Component-based Software

Introduction and overview

Slides by courtesy of Uwe Aßmann, IDA / TU Dresden

Revised 2005, 2006, 2007 by Christoph Kessler, IDA

Recommended Reading

- Szyperski: *Component Software – Beyond Object-Oriented Programming, 2nd edition*. Addison-Wesley, 2002.
- Douglas McIlroy. *Mass-produced software components.*
  http://cm.bell-labs.com/cm/cs/who/doug/components.txt
  in:
  P. Naur and B. Randell, "*Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*", Scientific Affairs Division, NATO, Brussels, 1969, 138-155.

Motivation for Component Based Development

- Managing system complexity:
  Divide-and-conquer (Alexander the Great)
- Well known in other disciplines
  - Mechanical engineering (e.g., German DIN 2221); IEEE standards)
  - Electrical engineering
  - Architecture
  - Computer architecture
- Outsourcing to component producers
- Goal: Reuse of partial solutions
- Easy configurability of the systems
  - Variants, versions, product families

Mass-produced Software Components

- Garmisch 1968, NATO conference on software engineering
- McIlroy:
  - Every ripe industry is based on components, since these allow to manage large systems
  - Components should be produced in masses and composed to systems afterwards

Mass-produced Software Components

In the phrase `*mass production techniques,*' my emphasis is on `*techniques*' and not on mass production plain.
Of course, mass production, in the sense of limitless replication of a prototype, is trivial for software.

But certain ideas from industrial technique I claim are relevant.
- The idea of subassemblies carries over directly and is well exploited.
- The idea of interchangeable parts corresponds roughly to our term `modularity,' and is fitfully respected.
- The idea of machine tools has an analogue in assembly programs and compilers.

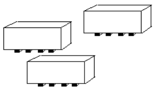
Yet this fragile analogy is belied when we seek for analogues of other tangible symbols of mass production.
- There do not exist manufacturers of standard parts, much less catalogues of standard parts.
- One may not order parts to individual specifications of size, ruggedness, speed, capacity, precision or character set.

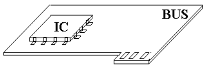
Mass-produced Software Components

- Later McIlroy was with Bell Labs ...
  - ... and invented pipes, diff, join, echo (UNIX).
  - Pipes are still today the most employed component system!
- Where are we today?

Real Component Systems

- Lego
- Square stones
- Building plans
- IC's
- Hardware bus
- How do they differ from software?





Definitions of “Component”

“A software component is a unit of composition
with contractually specified interfaces and
explicit context dependencies only.
A software component
can be deployed independently and
is subject to composition by third parties.”
- C. Szyperski, ECOOP Workshop WCOP 1997.

“A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction”
- Grady Booch

“A software component is a static abstraction with plugs.”
- Nierstrasz/Dami

Definitions of “Component” (cont.)

MetaGroup (OpenDoc):
“Software components are defined as prefabricated, pretested, self-contained, reusable software modules bundles of data and procedures - that perform specific functions.”

Sametinger:
“Reusable software components are self-contained, clearly identifyable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation, and a defined reuse status.”

Definitions of “Component” (cont.)

- Heineman / Councill [Ch.1]:

  “A *software component* is a software element
  that conforms to a component model
  and can be independently deployed and composed
  without modification according to a composition standard.

  A *component model* defines specific interaction and composition standards.

  Composition is the combination of two or more software components yielding a new component behavior at a different level of abstraction ... [which is] determined by the components being combined and the way how they are combined.”

Component as unit of composition

U. Assmann (2003):

- A *component* is a *container with*
  - *variation points*
  - *extension points*
  - that are adapted during composition
- A component is a reusable *unit for composition*
- A component underlies a *component model*
  - abstraction level
  - composition time (static or runtime?)

Are Objects Components??

Szyperski [CS 4.1]: **No!**

- An **object** is a unit of instantiation.
- It has a unique identity.
- It may have state, and this can be (externally) observed
- It encapsulates its state and behavior.

Components are rather prototypes / blueprints / plans from which (stateful) objects can be instantiated
  - e.g., a function definition, type definition, class or set of classes
- No (externally observable) state
  - Only one copy required per context (e.g., process)
- Unit of independent deployment
- Unit of third-party composition

What Is A Component-Based System?

A *component-based system*
has the following divide-and-conquer feature:

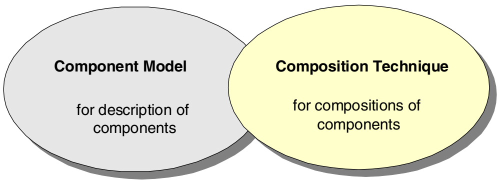
- A component-based system is a system in which a major relationship between the components is
  - tree-shaped
  - or reducible.
- Consequence:
  the entire system can be reduced to one abstract node
  - at least along the structuring relationship
- Systems with layered relations (dag-like relations) are not necessarily component-based.
  - Because they cannot be reduced

What Is A Component-Based System?

- Because it is divide-and-conquer,
  component-based development is attractive.
- However, we have to choose the structuring relation
- And, we have to choose the composition model
- Mainly, two sorts are known:
  - Modular decomposition (blackbox)
  - Separations of concerns (graybox)

Component Systems
(Component Platforms)

- We call a technology in which component-based systems can be produced a *component system* or *component platform*.
- A component system has



Component Model

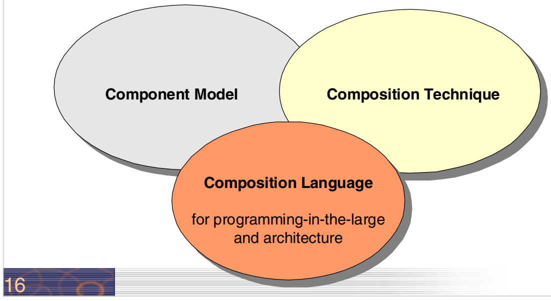
for description of components

Composition Technique

for compositions of components

Software Composition Systems

- A *composition system* has



Component Model

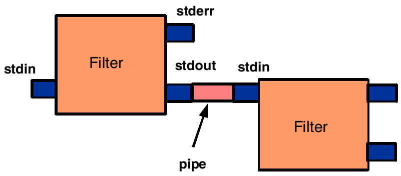
Composition Technique

Composition Language

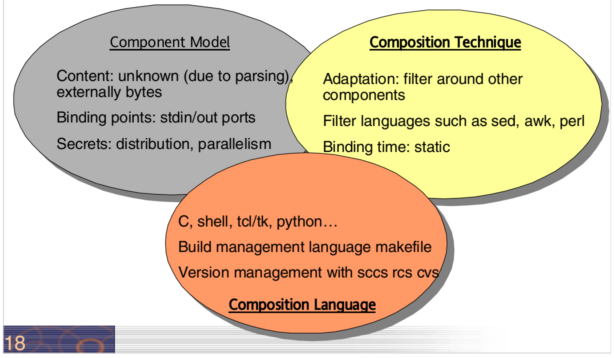
for programming-in-the-large and architecture

Example: UNIX Filters and Pipes [McIlroy]

- UNIX shells style still offers the most used component paradigm:
  - Communication with byte streams via standard I/O ports
  - Parsing and linearizing the objects
  - Extremely flexible, simple



Unix Filters and Pipes
as Composition System



Component Model

Content: unknown (due to parsing), externally bytes

Binding points: stdin/out ports

Secrets: distribution, parallelism

Composition Technique

Adaptation: filter around other components

Filter languages such as sed, awk, perl

Binding time: static

C, shell, tcl/tk, python…

Build management language makefile

Version management with sccs rcs cvs

Composition Language

Desiderata for Flexible Software Composition

- Component Model
  - How do components look like?
  - Binding points, binding time?
  - Secrets, interfaces, substitutability
- Composition Technique
  - How are components plugged together, composed, merged, applied?
  - Composition time (Deployment, Connection, ...)
- Composition Language
  - How are compositions of large systems described?
  - How are system builds managed?
- Be aware: This list is NOT complete!

Desiderata Component Model

- **Modularity**
  - M1 Component secrets (information hiding)
    - Location, lifetime, language
    - Explicit specification of interfaces (contact points, exchange points, binding points)
    - Provided and required interfaces
  - M2 Semantic substitutability (conformance, contracts)
    - Syntactic substitutability (typing)
  - M3 Content
    - Component language metamodel

Desiderata Component Model (cont.)

- **Parameterization** of components to their reuse context
  - P1 Generic type parameters
  - P2 Generic program elements
  - P3 Property parameterization
- **Standardization**
  - S1 Open standards – or proprietary ones
  - S2 Standard components
  - S3 Standard services

Desiderata Composition Technique

- **Connection and Adaptation**
  - C1: Automatic Component Adaptation:
    adapt the component interface to another interface
  - C2: Automatic Glueing: Generation of glue code for communication, synchronization, distribution. Consists of a sequence of adaptations
- **Extension**
  - E1: Base Class Extension: can base classes be extended?
    - E1.1 Generated factories: can factories be generated
    - E1.2 Generated access layers
  - E2: General Views. Use-based extensions: Can a use of a component extend the component?
  - E3: Integrated Extensions. Can an extension be integrated into a component?

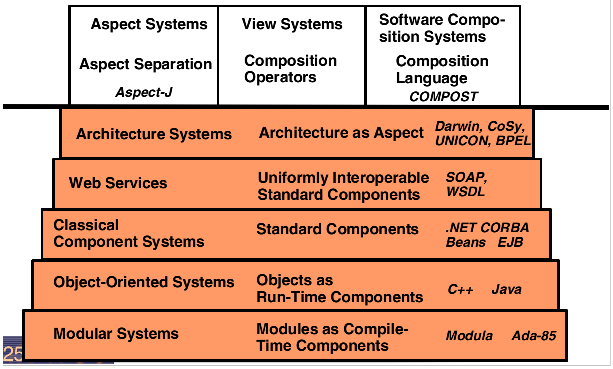
Desiderata Composition Technique

- **Aspect separation (aspect composition)**
  - AS1: Aspect weaving: Extension by crosscutting aspects
  - AS2: Multiple interfaces: Can a component have multiple interfaces?
- **Scalability (Composition time)**
  - SC1: Binding time hiding
  - SC2: Binding technique hiding
- **Metamodelling**
  - MM1: Introspection and reflection (metamodel).
    Can other components be introspected? The component itself?
  - MM2: Metaobject protocol:
    Is the semantics of the component specified reflectively?

Desiderata Composition Language

- CL1: Product Consistency
  - Variant cleanness: consistent configurations
  - Robustness: freedom of run-time exceptions
- CL2: Software Process Support
  - Build management automation
- CL3: Meta-composition
  - Is the composition language component-based,
    i.e., can it be composed itself?
  - Reuse of architectures
- CL4: Architectural styles (composition styles)
  - Constraints for the composition

The Ladder of Component and Composition Systems



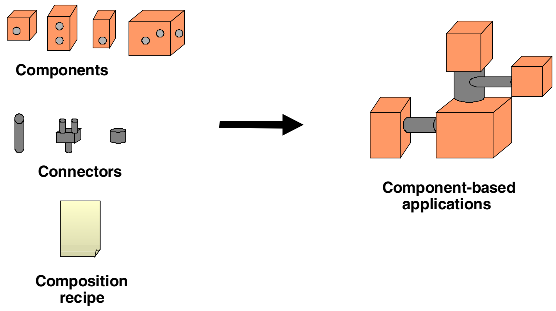
Aspect Systems Aspect Separation *Aspect-J*	View Systems Composition Operators	Software Composition Systems Composition Language *COMPOST*
Architecture Systems	Architecture as Aspect	*Darwin, CoSy, UNICON, BPEL*
Web Services	Uniformly Interoperable Standard Components	*SOAP, WSDL*
Classical Component Systems	Standard Components	*.NET CORBA Beans EJB*
Object-Oriented Systems	Objects as Run-Time Components	*C++ Java*
Modular Systems	Modules as Compile-Time Components	*Modula Ada-85*

The Essence of the 60s-90s: LEGO Software

- Procedural systems
- Modular systems
- Object-oriented technology
- Component-based programming
  - CORBA, EJB, DCOM, COM+, .NET
- Architecture languages

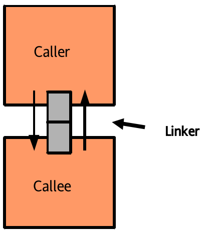
Blackbox composition

Blackbox Composition

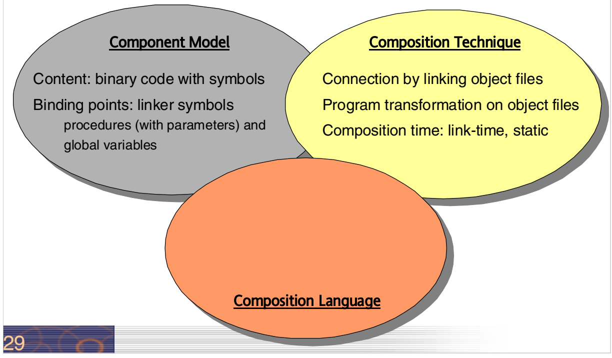


Procedure Systems

- Fortran, Algol, Pascal, C, ...
- The *procedure* is the component
- The *activation record* the instantiation
- Component model is supported by almost all processors directly
  - JumpSubroutine instruction
  - Return instruction



Procedures as Composition System



Content: binary code with symbols

Binding points: linker symbols procedures (with parameters) and global variables

Connection by linking object files

Program transformation on object files

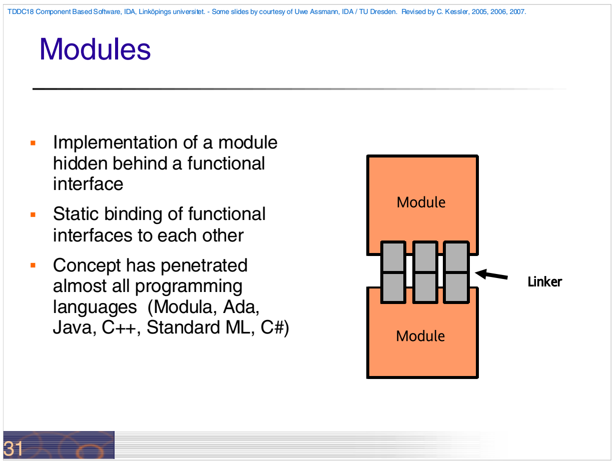
Composition time: link-time, static

Modules (a la Parnas)

We can attempt to define our modules “around” assumptions which are likely to change. One then designs a module which “hides” or contains each one.

Such modules have rather abstract interfaces, which are relatively unlikely to change.

- Every module hides an important design decision behind a well-defined interface which does not change when the decision changes.



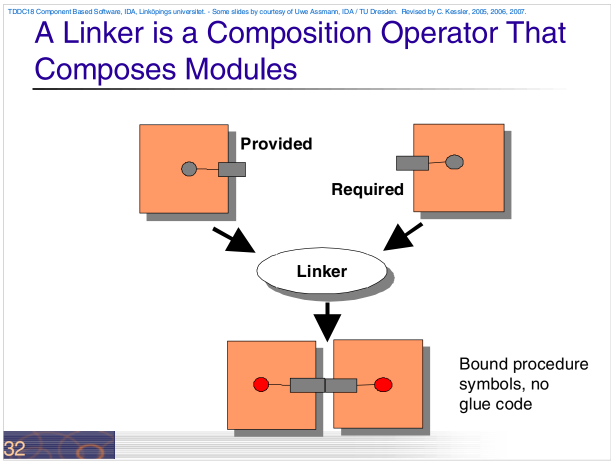
Modules

- Implementation of a module hidden behind a functional interface
- Static binding of functional interfaces to each other
- Concept has penetrated almost all programming languages (Modula, Ada, Java, C++, Standard ML, C#)

Module

Linker

Module



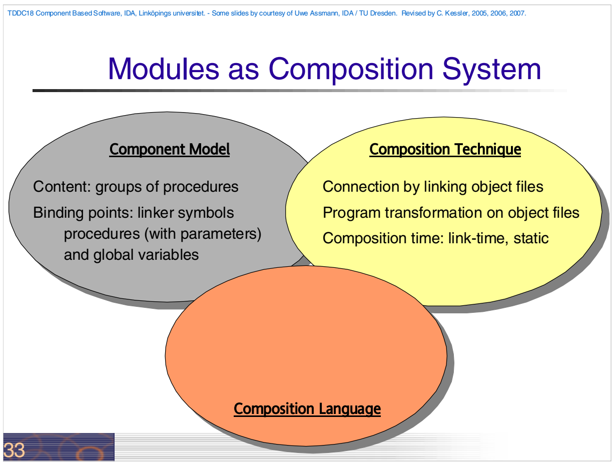
A Linker is a Composition Operator That Composes Modules

Provided

Required

Linker

Bound procedure symbols, no glue code



Modules as Composition System

Component Model

Content: groups of procedures

Binding points: linker symbols procedures (with parameters) and global variables

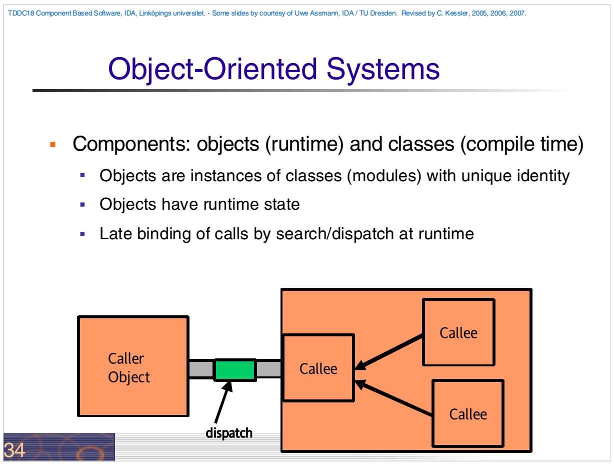
Composition Technique

Connection by linking object files

Program transformation on object files

Composition time: link-time, static

Composition Language



Object-Oriented Systems

- Components: objects (runtime) and classes (compile time)
  - Objects are instances of classes (modules) with unique identity
  - Objects have runtime state
  - Late binding of calls by search/dispatch at runtime

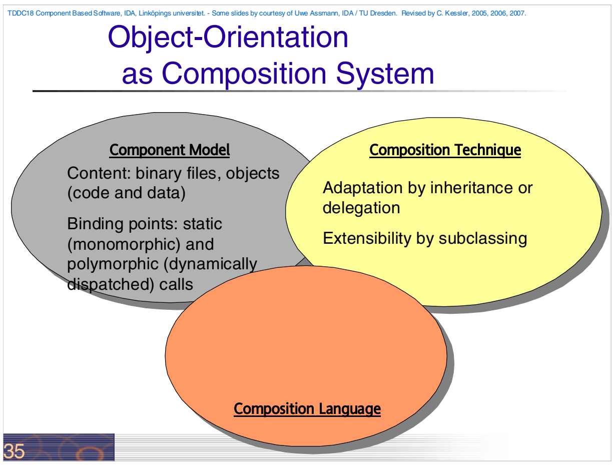
Caller Object

Callee

Callee

Callee

dispatch



Object-Orientation as Composition System

Component Model

Content: binary files, objects (code and data)

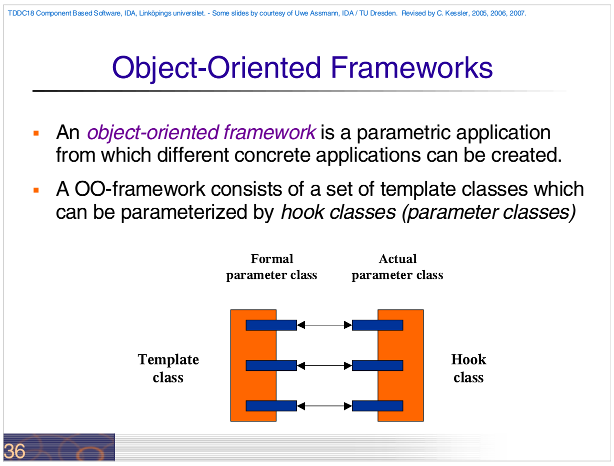
Binding points: static (monomorphic) and polymorphic (dynamically dispatched) calls

Composition Technique

Adaptation by inheritance or delegation

Extensibility by subclassing

Composition Language



Object-Oriented Frameworks

- An *object-oriented framework* is a parametric application from which different concrete applications can be created.
- A OO-framework consists of a set of template classes which can be parameterized by *hook classes (parameter classes)*

Formal parameter class

Actual parameter class

Template class

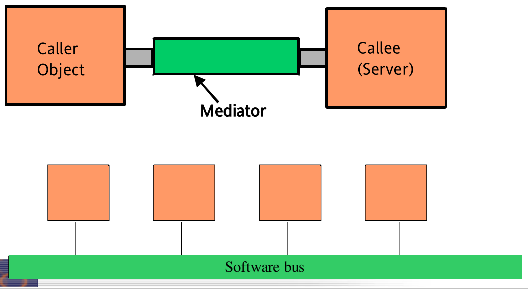
Hook class

Object-Oriented Frameworks

- Component Model
  - Binding points: Hot spots to exchange the parameter classes (sets of polymorphic methods)
- Composition Technique
  - Same as OO
- Compostion language
  - Same as OO

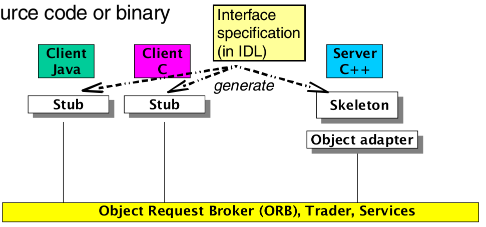
Commercial Component Systems

- CORBA / DCOM / .NET / JavaBeans / EJB
- Although different on the first sight, turn out to be rather similar



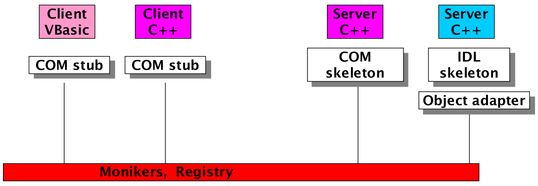
CORBA

- Language independent, distribution transparent
- interface definition language IDL
- source code or binary


Interface specification (in IDL)
Client Java
Client C
Server C++
generate
Stub
Stub
Skeleton
Object adapter
Object Request Broker (ORB), Trader, Services

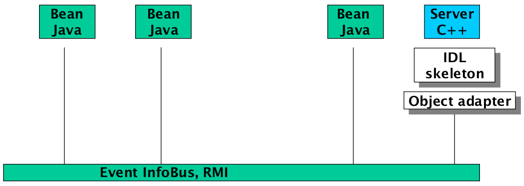
(D)COM, ActiveX

- Microsoft's model is similar to CORBA. Proprietary
- (D)COM is a binary standard



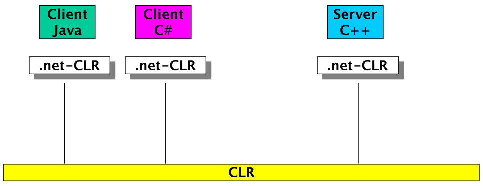
Java Beans

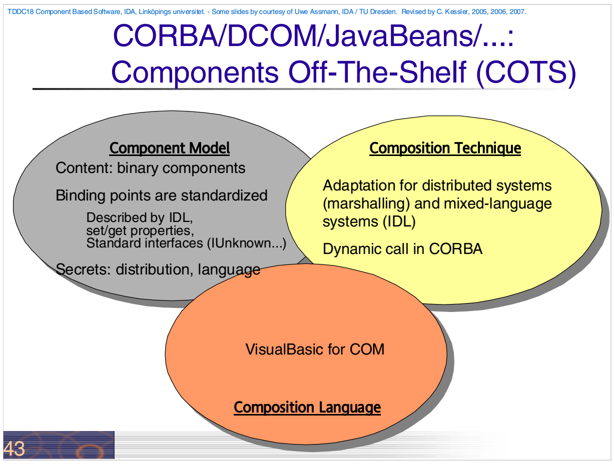
- Java only: source code / bytecode-based
- Event-based, transparent distribution by remote method invocation (RMI – includes Java Object Serialization)



DOT-NET

- Language independent, distribution transparent
- NO interface definition language IDL (at least for C#)
- source code or bytecode MSIL
- Common Language Runtime CLR





CORBA/DCOM/JavaBeans/...: Components Off-The-Shelf (COTS)

Component Model

Content: binary components

Binding points are standardized

Described by IDL,
set/get properties,
Standard interfaces (IUnknown...)

Secrets: distribution, language

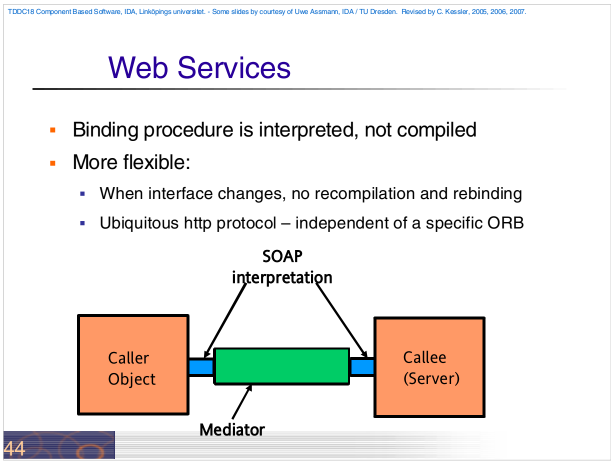
Composition Technique

Adaptation for distributed systems (marshalling) and mixed-language systems (IDL)

Dynamic call in CORBA

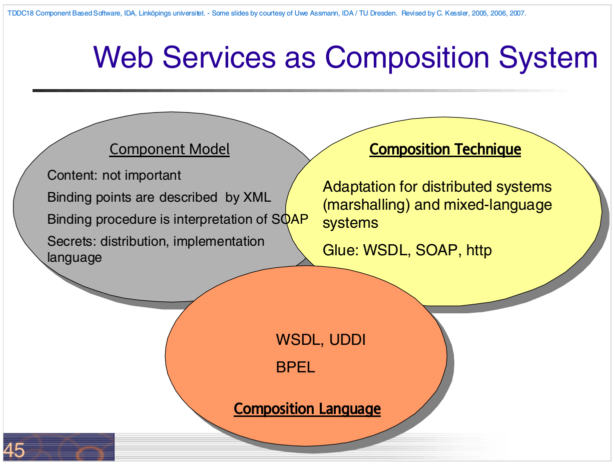
VisualBasic for COM

Composition Language



Web Services

- Binding procedure is interpreted, not compiled
- More flexible:
  - When interface changes, no recompilation and rebinding
  - Ubiquitous http protocol – independent of a specific ORB



Web Services as Composition System

Component Model

Content: not important

Binding points are described by XML

Binding procedure is interpretation of SOAP

Secrets: distribution, implementation language

Composition Technique

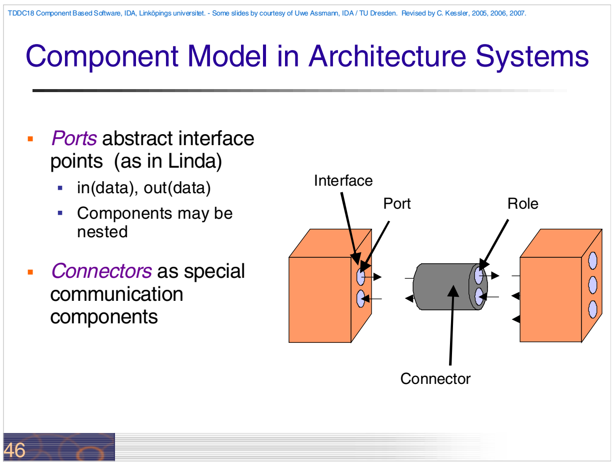
Adaptation for distributed systems (marshalling) and mixed-language systems

Glue: WSDL, SOAP, http

WSDL, UDDI

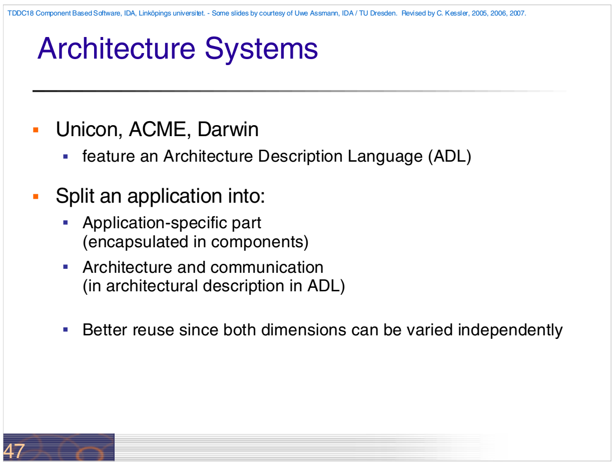
BPEL

Composition Language



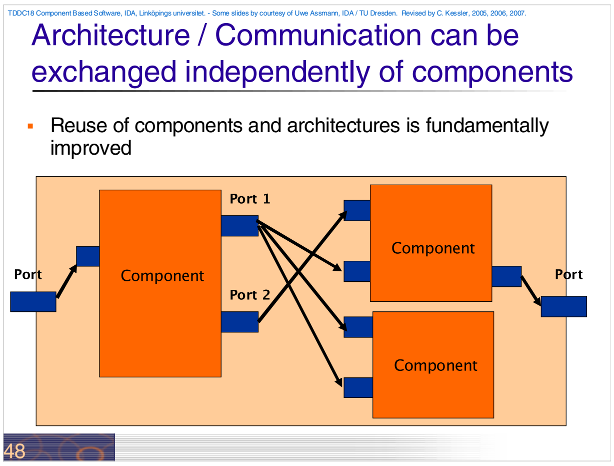
Component Model in Architecture Systems

- *Ports* abstract interface points (as in Linda)
  - in(data), out(data)
  - Components may be nested
- *Connectors* as special communication components



Architecture Systems

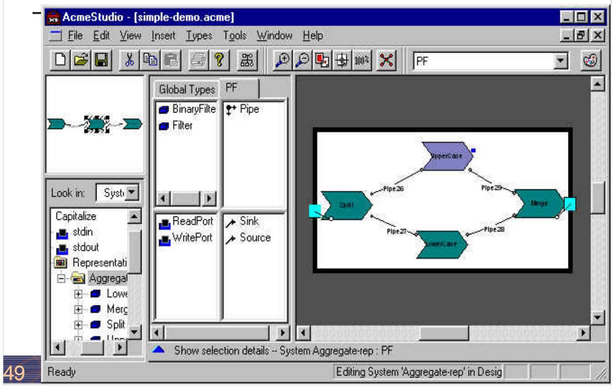
- Unicon, ACME, Darwin
  - feature an Architecture Description Language (ADL)
- Split an application into:
  - Application-specific part (encapsulated in components)
  - Architecture and communication (in architectural description in ADL)
  - Better reuse since both dimensions can be varied independently



Architecture / Communication can be exchanged independently of components

- Reuse of components and architectures is fundamentally improved

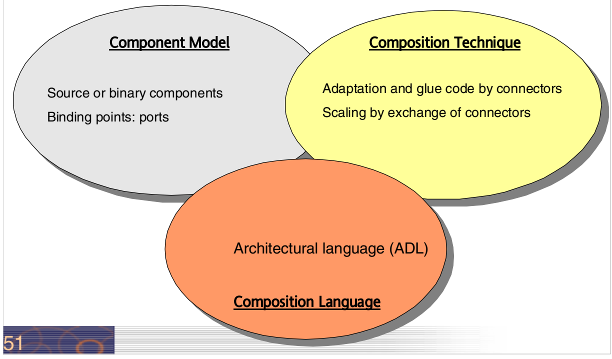
ACME Studio



The Composition Language: ADL

- Architectural description language, ADL
  - ADL-compiler
  - XML-Readers/Writers for ADL.
    XADL is a new standard exchange language for ADL based on XML
- Graphic editing of systems
- Checking, analysing, simulating systems
  - Dummy tests
  - Deadlock checkers
  - Liveness checking

Architecture Systems as Composition Systems



Component Model

Source or binary components

Binding points: ports

Composition Technique

Adaptation and glue code by connectors

Scaling by exchange of connectors

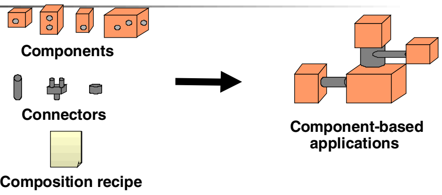
Architectural language (ADL)

Composition Language

What the Composition Language Offers for the Software Process

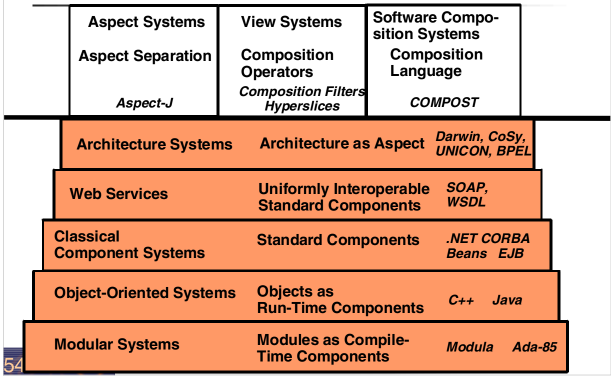
- Communication
  - Client can understand the architecture graphics well
  - Architecture styles classify the nature of a system in simple terms (similar to design patterns)
- Design support
  - Refinement of architectures (stepwise design, design to several levels)
  - Visual and textual views to the software resp. the design
- Validation: Tools for consistency of architectures
  - Are all ports bound? Do all protocols fit?
  - Does the architecture corresponds to a certain style ? Or to a model architecture?
  - Parallelism features, such as deadlocks, fairness, liveness
  - Dead parts of the systems
- Implementation: Generation of large parts of the implementation (the communications- and architecture parts)

The Essence of Blackbox Composition



- 3 Problems in system construction
  - Variability
  - Extensibility
  - Adaptation
- Blackbox composition supports variability and adaptation
  - not extensibility

The Ladder of Component and Composition Systems

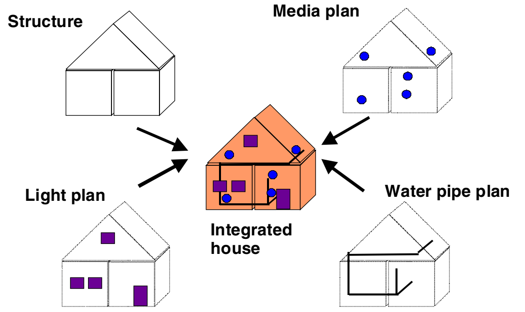


Aspect Systems	View Systems	Software Composition Systems
Aspect Separation	Composition Operators	Composition Language
Aspect-J	*Composition Filters Hyperslices*	*COMPOST*
Architecture Systems	Architecture as Aspect	*Darwin, CoSy, UNICON, BPEL*
Web Services	Uniformly Interoperable Standard Components	*SOAP, WSDL*
Classical Component Systems	Standard Components	*.NET CORBA Beans EJB*
Object-Oriented Systems	Objects as Run-Time Components	*C++ Java*
Modular Systems	Modules as Compile-Time Components	*Modula Ada-85*

Graybox Component Models

Component integration
- Aspect oriented programming
- View-based composition

Aspects in Architecture



Structure

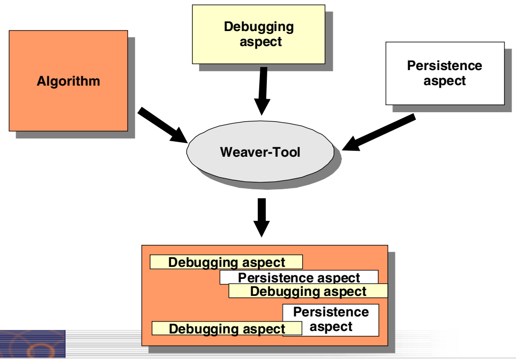
Media plan

Light plan

Integrated house

Water pipe plan

Aspects in Software



Algorithm

Debugging aspect

Persistence aspect

Weaver-Tool

Debugging aspect

Persistence aspect

Debugging aspect

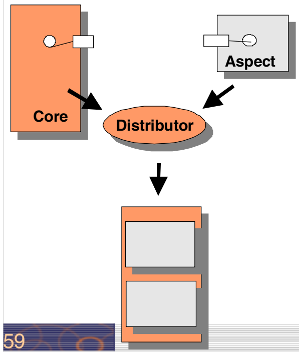
Persistence aspect

Debugging aspect

Aspect Systems

- Aspect languages
  - Every aspect in a separate language
  - Domain specific
  - Weaver must be built (is a compiler, much effort)
- Script-based Weavers
  - The weaver interprets a specific script or aspect program
  - This introduces the aspect into the core

Aspect Weavers Distribute Advice Components over Core Components



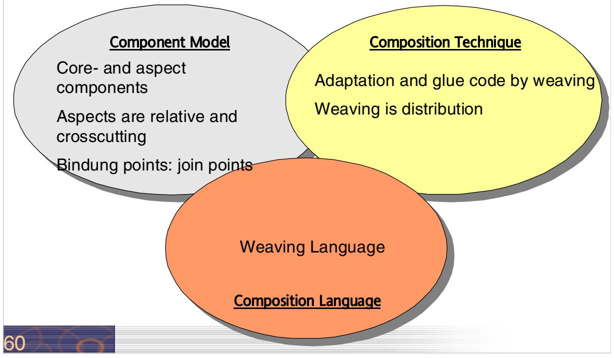
Core

Aspect

Distributor

- Aspects are *crosscutting*
- Hence, aspect functionality must be *distributed* over the core

Aspect Systems As Composition Systems



Component Model

Core- and aspect components

Aspects are relative and crosscutting

Bindung points: join points

Composition Technique

Adaptation and glue code by weaving

Weaving is distribution

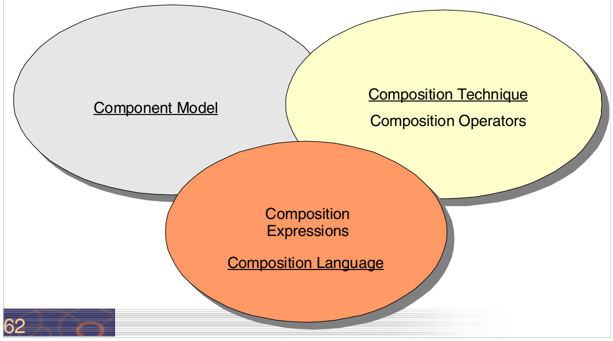
Weaving Language

Composition Language

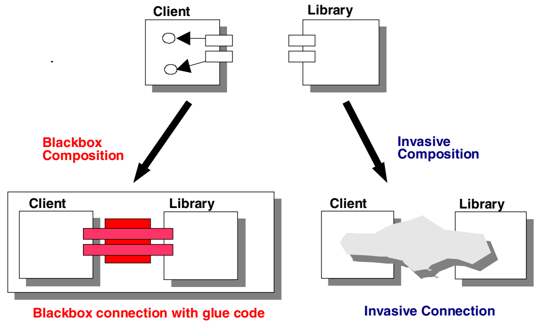
Composition Systems
with composition operators and expressions

- Hyperspace Programming [Ossher *et al.*, IBM]
- Piccola [Nierstrasz, et.al., Berne]
- Metaclass composition [Forman/Danforth, Cointe]
- Invasive software composition [Aßmann 2003]
- Formal calculi
  - Lambda-N calculus [Dami]
  - Pi-L calculus [Lumpe]

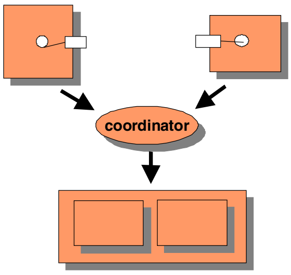
Composition Systems
with composition operators and expressions



Connectors are Composition Operators

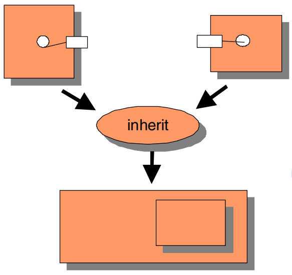


Composers can be used for Skeletons (Coordination functions)



- Instead of functions or modules, skeletons can be defined over fragment components
- CoSy coordination schemes (ACE compiler component framework www.ace.nl)
  - Compose basic components with coordinating operators

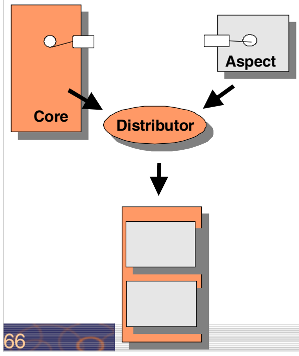
Composers can be used for inheritance



- Extension can be used for inheritance (mixins)

  A *mixin* is a class (i.e., a set of features) by which a superclass can be extended to derive a subclass. The mixin class itself is final, i.e., cannot be subclassed.
- Mixin-based inheritance**:**
  - copy first superclass
  - extend with fragments of second superclass (mixin)

Composers Generalize Aspect Weavers in AOP



- Complex composers distribute aspect fragments over core fragments
- *Distributors* extend the core
- Distributors are more complex operators, defined from basic ones

Composition Languages

- **Composition languages** describe the structure of the systen in-the-large ("programming in the large")
- **Composition programs** combine the basic composition operations of the composition language
- Composition languages can look quite different
  - Standard languages, such as Java
  - Makefiles
- Enables us to describe large systems

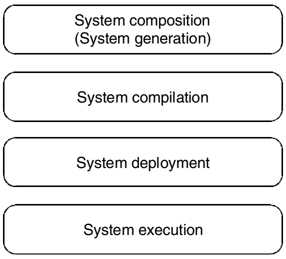
Composition program size	1
System size	10

Conclusions for Composition Systems

- Components have a *composition interface*
  - Composition interface is different from functional interface
  - The composition is running usually *before* the execution of the system
  - From the composition interface, the functional interface is derived
- System composition becomes a new step in system build

Steps in System Construction

- We need component models and composition systems for all levels of system construction



System composition (System generation)

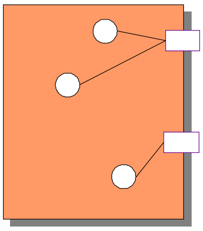
System compilation

System deployment

System execution

The Component Model of Invasive Composition

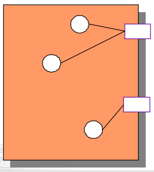
- **The component is a *fragment container (fragment box)***
  - a set of fragments/tag elements
- **Uniform representation of**
  - a fragment
    - a class, a package, a method
  - a set of fragments
    - an aspect
    - a meta description
    - a composition program



Fragment Box Components Have Hooks

Hooks are variation points of a component: fragments or positions, which are subject to change

- **Software variation points, hooks**
  - Method entries/exits
  - Generic parameters

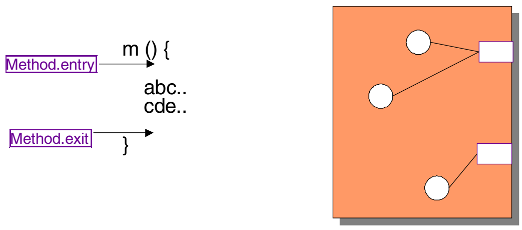


Invasive Composition

Invasive composition adapts and extends components at hooks by transformation

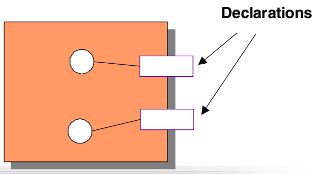
Implicit Hooks In Software

- Given by the programming language
- Example: Method entry/exit



Declared Hooks

Declared Hooks are declared by the component writer as code parameters

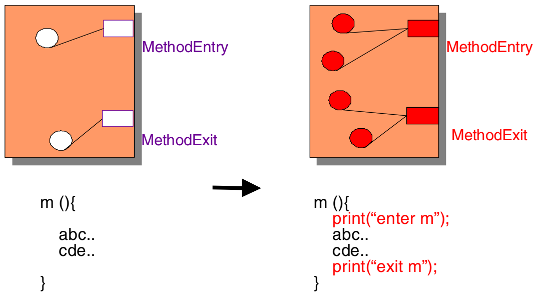


The Composition Technique of Invasive Composition

Invasive Composition
adapts and extends
components
at hooks
by transformation

A composer transforms unbound to bound hooks

***composer*: fragment box with hooks --> fragment box with bound hooks**

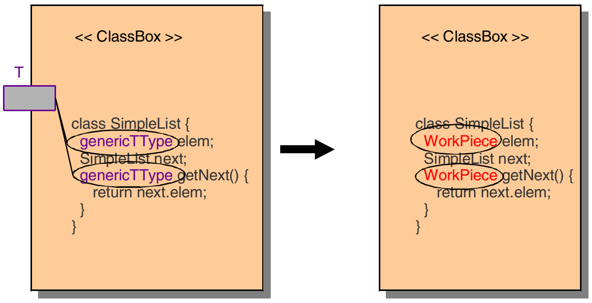


```
m (){

    abc..
    cde..

}
```

```
m (){
    print(“enter m”);
    abc..
    cde..
    print(“exit m”);
}
```

```
component.findHook(“MethodEntry”).extend(“print(\”enter m\”);”);
component.findHook(“MethodExit”).extend(“print(\”exit m\”);”);
```

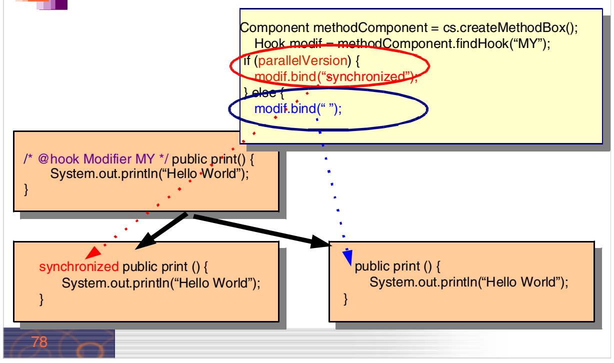
Generic Types



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

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

Generic Modifiers

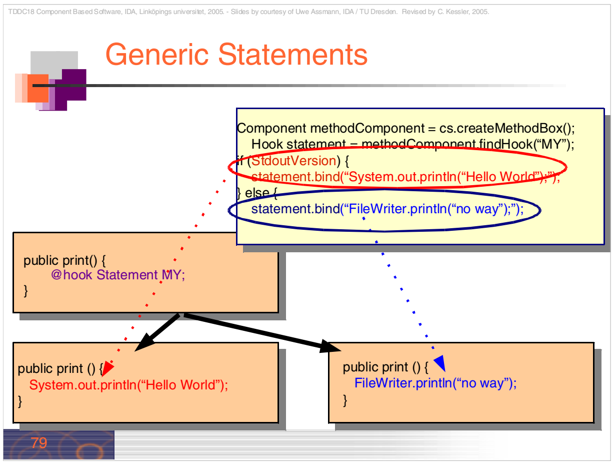


```
Component methodComponent = cs.createMethodBox();
   Hook modif = methodComponent.findHook(“MY”);
if (parallelVersion) {
   modif.bind(“synchronized”);
} else {
   modif.bind(“ ”);
```

```
/* @hook Modifier MY */ public print() {
     System.out.println(“Hello World”);
}
```

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

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



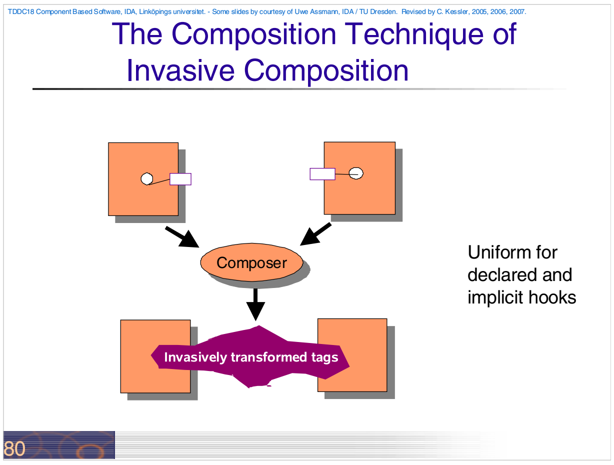
Generic Statements

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

```
public print() {
      @hook Statement MY;
}
```

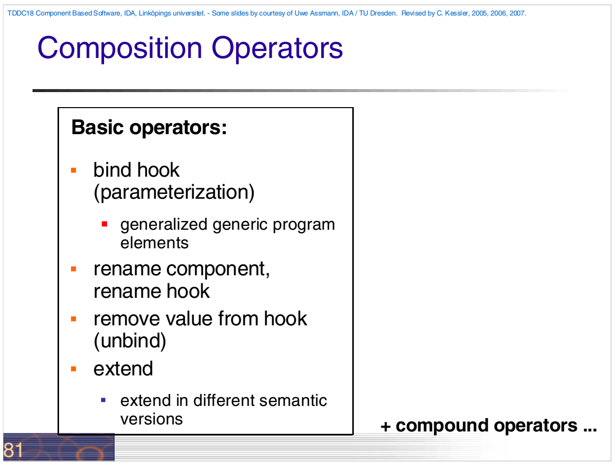
```
public print () {
  System.out.println("Hello World");
}
```

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



The Composition Technique of Invasive Composition

Uniform for declared and implicit hooks

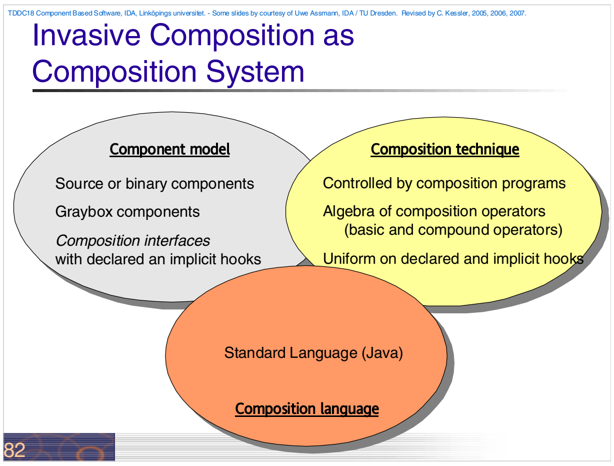


Composition Operators

Basic operators:

- bind hook (parameterization)
  - generalized generic program elements
- rename component, rename hook
- remove value from hook (unbind)
- extend
  - extend in different semantic versions

+ compound operators ...



Invasive Composition as Composition System

Component model

Source or binary components

Graybox components

Composition interfaces with declared an implicit hooks

Composition technique

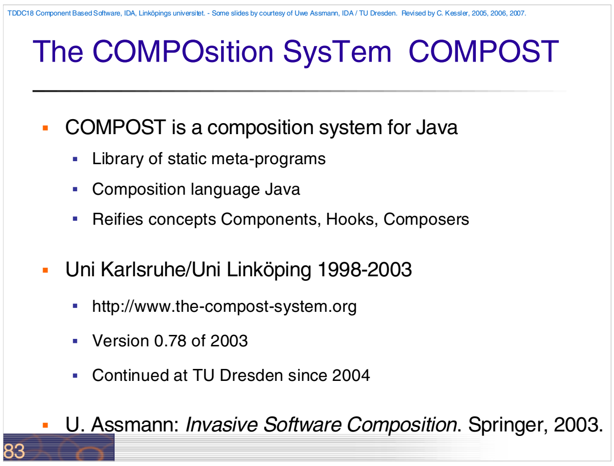
Controlled by composition programs

Algebra of composition operators (basic and compound operators)

Uniform on declared and implicit hooks

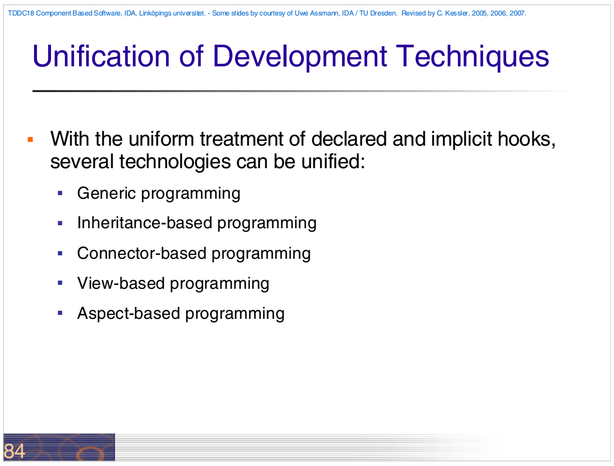
Standard Language (Java)

Composition language



The COMPOsition SysTem COMPOST

- COMPOST is a composition system for Java
  - Library of static meta-programs
  - Composition language Java
  - Reifies concepts Components, Hooks, Composers
- Uni Karlsruhe/Uni Linköping 1998-2003
  - http://www.the-compost-system.org
  - Version 0.78 of 2003
  - Continued at TU Dresden since 2004
- U. Assmann: *Invasive Software Composition*. Springer, 2003.



Unification of Development Techniques

- With the uniform treatment of declared and implicit hooks, several technologies can be unified:
  - Generic programming
  - Inheritance-based programming
  - Connector-based programming
  - View-based programming
  - Aspect-based programming

Summary:
Component-based Systems

- ... are produced by component systems or composition systems...
- ... support a component model
- Blackbox composition supports variability and adaptation
- Graybox composition also supports extensibility