**Carlo Ghezzi** DEEPSE Group DEI, Politecnico di Milano.
**Matteo Pradella** DEEPSE Group CNR IEIIT-MI.
**Guido Salvaneschi** DEEPSE Group DEI, Politecnico di Milano.

# PROGRAMMING LANGUAGE SUPPORT TO CONTEXT-AWARE ADAPTATION — A CASE STUDY WITH ERLANG

# OUTLINE

- Context Oriented Programming
- Erlang and OTP.
- ContextErlang
  - Dynamic Variation Activation
  - Variation Composition
  - Variation Transmission
- Implementation details
- Performance Evaluation
- Future work and open problems.

# CONTEXT ADAPTATION

- Context:
  - Temperature, weather, spatial/geographic position, bandwidth, computational power availability, protocols, other interacting software components…
  - More that “external conditions”.

- Need for an application to adapt to a changing context.

- Context depending behaviors typically crosscut the system functionalities.
  - Managing context-depending elements in a systematic and effective way is challenging.

# CONTEXT ORIENTED PROGRAMMING

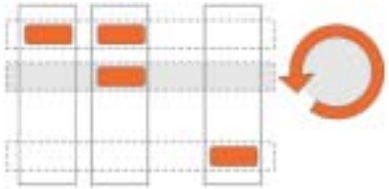
- A contribution to the issue of context depending behaviors
- Language-level support for context management

- Several proposes, mainly from: Software Architecture Group, Hasso Plattner Institut, Potsdam and Palo Alto, prof. Robert Hirschfeld

  - ContextL – Lisp
  - AmOS – Lisp
  - ContextPy – Python
  - ContextJS – Javascript
  - ContextG - Groovy
  - ContextR - Ruby
  - ContextS - Smalltalk
  - ContextScheme – Scheme
  - ContextJ – Java (library-based)
  - JCop – Java (compiler-based)




# CONTEXT ORIENTED PROGRAMMING



- Behavioral variations.
  - Variations typically consist of new or modified behaviors.
  - E.g. partial definitions of modules in the underlying programming model such as procedures or classes
  - Complete definitions represent a rare case.
- Layers.
  - Group related context-dependent behavioral variations.
  - First-class entities, they can be explicitly referred to.
- Activation.
  - Layers aggregating context-dependent behavioral variations can be activated and deactivated dynamically at runtime.
  - Code can enable or disable layers depending on the current context.

# CONTEXT ORIENTED PROGRAMMING

```
class Person {
private String name, address;
private Employer employer;

Person(String newName, String newAddress,  Employer newEmployer){
   this.name = newName;
    this.employer = newEmployer;
    this.address = newAddress;
}

String  toString() {return "Name: “ + name;}

layer Address {

   String toString() {
      return proceed() + "; Address: “ + address;}
}

layer Employment {

   String toString() {
      return proceed() + "; [Employer] “ + employer;}
}

}
```

```
class Employer {
   private String name, address;

Employer(String newName, String newAddress) {
   this.name = newName;
   this.employer = newAddress;
}

String toString() {return "Name: “ + name;}

layer Address {
   String toString() {
   return proceed() + "; Address: “ + address;}
} }
```

```
Employer vub = new Employer("VUB", "1050 Brussel");
Person somePerson = new Person("Pascal Costanza", "1000 Brussel", vub);

with (Address) {
   with (Employment) {
      System.out.println(somePerson);
   }
}

Output: Name: Pascal Costanza; Address: 1000 Brussel;
[Employer] Name: VUB; Address: 1050 Brussel
```

# ERLANG

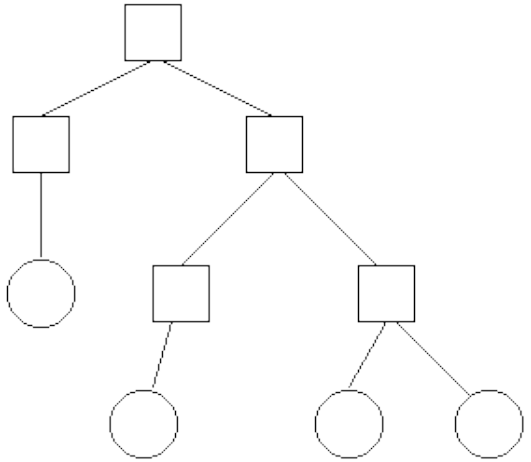
- General-purpose functional language and concurrent runtime system
  - Ericsson Computer Science Lab. (1986), open source from 1998
  - Single assignment, dynamic typing, no Objects.

- Many features such as concurrent processes, scheduling, memory management, distribution or networking are commonly associated to an operating system rather than to a language.
  - Concurrency: lightweight  processes: no shared memory, asynchronous message passing.
  - Distribution: processes on different nodes communicate in the same way as local processes.
  - Fault-tolerance: history of high availability communication systems.
  - Hot code upgrade: program code can be changed in a running system without downtime.

# THE OTP PLATFORM

- The OTP (Open Telecom Platform) is a set of libraries and procedures used for implementing fault-tolerant, large-scale, distributed applications.
  - Practically any *real* Erlang application.
  - “Set of principles for how to structure Erlang code in terms of processes, modules and directories”
  - Bridging the gap between coding and the application architecture.

- Supervision tree:
  - Base of fault-tolerance
  - Hierarchical arrangement of code into supervisors and workers.

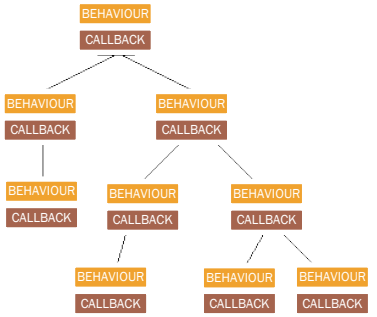


# ERLANG/OTP: BEHAVIORS

- Many processes have similar structure, e.g servers, FSM, event handlers or supervisors.
- Process code modularization:
  - Generic part: **behaviour** mod.
  - Specific part: **callback** mod.

BEHAVIOUR

CALLBACK




- OTP Behaviors implement this pattern.
  - gen _server, gen_fsm, gen_event, supervisor
  - The user only implements the callback module
  - The callback module exports a pre-defined set of functions..

# GEN_SERVER

```
-module(gen_server).
...
loop(Mod, State) ->
    receive
        {call, From, Request} ->
            {Response, State2} =
                    Mod:handle_call(Request, State),
                From ! {Mod, Response},
                loop(Mod, State2);
            {cast, Request} ->
                State2 = Mod:handle_cast(Request, State),
                loop(Mod, State2)
end.
...
```

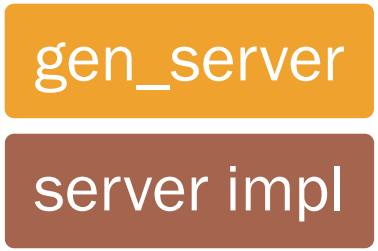
```
-module(server).

%% API functions
alloc(Item, Key) ->
    gen_server:call(?MODULE, {alloc, Item, Key}).
free(Key) ->
    gen_server:call(?MODULE, {free, Key}).
...
%% Callback functions
handle_call({alloc, Item, Key}, State) ->
    alloc(Item, Key, State),
    ...
handle_cast({free, Key}, State) ->
    free(Key, State),
    ...
...
```

gen_server

server

# COP IN ERLANG: CONTEXTERLANG

- The functionalities associated to the generic module, are (probably) not influenced by a context change.
  - message management
  - error handling
  - fault tolerance.




- The callback module implements the specific functionalities of the component, and it is directly influenced by a context change

# CONTEXTERLANG

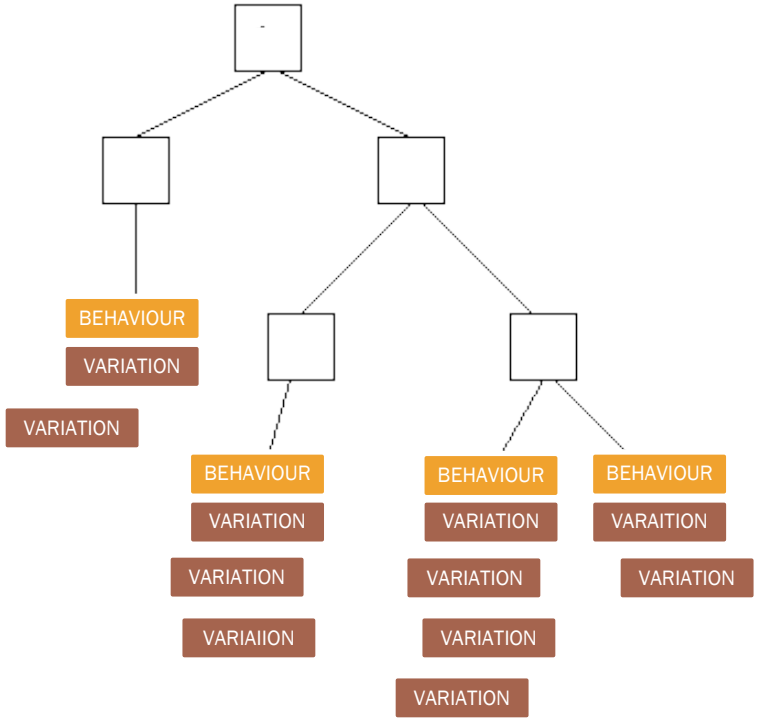
- ContextErlang application:
  - A set of components each made of a behavior module and several callback modules.
  - Each component maps on (at least) a process at run time.

- We call each callback module a *variation*.
  - It implements a set of behavioral changes for the component
  - It is bound with the behavior module on-the-fly.

# UNCHANGED ARCHITECTURAL STRUCTURE

- The behaviors and all the whole supervisor tree structure, i.e. the general architecture of the Erlang/OTP application does not change during the execution.

- What changes accordingly to the context is only the lower part of each process , which is referenced on-the-fly.

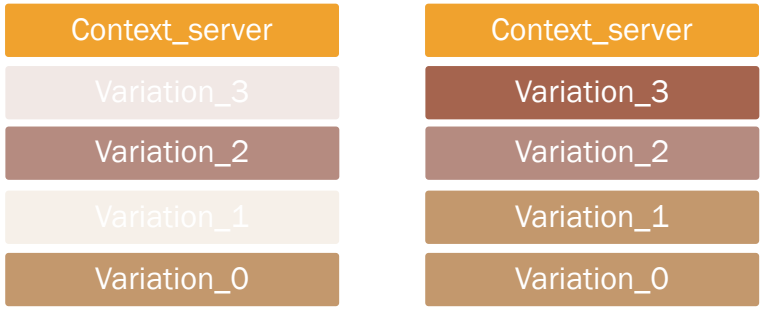


# DYNAMIC VARIATION ACTIVATION

- The a set of partial definitions starts affecting the program behavior.
- In COP can be achieved in various ways, e.g. :
  - Dynamically changing the delegation relationship between objects
  - Changing inheritance relationships of classes
  - In ContextErlang the behavior keep trace of which the set of callback modules which are eligible to receive a function call.

- **`change_context([…])`** must be called on a context-enabled process.

  - All the variations that must be active from that moment onwards are explicitly stated.



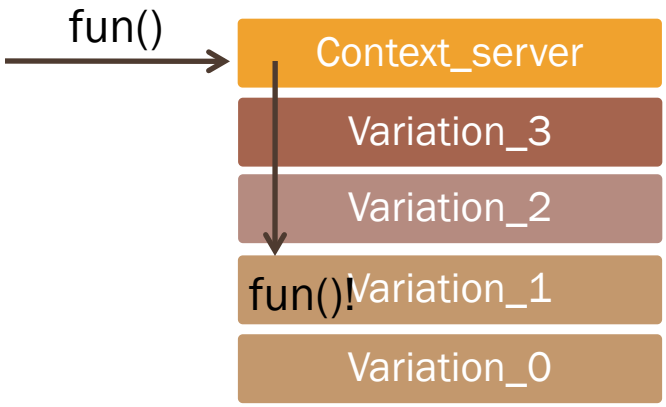
# DYNAMIC VARIATION ACTIVATION

```
-module(variation0).
-behavior(context_gen_server)
...
handle_call({funA}, State) ->
        io:format("[variation0]:
        funA executed~n"),
        Reply = ok,
        {reply, Reply, State};
handle_call({funB}, State) ->
        io:format("[variation0]:
        funB executed~n"),
        Reply = ok,
{reply, Reply, State};
...
```

```
-module(variation2).
...
handle_call({funA}, State) ->
        io:format("[variation2]:
        funA executed~n"),
        Reply = ok,
        {reply, Reply, State};
handle_call({funB}, State) ->
        io:format("[variation2]:
        funB executed~n"),
        Reply = ok,
        {reply, Reply, State};
handle_call({funC}, State) ->
        io:format("[variation2]:
        funC executed~n"),
        Reply = ok,
{reply, Reply, State};
...
```

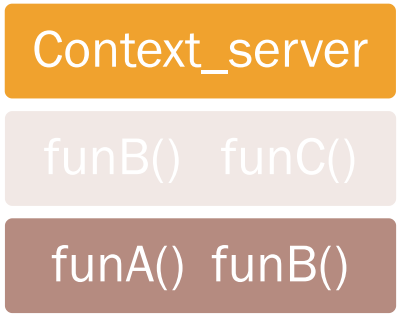
# DYNAMIC VARIATION ACTIVATION

- When the behavior module receives a call request, the corresponding function to be executed is searched inside the set of active variations.
- The composition logic is given by the lookup algorithm inside the behavior module.
  - Search in the top-of-stack variation, if found the call is performed, otherwise go on down along the stack.




- More general variations are arranged at the bottom of the stack
- More specific variations are in the upper part of the stack and have precedence in calls dispatching.

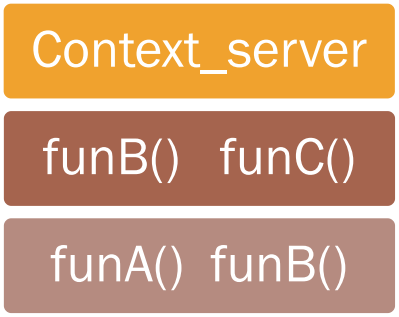
# DYNAMIC VARIATION ACTIVATION




%% In an external context manager
[agent:change_context([variation1])]

agent:funA()
agent:funB()
agent:funC()

$>[variation1]: funA executed.
$>[variation1]: funB executed. (var1)
$>Error: funC not exported



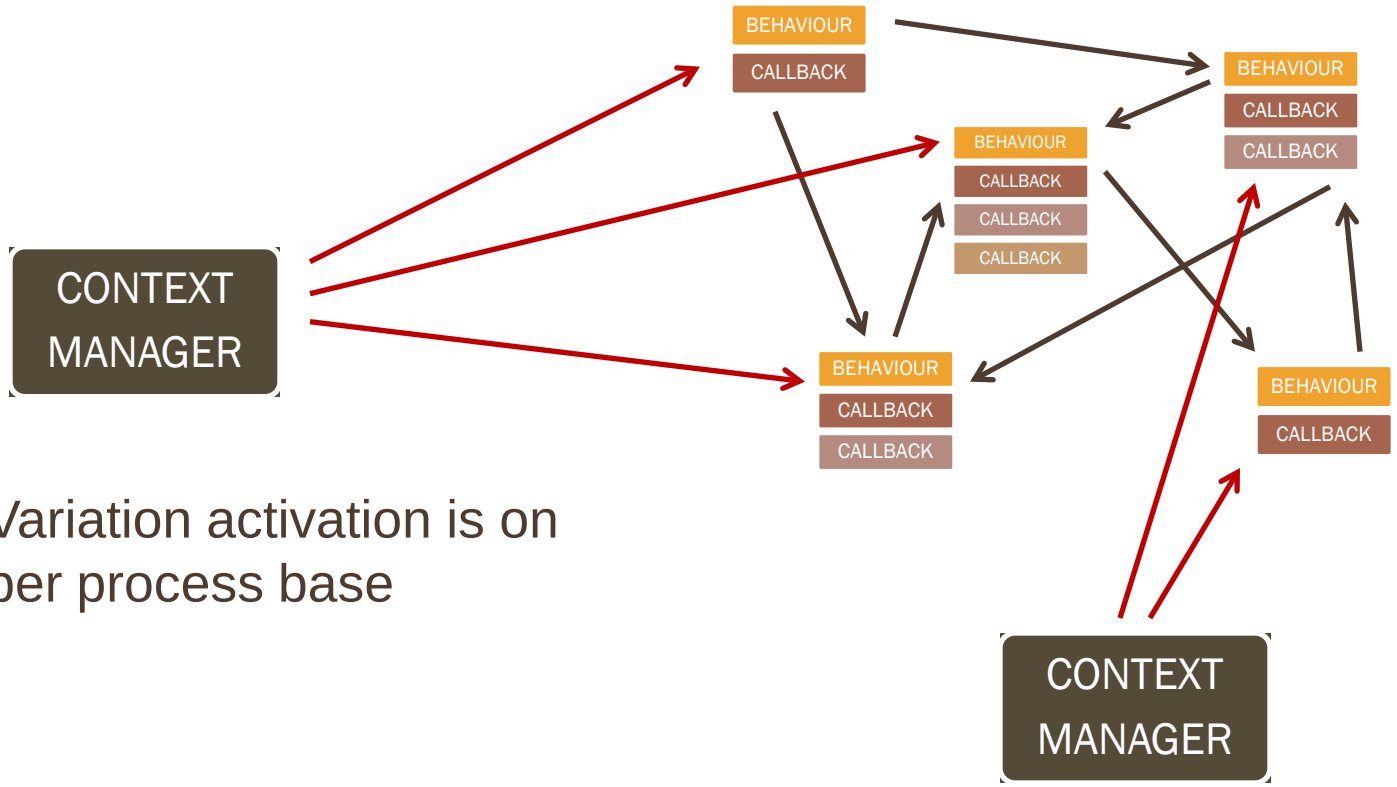

%% In an external context manager
[agent:change_context([variation2, variation1])]

agent:funA()
agent:funB()
agent:funC()

$>[variation1]: funA executed.
$>[variation2]: funB executed. (var2)
$>[variation2]: funC executed.

# DYNAMIC VARIATION ACTIVATION

- Context-change calls are managed by some processes in the system that masters the change of context in the processes that really do the computation.




- Variation activation is on per process base

# VARIATION COMPOSITION

- The context-stack is a simple form of composition: the behavior of a component comes from the arrangement of all the active variations.
- **`proceed()`** calls the subsequent eligible function in the context-stack.
  - If a function f() is implemented in different variations of the stack, only the first one is called.
  - If inside f(), a call to **`proceed()`** is found, the stack is searched for the next variation implementing f() and that implementation is called.

- One can add functionalities to an existing function by simply adding a new variation that wraps the calls to that function.
- Similar to the call to super() in object-oriented languages or to around methods in Common Lisp.

# EXAMPLE (FROM HIRSCHFELD ET AL.)

```
-module(person_base_variation).
...
handle_call({display}, From, State) ->
    io:format("[Person] Name: John; Surname: Smith ~n"),
    Reply = ok,
    {reply, Reply, State};
...
```

Context_server

Employment_var

Person_base_var

person

```
-person:display(),
%An ext. context manager
[person:change_context(
[employment_variation,person_base_variation,person])]
person:display()

$>[Person] Name: John; Surname: Smith
$>
$>[Person] Name: John; Surname: Smith
$>[Employer] Name: aFirm; City: London
```

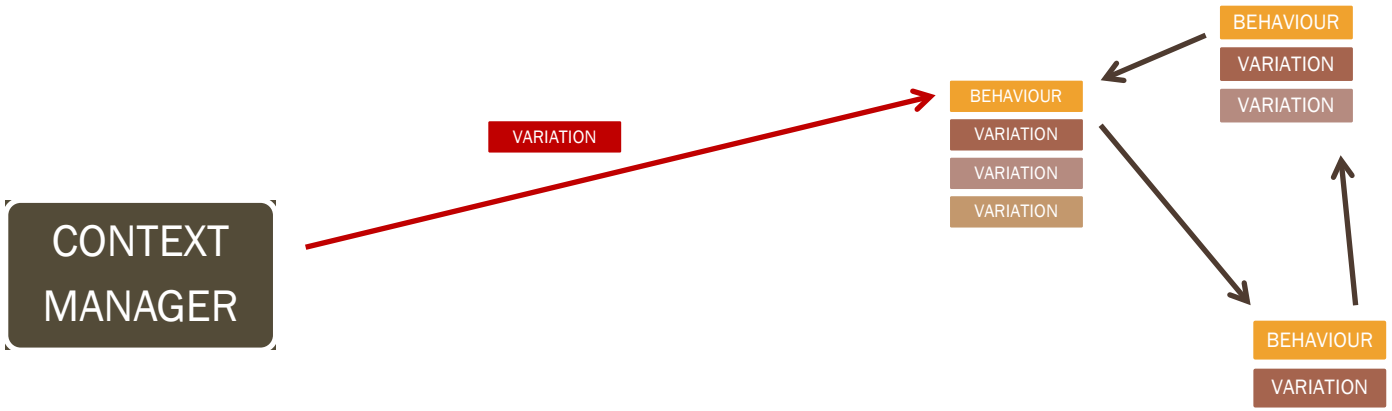
```
-module(employment_variation).
...
handle_call({display}, From, Tab) ->
    proceed(),
    io:format("[Employer] Name: aFirm; City: London ~n"),
    Reply = ok,
    {reply, Reply, Tab}.
...
```

# DYNAMIC VARIATION ACTIVATION

- The stack solution allows a simple set of desirable behaviors.
  - A variation can wrap functions declared in other (more generic) variations by being placed in an upper part of the stack.
  - Function overriding can be obtained implementing the new version of the function in an upper variation.
  - Adding new capabilities to a module is straightforward: activate a variation with new functions inside.

# VARIATION TRANSMISSION

- What if a system must react to a sequence of events that was not planned when the system was built or deployed?
  - The variation can be simply sent by a context manager
  - The variation is loaded on- the-fly on the other node.



# CODE MODULARIZATION

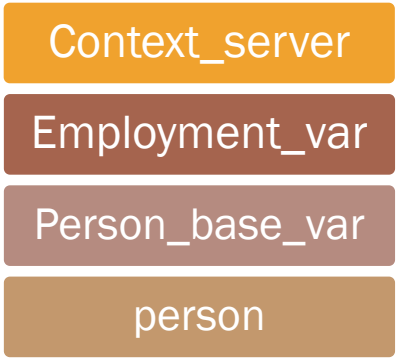
- A well known point addressed by COP is code modularization with respect to contexts.

- Our approach associates each variation to an Erlang module.
  - New variations can be easily added.
  - A module is made of a generic module ad a set of variations, which map on a process at run time.
  - Since processes are the structuring units in Erlang applications and activation is on per process base, modularization is preserved.

# IMPLEMENTATION

- COP extension to the Erlang/OTP platform is obtained with a new OTP behavior module **`context_gen_srv`**.
  - From **`gen_server`**: message management, error handling and fault tolerance.
  - Exposes the APIs for variation activation and composition.
  - The programmer is expected to simply implement the variation modules that can be dynamically activated during the execution.

- **`context_gen_srv`** exposes:
  - **`Proceed()`**
  - **`Change_context(…)`**



# VARIATION TRANSMISSION

- Erlang's dynamic code loading:
  - The binding between a fully-qualified call **`Module:function()`** and the module implementing the call is done at runtime.
  - Variations are modules: they can be dynamically loaded during system execution.

```
context:send_variation(node@host, newVariation)
rpc:call(node@host,component,change_context,[newVar,groudVar])
```

- Send_variation is something like:

```
{Module, Binary, Fname} = code:get_object_code(aModule)
rpc:call(p@teta, code, load_binary, [Module, Fname, Binary]).
```

# IMPLEMENTATION

- None of the modules implementing the variations has a privileged role.
  - In principle all the behaviors of the component can be put inside dynamically activated variations.
  - One variation must contain init() which is expected to initialize the process.
  - Is probably a good practice to keep almost one variation unchanged.
    - This is the natural place for the init() function.
    - Functions that do not change.

# PERFORMANCE

- A context_gen_server receiving a function call:
  - Five active variations, only the last one implementing the required function.
  - Comparison with a call to an OTP generic server with a single callback module implementing the function.
- Overhead introduced by the proceed() call.
  - Five active variations;
  - When the top one is called it calls proceed() on the context_gen_server which dispatches the call to the next variation.
  - The process is repeated up to the call to the last variation.
  - Compared with the same five variations simply calling each other.

| | OTP | | ContextErlang | |
|---|---|---|---|---|
| | Mean | Median | Mean | Median |
| Call Test | 16 | 18 | 34 | 32 |
| Proceed Test | 8 | 8 | 19 | 18 |

microsec

# FUTURE WORK

- Further develop ContextErlang covering all the OTP behaviors.
  - Context-aware event handlers, context-aware FSM…
- “Context” is often referred to user-related issues, such as the mouse position or the look and feel customization.
  - An interesting point is if COP applies well also for server-side software.
  - We are working on a chat server developed in ContextErlang.
- How to obtain in practice contextual information? Interface to a database, to sensors, ecc..