

**A Button-Led systems:
from objects to actors and services**

DISI-Cesena
edited by Antonio Natali

Alma Mater Studiorum – University of Bologna
via Venezia 52, 47023 Cesena, Italy

Table of Contents

A	Button-Led systems: from objects to actors and services	1
	<i>DISI-Cesena edited by Antonio Natali</i>	
1	Requirements	2
1.1	Goals	2
1.2	Software life cycle process	2
1.3	Bottom-up or Top-Down?	3
2	Technology-based design	4
2.1	Function-based software	4
2.2	Event-driven servless applications: AWS Lambda	4
2.3	Lexical Closures	5
2.4	Objects	6
2.5	Observable and Observers	6
2.6	Models	7
2.7	Modelling a Button	7
3	An object-based design	8
3.1	A Led as an object in Javascript	9
3.2	From BLS Mock devices to physical devices	10
3.2.1	Dependency injection and Inversion of Control	10
3.3	From a BLS local system to a BLS distributed	10
4	Evolving the BLS into a distributed, event-driven system	11
4.1	The BLS System in Vert.x	11
5	Evolving the BLS into a RESTful system	12
5.1	Spring	12
5.2	BLS using Spring	13
5.3	Running	13
6	Architectures	15
6.1	Layered Architectures	15
6.2	Hexagonal Architectures	15
7	Evolving the BLS into a message-based, distributed heterogeneous system	16
7.1	Technology-dependent application code (Bottom-up)	16
7.2	(Domain) business related application code (Top-down)	16
7.3	A custom meta-model (based on MOF)	16
7.3.1	qaakka18	17
7.4	A qa model for the BLS	17
7.5	Interaction MQTT-based	18
8	Devices as microservices	20

1 Requirements

Every **IOT** system usually performs a basic set of actions:

- Acquire data from sensor devices.
- Perform some control action.
- Send commands to actuator devices.

In this very basic demo, we use a **Button** as a **sensor** and a **Led** as an **actuator** and the control action represents our business logic. Examples of the business logic implemented by our Button-Led (**BLS**) system are:

1. **ROnOff**: the Led is **turned on/off** each time the Button is pressed.
2. **RBlink**: when the Button is pressed, the Led **starts blinking**. When the Button is pressed again, the Led **stop blinking**. And so on.
3. ...

Reference project: [it.unibo.bls.oo](https://github.com/anatali/IotUniboDemo) on <https://github.com/anatali/IotUniboDemo>.

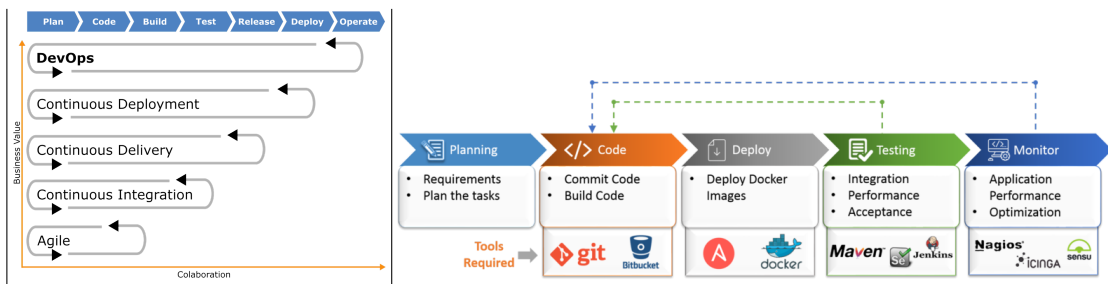
1.1 Goals

Our goals can be summarized as follows:

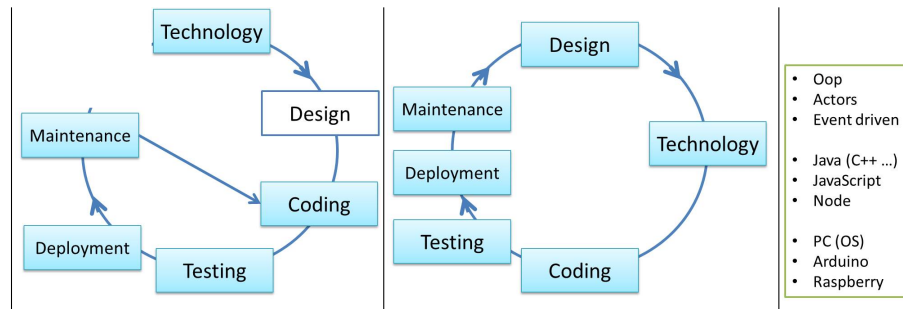
1. Define a 'technology-independent' architecture/prototype of the BLS in a local, **tightly coupled** environment.
2. Specialize the initial prototype according to different technologies. For example:
 - (a) The devices are implemented as **Mock** objects in a virtual environment.
 - (b) The devices are concrete things controlled by low-costs devices such as **Arduino/RaspberryPi**.
3. Modify the first working prototype into a **loosely-coupled** (distributed) system in which each device works within its own computational node: see Subsection 3.3.
4. Modify the distributed prototype by 'transforming' each device into a (**RESTful**) **service**.
5. ...

1.2 Software life cycle process

Discussed in **BBS-module5 : Software production**.



1.3 Bottom-up or Top-Down?



The function defined at `natAwsServices` named `blsOnOffEdge` has the following code:

```
1 var state = 'off'; //Initialization (state of a Led Mock)
2
3 exports.handler = (event, context, callback) => {
4   console.log('Received event:', JSON.stringify(event, null, 2));
5   console.log('edge =', event.edge);
6   if ('edge' in event) {
7     state = event['edge'] ;
8   }
9   var ledState = 'Led is ' + state ;
10  console.log(ledState); //Logging to Amazon CloudWatch Log
11  callback(null, ledState); // End and return
12};
```

The function can be called by the AWS Console on the Web in two ways

- without any testing event
- with a testing event expressed in Json:

```
1 {
2   "edge": "off"
3 }
```

As an alternative, we can call the function from a *AWS Command Line Interface* (**AWS-CLI**) Installed on our PC. Before doing that, we must configure a (client) user on the PC by using the command `aws configure` (in `C:/Program Files/Amazon/AWSCLI/bin`):

```
1 AWS Access Key ID [None]: ...BCG4A
2 AWS Secret Access Key [None]:...T8eW
3 Default region name [None]: eu-west-1
4 Default output format [None]: json
```

To find the Access Key (ID):

1. go to the security credentials on your account page (Click on your name in the top right corner -> My security credentials;
2. generate access keys over there and use those access keys in your credentials file (`aws configure`)

Finally we can invoke the function via the **AWS-CLI**:

```
1 aws lambda invoke --function-name blsOnOffEdge --payload '{"edge": "on"}' output.txt
2 aws lambda invoke --function-name blsOnOffEdge --payload '{"edge": "off"}' output.txt
```

in this case the function is executed via the **AWS Lambda Invoke API** call.

An important point at this stage, is that we can experiment in an easy way some fundamental aspects of functional programming (in JavaScript/Node), including the concept of lexical closure (`lambda` in Java8) .

2.3 Lexical Closures

Operationally, a **closure** is a record storing a function together with an environment: a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.

An example is the function named `ClosureExample`:

```
1 function ledClosure(state){
2   var doSwitch = function( ){
3     state = ! state;
4     return state
5   }
6 }
```

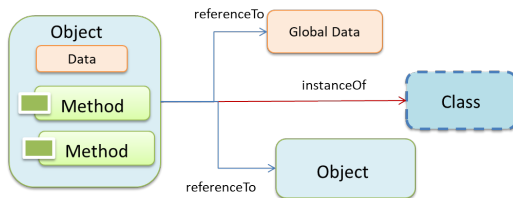
```

6   return doSwitch
7   }
8
9   var ledSwitch = ledClosure(false);
10
11  exports.handler = (event, context, callback) => {
12    console.log( ledSwitch() );
13    console.log( ledSwitch() );
14    console.log( ledSwitch() );
15    callback(null, "done"); // End and return
16  };

```

2.4 Objects

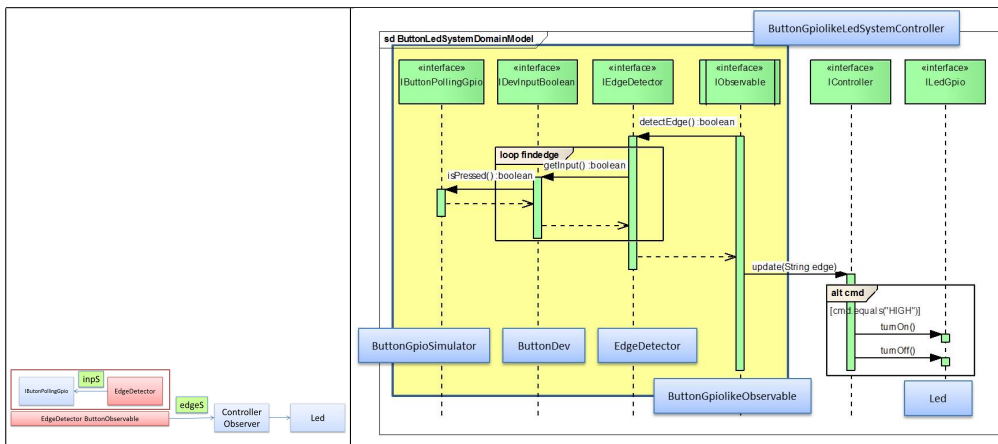
As an alternative of the function-based code of Subsection 2.1, both the *EdgeDetector* and the *Controller* can be modelled as finite state machines (FSM) working as *transducers*. They can be viewed as *objects* interacting via procedure-calls.



In any object-oriented model, all the computation usually takes place within a single thread. In our case the main thread could be the thread related to the component that performs the polling of the wave, i.e. the *EdgeDetector*. In this case, the *Controller* is called by the *EdgeDetector* that, must explicitly know the *Controller* in order to call it.

2.5 Observable and Observers

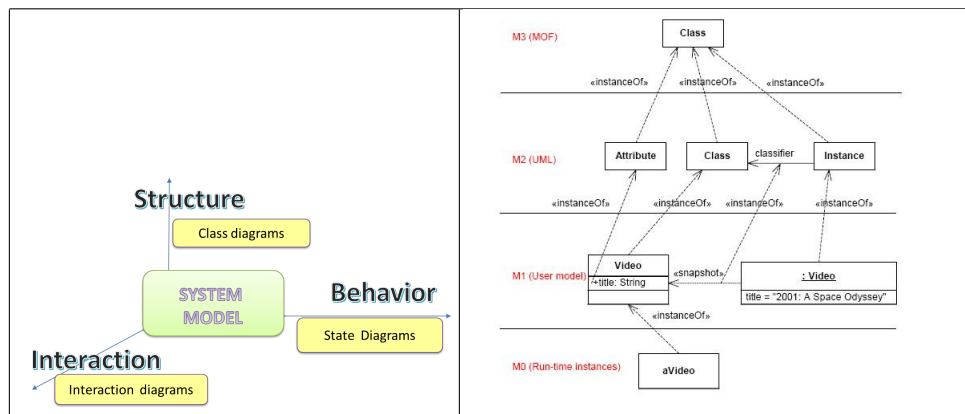
A more flexible architecture can be obtained (without changing the run-time interaction pattern) by conceiving the *Controller* as an *observer* that can be registered to the *EdgeDetector* information source.



However, starting from the idea that a Button is an 'edge detector' device is a too low-level approach for modern software applications. An effort has to be made to introduce a more appropriate **model** of the Button entity in terms of structure, interaction and behavior.

2.6 Models

Nel contesto dei processi di costruzione del software, il termine **modello** va primariamente inteso come un insieme di concetti e proprietà volti a catturare **aspetti essenziali** di un sistema, collocandosi in un preciso **spazio concettuale**. Per l'ingegnere del software quindi un modello costituisce una visione semplificata ma **coerente, consistente ed efficace** di un sistema che rende il sistema stesso più accessibile alla **comprensione e alla valutazione** e **facilita il trasferimento di informazione** e la collaborazione tra persone, soprattutto quando è espresso in forma visuale.



2.7 Modelling a Button

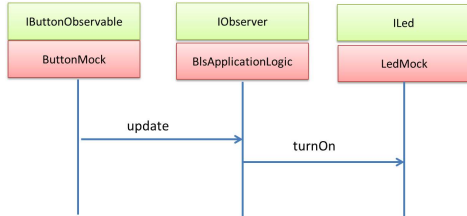
From the **structural** point of view, a button is intended by a customer as an **atomic** entity whose **behavior** can be modelled as a **finite state machine (FMS)** composed of two states ('pressed' and 'unpressed'). The state transition is performed by some agent *external* to the system (an user, a program, a device, etc.). From the **interaction** point of view, the button can expose its internal state in different ways:

- by providing a **property** operation (e.g. `boolean isPressed()`) that returns `true` when the button is in the **pressed** state. In this way the interaction is based on "**polling**";
- by providing a **synchronizing** operation (e.g. `void waitPressed()`) that blocks a caller until the button transits in the **pressed** state. In this way the interaction is based conventional "**procedure-call**";
- by working as an **observable** according to the *Observer* design pattern. In this way the interaction is based on "**inversion of control**" and involves observers (also called "*listeners*") that must be explicitly referenced (via a "*register*" operation) by the button.
- by emitting **events** handled by an event-based support. In this way the interaction is based on "inversion of control" that involves observers (usually known as "*callbacks*") referenced by the support and not by the button itself.
- by sending **messages** handled by a message-based support. In this way the interaction is based on message passing and can follow different "patterns" (in our internal terminology we distinguish between *dispatch*, *invitation*, *request-response*, etc.)

All these "models" could be appropriate in some software application. Thus, a very useful exercise is to define in a formal way each of these models by adopting (at the moment) a test-driven approach.

3 An object-based design

Working in the conceptual space of 'classical' object oriented software development, the logic architecture of the BLS can be summarized by the following UML interaction diagram:



The computation starts from the observable device (*Button*), that calls a method of the object devoted to implement the application logic (that works as the *Observer*). An example is given in the project `it.unibo.bls.oo` that is based on the Java language. The working directory of the project is structured as follows:

- The package `it.unibo.bls.interfaces` includes the definition of the object interfaces.
- The package `it.unibo.bls.devices` includes the implementation of the object interfaces related to the devices (*Button* and *Led*). For each device, two different implementations are given: a *Mock* device and a 'virtual' object implemented with a GUI.
- The package `it.unibo.bls.applLogic` includes the definition of the object that implements the application logic.
- The package `it.unibo.bls.appl` includes the *Main* programs.
- The `test` directory includes examples of test units.

A software system working with the *Mock* devices should include a **configuration phase** to create the system components (objects) and properly connect them, according to the system architecture design:

```
1 public class MainBlsMockBase {
2     private IButton btn;
3     private ILed led;
4     //Factory method
5     public static MainBlsMockBase createTheSystem(){
6         return new MainBlsMockBase();
7     }
8     protected MainBlsMockBase( ) {
9         createComponents();
10    }
11    protected void createComponents(){
12        led = LedMock.createLed( );
13        BlsApplicationLogic applLogic = new BlsApplicationLogic(led);
14        btn = ButtonMock.createButton( applLogic );
15        led.turnOff();
16    }
}
```

Listing 1.1. MainBlsMockBase.java: configuration

The system defines also some working activity, to cause the change of the state in the *ButtonMock*:

```
1 public void doSomeJob() {
2     System.out.println("doSomeJob starts" );
3     ((ButtonMock)btn).press();
4     Utils.delay(1000);
5     ((ButtonMock)btn).press();
6     System.out.println("doSomeJob ends" );
7 }
```

Listing 1.2. MainBlsMockBase.java: simulated action

Since every system is a **composed** (i.e. it is a **non-atomic**) entity, it provides also **selector-methods** to get its components:

```
1 public IButton getButton(){ //introduced for testing
2     return btn;
3 }
4 public ILed getLed(){ //introduced for testing
5     return led;
6 }
```

Listing 1.3. MainBlsMockBase.java: selectors

The selectors can be useful during the testing, to access the state of the single devices. Finally, there is the **main** method:

```
1 public static void main(String[] args) {
2     MainBlsMockBase sys = createTheSystem();
3     sys.doSomeJob();
4 }
5 }
```

Listing 1.4. MainBlsMockBase.java: the main method

3.1 A Led as an object in Javascript

Let us model a Led as an object in JavaScript

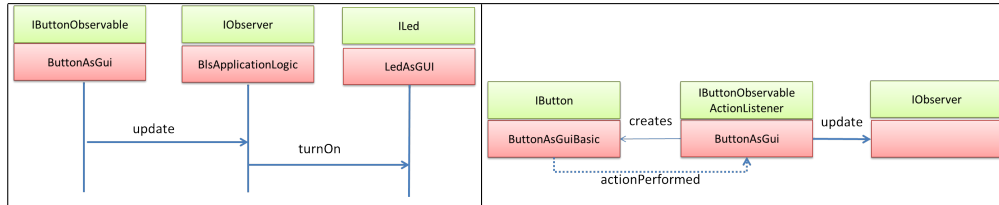
```
1 //State (specific)
2 var Led = function(name, guid){
3     //Led Constructor: instance data
4     this.name = name;
5     this.guid = guid;
6     this.ledState = false;
7 }
8
9 //Methods (shared)
10 Led.prototype.turnOn = function(){
11     this.ledState = true;
12 }
13 Led.prototype.turnOff = function(){
14     this.ledState = false;
15 }
16 Led.prototype.switchState = function(){
17     this.ledState = ! this.ledState;
18 }
19 Led.prototype.getState = function(){
20     return this.ledState;
21 }
22 Led.prototype.getName = function(){
23     return this.name;
24 }
25 Led.prototype.getDefaultRep = function(){
26     return this.name+"||"+ this.ledState
27 }
```

The function defined at natAwsServices named **blsR0n0ff** has the following code:

```
1 //Initialization
2 var ledMod = require("./Led");
3 var l1 = new ledMod.Led('l1', null);
4 //Entry
5 exports.handler = (event, context, callback) => {
6     console.log('Received event:', JSON.stringify(event, null, 2));
7     console.log('edge =', event.edge);
8     if ('edge' in event) { //change the led state at each edge
9         l1.switchState();
10    }
11    console.log( l1.getDefaultRep() ); //Logging to Amazon CloudWatch Log
12    callback(null, l1.getDefaultRep()); // End and return
13 };
```

3.2 From BLS Mock devices to physical devices

The logical architecture previously introduced does not change if we replace the **Mock** devices with concrete devices: For example, in the case of virtual devices implemented with a GUI:



The class `ButtonAsGuiBasic` implements a GUI-based Button by extending the class `java.awt.Button`. The class `ButtonAsGui` implements the concept of Button as Observable entity.

Note that `ButtonAsGui` re-uses the class `ButtonAsGuiBasic` but without exploiting inheritance. Rather, it creates an instance of `ButtonAsGuiBasic` and works as its listener. This behaviour is caused by the fact that Java does not support multiple inheritance for classes and `ButtonAsGui` already extends the class `java.util.Observable`.

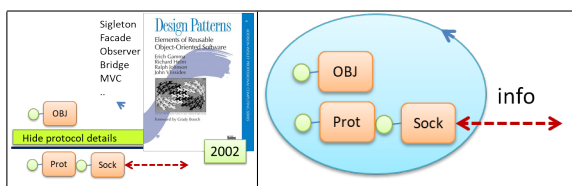
In a more general perspective, we could build this part of our software system by exploiting the **Dependency Injection** pattern.

3.2.1 Dependency injection and Inversion of Control .

In software engineering, an **injection** is the passing of a dependency to a dependent object (a client) that would use it as a service. In the **dependency injection** pattern, passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement.

Dependency injection is one form of the broader technique of **Inversion of Control**. The client delegates the responsibility of providing its dependencies to external code (the **injector**). The client is not allowed to call the injector code.

3.3 From a BLS local system to a BLS distributed



4 Evolving the BLS into a distributed, event-driven system

The author of section is [Angelo Croatti](#).

Project [it.unibo.bls.oo/it.unibo.bls.vertx](https://github.com/anatali/IotUniboDemo) on <https://github.com/anatali/IotUniboDemo>

This example is an evolution of the BLS system, using the Vert.x Framework (<https://vertx.io/>) and proposed abstractions - in terms of Event Loop and Event Bus - to evolve the BLS system to a distributed asynchronous system.

As a recall, in computer science, the **Event Loop** (sometimes referred also as Message Loop or Main Loop) is a programming structure that continually tests for external events and calls the appropriate routines to handle them. An **Event Bus** can be used in a distributed system to put in contact two or more Event Loops, dispatching events among them. In the Vert.x Framework, the Event Loop concept is encapsulated by the concept of **Verticle**: each verticle instances communicate with each other by sending messages on the event bus.

The Golden Rule: "Don't Block the Event Loop" - Vert.x APIs are non blocking and won't block the event loop, but the programmer have to avoid to block the event loop itself, e.g., in a handler. If the programmer blocks an event loop in Vertx instance then the event loop will not be able to do anything else while it's blocked. If a "blocking command" (e.g. `Thread.Sleep()`, a long-running calculation, I/O, ...) stops the event loop from doing anything else for a significant amount of time, vertx reacts throwing an exception.

4.1 The BLS System in Vert.x

The BLS system has been realized in Vert.x using two verticles, respectively for defining the Button (see `it.unibo.bls.vertx.ButtonVertx` class) and the Led (see `it.unibo.bls.vertx.LedVertx` class). This two verticles communicates through the Vert.x provided event bus.

First of all, the system needs to deploy each verticle using the Vert.x shared instance of the framework. Calling the `deployVerticle()` method, the vert.x instance starts the deployment of each verticle and, when the deployment process has been completed (as is, the `start()` method of the verticle instance has been called) a callback/handler is called (see method `initTheSystem()` within the `it.unibo.bls.vertx.MainVertxSystem` class).

Each verticle, in its deployment phase (handled by the `start()` method) register itself as a consumer on the event bus calling the method `consumer()` on the event bus instance in vertx (`vertx.eventBus()`). This means that the verticle registers an handler able to process events dispatched on the event bus (by other verticles or by itself) and received to be consumed. Each verticle can send events to the event bus exploiting the `send()` method (or the `publish()` ones) on the event bus instance, specifying the receiver/consumer address.

In this example, each verticle is registered as a consumer on a different address (respectively `"bls.button"` and `"bls.led"`). Anyway, it is also possible to use a "broadcast" address to send/publish events/messages to all verticles, regardless of who they effectively are. Moreover, in current BLS example, events dispatched on the event bus are modelled as JSON object and sent to a well defined verticle (identified by an address).

The application logic of the system is inherited by the previous example, so - as done in that example - the observation feature allow the button verticle to observe the gui button on the proposed GUI and to react to "pressed" event (see the `it.unibo.bls.vertx.BlsVertxControl` class).

5 Evolving the BLS into a RESTful system

The author of section is [Roberto Casadei](#).
Projects [it.unibo.bls.oo/it.unibo.bls.spring.ledService](#),
[it.unibo.bls.oo/it.unibo.bls.spring.buttonService](#)
on <https://github.com/anatali/IotUniboDemo>

A straightforward way to enable seamless distribution of the BLS as well as heterogeneity in the implementations of its components is to adopt a Service-Oriented Architecture (SOA), where the system components become services, with a well-defined interface (contract), that interact using interoperable protocols.

ReST (REpresentational State Transfer) is a particular architectural style for a SOA; it defines some key principles and constraints regulating how services have to be designed, most prominently, the stateless-ness of servers and uniform interface (based on HTTP methods, media types, URIs, manipulation of resources through representations, self-contained messages and [HATEOAS](#)).

5.1 Spring

Spring Boot is a framework that supports the development of RESTful services. It builds on the Spring Framework which, at its core, is an IoC framework; the idea is to declare "managed components" (called "beans" in Spring) through Java annotations or XML, wire those beans in other components (through field, constructor, or setter injection), and then let the framework deal with the injection of components and management of their lifecycle.

In order to enable Spring Boot in a Gradle project, you need to extend your build definition (build.gradle) with:

```
1 buildscript { // This block is e.g. for adding dependencies *to the build itself*
2     repositories {
3         mavenCentral()
4     }
5     dependencies {
6         classpath("org.springframework.boot:spring-boot-gradle-plugin:2.0.2.RELEASE")
7     }
8 }
9
10 apply plugin: 'org.springframework.boot'
11 apply plugin: 'io.spring.dependency-management'
12
13 sourceSets {
14     main { resources { srcDir 'resources' } } // Declares directory for resources
15 }
16
17 dependencies {
18     // .....
19     compile group: 'org.springframework.boot', name: 'spring-boot-starter-web'
20 }
21
22 bootJar {
23     mainClassName = 'it.unibo.bls.spring.buttonService.MainButtonService'
24     baseName = 'unibo-button'
25     version = '1.0.0'
26 }
```

The task `bootJar` builds a Fat JAR (i.e., the application code + all the dependencies) under `/build/libs`; the idea (cf., microservices) is to have a single artefact (also containing the webserver!) that can be easily deployed. (Unfortunately you can only define one "boot JAR"; this is coherent with the microservices philosophy, where each service gets its own repository and build!)

5.2 BLS using Spring

An example of the BLS using Spring is in package `it.unibo.bls.spring`: two services are defined for the button and the led, in sub-packages ‘button’ and ‘led’, resp. Each service consists of:

- A main class, annotated with ‘`@SpringBootApplication`’, which is the same as having ‘`@Configuration`’ (this class may declare beans), ‘`@EnableAutoConfiguration`’ (attempts to guess and configure beans that you are likely to need by analysing the classpath), ‘`@ComponentScan`’ (registers beans by looking for classes annotated with ‘`@Component`’ in this package and subpackages), that starts the Spring application for the service (i.e., we give control to Spring for managing the declared components).
- A resource file, under `/resources/application-xxx.yml`, defining some configuration for the service (e.g., the Spring application name, and the port to which the server should bind).
- A REST controller class (cf., MVC pattern), annotated with ‘`@RestController`’, which defines the (RESTful) HTTP interface for the service by annotating methods with ‘`@RequestMapping(value='uri/path')`’.
- A managed component class, annotated with ‘`@Component`’, that appropriately wraps the component (e.g., the button or the led).

Considers, for instance, the implementation of ‘`LedController`’

```
1 @RestController @RequestMapping(value = "/led")
2 public class LedController {
3     @Autowired private ILed led;
4
5     @RequestMapping(value = "/state")
6     public Boolean state(){
7         return led.getState();
8     }
9
10    @RequestMapping(value = "/turnOn")
11    public void turnOn(){
12        led.turnOn();
13    }
14
15    @RequestMapping(value = "/turnOff")
16    public void turnOff(){
17        led.turnOff();
18    }
19 }
```

Notice the use of annotation ‘`@Autowired`’ to enable field injection with a bean conforming interface `ILed`. Such bean is defined as follows:

```
1 @Component
2 public class LedComponent extends LedMock { }
```

i.e., it is a candidate for auto-detection when using annotation-based configuration and classpath scanning. Notice that by default the “scope” of a component is “singleton” (i.e., one instance per container); other scopes are e.g. “request” (per HTTP request) and “session” (per HTTP session).

5.3 Running

When the Spring application is started, the framework will start an embedded server (cf., application containers with WAR deployment), binding at the configured port, and will use the declared controllers to map HTTP requests to methods. From the point of view

of concurrency, it uses the Servlet model, i.e., one thread per request. Notice that the ‘ButtonComponent’, which internally reuses ‘BlsApplicationLogic’, is prone to race conditions (cf., [Isolation property](#) of [ACID](#)).

In order to run the system, launch the following main classes:

- ‘MainButtonService’ (NOTE: you also need to pass the JVM option ‘-Djava.awt.headless=false’)
- ‘MainLedService’

Then, you can press the button by performing a GET request at <http://localhost:8080/button/press>, and inspect the state of the led by performing a GET request at <http://localhost:8081/led/state>.

6 Architectures

Possible definition for the word 'Architecture' in the domain of software:

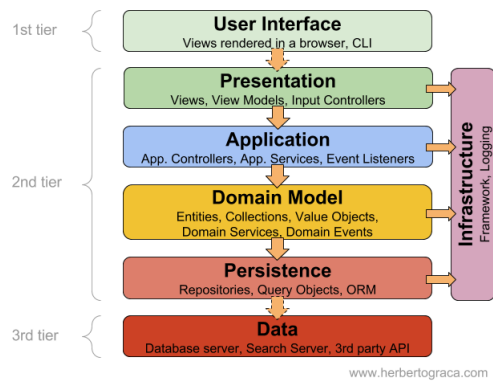
Open Group Architectural Framework: *"a set of elements depicted in an architectural model and a specification of how these elements are connected to meet the overall requirements of an information system"*.

Others say that: *"the software architecture of a program or computing system is the structure or structures of the system, which comprises software components, the externally-visible properties of these components and the relationships among them"*.

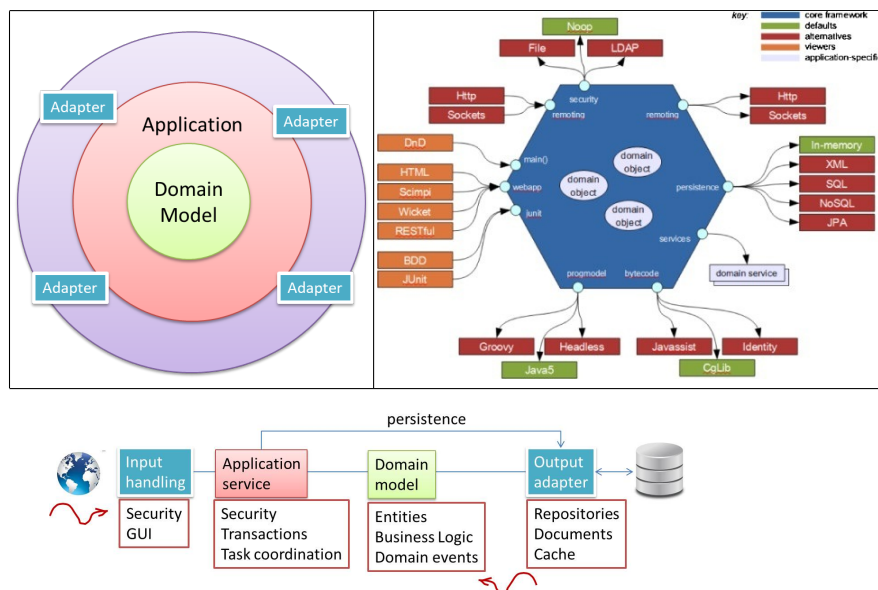
IEEE Computer society: (2000): *"the fundamental organization of a system embodied in its components their relationships to each other and to the environment, and the principles guiding its design and evaluation"*.

We can also say that: *L'architettura è ciò che rimane di un sistema quando non si può più togliere nulla, continuando a comprenderne la struttura e il funzionamento.*

6.1 Layered Architectures

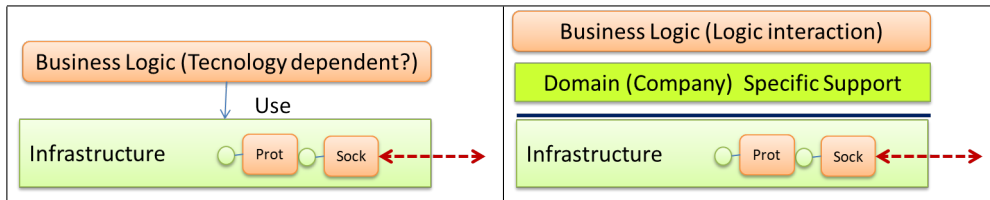


6.2 Hexagonal Architectures

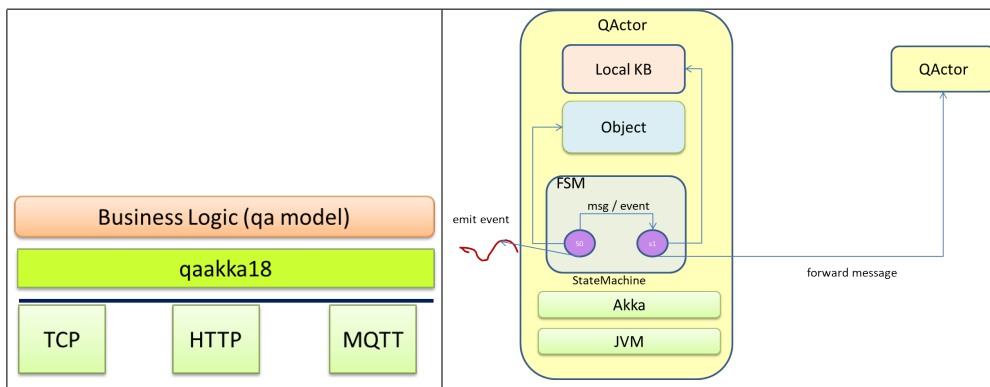


7 Evolving the BLS into a message-based, distributed heterogeneous system

7.1 Technology-dependent application code (Bottom-up)



7.2 (Domain) business related application code (Top-down)

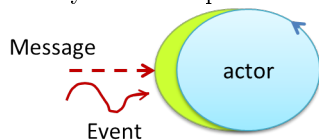


7.3 A custom meta-model (based on MOF)

From Fowler MDS:

Model Driven Software Development (MDS) is a style of software development that considers itself as an alternative to the traditional style of programming. The approach centers itself on building models of a software system. These models are typically made manifest through diagrammatic design notations. The idea is that you use these diagrams, to specify your system to a modelling tool and then you generate code in a conventional programming language.

QActor is the name given to a custom meta-model inspired to the actor model (as can be found in the **Akka** library). The **qa** language is a custom language that can allow us to express in a concise way the structure, the interaction and also the behaviour of (distributed) software systems composed of a set of **QActor**.



The actor can handle messages/events:

- 1) as a FSM
- 2) as a reactor

Events can be emitted by *asynchronous operations*

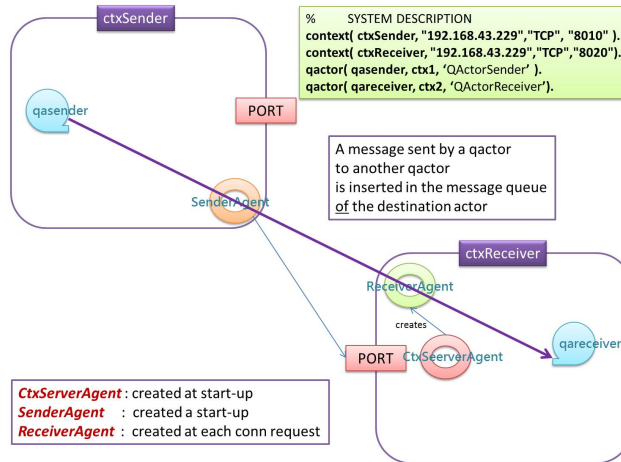
- write on a file
- send a request
- ...

The QActor language/metamodel is associated to a *software factory* that automatically generates the proper system run-time support (including system configuration code) so to allow Application designers to focus on application logic. The QActor software factory is implemented as a set of Eclipse plug-ins, built by using the XText framework.

The leading *Q/q* means 'quasi' since the QActor meta-model and the *qa* language do introduce (with respect to Akka) their own peculiarities.

7.3.1 qaakka18 .

The custom library provided by the file `qa18Akka.jar` defines the run-time support for a QActor system.



For each *Context*, the QActor software factory generates an Akka actor-system and a **SystemCreationActor** that work as the 'father' of all the other actors in that *Context*. More specifically, this actor creates:

- a Akka actor of class **EventLoopActor**
- a Akka actor of class **CtxServerAgent**
- a Akka actor for each QActor defined in the context

The **CtxServerAgent** allows messages exchanged among QActor working on different contexts to flow throw the context ports (using the TCP/IP protocol) and to deliver the message in the message-queue of the destination QActor.

For each connection, an Akka actor of class **ReceiverAgent** is created. This actor handles the message sent on that connection, by sending a *message* to the destination actor and an *event* to the **EventLoopActor**.

7.4 A qa model for the BLS

From the project `it.unibo.bls.qactor`:

```
1 System blsQactor
2 Event usercmd : usercmd(CMD) //from web gui
3 Dispatch turn : turn(X) // X = on/off
4
5
6
7 Context ctxBlsQactor ip [ host="localhost" port=8019 ] -httpserver
8 QActor buttonqa context ctxBlsQactor -g blue{
```

```

9   Plan init normal[
10     delay 1000;
11     println("buttonqa EMITS");
12     emit usercmd : usercmd("pressed");
13     delay 1000;
14     println("buttonqa EMITS");
15     emit usercmd : usercmd("pressed")
16   ]
17 }
18 QActor controlqa context ctxBlsQactor -g yellow{
19   Plan init normal[
20     println("controlqa STARTS")
21   ]
22   transition stopAfter 600000
23   whenEvent usercmd -> pressedOdd
24   Plan pressedOdd [
25     printCurrentEvent;
26     forward ledqa -m turn : turn(on)
27   ]
28   transition stopAfter 600000
29   whenEvent usercmd -> pressedEven
30   Plan pressedEven[
31     forward ledqa -m turn : turn(off)
32   ]
33   switchTo init
34 }
35 QActor ledqa context ctxBlsQactor -g green{
36   Plan init normal[
37     println("ledqa STARTS")
38   ]
39   transition stopAfter 600000
40   whenMsg turn : turn(X) do println (turnTheLed(X) )
41   finally repeatPlan
42 }

```

Listing 1.5. /blsQactor.qa

The model is executable. The code is generated by a custom software factory.

7.5 Interaction MQTT-based

```

1  System blsQactorMqtt
2  Event usercmd : usercmd(CMD) //from web gui
3  Dispatch turn : turn(X) // X = on/off
4
5  pubSubServer "tcp://localhost:1883" //for testing
6
7  Context ctxBlsQactorMqtt ip [ host="localhost" port=8019 ]
8
9  QActor buttonqamqtt context ctxBlsQactorMqtt -pubsub{
10   Plan init normal[
11     delay 1000;
12     println("buttonqa EMITS");
13     emit usercmd : usercmd("pressed");
14     delay 1000;
15     println("buttonqa EMITS");
16     emit usercmd : usercmd("pressed")
17   ]
18 }
19 QActor controlqamqtt context ctxBlsQactorMqtt -pubsub{
20   Plan init normal[
21     println("controlqa STARTS")
22   ]
23   transition stopAfter 5000
24   whenEvent usercmd -> pressedOdd
25
26   Plan pressedOdd [
27     forward ledqamqtt -m turn : turn(on)
28   ]
29   transition stopAfter 5000
30   whenEvent usercmd -> pressedEven

```

```
31
32   Plan pressedEven[
33     forward ledqamqtt -m turn : turn(off)
34   ]
35   switchTo init
36 }
37 QActor ledqamqtt context ctxBlsQactorMqtt -pubsub{
38   Plan init normal[
39     println("ledqa STARTS")
40   ]
41   transition stopAfter 5000
42   whenMsg turn : turn(X) do println (turnTheLed(X) )
43   finally repeatPlan
44 }
```

Listing 1.6. /blsQactorMqtt.qa

8 Devices as microservices

Deepened in [BBS-module7 : Web and Mobile](#).