

# An Evaluation of the Adaptation Capabilities in Programming Languages

Carlo Ghezzi  
Matteo Pradella  
Guido Salvaneschi



Dipartimento di Elettronica e Informazione  
Politecnico di Milano, Italy

# Outline

- Introduction
- Approach to the evaluation
- Evaluation of language features
- Conclusion
- Future work

# Self-adaptive Software

- Architectures
  - Component-oriented design, middlewares.
- Design patterns.
  - *State* pattern.
  - Ad-hoc patterns / variants of GoF patterns.
- *Special* language paradigms
  - AOP + behavioral reflection.
  - Dynamic AOP.
  - Context-oriented programming.

# And traditional languages ?

- Which features are useful for self-adaptation?
- An evaluation is not easy
  - Setting up a conceptual framework is one of the goals.

# Languages

- Erlang, Python, Ruby, Scala, F#.
  - Classes, objects, interfaces, traits, modules, multiple inheritance, parametric and ad hoc polymorphism, dynamic dispatching, first class functions, closures, algebraic data types, pattern matching, eval functions...
  - Procedural, object-oriented, and functional styles.
  - Static and dynamic typing.

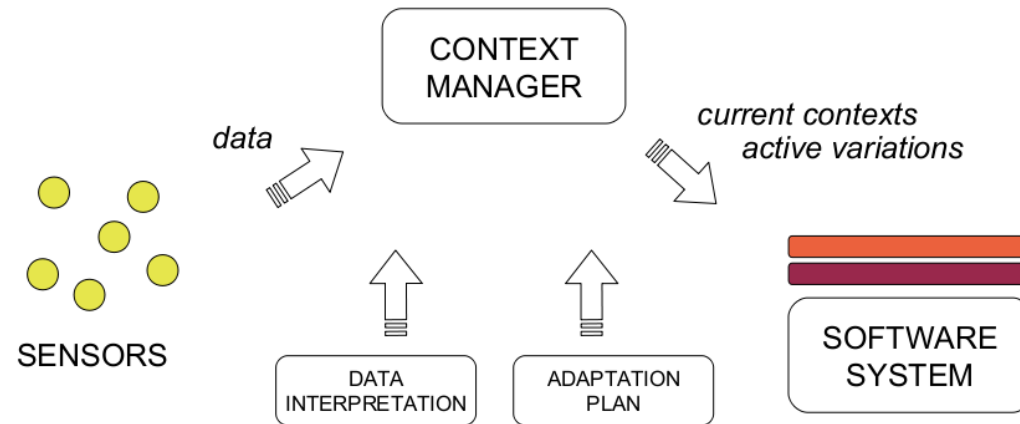
 **Scala**  
ERLANG**F#**

# Cache/storage Server Prototypes

- **Memory Constraint:** reduce RAM consumption.
- **Response Time Constraint:** minimize disk access
- **Low Bandwidth:** minimize bandwidth.
- **Privacy:** resources are ciphered.
- **Backup:** resources are saved on a persistent storage.
- **Security:** functionalities depends on client authentication.
- **Logging:** debug mode.

12 student teams (thanks!)

# Reference Model



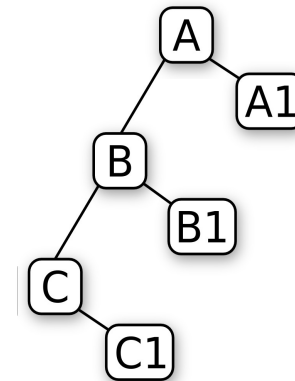
- Observed data are associated to *contexts*
  - E.g. 10 Mb/sec → *low\_band* context
- Contexts trigger *behavioral variations*

# Analysis Framework: Directions

- **Abstraction.**  
Which language abstraction implements the variations?
- **Generic code.**  
How to make adaptation locally “transparent”? Dynamically select proper behavior without scattered `if` chains.
- **Variation combination.**  
More than one variation can be active. How they are combined?
- **Variation activation.**  
How variations are activated to affect the system behavior?
- **Modularization.**  
How variations are organized in the code base?
- **Safety.**  
How the occurrence of context-adaptation errors is avoided?

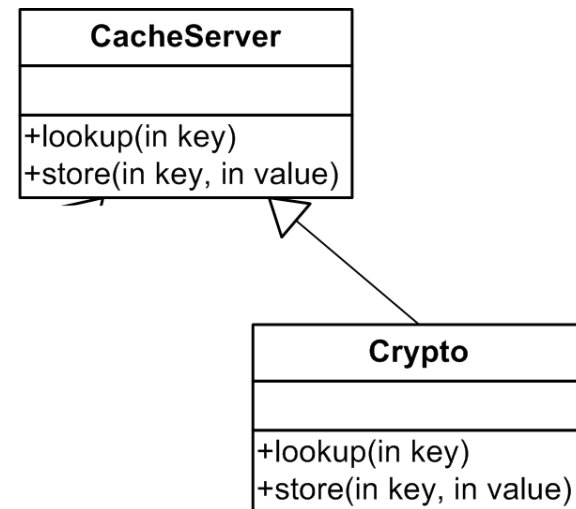
# Inheritance and Subtyping

- Classes are central in OO languages
  - Let's use them for the implementation of variations!
  - **Basic behavior** in class **A**,  
**variation** in class **A1**
    - A1 inherits from A.
    - *Super allows composition.*



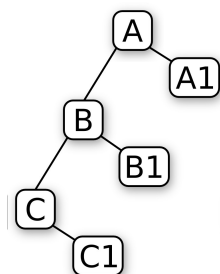
# Inheritance and Subtyping

- **Abstraction.** Classes.
- **Generic code.** Reference Polymorphism. The programmer uses CacheServer or Crypto servers transparently.
- **Variation combination.** *super* calls
- **Variation activation.** Instantiation of the the variation, instead of the base class. (Adaptation is only at instantiation time).
- **Modularization.** Classes.
- **Safety.** Static type systems assure safety of calls.

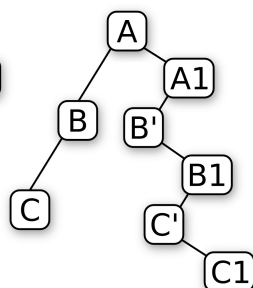


# Inheritance and Subtyping

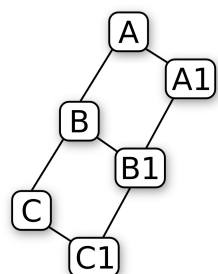
- Need for multiple inheritance:



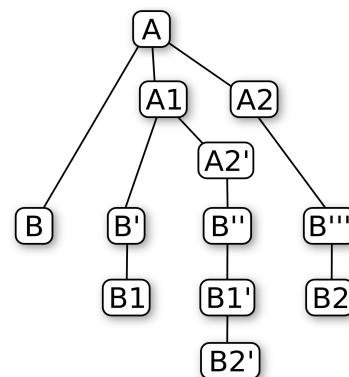
(1)



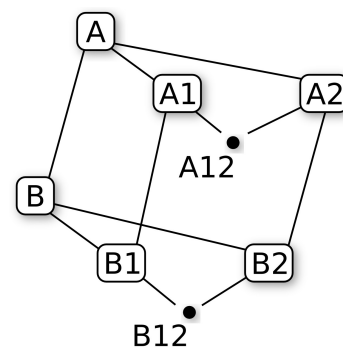
(2)



(3)



(4)



(5)

# First Class Functions

- Functions implement basic behavior or variations.
- Assign the proper function to the reference only once.
  - Avoid selecting the right implementation at each occurrence using `if` chains.

```

type Context =
  DebugContext
  | SafeContext
  | SimpleContext

```

```

let variationCombinator activeContext =

```

```

let loggingVar(key, value) =
  // logging...
  (key, value)

```

```

let cryptoVar(key, value) =
  // do encryption ...
  (key, "cryptoed" + value)

```

```

let backupVar(key, value) =
  // send to backup server ...
  (key, value)

```

```

let basicStore(key, value) =
  // save in cache ...

```

```

match activeContext with
| DebugContext -> loggingVar >> basicStore   □ □
| SafeContext -> backupVar >> cryptoVar >> basicStore   □ □ □
| SimpleContext -> basicStore   □

```

```

let operateOnCache store =
  // do stuff ...
  store ("key1", "val1") // Generic code
  // do stuff ...

```

```

// Real world: context asked to a ContextManager
operateOnCache (variationCombinator DebugContext)
operateOnCache (variationCombinator SafeContext)
operateOnCache (variationCombinator SimpleContext)

```

- **Abstractions.** Functions.
- **Generic code.** Function references.
- **Variation combination.** Combination of functions. E.g. `base(var2(var1(par)))`.
- **Variation activation.** Binding function bodies to function references.
- **Modularization.** Modules group the functions associated to the same context.
- **Safety.** With static typing, the compiler checks type safety.

# Modules as Values

- First class modules can be passed around.
  - Function calls select the implementation currently bound to the module reference:

```
Mod = ...  
Mod:funcall()
```

- Functionalities of the same context are placed in the same module.

- **Abstraction.** Modules.
- **Generic code.** Module references.
- **Variation combination.** Variation modules can call other variation modules.
- **Variation activation.** By binding module references with module values.
- **Modularization.** Modules.
- **Safety.** With static typing, by the compiler.

```

-module(client).

operate(ActiveContext) ->
  Mod = case ActiveContext of
    save_memory -> low_memory;
    normal -> high_memory;
    debug -> logging
  End,

  Mod:store("key1","val1"),
  % do stuff ...
  Mod:lookup("key1").
  % do stuff ...

```

```

-module(logging).

lookup(Key) ->
  % Logging ...
  high_memory:lookup(Key).

store(Key,Value) ->
  % Logging ...
  high_memory:store(Key,Value).
...

```

```

-module(high_memory).

lookup(Key) ->
  % Perform the lookup.

store(Key,Value) ->
  % Store the value
...

```

```

-module(low_memory).

lookup(Key) ->
  % Lookup value for Key

store(Key,Value) ->
  % Store (Key,Value)
...

```

# Dynamic Object Adaptation

- Objects are dictionaries
  - Signatures → method implementations.
- What about dynamically changing the dictionary?
- This is too fine-grained: change the class!
  - Classes become templates for the adaptation..

- **Abstraction.** Classes.
- **Generic code.** Object References.
- **Variation combination.** Run time combinations via dynamic class definitions.
- **Variation activation.** By assigning classes to objects.
- **Modularization.** Classes
- **Safety.** No type safety

```
class CacheServer(object):
    def store(self, key, value):
        # Store value
```

```
    def lookup(self, key):
        # Lookup value for key
        return value
```

```
    def adapt(self, newClass):
        self.__class__ = newClass
```

```
class BackupServer(CacheServer):
    def store(self, key, value):
        # Backup (key,value)
        CacheServer.store(self, key, value)
```

```
# Generic code
srv = CacheServer()
def run():
    srv.store("key1","val1")
    srv.adapt(BackupServer)
    srv.store("key1","val1")
```

# Context-oriented Programming

- Language abstractions for context adaptation.
  - R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, Journal of Object Technology, 7, (2008).
  - Abstract notion of context makes it convenient for self-adaptive software in general.
- “Validation” of the analysis framework with a paradigm specific for context and self-adaptation.
- Performance ?

```
class CacheServer {
```

```
...
```

```
layer logging{
```

```
  Object lookup(Key key){
    System.out.println("Logging ...");
    return proceed(Key key);
  }
```

```
...
```

```
}
```

```
layer crypto{
```

```
  Object lookup(Key key){
    Object o = proceed(Key key);
    System.out.println("Do decription ...");
    Return o;
  }
```

```
}
```

```
}
```

```
Object lookup(Key key) {
```

```
  System.out.println("Lookup")
```

```
...
```

```
}
```

```
}
```

```
CacheServer cs = new CacheServer();
```

```
with (crypto) {
```

```
  with (logging) {
    cs.lookup(key);
  }
```

```
}
```

```
Logging...
```

```
Do decription...
```

```
Lookup
```

- **Abstraction.** Layers.
- **Generic code.** Inside the `with` statement.
- **Variation combination.** Layers are dynamically combined.
- **Variation activation.** Via `with` statement.
- **Modularization.** Along classes (layer-in-class). Layers can be also defined apart (class-in-layer).
- **Safety.** Compiler + COP extension.

# Performance

- From the prototypes in various languages: microbenchmarks implemented in Python

- Python Vs ContextPy.
  - Python is mainstream
  - All functionalities supported
  - Fast COP implementation

	$t_{pass}$	$t_1$	$t_{10}$	$t_{100}$	$fn$
Inheritance Layers	0.53 5.01	1.33 5.64	1.90 6.76	21.4 26.8	9
First class functions Layers	0.44 4.24	0.75 4.82	1.18 5.26	14.45 18.54	6
Modules as values Layers	0.35 5.44	0.82 6.04	1.42 6.75	18.35 24.08	8
Dynamic obj. adapt. Layers	0.25 1.95	0.47 2.22	0.75 2.44	7.10 8.87	3

- Consistent overhead of COP
  - further motivates the investigation of native approaches.

# Conclusion

- Support given by traditional languages to self-adaptive software is important.
  - Avoid ad-hoc paradigms.
  - Performance
- Language functionalities are taken to the extreme consequences
  - Unintuitive code
- This work is an initial investigation.

# Future Work

- Other abstractions:
  - Predicate dispatching/multimethods, events, agents, ...
- Other paradigms:
  - (Dynamic) aspect-oriented programming, functional-reactive programming, implicit invocation
- Extend / further validate the evaluation framework.



Questions ? ...