Software composition, interaction and architecture

L.S. Barbosa

Dept. Informática,
Universidade do Minho
Braga, Portugal

DI-CCTC, UM, 2009

What is software architecture?

[Garlan & Shaw, 1993]

the systematic study of the overall structure of software systems

[Perry & Wolf, 1992]

SA = { Elements (*what*), Form (*how*), Rationale (*why*) }

[Kruchten, 1995]

deals with the design and implementation of the high-level structure of software

[Britton, 2000]

a discipline of generic design and composition

What is software architecture?

[Garlan & Shaw, 1993]

the systematic study of the overall structure of software systems

[Perry & Wolf, 1992]

SA = { Elements (*what*), Form (*how*), Rationale (*why*) }

[Kruchten, 1995]

deals with the design and implementation of the high-level structure of software

[Britton, 2000]

a discipline of generic design and composition

What is software architecture?

[Garlan & Shaw, 1993]

the systematic study of the overall structure of software systems

[Perry & Wolf, 1992]

SA = { Elements (*what*), Form (*how*), Rationale (*why*) }

[Kruchten, 1995]

deals with the design and implementation of the high-level structure of software

[Britton, 2000]

a discipline of generic design and composition

What is software architecture?

[Garlan & Shaw, 1993]

the systematic study of the overall structure of software systems

[Perry & Wolf, 1992]

SA = { Elements (*what*), Form (*how*), Rationale (*why*) }

[Kruchten, 1995]

deals with the design and implementation of the high-level structure of software

[Britton, 2000]

a discipline of generic design and composition

What is software architecture?

[Garlan & Perry, 1995]

the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time

[ANSI/IEEE Std 1471-2000]

the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

What is software architecture?

[Garlan & Perry, 1995]

the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time

[ANSI/IEEE Std 1471-2000]

the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

What is software architecture?

The architecture of a system describes its gross structure which illuminates the top level design decisions, namely

- how is it composed and of which interacting parts?
- where are the pathways of interaction?
- which are the key properties of the parts the architecture rely and/or enforce?

Note:
non functional properties, e.g.

- performance, reliability, dependability, portability, scalability, interoperability ...

are not covered in this course.

What is software architecture?

The architecture of a system describes its gross structure which illuminates the top level design decisions, namely

- how is it composed and of which interacting parts?
- where are the pathways of interaction?
- which are the key properties of the parts the architecture rely and/or enforce?

Note:
non functional properties, e.g.

- performance, reliability, dependability, portability, scalability, interoperability ...

are not covered in this course.

What is software architecture?

But what kind of structure have we in mind in this course?

- code-based structures: such as modules, classes, packages and relationships like uses, inherits from or depends on.
- run-time structures: such as object instances, clients, servers, databases, browsers, channels, broadcasters, software buses, ...
- allocation structures: intended to map code-based and run-time structures to external items, such as network locations, physical devices, managerial structures ...
- entails the need for architectural views

this course focus on run-time structures

What is software architecture?

But what kind of structure have we in mind in this course?

- code-based structures: such as modules, classes, packages and relationships like uses, inherits from or depends on.
- run-time structures: such as object instances, clients, servers, databases, browsers, channels, broadcasters, software buses, ...
- allocation structures: intended to map code-based and run-time structures to external items, such as network locations, physical devices, managerial structures ...
- entails the need for architectural views

this course focus on run-time structures

What is software architecture?

Components:	*Loci* of computation and data stores, encapsulating subsets of the system's functionality and/or data; Equipped with run-time interfaces defining their interaction points and restricting access to those subsets; May explicitly define dependencies on their required execution contexts; Typically provide application-specific services
Connectors:	Pathways of interaction between components; Ensure the flow of data and regulates interaction; Typically provide application-independent interaction facilities; Examples: procedure calls, pipes, wrappers, shared data structures, synchronisation barriers, etc.

What is software architecture?

Components:	*Loci* of computation and data stores, encapsulating subsets of the system's functionality and/or data; Equipped with run-time interfaces defining their interaction points and restricting access to those subsets; May explicitly define dependencies on their required execution contexts; Typically provide application-specific services
Connectors:	Pathways of interaction between components; Ensure the flow of data and regulates interaction; Typically provide application-independent interaction facilities; Examples: procedure calls, pipes, wrappers, shared data structures, synchronisation barriers, etc.

What is software architecture?

Configurations:	Specifications of how components and connectors are associated; Examples: relations associating component ports to connector roles, mapping diagrams, etc.
Properties:	Set of non functional properties associated to any architectural element; Examples (for components): availability, location, priority, CPU usage, ... Examples (for connectors): reliability, latency, throughput, ...

What is software architecture?

Configurations:	Specifications of how components and connectors are associated; Examples: relations associating component ports to connector roles, mapping diagrams, etc.
Properties:	Set of non functional properties associated to any architectural element; Examples (for components): availability, location, priority, CPU usage, ... Examples (for connectors): reliability, latency, throughput, ...

What is software architecture?

Keywords

- composition
- interaction

A mix of examples

from the micro level (a Unix shell script)

```
cat invoices | grep january | sort
```

- Application architecture can be understood based on very few rules
- Applications can be composed by non-programmers
- ... a simple architectural concept that can be comprehended and applied by a broad audience

A mix of examples

to the macro level (the WWW architecture)

- a collection of resources with unique names (URL)
- URIs used to determine the identity of a machine on the web
- Communication is initiated by clients (e.g. a web server) who make requests to servers.
- Resources manipulated through representations (e.g. HTML)
- All communication is performed by a simple, generic protocol (HTTP), offering methods, e.g. GET, POST, etc.
- All communication between user agents and servers is fully self-contained

A mix of examples

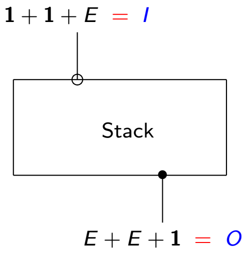
to the macro level (the WWW architecture)

- Architecture is totally separated from the code
- There is no single piece of code that implements the architecture
- There are multiple pieces of code that implement the various components of the architecture (e.g., different browsers)
- One of the most successful applications is only understood adequately from an architectural point of view

A mix of examples

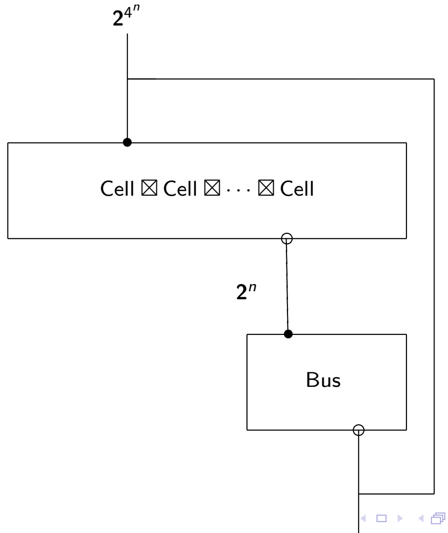
$\boxed{\text{components \& ports (the Stack diagram in a component calculus)}}$

$$\begin{cases} \text{pop}: & \mathbf{1} \longrightarrow E \\ \text{top}: & \mathbf{1} \longrightarrow E \\ \text{push}: & E \longrightarrow \mathbf{1} \end{cases}$$



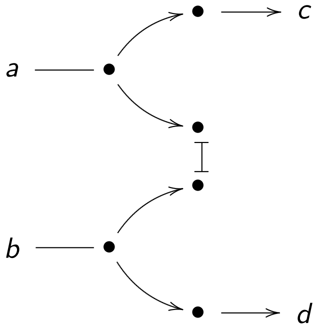
A mix of examples

new components from old (assembling the Game of Life)



A mix of examples

a connector (synchronization barrier in REO)



A mix of examples

a configuration (client-server in ACME)

```
System CS = {
   component client = { port call }
   component server = { port request }
      property max-clients-supported = 10;
   connector rpc = { role plug-cl; role plug-sv}
   }
   attachments = {
      { call to plug-cl ; server to plug-sv }
   }
```

Architectural style (or pattern)

... emerged as a main issue in SA research to ...

- classify families of software architectures
- act as types for configurations
- provide
  - domain-specific design vocabulary (eg, set of connector and component types admissible)
  - a set of constraints to single out which configurations are well-formed. Eg, a pipeline architecture might constraint valid configurations to be linear sequences of pipes and filters.

Examples

- Layers
- Client & Server
- Master & Slave
- Publish & Subscribe
- Peer2Peer
- Pipes and Filters
- Event-bus
- Repositories
  - triggering by transactions: databases
  - triggering by current state: blackboard
- Table-driven (virtual machines)
- ...

Pattern: Layers

- helps to structure applications that can be decomposed into groups of subtasks at different levels of abstraction
- Layer n provides services to layer $n+1$ implementing them through services of the layer $n-1$
- Typically, service requests resort to synchronous procedure calls

Examples:

virtual machines (eg, JVM)
APIs (eg, C standard library on top of Unix system calls)
operating systems (eg, Windows NT microkernel)
networking protocols (eg, ISO OSI 7-layer model; TCP/IP)

Pattern: Client-Server

- permanently active servers supporting multiple clients
- requests typically handled in separate threads
- stateless (session state maintained by the client) vs stateful servers
- interaction by some inter-process communication mechanism

Examples:

remote DB access
web-based applications
interactive shells

Pattern: Peer-2-Peer

- symmetric Client-Service pattern
- peers may change roles dynamically
- services can be implicit (eg, through the use of a data stream)

Examples:

multi-user applications
P2P file sharing

Pattern: Publish-Subscribe

- used to structure distributed systems whose components interact through remote service invocations
- servers publish their capabilities (services + characteristics) to a broker component, which accepts client requests and coordinate communication
- allows dynamic reconfiguration
- requires standardisation of service descriptions through IDL (eg CORBA IDL, .Net, WSDL) or a binary standard (eg, Microsoft OLE — methods are called indirectly using pointers)

Examples:

web services
CORBA (for cooperation among heterogeneous OO systems)

Pattern: Master-Slave

- a master component distributes work load to similar slave components and computes a final result from the results these slaves return
- isolated slaves; no sharing of data
- supports fault-tolerance and parallel computation

Examples:

dependable systems

Pattern: Event-Bus

- event sources publish messages to particular channels on an event bus
- event listeners subscribe to particular channels and are notified of message availability
- asynchronous interaction
- channels can be implicit (eg, using event patterns)
- allows dynamic reconfiguration
- variant of so-called event-driven architectures

Examples:

process monitoring
trading systems

Pattern: Pipe & Filter

- suitable for data stream processing
- each processing step is encapsulated into a filter component
- uniform data format
- no shared state
- concurrent processing is natural

Examples:

compilers
Unix shell commands

Pattern: Blackboard

- suitable for problems with non deterministic solution strategy known
- all components have access to a shared data store
- components feed the blackboard and inspect it for new partial data
- extending the data space is easy, but changing its structure may be hard

Examples:

complex IA problems (eg, planning, machine learning)
complex applications in computing science (eg, speech recognition; computational chemistry)

- Introduction
- Software Architecture
- Architectural Styles
- SA: Evolution & Challenges
- Our Approach to SA

Origins

- Until the 90's, SA was largely an ad hoc affair (but see [Dijkstra,69], [Parnas79], ...)
- Descriptions relied on informal box-and-line diagrams, rarely maintained once the system was built

As a discipline it emerged from

- the recognition of a shared repertoire of methods, techniques and patterns for structuring complex systems
- the quest for reusable frameworks for software development
- the attempt to conceptualise and classify composition/interaction patterns as architectural styles

Origins

- Until the 90's, SA was largely an ad hoc affair
  (but see [Dijkstra,69], [Parnas79], ...)
- Descriptions relied on informal box-and-line diagrams, rarely maintained once the system was built

As a discipline it emerged from

- the recognition of a shared repertoire of methods, techniques and patterns for structuring complex systems
- the quest for reusable frameworks for software development
- the attempt to conceptualise and classify composition/interaction patterns as architectural styles

The last 15 years

- Formal notations for representing and analysing SA: ADL
- Examples: Wright, Rapide, SADL, Darwin, C2, Aesop, Piccola …

ADLs provide:

- conceptual framework + concrete syntax
- tools for parsing, displaying, analysing or simulating architectural descriptions

- ACME [Garlan et al, 97] as an architectural interchange language (a sort of XML for architectural description)
- Use of model-based prototyping tools (eg Z, VDM) or model-checkers (eg Alloy) to analyse architectural descriptions

The last 15 years

- Classification of architectural styles characterising families of SA and acting as types for configurations
- Standardisation efforts: ANSI/IEEE Std 1471-2000, but also 'local' standards (eg, Sun's Enterprise JavaBeans architecture)
- Impact of the emergence of a general purpose (object-oriented) design notation — UML — closer to practitioners and with a direct link to OO implementations
- SA becomes a mature discipline in Software Engineering; new fields include documentation and architectural recovery from legacy code

Current trends

Not only the world of software development, but also the contexts in which software is being used are changing quickly and in significant ways ...
... whose impact on Software Engineering, in general, is still emerging

Which trends can be identified?
In which way are reshaping our understanding of software composition, interaction and architecture?

Current trends

Not only the world of software development, but also the contexts in which software is being used are changing quickly and in significant ways ...
... whose impact on Software Engineering, in general, is still emerging

Which trends can be identified?
In which way are reshaping our understanding of software composition, interaction and architecture?

Current trends

- From object-oriented to component-based development:

  - In OO the architecture is implicit: source code exposes class hierarchies but not the run-time interaction and configuration;
  - Objects are wired at a very low level and the description of the wiring patterns is distributed among them;
  - CBD retains data and code encapsulation, but shifts the emphasis from class inheritance to object composition ...
  - ... to avoid interference between inheritance and encapsulation and pave the way to a development methodology based on third-party assembly of components.

Current trends

- The emergence of service-oriented computing:

  - software is composed of services which reside (and are maintained) on third-party machines/organisations;
  - composition becomes inherently open, dynamic (services can move, to reconfigure themselves, ...) and asynchronous
  - ...
  - subtle change of emphasis: software is understood as a service, rather than as a product.

Current trends

- New business structures in the IT (global) market

for example,

software sub-contracting:
many companies look at themselves more as system integrators than as software developers: the code they write is essentially glue code ...
which entails the need for common frameworks to reduce architectural mismatchs

Current trends

- From programming-in-the-large to programming-in-the-world:

'not only the complexity of building a large application that one needs to deliver, in time and budget, to a client, but of managing an open-ended structure of autonomous components, possibly distributed and highly heterogeneous.
This means developing software components that are autonomous and can be interconnected with other components, software or otherwise, and managing the interconnections themselves as new components may be required to join in and others to be removed.'
(Fiadeiro, 05)

Current trends

Such trends entails a number of challenges to the way we think about SA:

Concurrency is ubiquitous!

- requires applications to be structured as parts of interacting computations ...
- which must coordinate multiple activities, manage communication and handle failure, interrupts and time-outs as they leverage concurrent services;
- the internet paradigm: from a communication service provider to a truly global computer;
- need to handle stringent real-time constraints, e.g. in embedded systems from video games to avionics

Current trends

Interaction as a first-class citizen

- computational units as black boxes *vs* complex interaction patterns
- ... which may be modified dynamically;
- historically, interaction has been influenced by the available hardware (from *semaphores* to *monitors* and *transactions* ... but such mechanisms do not scale;
- since the 1980's: major theoretical advancements on calculus for describing interacting behaviours (CCS, CSP, π, ...), envisaging a similar role to that of λ-calculus in sequential computing; inspired new mechanisms in programming languages (e.g. $C\sharp$).

Current trends

Interaction as a first-class citizen

- computational units as black boxes *vs* complex interaction patterns
- ... which may be modified dynamically;
- historically, interaction has been influenced by the available hardware (from *semaphores* to *monitors* and *transactions* ... but such mechanisms do not scale;
- since the 1980's: major theoretical advancements on calculus for describing interacting behaviours (CCS, CSP, π, ...), envisaging a similar role to that of λ-calculus in sequential computing; inspired new mechanisms in programming languages (e.g. $C\sharp$).

Behaviour & Interaction

In the era of global computing, software architecture deals with

... objects, components, processes, services, ...

which emphasises behavioural rather than informational structures and assigns a fundamental role to interaction:

Behaviour & Interaction

[R. Milner, 1997]
Thus software, from being a prescription for how to do something — in Turing's terms a "list of instructions" — becomes much more akin to a description of behaviour, not only programmed on a computer, but occurring by hap or design inside or outside it.

[B. Jacobs, 2005]
The subject of Computer Science is not information processing or symbol manipulation, but generated behaviour.

Behaviour & Interaction

[R. Milner, 1997]
Thus software, from being a prescription for how to do something — in Turing's terms a "list of instructions" — becomes much more akin to a description of behaviour, not only programmed on a computer, but occurring by hap or design inside or outside it.

[B. Jacobs, 2005]
The subject of Computer Science is not information processing or symbol manipulation, but generated behaviour.

Architecture as coordination

Architectural design as a coordination problem

- The main architectural challenge is to coordinate multiple, heterogeneous, distributed, loosely-coupled, autonomous entities with limited access through (often fragmentary) *published interfaces*.
- recall web-service orchestration, choreography, etc.

The scenario:

- a palette of computational units treated as black boxes
- and a canvas into which they can be dropped
- connections are established by drawing wires

Architecture as coordination

The coordination paradigm

The purpose of coordination is to find solutions to the problem of managing the interaction among concurrent programs.
(*F. Arbab, 1998*)

- emerged in the 1980's from the need to exploit the full potential of massively parallel systems, as a solution to the problem of managing interaction among concurrent activities;
- enforces a strict separation between effective computation and its control (components and interactions) and focus on their joint emergent behaviour.

Architecture as coordination

The coordination paradigm

- From *passive, data-driven* coordination: the Linda model (Carriero et al, 1985) and its shared dataspace – a memory ab- straction accessible for data exchanging to all processes cooperating towards the achievement of a common goal,
- to *active, control-driven, exogenous* coordination: Reo (Arbab, 2003) and Orc (Misra, 2004)

Our programme

The coordination paradigm

- Express architectural designs as coordination patterns
- Introduce two complementary perspectives:
  - Orc: focus on action patterns, with ephemeral interaction (in the tradition of process algebra)
  - Reo: focus on (continued) interaction as a first-class citizen
- Emphasise formal models for the key notions: interaction, behaviour, concurrency, building on top of process calculi and automata theory.

Our starting point

SA as studied at MFES (until now):

$\boxed{\text{the architecture of functional designs}}$

Interfaces:	$f :: \cdots \longrightarrow \cdots$
Components:	$f = \cdots$
Connectors:	$\cdot, \langle\ ,\ \rangle, \times, +, \ldots$
Configurations:	functions assembled by composition
Properties:	invariants (pre-, post-conditions)
Behavioural effects:	monads and Kleisli compostion
Underlying maths:	universal algebra and relational calculus

To be extended to

- process-based designs: interfaces are patterns of actions whose transformational meaning is largely ignored; richer composition mechanisms; interaction by action handshake.
- service-based designs: interfaces as sets of ports through which data flows; interaction is anonymous and handled by complex connectors; clear separation between computation and coordination

To be extended to

- process-based designs: interfaces are patterns of actions whose transformational meaning is largely ignored; richer composition mechanisms; interaction by action handshake.
- service-based designs: interfaces as sets of ports through which data flows; interaction is anonymous and handled by complex connectors; clear separation between computation and coordination

Mathematical foundations

- No time in this course to go deep into the semantics of Reo or Orc,
- emphasise, instead, the key semantical notions and tools able to take into account
    - persistence, i.e., internal state and state transitions
    - continued interaction along the whole computational process
    - potential infinite behaviour
    - equational and inequational reasoning in terms of observational preorders

... transition systems, bisimulation, coinduction ...

Mathematical foundations

- No time in this course to go deep into the semantics of Reo or Orc,
- emphasise, instead, the key semantical notions and tools able to take into account
  - persistence, i.e., internal state and state transitions
  - continued interaction along the whole computational process
  - potential infinite behaviour
  - equational and inequational reasoning in terms of observational preorders

... transition systems, bisimulation, coinduction ...

Syllabus

1. Introduction to Software Architecture
2. Foundations: behaviours, transition systems and coinduction
3. Architecture as coordination: the Reo perspective
4. Architecture as coordination: the Orc perspective