

Software architecture for reactive systems (introduction)

Luís S. Barbosa

HASLab - INESC TEC
Universidade do Minho
Braga, Portugal

27 February, 2013

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

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.

[Garlan, 2003]

a bridge between requirements and code (...) a blueprint for implementation.

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?

What is software architecture?

Properties

- functional vs non functional
(e.g. performance, reliability, dependability, portability, scalability, interoperability ...)
- structural vs behavioural

What is software architecture?

Which structure? \rightsquigarrow Architectural views

- **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** ...

This course

- focus on **run-time structures**
- and entails a particular **view**: **components & glue**

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?

Constraints:

Represent claims about an architectural design that should remain true even as it evolves over time. Typical constraints include restrictions on allowable values of properties, topology, and design vocabulary. For example, **the number of clients of a particular server is less than some maximum value.**

Styles:

Styles represent **families of related systems**. A style defines a vocabulary of design element types and rules for composing them. Examples include dataflow architectures based on pipes and filters, blackboard architectures based on shared data space and a set of knowledge sources, and layered systems.

Two 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

Two 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

Architectural styles (or patterns)

- 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)

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

Challenges

- recognition of a shared **repertoire** of methods, techniques and patterns for structuring complex systems
- quest for **reusable frameworks** for the development of product families

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

- **Software sub-contracting**: many companies look at themselves more as system **integrators** than as software **developers**:

the code they write is **glue** code ...

which entails the need for common frameworks to reduce **architectural mismatches**

Current trends

- **Software as a service**
- From **object-oriented** to **component-based** development:

- In OO the architecture is **implicit**: source code exposes **class hierarchies** but not the **run-time interaction**
- Objects are wired at a very low level and the description of the wiring patterns is distributed among them
- CBD retains the basic encapsulation of **data** and **code** principle to increase modularity 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

- 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)

Challenges

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

- new **target**: need for an architectural discipline for **reactive systems**
(often **complex**, **time critical**, **mobile**, etc ...)
- from **composition** to **coordination** (orchestration)
- relevance of **wrappers** and component **adapters**: integration vs incompatible assumptions about component interaction
- continued **interaction** as a first-class citizen and the main form of software composition

Our approach

There is no **general-purpose, universally tailored**, approach to architectural design ...

Therefore, the course

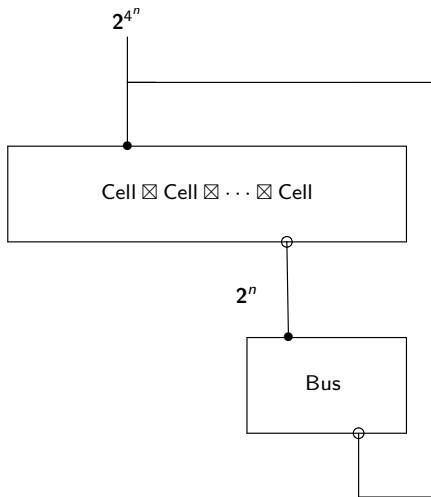
- introduces 3 different approaches
- addressed from both a **foundational and methodological perspective**
- with (reasonable) **tool support** for modelling and analysis

Syllabus

- Introduction to software architecture
- **Component-oriented architectural design**
 - **Paradigm:** Software components as monadic Mealy machines
 - **Foundations:** Coalgebra theory as a semantic framework for state-based systems a component calculus
 - **Method:** The MMM calculus; prototyping in HASKELL
- **Process-oriented architectural design**
 - **Paradigm:** Overview of process-oriented ADLs
 - **Foundations:** Interactive Markov chains
 - **Method:** Specification and analysis of architectures with stochastic constraints
- **Coordination-oriented architectural design**
 - **Paradigm:** The REO exogenous coordination model
 - **Foundations:** Constraint automata and variants
 - **Method:** Compositional specification of the glue layer

Component-oriented architectural design

new components from old in MMM



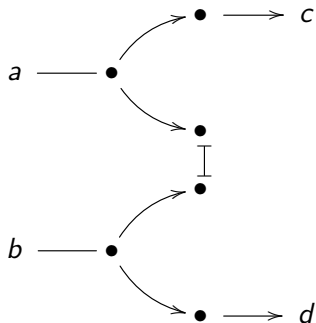
Process-oriented architectural design

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 }  
}
```


Coordination-oriented architectural design

a connector (synchronization barrier) in REO



Pragmatics ...

- **References:** (tba)
- **Assessment:** (tba, but essentially **continuous**)
- **Research context:** Projects
 - MONDRIAN — 2011-13
on **foundations for architectural design**
 - QAIS — 2012-14
on **coalgebraic quantitative reasoning**
 - NASONI — 2012-15
on **heterogenous software coordination**
(**continuous vs discrete systems**)

GRANTS available!

(with U. Nijmegen, U. Aveiro, CWI, INESC TEC)