Component based software – introduction

Lars-Åke Fredlund

2011–2012

Course Information

- A series of **lectures** in English:
  - Designing components
  - Describing components (interfaces, . . .)
  - Implementing components
  - Validating components
  - Programming using components
  - Example of (distributed) component frameworks
- 1–2 **obligatory** exercises
- My email: `lfredlund@fi.upm.es`
- Course web page:
  `http://babel.ls.fi.upm.es/~fred/sbc/`

Course Information

- The course will be an overview of component-based software
- We will mention a lot of different languages, frameworks, techniques, etc
- To get something out of the course you have to be **active**:
  ask questions during class,
  read about items items mentioned in class
  (starting at `wikipedia` and `google`)
  Write programs, install tools and try them out!
- Be ambitious with the exercise: do a thorough investigation of the problem and technique you choose

Lecture Plan

- Today: introduction to component based systems
- Component specification
- Validation and verification of components:
  testing, formal verification
- Software Architectures (for components)
  software buses, multitier architectures, ...
- Examples of (distributed) component frameworks:
  Erlang, Web Services, Mashups, Autosar, ...
- Extras
- Your lectures

About the exercise

- Study and use one of the component frameworks,
  `or` specify, implement, and validate a set of components,
  `or` study the impact and/or problems (economic, timewise) of introducing components in software development,
  `or` study and use software architecture description methods
- Mail suggestions to us beforehand!

Document result:

- Give a presentation (around 30 minutes)
- A report (15–20 pages) – Spanish allowed
- Participate (ask questions) at other presentations

About the exercise

It is **not** just a literature study; we do not want to read 12 pages of an introduction to Web Services extracted from Wikipedia

- **Learn** a framework
- **Apply** the framework to an interesting example, as part of a critical **Evaluation**

  Program a solution, write a specification and test, use an architecture description language to specify an architecture, study a development process, . . .
- **Document** the **result** of applying the framework to the example, with criticism resulting from **your** study: *did things work?*, *what were the benefits compared to not using the framework?*, *what were the problems?*, etc
- **Do not be afraid to include concrete details in the report:** source code, specifications, etc.

Motivation: why component-based software

- Classic argument: **Cost of software development**
  - need to re-use software to reduce costs
  - better to buy off-the-shelf than reimplementing

Motivation: why component-based software

- Classic argument: **Cost of software development**
  - need to re-use software to reduce costs
  - better to buy off-the-shelf than reimplementing
- **More reliable software**
  - more reliable to reuse software than to create
  - system requirements can force use of certified components (car industry, aviation, . . .)

Motivation: why component-based software

- Classic argument: **Cost of software development**
  - need to re-use software to reduce costs
  - better to buy off-the-shelf than reimplementing
- **More reliable software**
  - more reliable to reuse software than to create
  - system requirements can force use of certified components (car industry, aviation, . . .)
- **Emergence of a component marketplace**
  Apple's App Store, Android Market, . . .

Motivation: why component-based software

- Classic argument: **Cost of software development**
  - need to re-use software to reduce costs
  - better to buy off-the-shelf than reimplementing
- **More reliable software**
  - more reliable to reuse software than to create
  - system requirements can force use of certified components (car industry, aviation, . . .)
- **Emergence of a component marketplace**
  Apple's App Store, Android Market, . . .
- **Emergence of distributed and concurrent systems**
  we need to build systems composed of independent parts, by necessity

Trends in SW design

- **Concurrency** – multiple activities at the same time
- **Distribution** – multiple activities at the same time, at different locations

Today component frameworks needs to address concurrency and distribution because of

- **Hardware developments:** microprocessors with many cores (Intel quad –4– cores..., ARM processors for mobile phones)

  Leading to renewed interest in concurrent programming

- **Software developments:** Web services communicate to offer composite services (business processes)

  Distribution and fault tolerance to handle 24/7 availability requirements

Some History (towards component-based software)

- Distributed systems
- Open systems
- The problem of re-use
- Evolution of programming models (including web)

Distributed Systems

Concurrent programs executing on different hosts that do not share memory

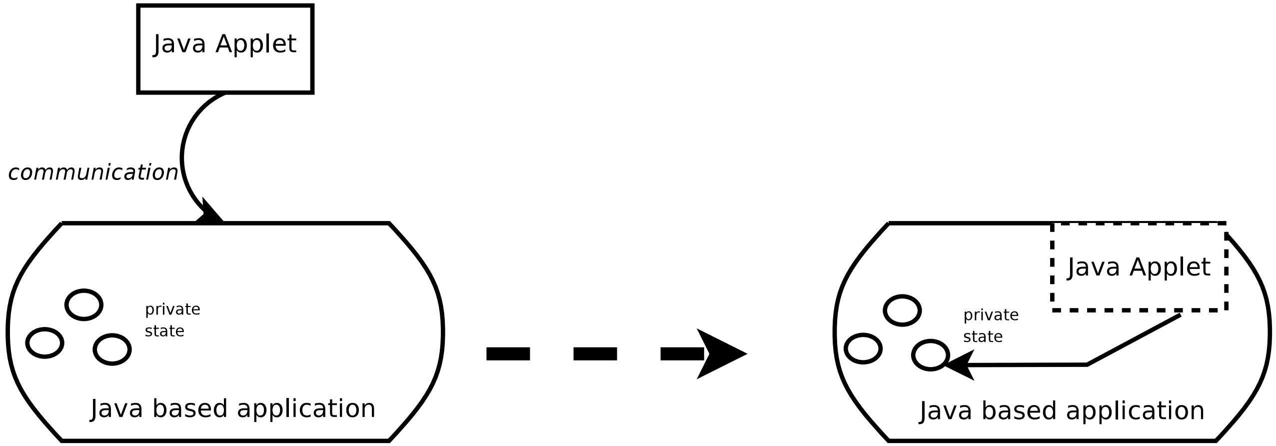
- Different communication mechanisms: message passing, RPC (remote procedure calls), . . .
- Typically systems that are online 24/7
- Reliability and fault tolerance is a key concern: hardware and software *will fail*, network links *will fail*, software has to recover from failures

Open Systems

- Distributed systems consisting of *heterogeneous* programs
- Programs programmed in different languages, running under different operating systems, . . .
- Some programs already exists (legacy systems)
- Other programs enter and leave the system during its execution

Open Systems

- Distributed systems consisting of *heterogeneous* programs
- Programs programmed in different languages, running under different operating systems, ...
- Some programs already exists (legacy systems)
- Other programs enter and leave the system during its execution
- Example: a Java based system accepting a new applet:



Open Systems: Actors

Actors is a classical programming model for open systems

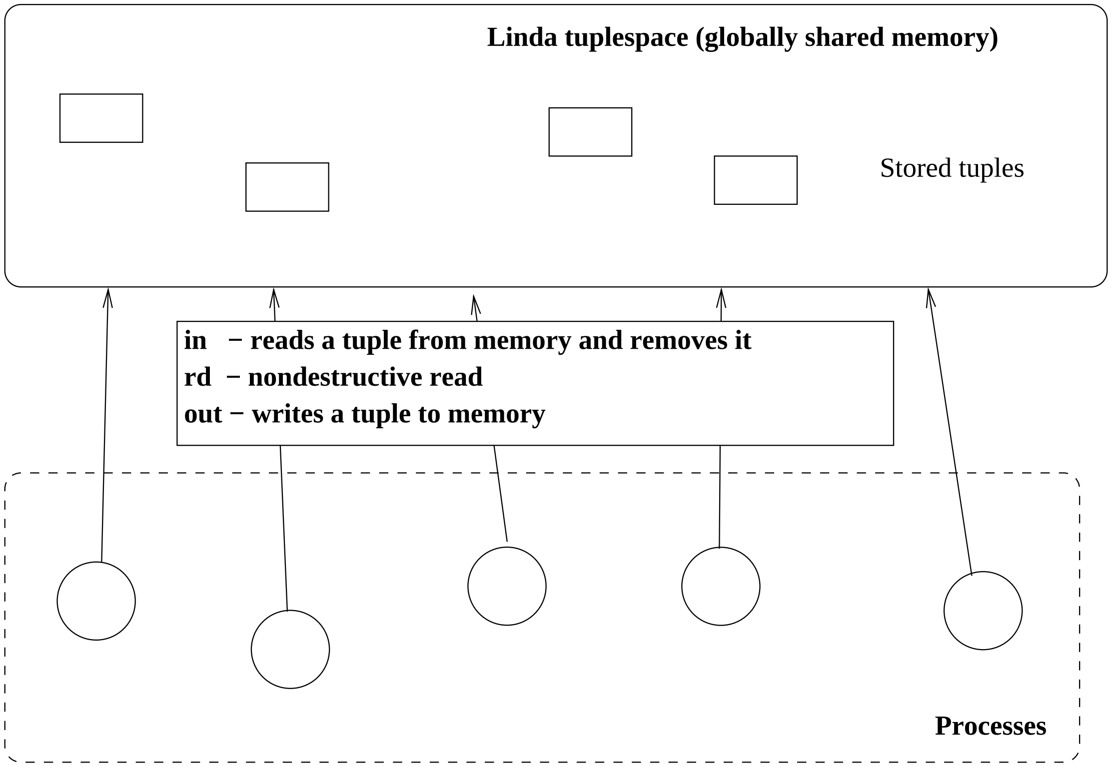
- Active objects
- Asynchronous messages
- Point-to-point communication
- Actors can create other actors (dynamically)
- Communication patterns are dynamic too (communication endpoint identifiers can be transmitted)
- Languages using an Actor-like communication model: Erlang

Open Systems: Coordination Models

- Entities (programs, processes) to control + coordination medium + coordination laws

Open Systems: Coordination Models

- Entities (programs, processes) to control + coordination medium + coordination laws
- Oriented towards data-sharing: Linda



Linda example

Operations:

$out(\langle v_1, \ldots, v_n \rangle)$	writes the tuple $\langle v_1, \ldots, v_n \rangle$ to memory
$in(tupletemplate)$	destructively reads a tuple from memory (blocking)
$rd(tupletemplate)$	nondestructive tuple read (blocking)
$eval(process)$	creates a new process

Examples:

$$out\langle \texttt{`person'}, \texttt{`juan'}, \texttt{22} \rangle$$
$$in\langle \texttt{`person'}, ?name, ?age \rangle$$

Linda example

Operations:

$out(\langle v_1, \ldots, v_n \rangle)$	writes the tuple $\langle v_1, \ldots, v_n \rangle$ to memory
$in(tupletemplate)$	destructively reads a tuple from memory (blocking)
$rd(tupletemplate)$	nondestructive tuple read (blocking)
$eval(process)$	creates a new process

Examples:

$$out\langle \texttt{'person'}, \texttt{'juan'}, 22 \rangle$$
$$in\langle \texttt{'person'}, ?name, ?age \rangle$$

How can we change the age of `'juan'`?

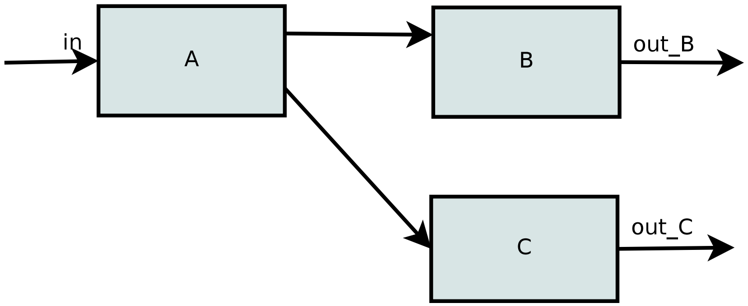
Open Systems: Coordination Models

- Entities (programs, processes) to control + coordination medium + coordination laws

Open Systems: Coordination Models

- Entities (programs, processes) to control + coordination medium + coordination laws

- Oriented towards control: filter/flow-based programming



- Data arrives as messages at the filter input

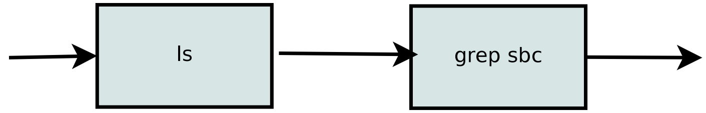
- A filter either manipulates a data item or lets it through unchanged to its outputs

Open Systems: Coordination Models

- Entities (programs, processes) to control + coordination medium + coordination laws
- Real-world example: pipes in UNIX

`ls | grep sbc`

`ls`	creates a file with a filename per line
`grep sbc`	removes all lines that do not contain `sbc`

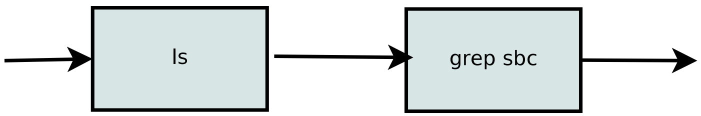


Open Systems: Coordination Models

- Entities (programs, processes) to control + coordination medium + coordination laws
- Real-world example: pipes in UNIX

`ls | grep sbc`

`ls`	creates a file with a filename per line
`grep sbc`	removes all lines that do not contain `sbc`



- Other example: MapReduce for distributed computing on large data sets

Coordination Systems: Open Documents

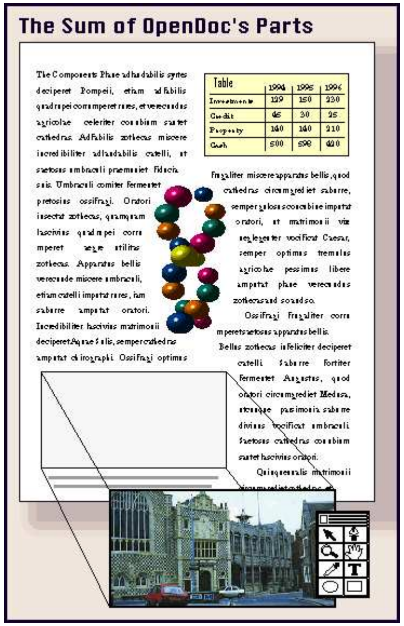
OpenDoc: one of the first component-based systems

- **Document centric**: no main application exists, the document is the central information store (compare Linda)
- **Compositional**: documents are composed from (possibly) distributed elements that themselves may be documents

Coordination Systems: Open Documents

OpenDoc: one of the first component-based systems

- **Document centric**: no main application exists, the document is the central information store (compare Linda)
- **Compositional**: documents are composed from (possibly) distributed elements that themselves may be documents



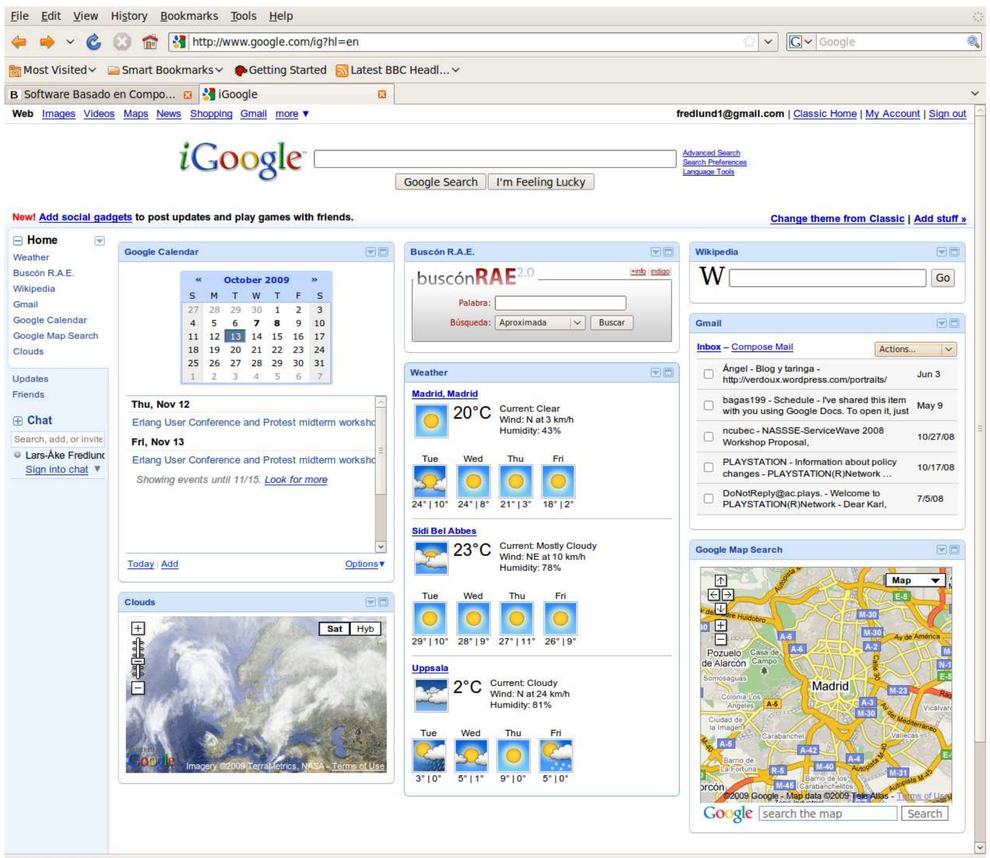
- Document elements can be active entities. Every element item has an editor (application) associated with it.
- Created at Apple in the 1990s (compare Microsoft OLE)
- Very ambitious goals: difficult to realise then and probably even today (compare the state of web browsers/servers)

Coordination Systems: Mashup web application

- “A web application that combines data from external sources to create a new service”

- “A web application that combines data from external sources to create a new service”

- Example: a customized google page:



Reuse of Software

The age-old problem in software industry: how to reuse software

- At the most basic level: **source code reuse**

Reuse of Software

The age-old problem in software industry: how to reuse software

- At the most basic level: **source code reuse**
- Old solution example: reuse of code for regular expressions evaluation in UNIX (replicated in many applications: `grep`, `bash`, `sed`, ...)

Reuse of Software

The age-old problem in software industry: how to reuse software

- At the most basic level: **source code reuse**
- Old solution example: reuse of code for regular expressions evaluation in UNIX (replicated in many applications: `grep`, `bash`, `sed`, ...)
- Advantages:
    - Good productivity
    - Consistency (regular expressions work the same)
    - No need to test re-used software pieces
- Everything is reused (analysis, design, code, documentation)
- Normally put in code libraries

Software Reuse: Source Code Libraries

Problems with re-use at the source code library level:

- When a library is modified one has to recompile and relink all applications making use of the library piece
- Hard to maintain different library versions for different applications
- Difficult to sell

Software Reuse: Binary Libraries

- No need to recompile and relink applications upon library change (dynamic libraries)
- Easier to sell (no need to distribute code)

But:

- Because of weak interfaces (at most type checked) it is difficult to know what impact a library change has on the corresponding application (we have to test and test and test...)
- Difficult to have cross-language libraries (although works to some extent...)
- Binaries usable on one (processor, OS) architecture only
- The result will be multiple library versions in a running system (hard to maintain)

Solving the problems of Binaries

A common solution to the problem of binary compatibility is to use intermediate code instead of native (Intel X86) machine code

- A compiler translate a high-level programming language to intermediate code (not specific to the target architecture)
- An abstract machine (virtual machine) executes intermediate code (probably somewhat specific to the target architecture)
- Example of languages that use such an implementation strategy: Java (Java Virtual Machine), C#, Erlang
- Using an abstract machine technology can be a way in which to permit multiple languages to communicate: example CRL (Common Language Runtime) for C#

Programming Models

Natural evolution:

- Module-based programming (Modula)
- Object-oriented programming (Java,C++)
- Aspect-oriented programming (AspectJ)
- Component-based programming (WWW example)

Programming Models

Example: Java

- Object-oriented language
- Single inheritance
- Automatic garbage collection: no pointers
- Abstract machine technology: Java Virtual Machine (JVM)
- Applets: small Java applications that can be sent between computers, and executed at the receiving side

Security Models for Java Applets

- Applets: small Java applications that can be sent between computers, and executed at the receiving side

- Different security models for applets:

Security Models for Java Applets

- Applets: small Java applications that can be sent between computers, and executed at the receiving side

- Different security models for applets:
  - Applets are put in a **sandbox**, where they cannot harm the host (so can only do limited actions)

Security Models for Java Applets

- Applets: small Java applications that can be sent between computers, and executed at the receiving side
- Different security models for applets:
  - Applets are put in a **sandbox**, where they cannot harm the host (so can only do limited actions)
  - Applets come with a (behaviour) certificate issued by some **authority** (which one can trust or not)

Security Models for Java Applets

- Applets: small Java applications that can be sent between computers, and executed at the receiving side

- Different security models for applets:
  - Applets are put in a **sandbox**, where they cannot harm the host (so can only do limited actions)
  - Applets come with a (behaviour) certificate issued by some **authority** (which one can trust or not)
  - Applets come with a description of their behaviour, and a checkable proof of compliance (**proof-carrying code**)

Programming Models: Aspect-oriented programming

- Programs are decomposed into different aspects, each aspect responsible for one requirement (security, logging, fault-tolerance, concurrency, . . .)
- The aspects can be largely independently developed, sometimes even in different programming languages
- **Weaver**: the task of combining different aspects into a whole program
- Attractive development model but still not very mature
- Example: **AspectJ** for aspect-oriented programming in Java

WWW for component-based programming

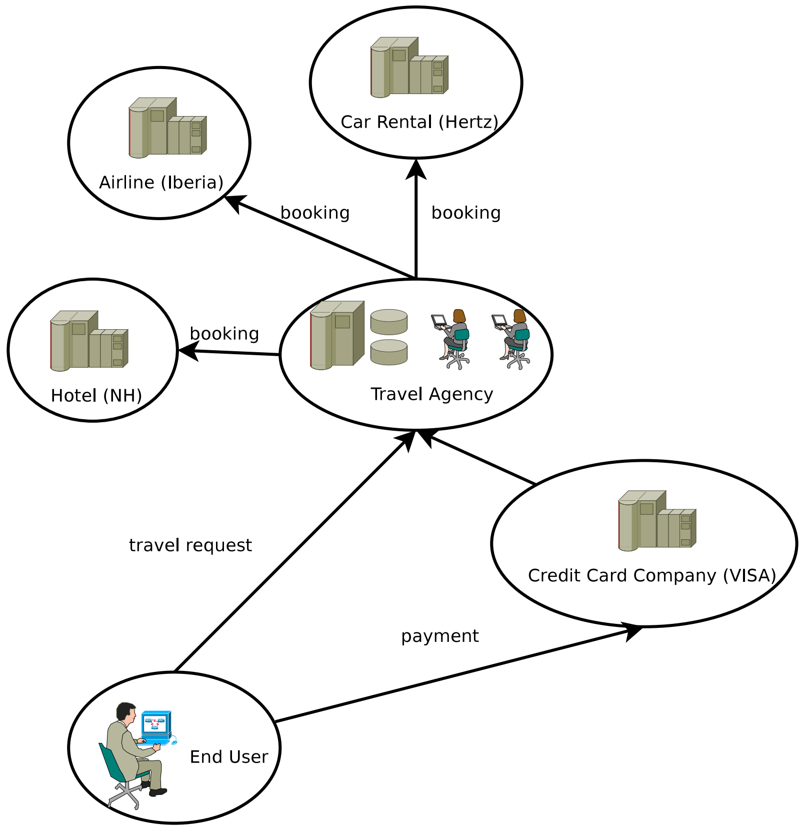
- First WWW generation: documents published using HTTP/HTML
- Second generation: dynamic generation of documents, using forms and databases (CGI)

WWW

- Third generation: everything is part of the Web
- Data is structured in a standard way (XML)
- Documents become (web) services
- Web services become accessible by other (web) services

A web-based service development model

Web services communicating using Web standards:



- Web connection: HTTP
- Web service search: UDDI
- Data definition: XML
- Messaging: SOAP
- Web service interface: WSDL
- Transactions: WS-Transaction
- Service composition: WS-BPEL

Components

What is a component?

One definition:

- **Encapsulated** i.e., with well defined **interfaces** and with an unknowable interior
- **Composable** with other components (using a well establish composition mechanism)
- **Multiple-use** (i.e., not a restricted resource)
- **Not context dependent** (usable in multiple systems)
- A unit of **independent deployment** and versioning (independent of other components)

Fundamental Concepts

- **Component interface**: describes the operations (method calls, messages, . . .) that a component implements and that other components may use
- **Composition mechanism**: the manner in which different components can be composed to work together to accomplish some task.
  For example, using message passing
- **Component platform**: A platform for the development and execution of components

Component-based Applications

Example: The **Firefox** web browser:

- Extensible architecture (using **plugins** – components)
- New plug-ins can be added (Adobe flash, spell checkers, . . .)
  At runtime?
- A **well-defined plugin architecture**: no need for plug-in developers to know all the internals of Firefox
- **Separation** of plug-ins from other plugins and the main application: a faulty plug-in should not crash Firefox (compare **Google Chrome**)
- Different providers

Component-based Systems

Linux:

- New hardware drivers from different providers
  (can be added at runtime?)
- Isolation of core OS and drivers very important (but difficult)
- Language independent?

GNOME (desktop environment):

- Consistent application configuration (`gconf`)
- **Reuse** of components for **consistency**: file browser, printer selector, secret key storage (keyring) . . .
- `D-Bus` for component intercommunication

Component-based Systems: examples

Autosar:

- A software architecture for the car industry
- Goal: reduce costs
- Vehicle producer's want third-party companies to develop their software (but are still responsible for the *overall quality*)
- Or use standard software pieces (components), but adapted to the vehicle manufacturer, moving towards a software component marketplace
- Problems: cost reductions, complex standards

Why build software using components?

An **economic argument** and a **safety argument**. . .

- *Developing* components is hard: a job for (expensive) experts
- Constructing systems by *composing* components is easier: let less expensive programmers do the job
- Or: **Buy** components off-the-shelf instead of constructing them

Tasks

Tasks

- How to **program** a component?

Tasks

- How to **program** a component?
- How to accurately describe the **interface** of a component?

Tasks

- How to **program** a component?
- How to accurately describe the **interface** of a component?
- How to **check** that a component fulfills its interface specification?

Tasks

- How to **program** a component?
- How to accurately describe the **interface** of a component?
- How to **check** that a component fulfills its interface specification?
- How to **compose** components?

Tasks

- How to **program** a component?
- How to accurately describe the **interface** of a component?
- How to **check** that a component fulfills its interface specification?
- How to **compose** components?
- And vitally important: how to **maintain** a system constructed from components . . .

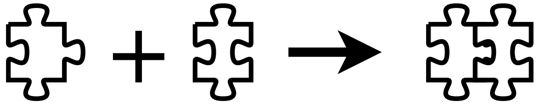
Software using components: criticism

Does the economic argument about the facility/cheapness of composition hold?

Software using components: criticism

Does the economic argument about the facility/cheapness of composition hold?

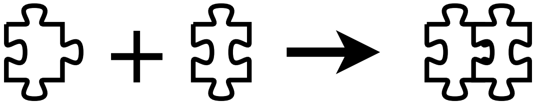
- Puzzle pieces may be easy to compose; we can tell just by the *shape* if it composes with another piece



Software using components: criticism

Does the economic argument about the facility/cheapness of composition hold?

- Puzzle pieces may be easy to compose; we can tell just by the *shape* if it composes with another piece

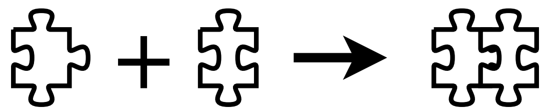


- And so there are attempts to do the same for software: *give components a shape by characterising the type of inputs and outputs*

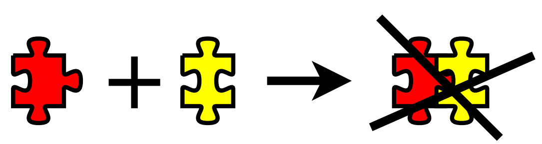
Software using components: criticism

Does the economic argument about the facility/cheapness of composition hold?

- Puzzle pieces may be easy to compose; we can tell just by the *shape* if it composes with another piece



- And so there are attempts to do the same for software: *give components a shape by characterising the type of inputs and outputs*
- But even for puzzles things are not so easy:



Component Specification: Dimensions

- Software components are hard to compose;
  there are many extra *dimensions* to a software component
- A user has to consider these extra dimensions when deciding whether to use a component

"Dimensions" of components:

Component Specification: Dimensions

- Software components are hard to compose;
  there are many extra *dimensions* to a software component
- A user has to consider these extra dimensions when deciding whether to use a component

"Dimensions" of components:

- **Input/output types**
- **Functional behaviour**
- **Concurrent behaviour**
- **Timing behaviour**
- **Resource usage**
- **Security**

Component Specification Examples: Input/Output

Input/output types

- Lets specify the operations on a component storing a set of integers:

```
initialise()
add(Integer)
member(Integer) -> Bool
...
```

- We also may need **exceptions** – handling exceptional (nonstandard) behaviour
- The operation `remove` is used to remove an existing element from a set

```
remove(Integer)
    throws exception
    // when element to remove is absent
```

Component Specification Examples: Functionality

Functionality: what is the behaviour of an operation?

- What is the relation between input and output parameters of a component and its state?
- Lets describe the integer set component again (not a program):

```
component integer_set
  var state : set

  initialise():
      state′ = ∅

  add(element):
      state′ = state ∪ {element}
```

Component Specification Examples: Concurrency

Concurrent behaviour

- Are concurrent calls to operations permitted?
- If yes, how are concurrent calls coordinated?
- What happens if a component invokes the operation `add(2)` at the same time as another component invokes the operation `initialise()`?
- Does the resulting set contain `2` or not?

Component Specification Examples: Timing

Timing behaviour

- What is the time complexity of invoking an operation? (when is an answer returned)
- For example, what is the worst-case time complexity of invoking the operation `member(element)`?

  Constant time (some hashing scheme used) or linear time (a list used in the implementation)?
- Are there any timers associated with the behaviour of the component?

Component Specification Examples: Resource Usage

Resource Usage

- Example: how much memory does a component consume?
- For example, how much memory is used to store a hundred million integers using the operation `add(element)`?

Component Specification Examples: Security

Security – what are the security implications of operations?

- Example: assume that a credit card component provides `validateCard(CardNumber,Pin) -> Bool` for checking a pin code against a credit card
- To use the `validateCard` operation we want to know that the pin code is not leaked in any way from the credit card component:

Component Specification Examples: Security

Security – what are the security implications of operations?

- Example: assume that a credit card component provides `validateCard(CardNumber,Pin) -> Bool` for checking a pin code against a credit card
- To use the `validateCard` operation we want to know that the pin code is not leaked in any way from the credit card component:
  - the operation communicates the pin to a third party

Component Specification Examples: Security

Security – what are the security implications of operations?

- Example: assume that a credit card component provides
  `validateCard(CardNumber,Pin) -> Bool`
  for checking a pin code against a credit card
- To use the `validateCard` operation we want to know that the pin code is not leaked in any way from the credit card component:
  - the operation communicates the pin to a third party
  - if the operation saves the pin, and lets another operation communicate it

Component Specification Examples: Security

Security – what are the security implications of operations?

- Example: assume that a credit card component provides
  `validateCard(CardNumber,Pin) -> Bool`
  for checking a pin code against a credit card

- To use the `validateCard` operation we want to know that the pin code is not leaked in any way from the credit card component:
  - the operation communicates the pin to a third party
  - if the operation saves the pin, and lets another operation communicate it
  - if the operation saves the pin, and reveals it by clever timing of operations

Component Specification Examples: Security

Security – what are the security implications of operations?

- Example: assume that a credit card component provides
  `validateCard(CardNumber,Pin) -> Bool`
  for checking a pin code against a credit card
- To use the `validateCard` operation we want to know that the pin code is not leaked in any way from the credit card component:
  - the operation communicates the pin to a third party
  - if the operation saves the pin, and lets another operation communicate it
  - if the operation saves the pin, and reveals it by clever timing of operations
  - if the operation saves the pin, and reveals it by clever use of resources (memory, power usage!)

Component Specification Examples: Security

Security – what are the security implications of operations?

- Example: assume that a credit card component provides
  `validateCard(CardNumber,Pin) -> Bool`
  for checking a pin code against a credit card
- To use the `validateCard` operation we want to know that the pin code is not leaked in any way from the credit card component:
  - the operation communicates the pin to a third party
  - if the operation saves the pin, and lets another operation communicate it
  - if the operation saves the pin, and reveals it by clever timing of operations
  - if the operation saves the pin, and reveals it by clever use of resources (memory, power usage!)
- An **information-flow** property (hard to verify)

Component Specification Examples

Maintainability: components may have a long lifetime – how do we maintain them?

- **Inspection:**
  - What are the interfaces of a component?
  - What is the state of a component, or a component interconnection mechanism?
    - How many requests has the component served?
    - Average waiting time until a request is served?
    - How many times has the component been restarted?
    - Are the queues used for component communication overloaded? (memory usage)

Component Specification Examples

Maintainability: components may have a long lifetime – how do we maintain them?

- **Inspection:**
  - What are the interfaces of a component?
  - What is the state of a component, or a component interconnection mechanism?
    - How many requests has the component served?
    - Average waiting time until a request is served?
    - How many times has the component been restarted?
    - Are the queues used for component communication overloaded? (memory usage)
- **Code upgrade:** how to update components on-line, without taking down the whole system

Another Component Dimension: Reliability

- Many component-based systems has to work 24/7, with **high reliability** (5 nines, i.e., 99.999%)
- **Fault tolerance:** can the component recover from hardware failures?
- A **good** component framework provides support to design and use components that are *reliable*, *fault tolerant* and *maintainable*

Next Lecture

Component specifications

- Specifying components
- Using abstractions (modelling), using formal methods
- Special emphasis on concurrent aspects