

Software Composition

Component-Based Software Engineering

www.win.tue.nl/~mchaudro/cbse2007

M.R.V. Chaudron
TU Eindhoven

1

Questions ?

Homework:

- Read about pipes and filters
http://en.wikipedia.org/wiki/Pipes_and_filters
- Think about:
 - Composing/Binding software?
 - What do architects/engineers need?
 - In what ways is software currently composed?
 - How does binding work in Pipes and Filters?

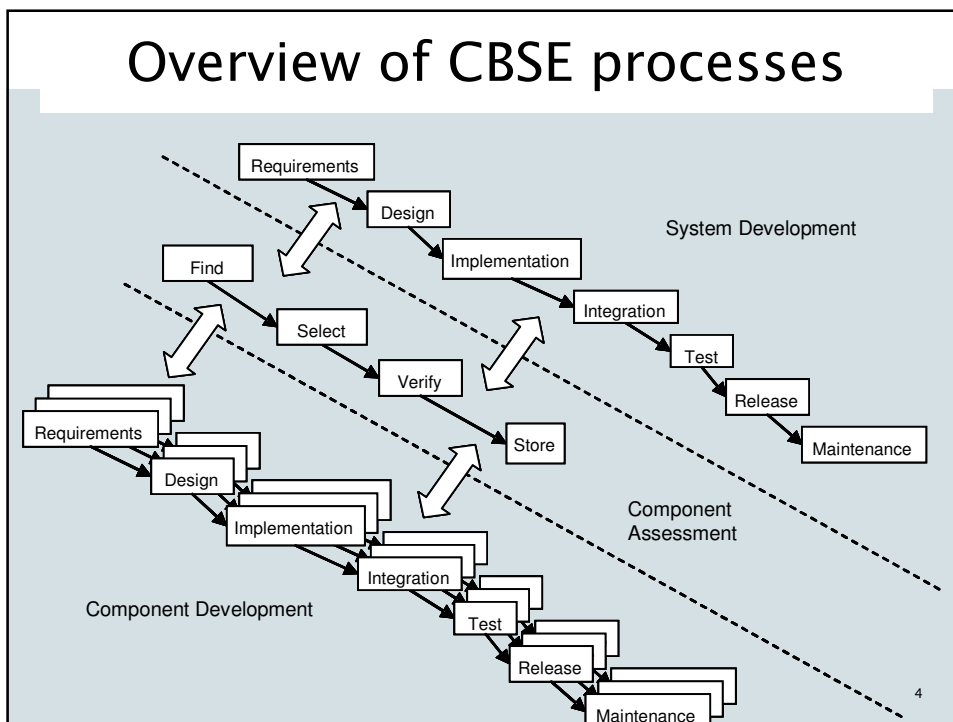
2

Today's Lecture

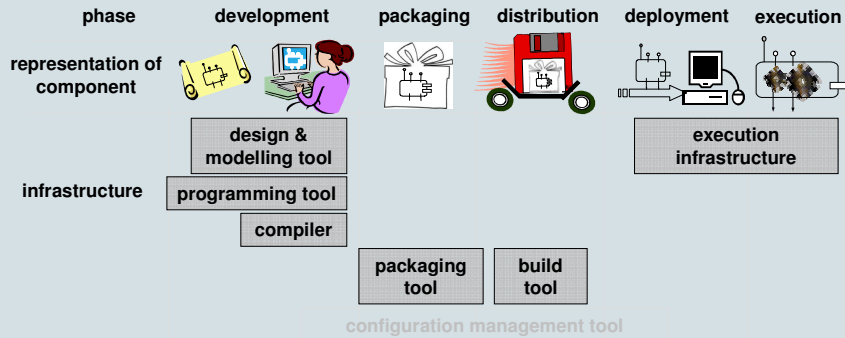
- ▶ Recap Component Models
- ▶ Composition

Feedback on lecture notes are welcome.

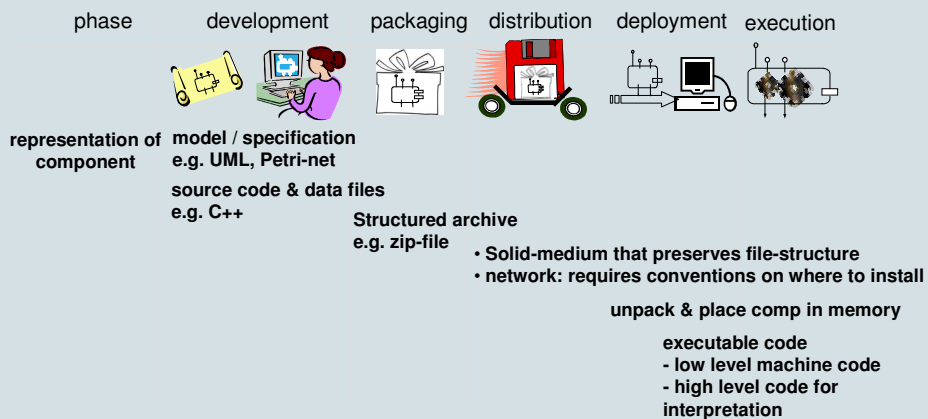
Overview of CBSE processes



Component Development Lifecycle



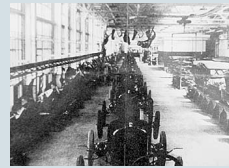
Component Development Lifecycle



In different processes, components are composed in different phases of development

Modularity in different phases

- **Modular in Design**
 - Modern computers
 - Eclectic Furniture (not “modular” furniture)
 - Recipes in a cookbook
- **Modular in Production**
 - Engines and Chassis
 - Hardware and software
 - NOT chips, NOT a cookbook
- **Modular in Use**
 - “Modular” furniture, bedding
 - Suits and ties
 - Recipes in a cookbook



By Baldwin & Clarke 7

COMPONENT MODELS

Component Model

Definition A *Component Model* specifies the standards and conventions that are needed to enable the composition of independently developed components.

Typically:

- The composition mechanism
- The conventions that components must adhere to in order to enable successful composition

Definition A *Component* is a building block that conforms to a component model.

9

Aspects of Component Models

- A component model typically addresses
- Life-cycle management:
 - instantiation, (de)activation, removal
- Binding mechanisms
- Interaction style
- Data exchange format
- Process model

10

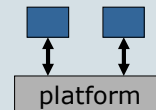
Component Model

What conventions does a component model need to specify to enable **composition** ?

Standards for

- Implementation technology
- Standard interfaces
- Specification/documentation/meta-data

Component Execution Platform



Component Platforms may provide support for

Component lifecycles

- install, create, replace, start, stop, remove

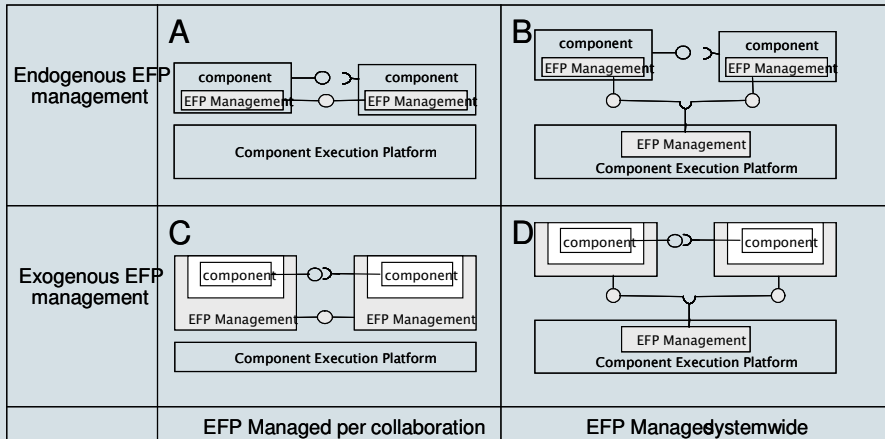
Inter-component services

- binding
- interaction

Extra-functional / resource aspects

- interoperability (language/OS)
- scheduling
- quality of service management
 - (dynamic) load balancing
 - (re)negotiation
- security
- fault tolerance (replication)

How does each of these aspects affect composability?



Composition



Substitution versus Composition

Substitution	Composition
Replace one ' <i>piece of software</i> ' with another that has the same observable properties - in particular the same interface.	Compose ' <i>pieces of software</i> ' such that they may interact.
This is 'do-able' and often sufficient for reuse of software.	
Context is static, Semantic context is known	Context may be dynamic, Semantic context is not known
Typically done before compilation	Typically done after compilation: run-time

15

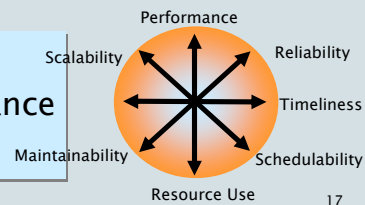
In what ways is software currently composed?

16

What should be the aim of composing software ?

- Extension?
- Interoperation?
- Functional composition?
- Synchronisation?
- Containment?
- ... Suggestions from the audience ...

Composition = the combining of functions & features under invariance of quality properties



What is the desired unit of composition?

- What units make sensible software building blocks?
- To whom?

What do you want to compose?

Aspect of Scale / Granularity:

- Instructions
 - Methods
 - Modules / Classes
 - Frameworks / Libraries
 - Services
 - Components / Aspects
 - Features
 - Systems
- } intra-language
- } intra-architecture
- } across systems

Composition: the programming language level

- Functional Programming Languages
- Logic Programming Languages
- Imperative Programming Languages
- Object Oriented Programming Languages

Composition in Functional Prog. Lang's

Functional Programming: Haskell, ML, LISP, ...

Philosophy:

Software is for computing functions.

The composition of functions is again a function.

- $(f \circ g) x$ or $f(g(x))$
- The output of function g is input to function f
- Requires compatible types
 - 'o' is of type $(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$
- Has nice algebraic properties (distr, assoc, comm, idem)
- Function is also unit of abstraction/structuring
- Has elegant semantics
 - for the functionality, not for the behaviour!
 - not good at handling state

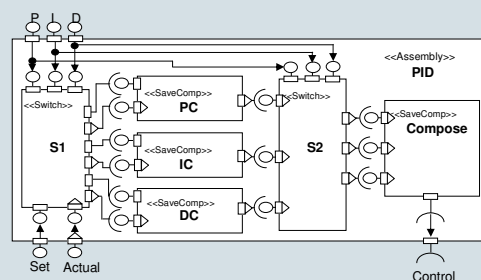
21

Invariance of form & behaviour

- $(f \circ g)$ is again a function
with similar input, output behaviour
- composition of filters is again a filter

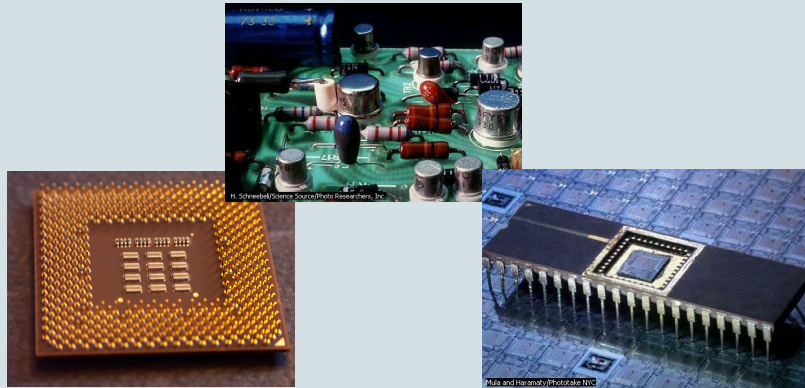


- How about components with many interfaces?



22

Streaming has uniform behaviour on the interfaces....



23

- How does the existence of state affect composition? ($f \circ g$)
- A component instance may be in one of several states.
- How does this affect other components?

24

Composition in Logical Prog. Lang's

Logical Programming: Prolog

Philosophy: *Computing = Theorem Proving*

A program is a set of deduction rules (in predicate logic)

Composition is the union of deduction rules.

- “ $B(x) \Leftarrow A(x)$ ” \wedge “ $C(y) \Leftarrow B(y)$ ”
- The result of ded-rules are the facts that can be derived from them
- Semantics is elegant for small programs only
 - The interaction of deduction rules becomes incomprehensible
 - There is no intuitive relation between a program and its behaviour

25

Composition in Imperative Prog. Lang's

Programming Languages: Fortran, Pascal, C, ADA, ...

Philosophy:

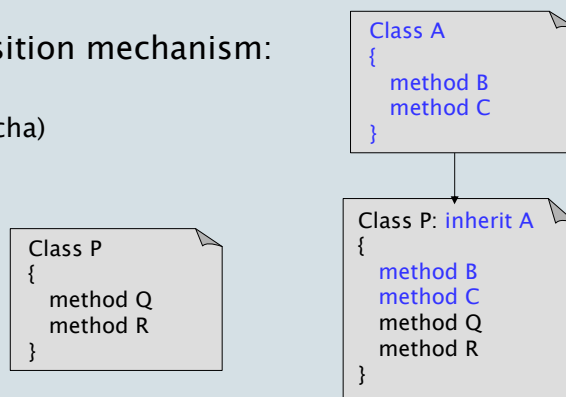
- A program is a sequence of elementary instructions for a processor.
- Composition is the appending/merging of the sequence of instructions.
- Elementary instruction are: assignment & expression evaluation
 - $x := y * y$
 - instructions can be composed using control-flow statements:
 - ; , **if .. then .. else**, **while .. do**, **repeat** ,
- Semantics is elegant for small programs only; reasoning is difficult
 - There is a direct relation between a program and its behaviour
 - This lack of abstraction complicates reasoning about large program

Composition: the programming language level

- Technically we have:
 - FP: functions; LP: clauses; OO: classes
 - Issues in composition:
 - Syntax: form of the interface
 - Execution model: control flow, data flow
 - Data model: representation of data, meaning of data
 - Often incomplete information in the program text

Composition in OO Prog. Lang's

- OO Prog. Lang's: Simula, C++, Java, ...
- Objects are essentially a structuring/abstraction mechanism
- Main composition mechanism:
 - Inheritance
 - Mixin's (Bracha)

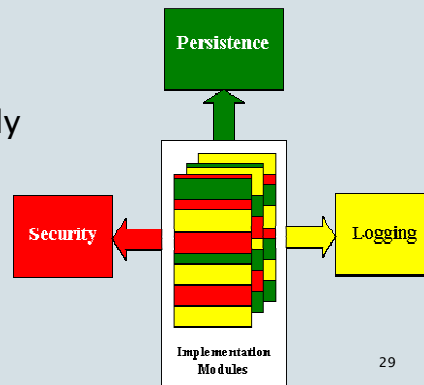


Aspect Oriented Composition

Design & maintain concerns in isolation

Automatically construct implementation by 'weaving' concerns

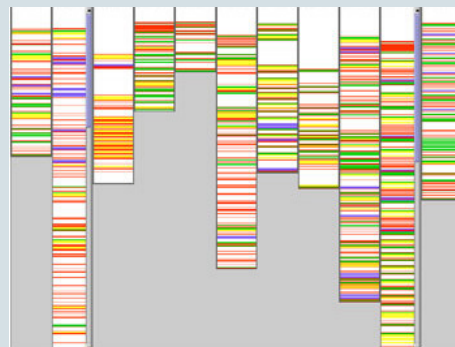
Current technologies are mainly oriented at code level,
But principle is also applicable at design / modeling level



29

Example Cross Cutting Concerns

- ASML software component consisting of 19 KLOC of C code
- each column represents a module within the component
- each colour represents one of four Cross cutting concerns:
 - tracing (green),
 - pre- and post-condition checking (yellow),
 - memory-error handling (blue), and
 - general-error handling (red).
- the CCCs considered comprised roughly 31% of the code.



Composition in Aspect-orientation

Aspect = piece of code related to one *concern*

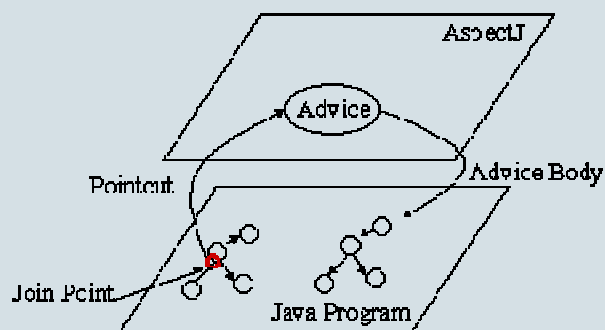
Pointcut = an expression that identifies a set of join-points in the program text; typically:

- » Procedure-call
- » Assignment to a specific variable

Advice = instruction as to how to combine an aspect with a joinpoint

For example: **before, around, after**
entry, exit

Typical examples: tracing & logging calls,



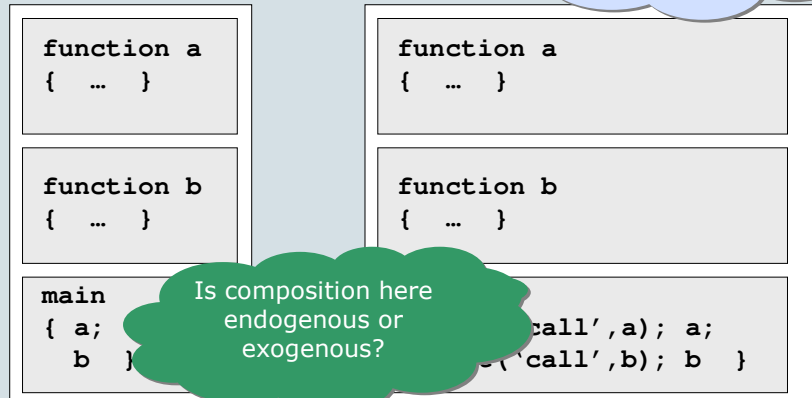
Aspect-oriented composition

Aspect: { print('call',f) }

Pointcut: at every function call f

Advice: before

Aspect extentions to many languages exist. Most popular: AspectJ



33

Questions?

34