Compose

Aspect-Oriented Composition Tools
for Composition Filters

Lodewijk Bergmans, Istvan Nagy, Gurcan Gulesir & Mehmet Aksit: TRESE project, University of Twente

Sverre Boschman, Raymond Bosman, Pascal Durr, Frederik Holljen, Carlos Noguera, Tom Staijen, Christian Vinkes, ..
Introduction

About Composition Filters

- **Goal**: support and reason about robust, scalable, composition
  - most AOP is language-specific
    - CFs are a language-independent extension
    - But the native type systems do not like us always..
  - most AOPLs allow breaking encapsulation
    - CFs do only ‘interface programming’
    - this even allows for extending precompiled objects..
  - declarative specifications
    - support conflict detection & verifying consistency
    - including reasoning about semantics
Introduction

About Compose*

- A platform for experimentation & proof of concept
  - but also for third parties
- (mostly) independent of target platform
- Supports multiple target languages/component models
- Framework supports any (mixed) approach from interpreter to inlining compiler
Outline of this presentation

- Introduction
- Example Application: Pacman
- Composition Filters in a Nutshell
- The Compose* tool
- Demonstrating some Features
The example application: Pacman
package pacman;

class Glyph {
    private int speed = 0;
    private int direction = 3;
    protected int x = 0;
    protected int y = 0;
    protected int dx = 0;
    protected int dy = 0;
    protected int vy = 0;
    protected int vx = 1;
    protected World world;

    public Glyph(World world) {
        this.world = world;
        this.reset();
    }

    public int speed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }

    public int direction() {
        return direction;
    }

    public void setDirection(int direction) {
        this.direction = direction;
    }

    public int x() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int y() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    public int dx() {
        return dx;
    }

    public void setDx(int dx) {
        this.dx = dx;
    }

    public int dy() {
        return dy;
    }

    public void setDy(int dy) {
        this.dy = dy;
    }

    public int vy() {
        return vy;
    }

    public void setVY(int vy) {
        this.vy = vy;
    }

    public int vx() {
        return vx;
    }

    public void setVX(int vx) {
        this.vx = vx;
    }

    public void reset() {
        x = 0;
        y = 0;
        dx = 0;
        dy = 0;
        vy = 0;
        vx = 1;
    }
}

Public Class Score
// This is Visual Basic
    Private score As Integer
    Public Function intValue() As Integer
        Return score
    End Function
    Sub increase(ByVal points As Integer)
        score = score + points
    End Sub
    Sub setValue(ByVal points As Integer)
        score = points
    End Sub
End Class
Composition Filters in a Nutshell

- filters
  - input~ & output~
- internals & externals
- conditions
- superimposition
  - selectors
  - filtermodules
  - binding
filtermodule dynamicstrategy {
  internals
    flee_strategy : FleeStrategy;
    stalk_strategy : StalkStrategy;
  conditions
    isEvil
  inputfilters
    flee_filter : Dispatch = { isEvil =>
        <*.getNextMove>flee_strategy.getNextMove };
    stalk_filter : Dispatch = { True =>
        <*.getNextMove>stalk_strategy.getNextMove };
}

superimposition{
  selectors    strategy = { *=RandomStrategy };
  conditions   strategy <- isEvil;
  filtermodules strategy <- dynamicstrategy;
}
Picture After Superimposition of DynamicStrategy

- fleeStrategy
- stalkStrategy

dynamicstrategy

FleeFilter
StalkFilter

isEvil

get Next Move

RandomStrategy

DynamicStrategy.isEvil
concern Tracing{
    filtermodule trace{
        inputfilters
            trace: Meta = { /*.**/inner.traceMessage };
    }
    superimposition {
        selectors
            tracedItems = {*=pacman_Ghost,
                *=pacman_RandomStrategy };
        filtermodules
            tracedItems <- trace;
        methods
            tracedItems <- inner.traceMessage; //
    }
...
Picture after superimposition of Tracing

- `trace`: The main strategy
  - `traceMessage`: Message trace
  - `dynamicstrategy`: Dynamic strategy
    - `isEvil`: Evil strategy check
    - `FleeFilter`: Flee filter
    - `StalkFilter`: Stalk filter
  - `get Next Move`: Next move strategy
    - `RandomStrategy`: Random strategy

- `fleeStrategy` and `stalkStrategy`: Strategies for fleeing or stalking.
Composition Filters

- Compose* is a modular language extension
- Current target: extend .NET

Implementation

- implementation is crucial
  - for our own verification
  - for the trust of the community
  - for the learning process internal/external
- many past implementations
  - throw-away prototypes; wasted effort
Current project: Compose*

- A platform for experimentation & proof of concept
  - but also experimentation for third parties
- (mostly) platform independent
- target language/component model independent (multiple)
- targeting any approach from interpreter to inlining compiler
- currently 6 MSc students active, 1 finished
- Current state: pre-alpha (…)

© 2004 Lodewijk Bergmans [lbergmans@acm.org]
Execution Model

- **Currently: interpreter**
  - of the filtering & superimposition only!
  - focus on functionality
  - experiment with dynamic composition

---

**TRESE**

© 2004 Lodewijk Bergmans [lbergmans@acm.org]
Preprocessing & Analysis

Superimposition Preprocessor (SUPRE)

- Resolve References
- Superimposition Analysis
- Filter Ordering
- ...

Repository:
- Very Abstract Syntax Tree + Extensions

Main Controller (MASTER)

- Filter Reasoning
- Selector Reasoning
- ...

Superimposition configuration file:
- Parse
- Syntactic sugar
- Semantic checks
- ...

Generate/Transform Repository Info

Domain-Specific Repository Interfaces

COde geNEration (CONE)

- Filter Consistency
- Filter Composition
- Semantic consistency
- ...

Analysis Tools

© 2004 Lodewijk Bergmans [lbergmans@acm.org]
Some Important Characteristics of Compose*

- **Language/Model & Platform are variable**
  - e.g. .NET (any .NET language), Java (JVM), C, ..
  - The filter specifications remain unaffected!

- **Various execution engines possible:**
  - only for filters & superimposition
  - interpreter … code transformation … inlining compiler

- **Set of Analysis Tools**
  - e.g. superimposition analysis, semantic conflict analysis

- **Architecture is a repository-centered toolkit**
  - adding e.g. new analysis tools is flexible

*(make and run)*
How does it work?

- The filter specifications are language independent
  - so reuse base and aspect code in any language.
- Notations are based on UML and OCL
- .NET has a language-agnostic component model;
  the implementation is ‘for free’
- But: the native compilers have to compile against
  enhanced interfaces due to introductions!
Demonstration of some Features:

Filter Reasoning (‘FIRE’)

- The Filter Language is declarative
  - limited expressiveness, e.g. not Turing-complete
  - domain specific semantics are encapsulated in the filter type

```java
flee_filter : Dispatch = { isEvil =>
  <*.getNextMove>flee_strategy.getNextMove };  
```

- hence we can ‘predict’ when a filter (or a composition of filters) will do what

(FIRE output)
Demonstration of some Features:

Filter Composition (‘FILTH’)

- As filtermodules are superimposed upon the same join point, there are several ways to compose them, such as ordering.

- This is not in the language (yet), but as a separate specification:
  - Define individual ordering constraints between superimposed filtermodules.

- If not restrained, all possible orderings are allowed

(FILTH output before and after constraint)
Demonstration of some Features:
Semantic Reasoning (‘SECRET’)

- Since:
  - Filters are declarative
  - Filter types encapsulate (domain-specific) behaviour
- By annotating the filter types with a (common) behavioral model/abstraction
- And analyzing superimposition and possible orderings
- We can detect possible semantic conflicts

(SECRET output)
Conclusion

- Filters aim at robust, predictable composition
  - they are declarative, language independent and highly composable

- Compose* is a tool/framework for composition filters
  - architecture supports multiple targets, repository-based set of analysis tools

- We demonstrated a few simple examples and current output of the tools
Conclusion

Future

- beta-release this summer
- open source (composestar.sf.net)
- things to do before release:
  - stabilize & missing features (e.g. filter types)
  - improve semantic detection by integrating FIRE (filter reasoning) and SECRET (semantic reasoning)
  - full integration with .NET
- Subsequent releases:
  - full OCL support for conditions & selectors (PCDs)
  - more advanced aspect composition at shared join points,
    - specifications integrated with language.
  - support abstract specifications of semantics in ACTs & user-defined filter types
  - experiment with dynamic weaving
Credits

The Development Team
Lost Puppies After this slide
Motivation

- The OO model has known deficiencies,
  1. need for additional mechanisms, e.g. AOP techniques
     - solving it by extending CIL is not feasible because languages do not support new concepts
     - solving this at the language level requires extending all prog. languages
       → use selection of email example to illustrate -an AOP- problem?
       → solution is a language neutral modular extension to CIL
  2. new challenges caused by language-heterogeneous composition (related)
     - because languages have different abstractions
     - semantic consistency checks really hard (n^2 language combinations)
  3. more advanced consistency checking needed, e.g. for accidental composition conflicts
Characteristics of CF approach

- **most AOP is language-specific**
  - CFs are a language-independent extension
  - But the native type systems do not like us always..

- **most AOPLs allow breaking encapsulation**
  - CFs do only ‘interface programming’
  - this even allows for extending precompiled objects..
    - but CFs are not meant for ‘patching’

- **declarative specifications**
  - support conflict detection & verifying consistency