public class composeTemplate {  
    String use =  
    String imports=  
    compose() {  
        String year =  
        String day =  
        greeting =  
    }  
}
```

Composition Program (Architecture)

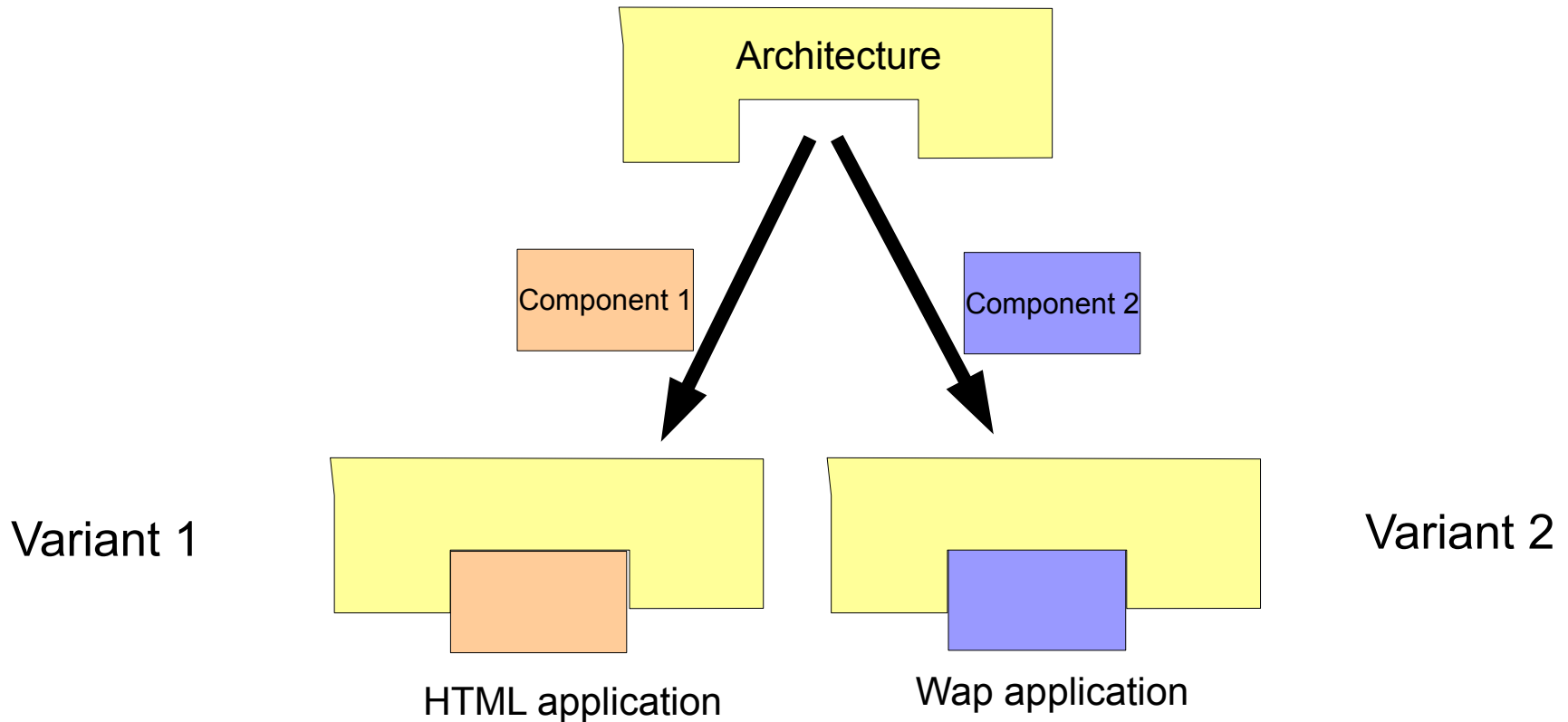
Component 1

Component 2

```
<html>  
    <hook id="imports">  
<h1> Welcome! </h1>  
    <hook id="use">  
<p> Today is <hook id="year"/>  
    -<hook id="day"/>  
</p><p> <hook id="greeting"/>  
</p>  
</html>
```

```
<wap>  
    <hook id="imports">  
<b> Welcome! </b>  
    <hook id="use">  
<p> Today is <hook id="year"/>  
    -<hook id="day"/>  
</p><p> <hook id="greeting"/>  
</p>  
</wap>
```

# ■ Variants in a Product Line



# Variant Management by Control Flow in Architectural Programs

```
public class composeTemplate {  
    if (HTML) then use component 1  
        else use component 2  
  
    String use =  
    String imports=  
    compose() {  
        String year =  
        String day =  
        greeting =  
    }  
}
```

Variant 1

Variant 2

```
<html>  
    <hook id="imports">  
<h1> Welcome! </h1>  
    <hook id="use">  
<p> Today is <hook id="year"/>  
    -<hook id="day"/>  
</p><p> <hook id="greeting"/>  
</p>  
</html>
```

```
<wap>  
    <hook id="imports">  
<s1> Welcome! </h1>  
    <hook id="use">  
<p> Today is <hook id="year"/>  
    -<hook id="day"/>  
</p><p> <hook id="greeting"/>  
</p>  
</wap>
```



# Advantages for Separating Architecture From Application Components

Isolation of commonalities into frameworks

Comprehensibility

Programming-in-the-large is separated from programming-in-the-small, components can be abstracted away

Less spaghetti

Easy variability (variant configuration)

# Definition: Staged Processing Architectures

Staged processing architectures add an explicit architectural level to staged template processing

Every stage is executed to produce **data** for the next stage

Every stage is executed at a specific time

On every stage, there is

- an architecture,

- a component model

- a composition technique,

- and a composition language

Every composition language has its own interpreter

- and is reduced (expanded) at different interpretation times



# Web Programming needs Staged Processing Architectures

It would be nice to extend staged typed template expansion to staged processing architectures.



# ■ Definition: Staged Architectures

Staged architectures combine *staged programming with an explicit architectural level*

Every stage is executed to produce **code** for the next stage

The final runtime code (architecture and components) is computed over several stages

The initial architecture is very small, the final architecture can be very large

Composition expressions, specifications, or programs may be hidden in components of a previous stage

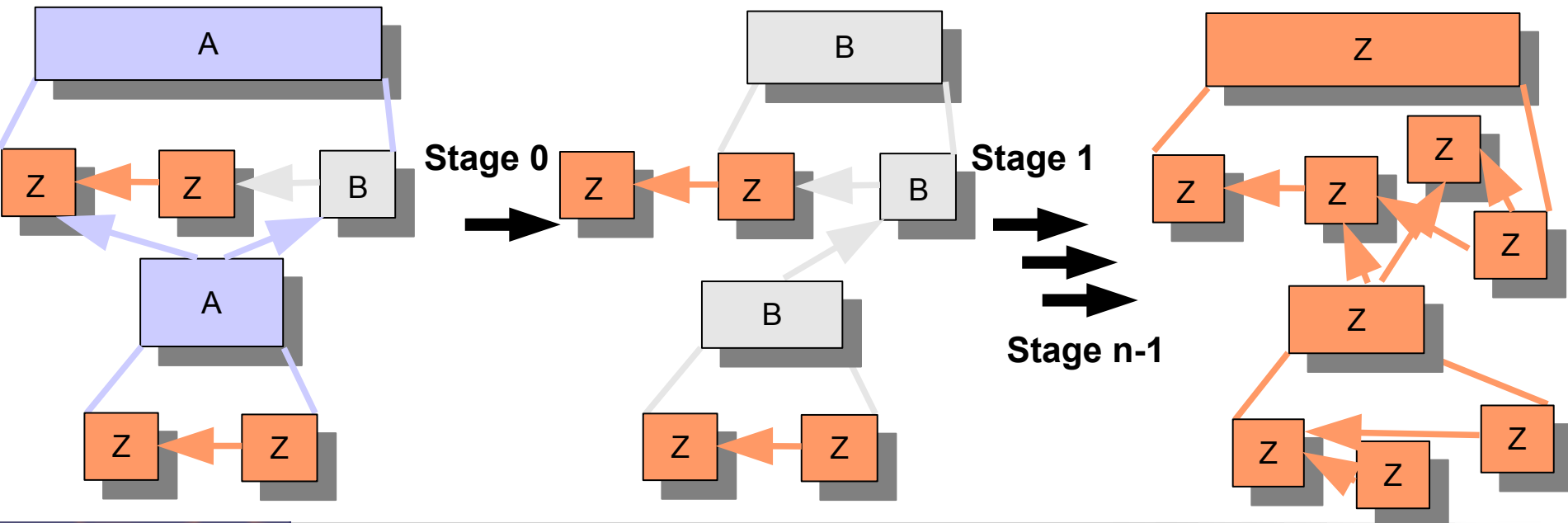


# Staged Architectures and Different Component Models

Stage-0 architecture in composition language A0  
Component language C0

Stage 0 produces Stage-1 architecture in composition language A1  
Component language C1

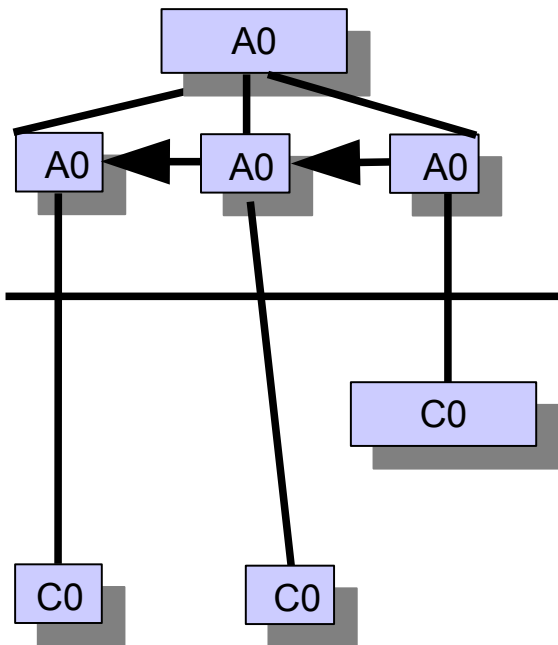
Stage n-1 produces Stage-n architecture in composition language An  
Component language Cn





# Staged Architectures Separate Large from Small

**Stage-A0 architecture in composition language A0**  
**Component language C0**



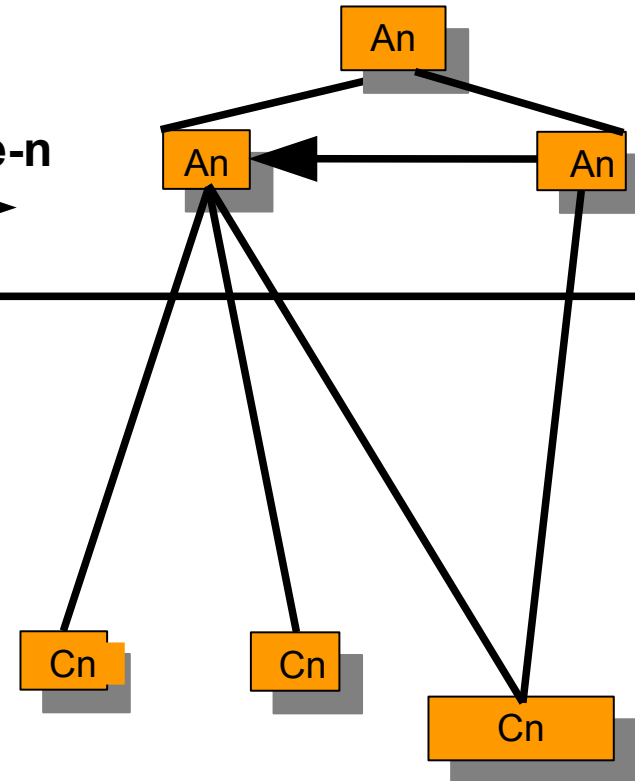
**Stage-0**

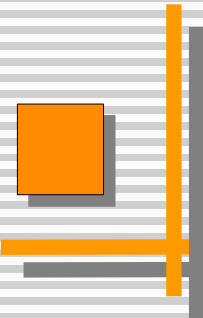


**Stage-n**



**Generated Stage-An architecture in composition language An**  
**Component language Cn**





# Staged Architectures in Software Engineering



# Build Management is Staged Composition

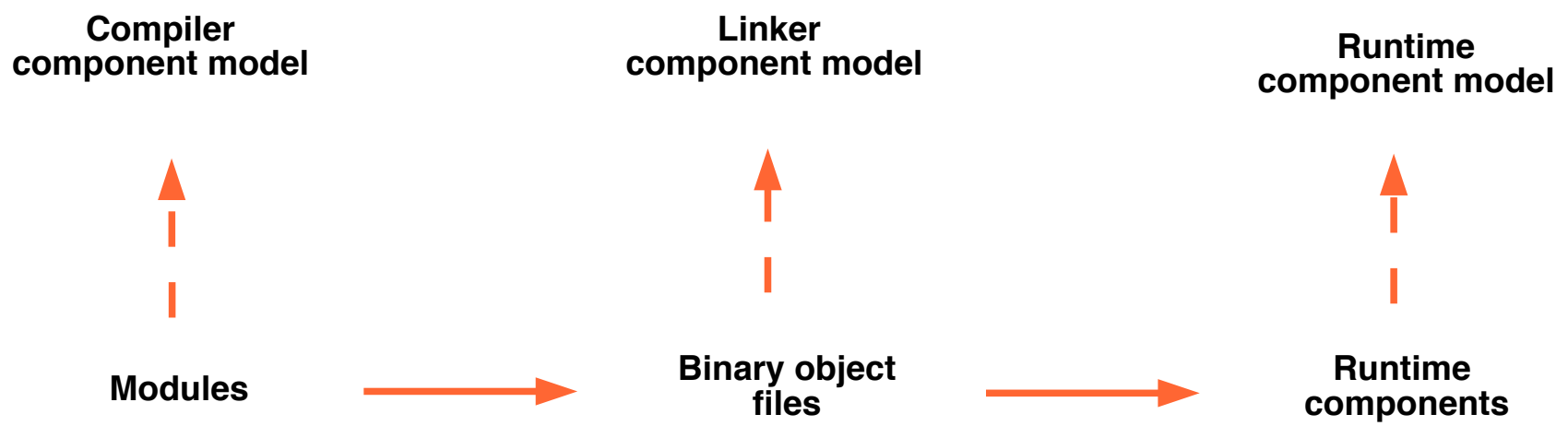
Software build management is code composition in several stages

Make is a composition tool with a lazy rule-based language

Composition language is

Expressions are applications of UNIX tools (compiler, linker, generator, preprocessor)

Different component models on all stages



# Invasive Software Composition

Produces code from typed templates by parameterization and expansion

**Stage-0**  
**Composition level**  
**language: Java**

**Stage-1**  
**language: Java**

**Fragment  
component model**

**Runtime  
component model  
(objects)**



**Stage-0**

**Code Fragment  
Components**

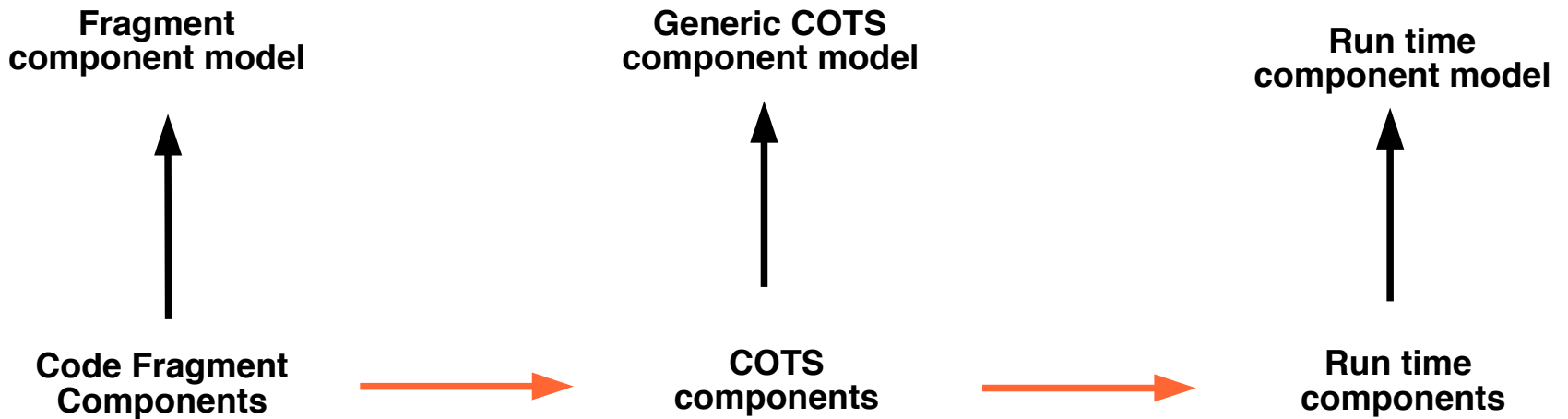


**Runtime  
components**



# Component Models on Different Levels in the Software Process

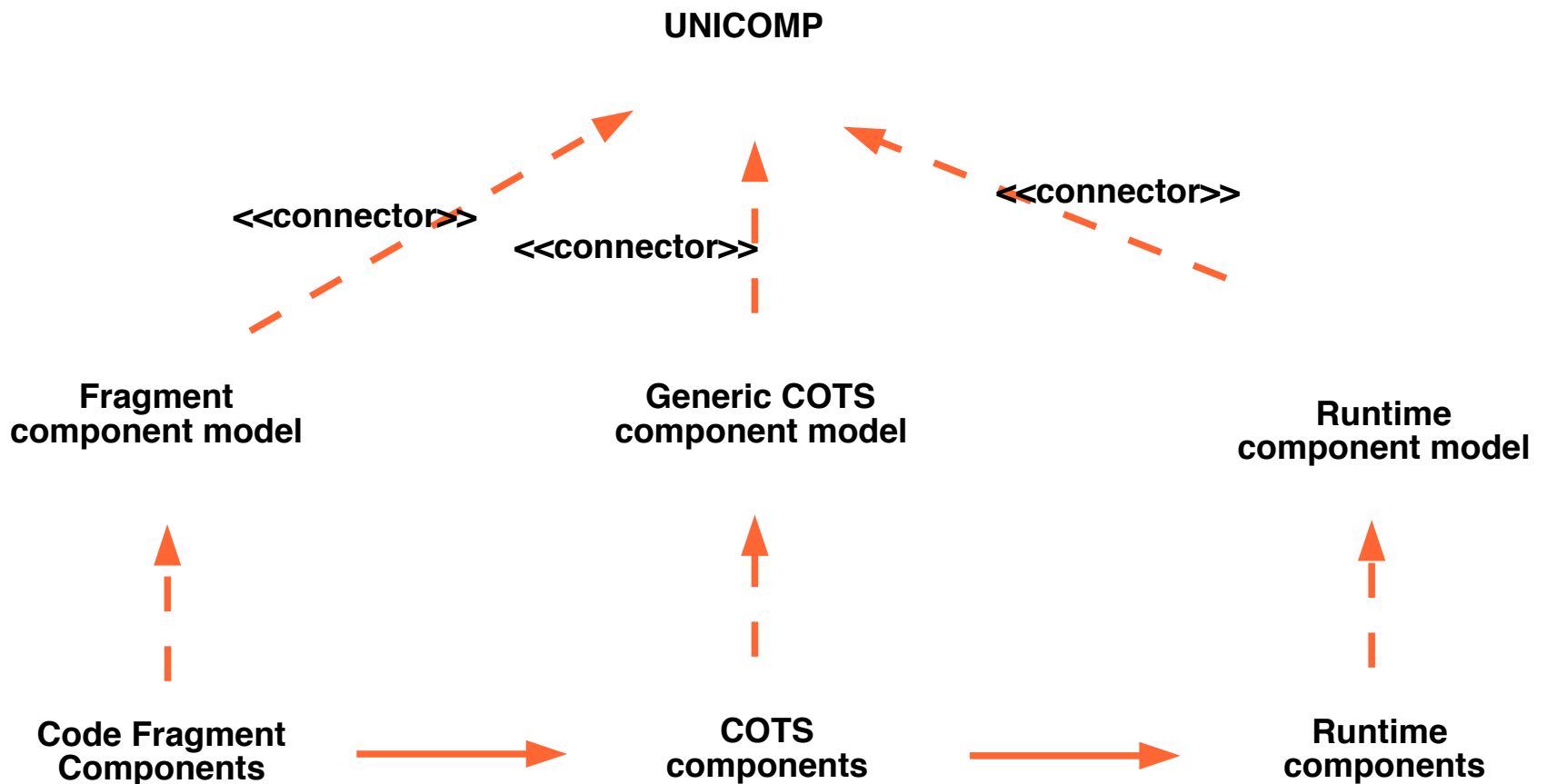
Standard COTS models are just models for binary code



# Uniform Component Models

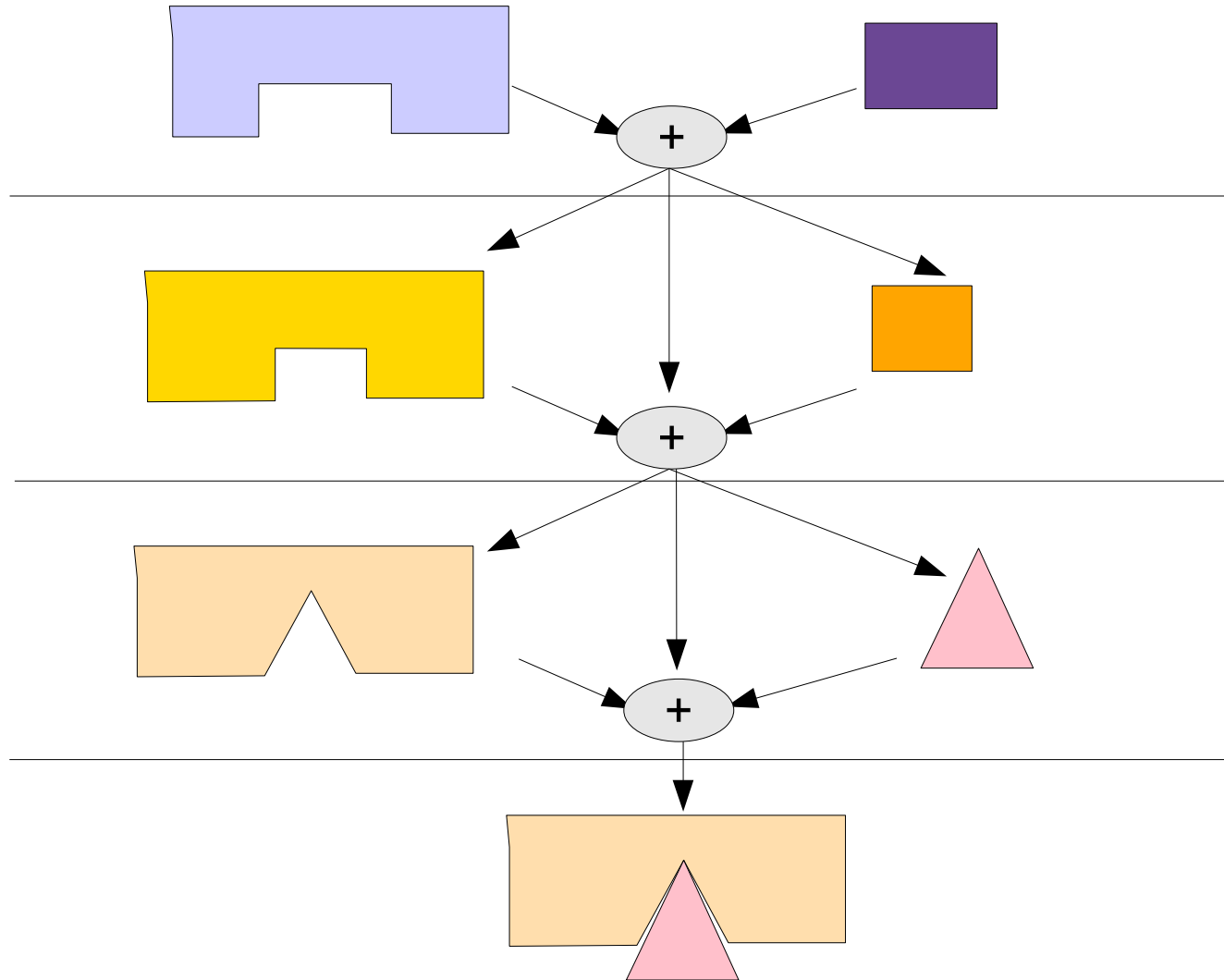
It is possible to build generic component models

The Uniform Composition Model of the EASYCOMP Project





# Staged Architectures Written as Layers

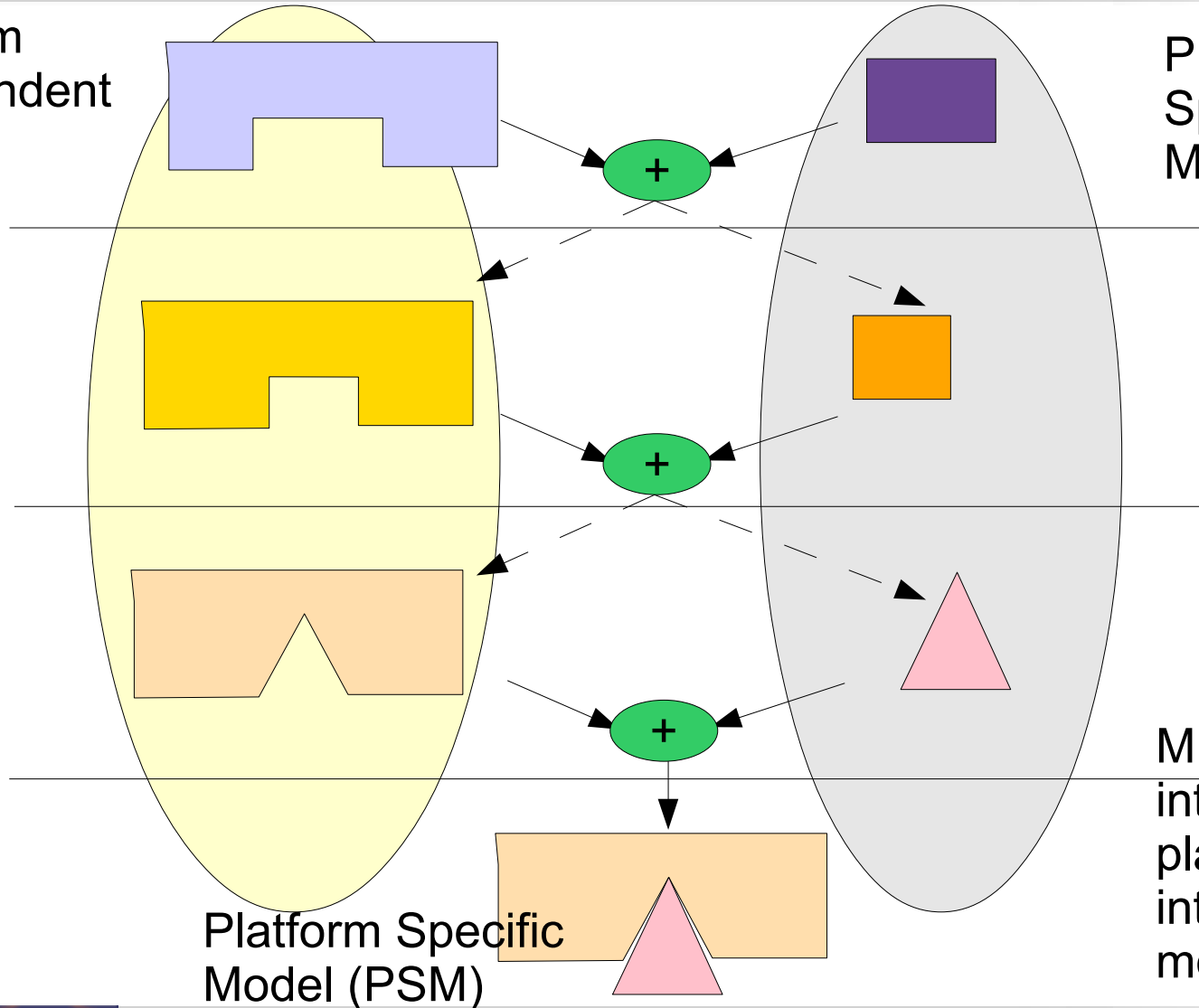




# A Staged Processing Architecture: MDA

Platform Independent Models (PIM)

Platform Specific Models



Platform Specific Model (PSM)

MDA weavers integrate platform variants into staged models

# ■ Staged Architectures vs MDA


MDA is a product line approach (hence the stages)

MDA has *not* a staged architecture, since the weavers are *fixed*

MDA has no architecture, no component models

... but a staged processing technology for variant selection

... but we can build more powerful forms of MDA by taking in the ideas of staged programming and staged architectures



# The Dresden Staged Architecture Development Process

Fix the stages

Decide on a staged processing or programming architecture

Fix the component models for every stage

Interface concepts, composition operations, composition language

Fix the architectures

Fix the variant management

Fix the components

And you'll have a pretty comprehensible product line!

# The Vision

The staged programming principle is powerful, so future systems will employ it

We need tools to support staged architectures

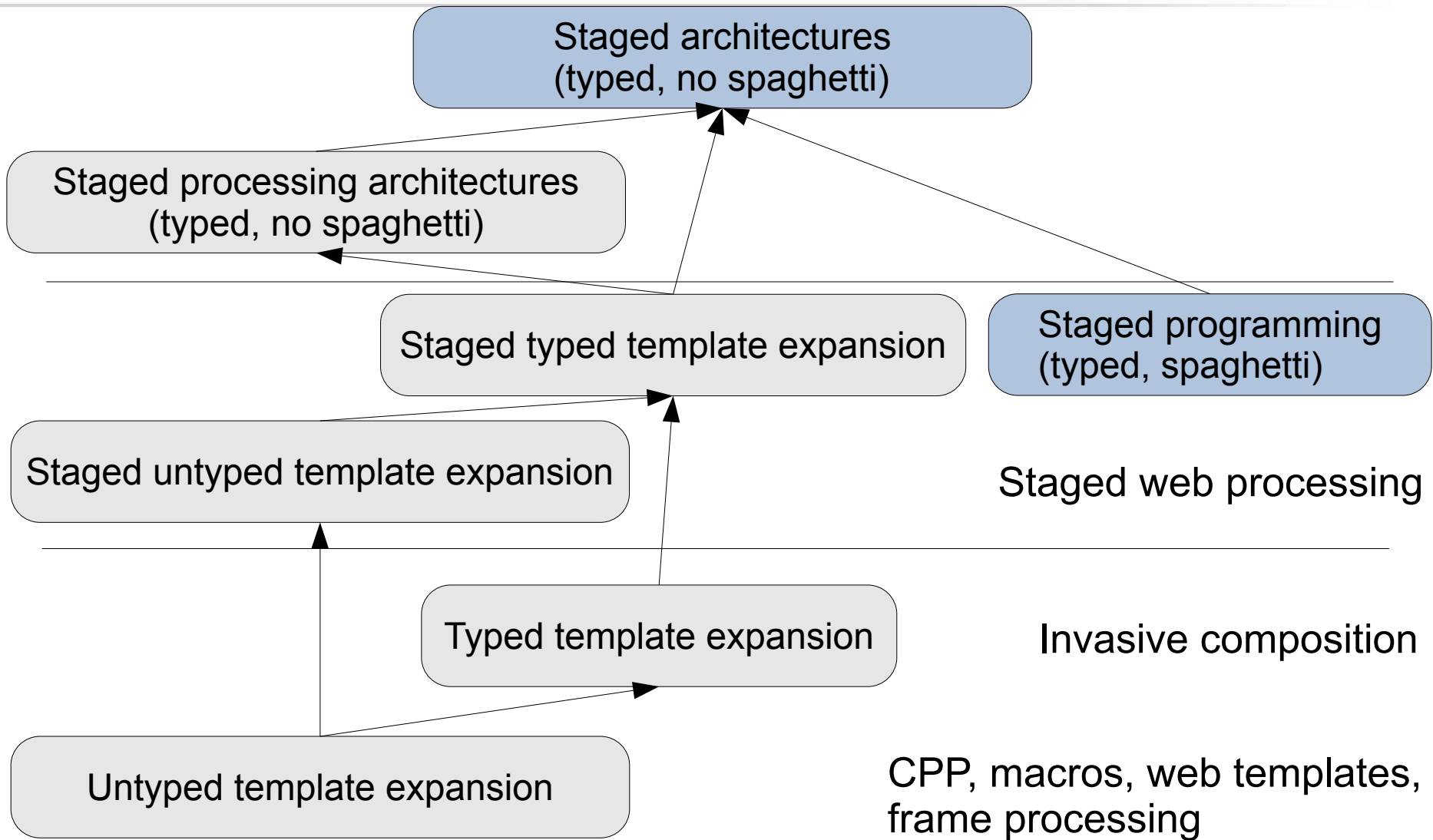
- Visualize them

- Debug them

- Support the component models

.... that's a lot of work...

# The Hierarchy of Staged Architectures



# ■ The Beauty of a Staged Architecture



# The End

[www.easycomp.org](http://www.easycomp.org)

<http://www.the-compost-system.org>

U. Aßmann. Invasive Software Composition, 2003, Springer.

U. Aßmann. Architectural Styles for Active Documents. Special Issue “Software Composition” Science of Computer Programming, Elsevier, 2005.

Walid Taha. A Gentle Introduction to Multi-Stage Programming.





# Invasive Composition Produces Functional from Composition Interfaces

Two different component models

