Goals of this lecture

• Students have an overview of motivation and concepts of Component-Based Software Engineering

• Students have an understanding of how a CBSE system works in practice
Agenda

- Motivation
- Component models and frameworks
- Composition
- Examples
Design process – four elements

1. (Domain) analysis
   • increase knowledge, make models
     – use cases, based on stakeholder viewpoints
   • feedback to stakeholders
     – validation of requirements: “Do we solve the right problem?”

2. Apply strategies
   • hierarchical decomposition:
     – top-down (factorization): specify advanced building blocks (decompose functional specification, and derive extra-functional properties for the parts)
     – bottom up: design advanced building blocks
   • apply patterns, styles
     – pattern, style: coherent set of design decisions
   • generate alternatives

3. Synthesis
   • evaluate and choose alternatives, combine partial solutions

4. Verification
   • is the system according to specification? (“Did we solve the problem right?”)
Motivation for Software Components

1. Separate *application development* from *component development*
   - manufacturing rather than engineering
   - bring standardization, facilitate a new *industry* and a *market*
   - bring re-use: improve productivity (time to market) and cost

2. Modularity
   - decomposition, and localizing *dependencies*
3. Flexibility – facilitate *change*: easy removal and addition of functionality
   • the methods and processes for this are explicitly defined
   • facilitates *product lines*, different versions of a product

4. Predictability – Similar to other engineering disciplines
   • properties of the whole derived from properties of the parts

   • Early ideas: Douglas McIlroy, NATO conference ’68
   • (“*Mass produced software components*”)
Sample system components

- Entire computer systems
  - put together into a distributed system
- Components on a motherboard
- CPU platforms
  - ISA, interfaces to devices, ..... 
- Operating Systems
  - OS-API, process model, file model, GUI, ..... 
- Source code libraries
  - standard template library
- Static, compiled libraries
  - Math functions library, file io, concurrency support functions
- Dynamic libraries, DLLs
- Executable programs

*Picture by A.S. Tanenbaum*
Agenda

- Motivation
- Component models and frameworks
- Composition
- Examples
According to SEI in CMU/SEI-2000-TR-008:
http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5203

- Component-based software engineering is concerned with the **rapid assembly** of systems from components where
  - components and frameworks have **certified properties**;
  - and these certified properties provide the basis for predicting the **properties of systems** built from components
CBSE: Component definition

• What is a component?
  • Szyperski: '97: A software component is a *unit of composition* with *contractually specified interfaces* and *explicit context dependencies only*....
    – explicit, i.e., no dependencies other than through interfaces
  • ... A software component is *independently deployable* and subject to *composition by third parties*.
    – no partial deployment
    – can a component be defined at source code level?

• Components have two aspects
  • they implement functionality
    – one or more services
    – in various modes (a certain quality level for a certain resource footprint)
  • they represent an abstraction, a style
CBSE: Component Model

- Component *model*
  - defines what is, and what is not a component
  - specifies how to use metadata
  - defines a series of concepts
    - coordination, composition
    - quality attributes
    - typing: interfaces
    - set of standard interfaces
    - binding and instantiation
    - interaction style
- A component model specifies the standards and conventions that enable composition of independently developed components
- A component conforms to this model

From: *Technical Concepts of Component-Based Software Engineering, CMU/SEI-2000-TR-008*
CBSE: Component Framework

- Component framework
  - a framework to work with a certain component model, as a style
  - defines application life cycle
  - platform, run-time services, component “docking”
  - process model (of the running system)

- A component has three obvious dependencies:
  - to the platform, using standard interfaces
  - to other components: provided/required
    - both can be managed through the run-time


Remember: Framework

Szypersky: the primary task of a component framework is to enforce architectural principles (policies, communication styles)

• A framework consists of
  • a ‘static’ part
    – programming model, data model
      – Libraries, IDLs
    – life cycle model
    – methods or tooling for development
  • a ‘dynamic’ part
    – a run-time system, or platform
      – entirely separate entity or a library
    – a set of services
      – provided by the platform
      – e.g. binding, discovery, installation
    – a process model
Component framework services

blue = mandatory

Taken from: A Component Framework for Consumer Electronics Middleware
Component life cycle

- **Component life cycle**
  - design
  - distribution
  - create instance, start/stop, destroy

  **requirements** → **modelling** → **implementation** → **packaging** → **deployment** → **execution**

  - Specification
    - interface
    - models
    - meta data
  - Code
    - source code
    - executable code
    - executable models
  - Storage
    - repository
    - package
    - meta data
  - Installed files
  - Executable code

- Notice that a component can be regarded as *a set of models*
  - source code, binary code, performance model, simulation model,.....
  - .....different aspects of what the component is
  - this is the perspective of the *ROBOCOP* component model

**A Classification Framework for Software Component Models,** Ivica Crnkovic, Severine Sentilles, Aneta Vulgarakis, and Michel R.V. Chaudron
Example: representations of an executable program

Component:
- executable program

- Requirements/modelling/design phase
  - interfaces, and behavior
    - input, output
    - GUI
    - file system, network assumptions

- Implementation phase
  - Source code, executable code
    - possibly: executable model

- Packaging phase
  - (machine readable) description

- Deployment / distribution phase
  - put executable file at certain location
Example: Corba Component

- Component model:
  - generalization of an object
- Requirements/modeling
  - model of requires/provides interfaces ("ports"), and interaction styles
- Implementation
  - using interface-definition files and compiler support to implement the ‘standard’ part
  - instances managed at runtime by a *home*
  - a *container* acts as a process context
- Packaging
  - zip, containing code and description
- Deployment
  - start ‘container’ and register with ORB (object request broker)

**Note:** standards document: 350 pages; tutorial: 216 slides.
Application life cycle (I)

- An application is a set of connected and cooperating components
  - no ‘dangling’ interfaces
- Connecting components is called *composition*.
  - *Horizontal*: simply connect interfaces without further rules for composite
  - *Vertical*: a composite (‘partial application’) is again a component,
    - *aggregation* (bring internal interfaces outside)
    - *delegation* (map external interfaces to internal)
Application life cycle (II)

• Phases in composition:
  • discover (lookup) components, by interface
  • deploy
    – distribute: components to machines
    – instantiate (start) components
  • bind interfaces
    – first party: control resides in either of the bound parties
    – third party: binding control lies outside the bound parties – requirement for CBSE

• run-time management:
  – monitoring, (re-)allocation, fault recovery, destroy
Requirements: on model and framework

- The component framework must include the services for composition
  - discovery, perhaps a registry, or a repository
  - deployment: component allocation and instantiation, further management
    - typically, a framework component called a *dock*
    - that implements *policies* for this part of the lifecycle, e.g. when to create/destroy, security
  - possibly: resource management, monitoring
- The component model must include the interfaces to perform the component-related tasks of composition
  - third party binding, perhaps monitoring, start/stop
Example:

- **Component type:**
  - executable program

- **Composition**
  - discover: OS finds location of command (i.e. executable program) based on name
  - distribute: depends on whether the OS manages several processors
  - start: regular program start
  - bind: e.g. Unix pipes connect output of one command to input of next
    - `cat file | sort | unique | wc`

**Note:** this was not *designed* to be a component framework
Example

- **Component**
  - Corba Component

- **Composition**
  - **discover**: registration with ORB
    - POAs register object reference in ORB naming service
    - Clients lookup references using the naming service
    - HomeFinder service (to find the Home interface)
  - **distribute**: not in the framework
  - **start**: control lies with an ‘Assembler’ tool that takes a descriptor and activates home and containers on the relevant nodes
  - **bind**: through Corba’s ORB mechanism
    - ORBs are connected
Runtime

Provides client control over the component’s lifecycle

from Corba Component Model Specification 4.0(OMG)

ORB: Object Request Broker
POA: Portable Object Adapter
Application deployment scenario

Assumption:
Allocation and installation of components A and B already done

Taken from: this CCM tutorial
# Evaluation Existing Component Frameworks

<table>
<thead>
<tr>
<th>Component Framework</th>
<th>COM</th>
<th>DCOM</th>
<th>EJB</th>
<th>.NET</th>
<th>CORBA</th>
<th>Koala</th>
<th>Robocop</th>
<th>PECOS</th>
<th>Autocomp</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Infrastructure</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Binding</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Communication</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Distribution</td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Announcement of Cap.</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Discovery</td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Extra Functional Properties</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Robust &amp; Reliable operation</td>
</tr>
<tr>
<td>Security</td>
</tr>
<tr>
<td>Low resources</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Upgrading and Extension</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Design time</td>
</tr>
<tr>
<td>Compile time</td>
</tr>
<tr>
<td>Run time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Development Support</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Language Independence</td>
</tr>
<tr>
<td>Platform independence</td>
</tr>
<tr>
<td>Analysis techniques</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Trading</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Taken from: [A Component Framework for Consumer Electronics Middleware](#)
Component and service

• In networked systems, it is fruitful to discriminate between *component* and *service*
  • separation of functionality (the service)
  • and how it is implemented (the components)

• Applications are a set of connected *services*
  • steps remain the same:
    – lookup components: service discovery
    – deploy = allocate and start service, where allocation is twofold:
      – of component to service and
      – of host to component
    – bind: connect service interfaces

• This results in the same style for application composition, *without* requiring a component-based realization.
  • similar to SOA style; however, SOA requires also independence of services
Example: UPnP

- **Component model**
  - none

- **Composition**
  - **discover**: SSDP (immediate, local discovery protocol)
  - **allocate**: fixed;
    - actually, the service is delivered by the device; this is defined by the manufacturer
  - **start**: fixed
    - the service is active upon starting the device
  - **bind**: calling upon the binding interface of the services
Example: UPnP services

- UPnP services are accessed using a REST-like style
  - although there is some debate on this, see e.g. the paper by Newmarch
- Service implementation is
  - entirely hidden, as are the OS and the implementation language

Interactions: control points call actions on services (top) or establish connections (binding) between services (bottom)
Agenda

• Motivation
• Component models and frameworks
• Composition
• Examples
Certified properties

- Extra-functional properties of a system are ‘emergent’ properties
  - a property of the system as a whole....
  - ....determined by various properties of many or all of its parts
- For component based systems:
  - it must be possible to determine the properties of an assembly from the properties of the components
    - (actually, this is the goal, as for other engineering disciplines)
    - then these properties have to be specified as a model....
      - as ‘metadata’ of interfaces, or of a component as a whole
      - determined using some ‘testbank’ or modeling method
    - examples:
      - resource model: amount of processing per call, amount of memory
      - information leaking, reliability/availability
- methods are required to manage trust in such properties
  - certification (by an authority), monitoring (by the platform)
Reasoning about compositions

Non-trivial, examples are

• **Performance**: response time to an event (say key press), in the context of concurrent processes competing for platform resources
  • Model
    • Set of tasks with deadlines and WCET (worst-case execution time)
    • Scheduling policy, e.g. EDF (earliest deadline first)

• **Security**: protection of a website
  • Tactic: add dedicated components
    • DMZ-pattern (demilitarized zone)
    • see next slide

• **Availability**:
  • Model
    • Component availability given by $a_i, 0 \leq a_i \leq 1$
    • System availability $\prod a_i$, assuming independent components
Example: screened subnet architecture

- Protection is achieved:
  - by combining components, with particular security properties
    - packet filtering firewalls
    - secure communication
  - by having behaviour policies
- Network between the two filtering firewalls: a DMZ

- Policies
  - Servers in the DMZ operate as reverse proxy (i.e., proxy at server side)
  - Traffic to the internal network is only allowed from the DMZ
  - Internal servers cannot initiate traffic
  - All traffic is routed (as opposed to flooded)
- Functions of the DMZ servers include access control (e.g. password checking, secure communication) and request verification
Agenda

- Motivation
- Component models and frameworks
- Composition
- Examples
Example: Robocop component model

- A Robocop component is a set of related models
  - similar to architectural views: different aspects of the same underlying entity
  - models can be left out and added
    - not always all models are required
  - example relations:
    - source code S and executable E are related by compiler C
    - executable E on platform P has resource model R

- The models are used:
  - to select components
  - to determine properties of compositions on platforms
Layered, run-time view of a terminal

- **Application Layer**
  - Applications, composed of components
- **Middleware Layer**
  - Run Time Environment, providing management services for
    - both applications
    - and components
  - Executable components
- **Platform Layer**
  - OS Abstraction
  - Device & HW drivers
Predictable assembly, exploration

- Tooling, by Bondarev, based on the Robocop model
- Adding besides the component model a platform model
  - allows investigation of different mappings to different hardware
  - allows tradeoff among several platforms
  - using a simulation techniques based on ‘critical traces’.

Deepcompass framework

Bondarev, Chaudron, de With
A component framework (only available as prototype)

- **Targets I³ - applications**
  - interdisciplinary,
  - intrinsically distributed,
  - intricate resource management

- **Component model:**
  - Plugins in dll-libraries
  - Resource demand model

- **Lifecycle model:**
  - Service oriented application composition and orchestration
  - Various dynamic reconfigurations.

- **Platform model:**
  - Hosts on a LAN
  - Docks (containers) for executing components
FLUENT: entities and concepts

SYSTEM ABSTRATIONS

APPLICATION LAYER

Application

1

Orchestrator

* 1

Component instance

* 1

Service

1..*

Repository

1

ServiceType

1..*

ComponentType

1..*

FRAMEWORK ABSTRATIONS

SYSTEM LAYER

Platform

1

Resource manager

1

Resource

* 1

Resource budget

* 1

Device manager

1

Dock

+ 1

Platform node

1

Network

* 1

Memory

* 1

CPU

* 1

RESOURCE LAYER

R.H. Mak

2IMN10-CBSE page 39 25-Oct-18

TU/e

Technische Universiteit Eindhoven

University of Technology
FLUENT: runtime configuration
FLUENT: lifecycle models

APPLICATION LIFECYCLE

A: application made of services
B: map services to components

APPLICATION DEPLOYMENT

C: map components to nodes
D: set QoS levels
E: allocate resource budgets
- resource usage negotiation
- install & instantiate components
- bind components
- activate components

APPLICATION OPERATION

Analyze resource supply & usage
make dynamic reconfiguration choices

COMPONENT LIFECYCLE

COMPONENT DEVELOPMENT

• specification
• implementation
• instrumentation

REPOSITORY

COMPONENT DEPLOYMENT

needed in some configuration

COMPONENT OPERATION

binding → activation
running
unbinding ← deactivation

installation → instantiation
uninstalling ← termination