Software Composition Paradigms

Sommersemester 2015

Radu Muschevici

Software Engineering Group, Department of Computer Science

TECHNISCHE UNIVERSITÄT DARMSTADT

2015-06-02

Context-Oriented Programming

Context Awareness

Context is the physical, technical and human environment in which a software application is used.

Software…

- needs to be increasingly **aware of the context** in which it runs,
- should **adapt dynamically** to such context,
- to provide a service that **matches client needs** to the best extent possible.

Problem

There is **little explicit support for context awareness** in mainstream programming languages.

We still program this…

(2015)

using the programming models conceived for this…

(1980)

Programming with Context

A new paradigm is needed that puts programmers in the right state of mind to build **dynamically adaptable applications** from the ground up.

Context: Examples

spatial state

position, orientation, movement

location semantics

nearby objects & facilities

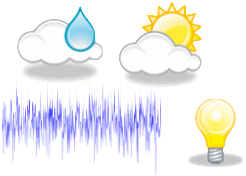
users

expertise, preferences

network peers & services

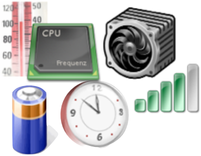
projector, GPS, storage

environmental properties



humidity, light, noise, lighting

internal state



load, time, battery

Context Adaptation Examples

peer service
take advantage of room projector for presentation

location semantics
disable phone ringtone in quiet places

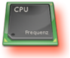
internal state
decrease playback quality when battery power is low

user task
show parking spots and gas stations (only) when driving

environmental conditions
give more detailed indications when visibility is low

Contexts as Situation Reifiers

- Any information that is computationally accessible can be context.
- To act based on context, the meaning of that information needs to be well-defined.

Z axis = 0.03	Landscape orientation	
Battery charge = 220 mAh	Low battery charge	
Idle cycles = 100 MHz	High CPU load	
User agent = "Mozilla/5.0..."	Firefox	
no semantics	**action can be taken**	

Contexts are reified situations
for which adapted application behaviour can be defined.

Manual Context Orientation

Any language can be use to build software that can adapt its behaviour based on dynamically changing context.

Context-independent behaviour

```
class Person {
    String name, address;
    Employer employer;
    void display() {
        println(name); println(address); println(employer);
    }
}
```

Context-dependent behaviour

```
class Person {
    String name, address;
    Employer employer;
    void display(boolean showAddress, boolean showEmployer) {
        println(name);
        if (showAddress) println(address);
        if (showEmployer) println(employer);
    }
}
```

Manual Context Orientation

But manual context adaptation leads to code that is…

- Tangled
- Scattered
- Complex, hard to comprehend
- Difficult to change/refactor/extend/evolve
- Difficult to reuse
- Error-prone

Context-Oriented Programming (COP)

Problem

- Context is increasingly important for many application domains: pervasive and mobile computing, business applications, security, personalisation, internationalisation, …

Solution: Context-Oriented Programming

- Make context an explicit element of the programming language.
- Provide mechanism to dynamically adapt behaviour in reaction to changing context.

COP Languages

- Context-oriented programming languages exist in several shapes and sizes[1].
- ContextJ, ContextL, ContextR, ContextJS, ContextPy, PyContext, ContextG, …

- Example code will be in ContextJ (Java variant)[2]

[1] [Appeltauer, Hirschfeld, Haupt, Lincke, et al. 2009]
[2] [Appeltauer, Hirschfeld, Haupt, and Masuhara 2011]

ContextJ Example

```
class Person {
    private String name, address;
    private Employer employer;
    Person(String name, String address, Employer employer) {
        this.name = name;
        this.address = address;
        this.employer = employer;
    }
    String toString() { return(name); }

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

    layer Employment {
        String toString() {
            return(proceed() + "; Employer: " + employer);
        }
    }
}
```

ContextJ Example (cont.)

```
class Employer {
    private String name, address;
    Employer(String name, String address) {
        this.name = name;
        this.address = address;
    }
    String toString() {
        return(name);
    }

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

COP: Programming with Layers in ContextJ

- Layers add **behavioural variations**[3] to classes.
- Layers are first class language entities.
- Layers are defined within scope of a class,
  thus can access private fields.
- Issue: layer functionality scattered across classes.

[3] [Hirschfeld, Costanza, and Nierstrasz 2008]

Behavioural Variations

- Add, modify or remove behaviour of objects.
- Are expressed as **partial definitions** of methods.
- Example:

```
String toString() {
    return(proceed() + "; Employer: " + employer);
}
```

- Method definition is **partial** because (in general) it is not complete: call to **proceed** requires the existence of a **base** method definition.
- Base method defined outside of layer.

Layers

- Layers represent contexts
- Modularisation mechanism, orthogonal to classes.
- Group related context-dependent behavioural variations.

Dynamic Layer Activation & Deactivation

- **with** and **without** statements
- **with** activates a layer (if required) and moves it to the position of most recently activated.
- **without** deactivates a layer.
- Activation is **dynamically scoped**:

  ```
  with(L) { stmt; without(L) { stmt; } }
  ```

  Layers are only active for the dynamic extent of the **with** block.
- Layer (de)activations can be nested.

Dynamic Layer Activation

```
Person ben = new Person("Ben", "Darmstadt",
                        new Employer("Lufthansa", "Frankfurt"));
```

```
with (Address) { println(ben); }
// Ben, Address: Darmstadt
```

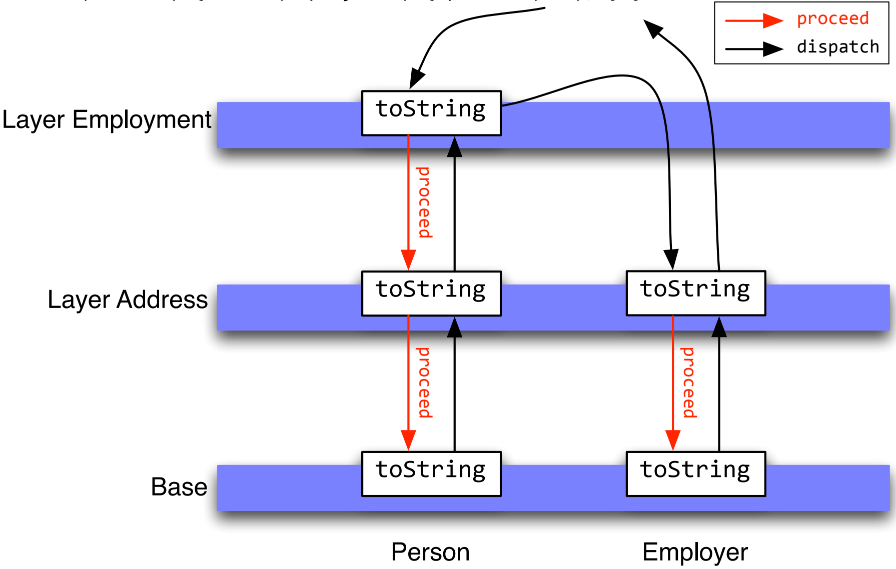
```
with (Employment) { println(ben); }
// Ben; Employer: Lufthansa
```

```
with (Employment) { with (Address) { println(ben); } }
// Ben; Employer: Lufthansa, Address: Frankfurt, Address: Darmstadt
```

```
with (Address) { with (Employment) { println(ben); } }
// Ben, Address: Darmstadt; Employer: Lufthansa, Address: Frankfurt
```

Calling Sequence

```
with (Address) { with (Employment) { println(ben); } }
```



Calling Sequence (cont.)

- The given layer order determines the order in which partial methods definitions are traversed: The most recently activated layer is first in the **proceed** chain.
- A partial method definition **proceed**s to a corresponding partial method in the next-in-order active layer, or to the base method.

```
with(Address, Employer) {
   println(ben);
}
```

proceed chain: `Employer` → `Address` → base method definition

```
with(Employer) { with(Address) {
   println(ben);
} }
```

proceed chain: `Address` → `Employer` → base method definition

Dynamic Scoping

- Layers change the semantics of message dispatch for a given dynamic scope.
- The scope in which layers are active or inactive can be controlled explicitly.
- The same behavioural variation may be simultaneously active and inactive within different scopes of the same running application.

Dynamic Scoping (cont.)

Behavioural variation of class `Person` in layer `Address`:

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

Create a `Person` object:

```
Person ben = new Person("Ben", "Darmstadt",
                        new Employer("Lufthansa", "Frankfurt"));
```

Behavioural variation is active:

```
with (Address) { println(ben); }
// Ben, Address: Darmstadt
```

Inactive:

```
without (Address) { println(ben); }
// Ben
```

Multi-dimensional Method Dispatch

One-dimensional dispatch In procedural programming, procedure calls are dispatched only based on the procedure's **name**.

Two-dimensional dispatch Object-oriented programming dispatches messages based on message **name** and the **receiver** object.

Three-dimensional dispatch Subjective programming[4] dispatches messages based on message **name**, the **receiver** and the message **sender**.

Four-dimensional dispatch Context-oriented Programming adds a fourth message dispatch dimension: the **context** of the message send.

[4] [Smith and Ungar 1996]

Code Modularisation: Layer Declaration Strategies

Layer inside Class

```
class C1 {
    layer L1 {...}
    layer L2 {...}
}
class C2 {
    layer L1 {...}
    layer L2 {...}
}
```

- Classes are completely specified
- Layers scattered across classes
- Can access privates

Class inside Layer

```
layer L1 {
    class C1 {...}
    class C2 {...}
}
layer L2 {
    class C1 {...}
    class C2 {...}
}
```

- Layers are completely specified
- Class is scattered across layers
- Should privates be accessible?

Code Modularisation (cont.)

Layer-in-Class vs. Class-in-Layer. What is better for…

- Code analysis, readability?
- Code evolution & extension?

- Some COP languages (e.g. ContextL) allow both
  $\Rightarrow$ programmer decides
- Alternative: Independent layer-class fragments

Stateful Layers

- Problem: Fields used in layer-specific behaviour need to be declared and can be accessed outside the layer.

```
class Person {
    private String name, address;
    private Employer employer;
    Person(String name, String address, Employer employer) {
        this.name = name;
        this.address = address;
        this.employer = employer;
    }

    layer Address {
        public String getAddress() { return address; }
    }
    layer Employer {
        public Employer getEmployer() { return employer; }
    }
}
```

- Some languages (e.g. ContextL) allow introduction of new state via layers.

A Comparison to Aspect-Oriented Programming

Cross-cutting concerns Both aspects and context layers modularise cross-cutting concerns.

Code scattering Aspects prevent code scattering by collecting code in a single location; Context layers are spread across classes (with a layer-in-class model); With a class-in-layer model, class fragments are scattered across layers.

Dynamicity Aspects are composed (weaved) statically; Layers are activated and deactivated dynamically – in response to dynamic changes in context.

Context-Oriented Programming: Summary

Benefits

- Context is modularised using layers.
- Dynamic activation & deactivation of layers in response to dynamic context change.
- Order of layer activation is explicit (user-defined).
- Objects of the same class behave differently, according to context.

Issues

- How to modularise context: layer-in-class, class-in-layer, both?
- Code scattering

This Week's Reading Assignment

- Hirschfeld, R., Costanza, P., and Nierstrasz, O. **Context-oriented Programming**. Journal of Object Technology 7, 3 (Mar. 2008), 125–151.

- Download link:
  `http://www.jot.fm/issues/issue_2008_03/article4/`

References I

Appeltauer, Malte, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid (2009). "A Comparison of Context-oriented Programming Languages". In: **International Workshop on Context-Oriented Programming**. COP '09. ACM Press, 6:1–6:6.

Appeltauer, Malte, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara (2011). "ContextJ: Context-oriented Programming with Java". In: **Information and Media Technologies** 6.2, pp. 399–419.

Hirschfeld, Robert, Pascal Costanza, and Oscar Nierstrasz (2008). "Context-oriented Programming". In: **Journal of Object Technology** 7.3, pp. 125–151.

Smith, Randall B. and David Ungar (1996). "A Simple and Unifying Approach to Subjective Objects". In: **Theory and Practice of Object Systems: Special issue on subjectivity in object-oriented systems** 2.3, pp. 161–178.

References II

Slides 3–8 are based on the course LSINF2335 – “Programming paradigms and applications” by Kim Mens, Université Catholique de Louvain.