Composing Configurable (Java) Components

Tijs van der Storm

31st March 2005

Introduction

Project Deliver:

- Intelligent Knowledge Management for Software Delivery
- Focus on release and delivery for product lines

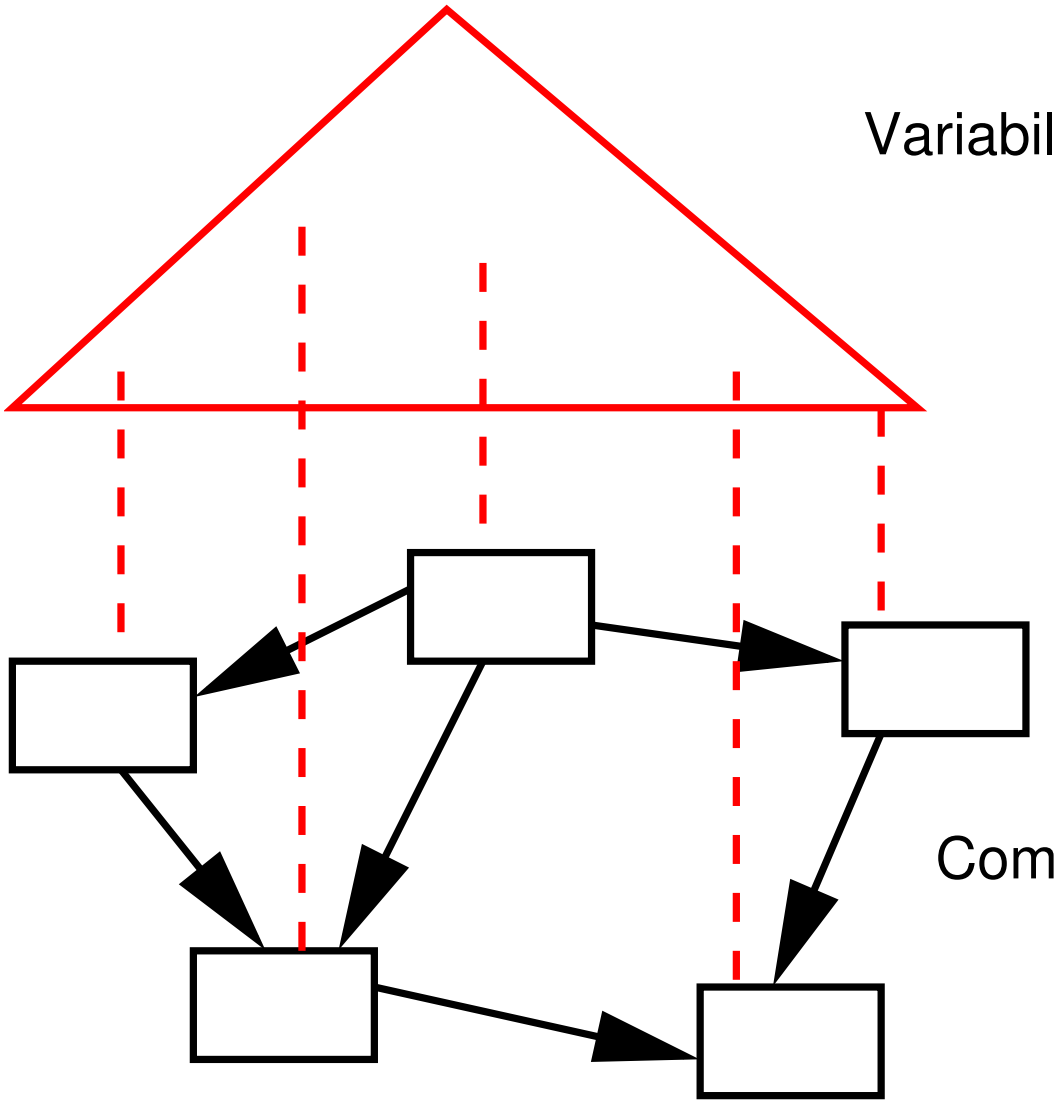
This talk:

- Composing Configurable (Java) Components
- *How to automatically deliver systems composed of configurable java components?*

Outline

- Configurable components
  - Component variability vs. system variability
  - Example Java component: `tree`
  - Implementing variability with AOP
- Automating composition
  - Specification of variability
  - Relating features to implementation
  - Deriving configured compositions

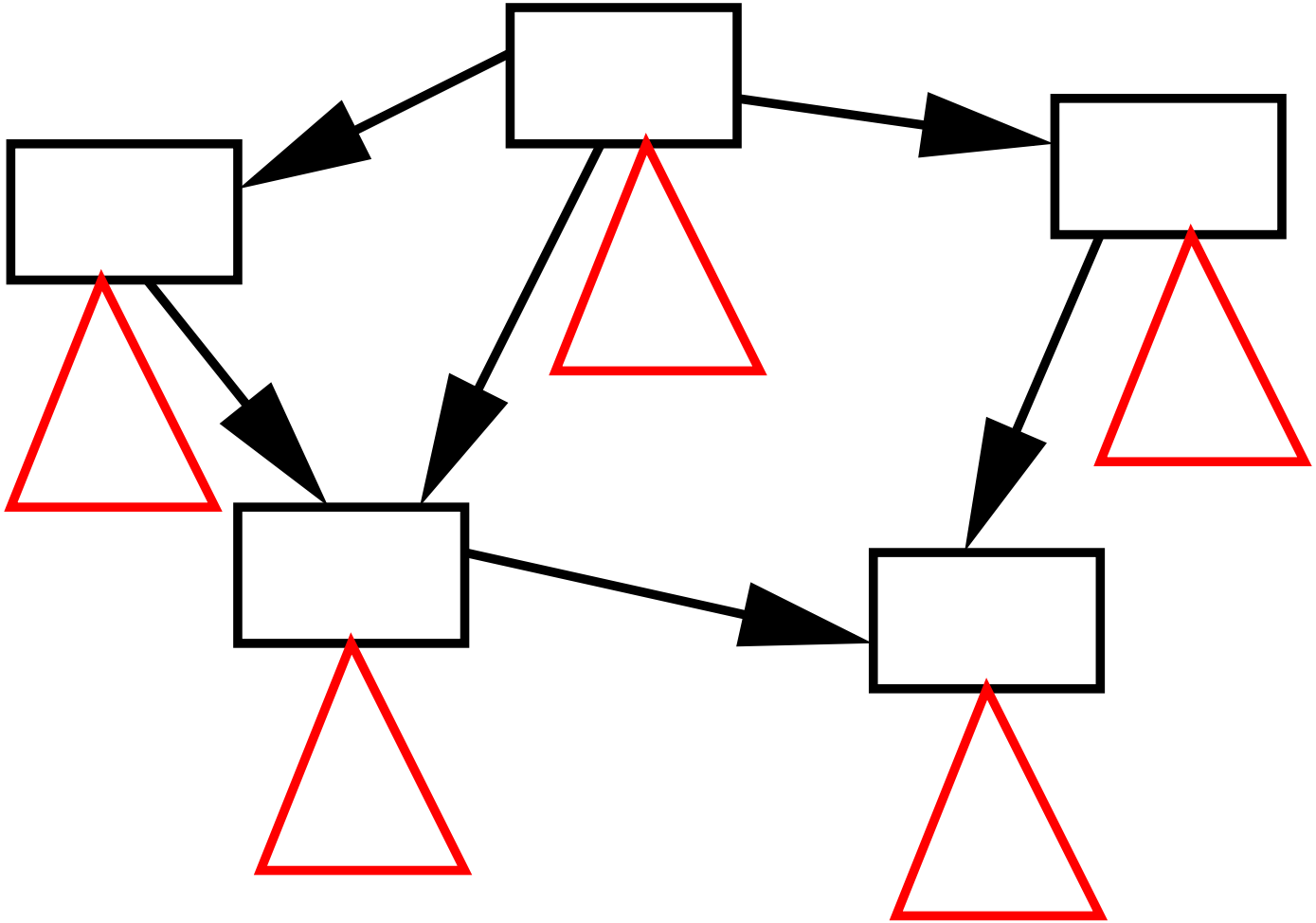
Variability described at level of system



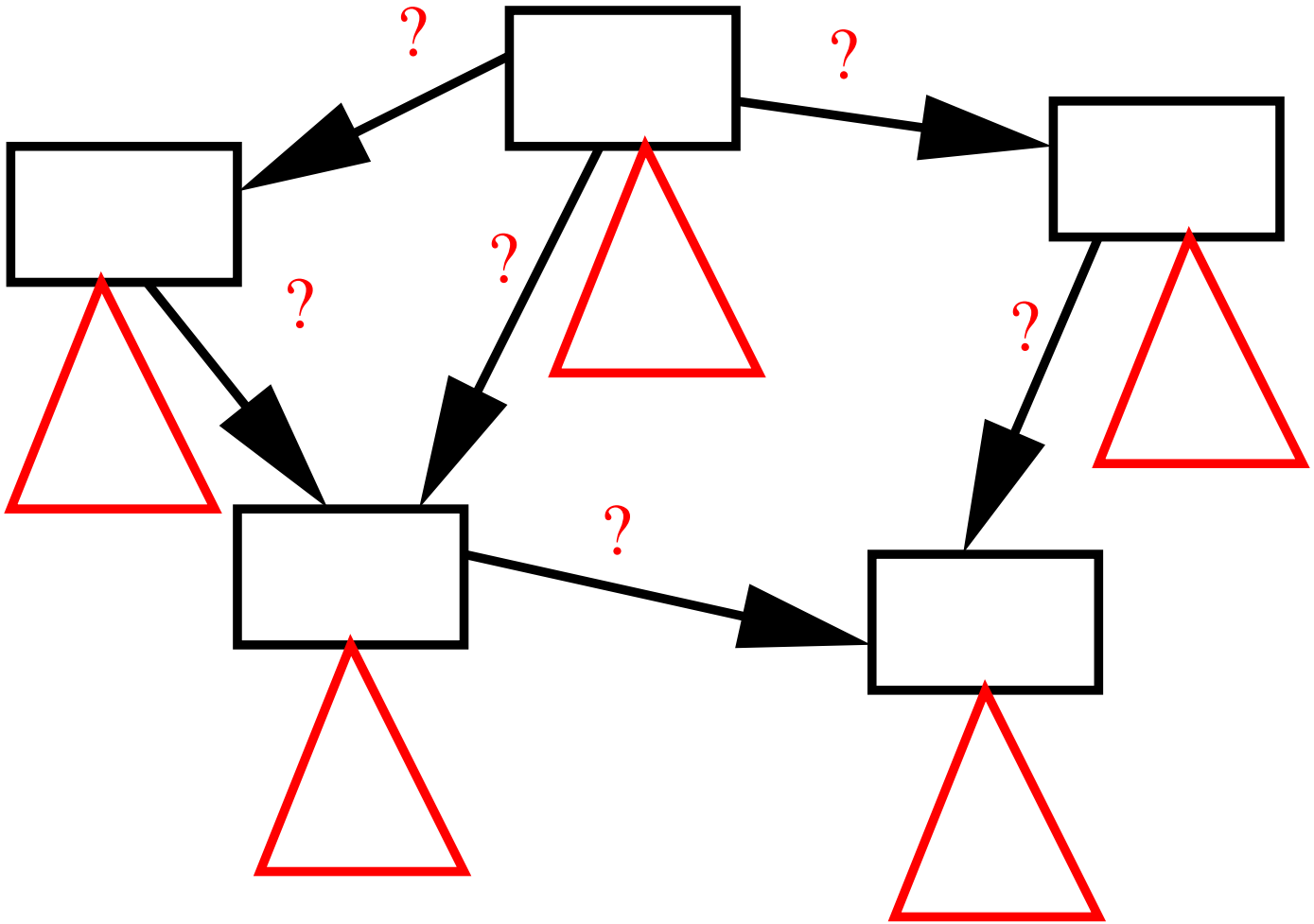
Variability description

Components

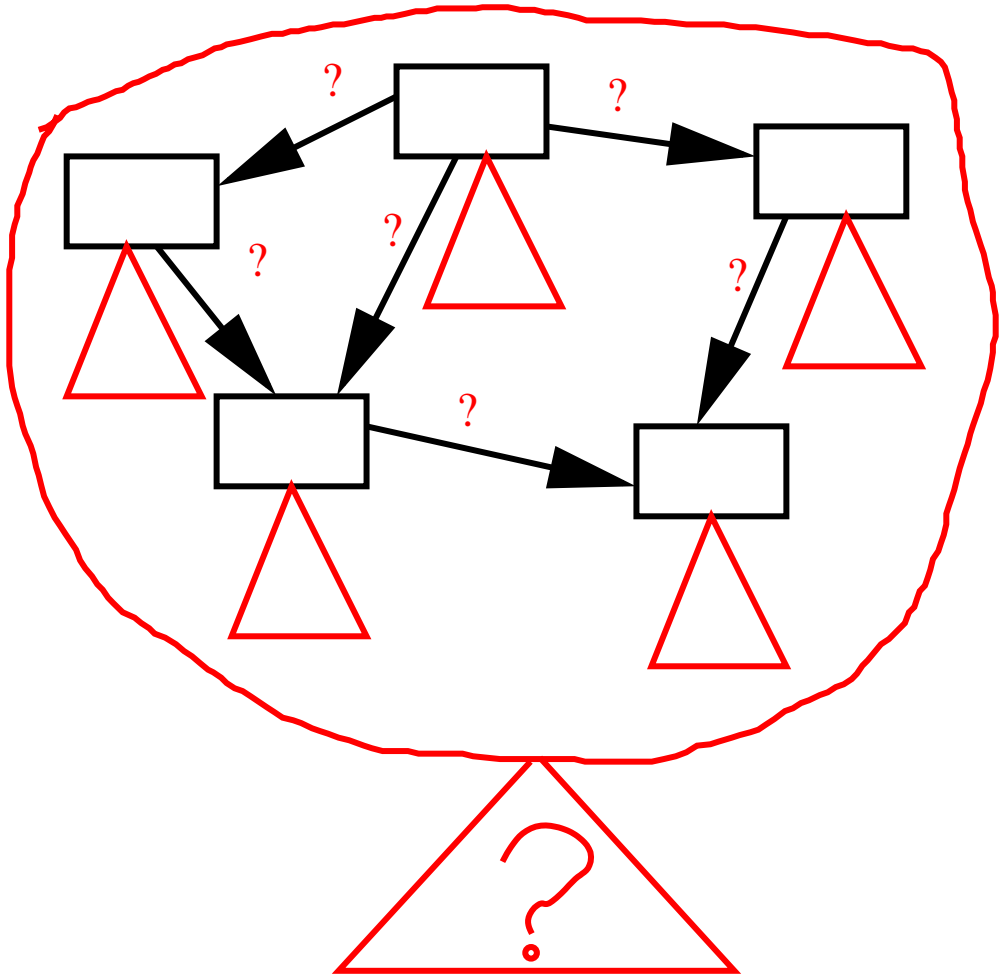
Variability described per component



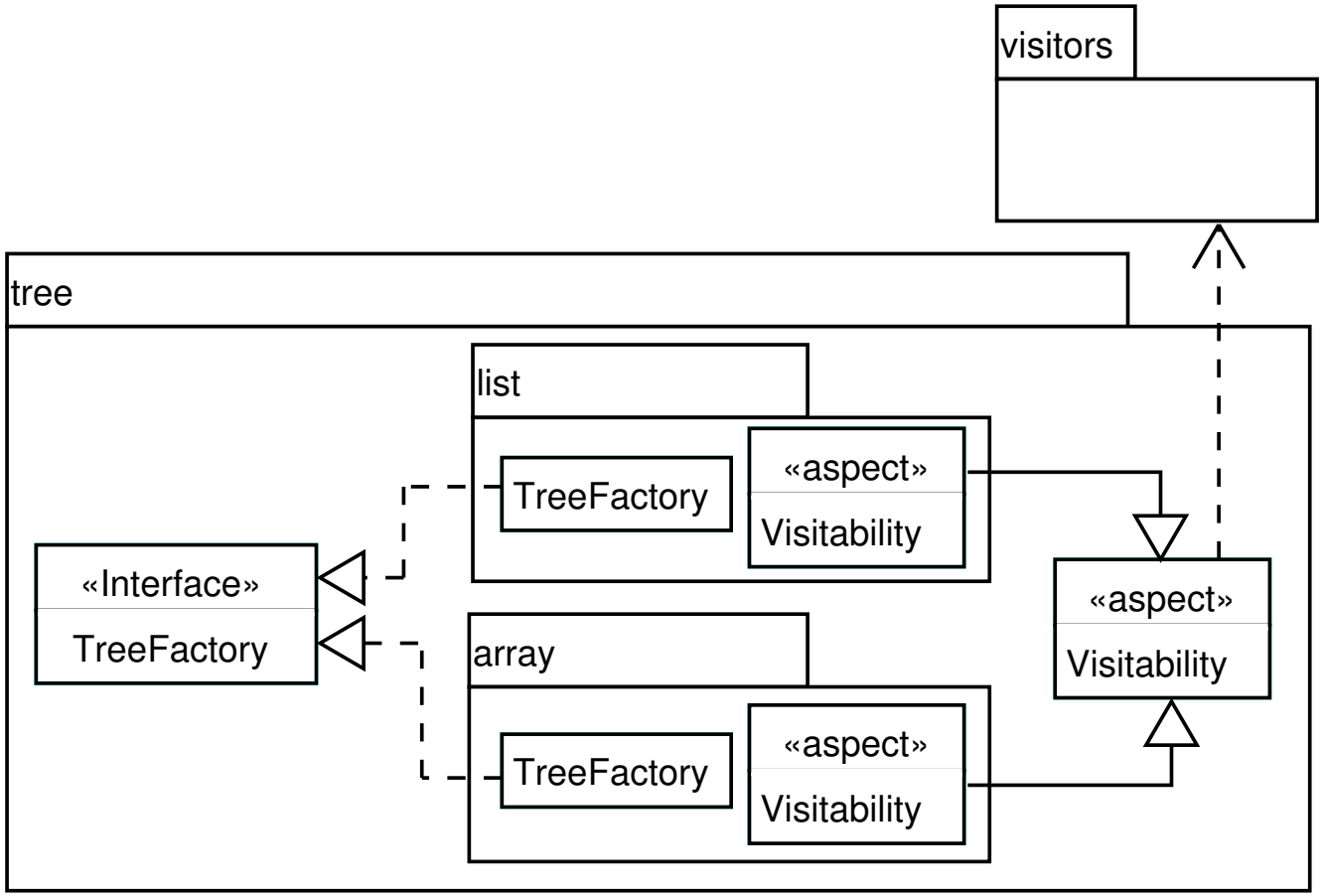
What happens at these junctures?



How to configure a composition?



A `tree` component with two implementations: array and list, and optional visiting functionality.



Implementing variability

Programming language imposes restrictions on variation mechanisms

Aspect-Oriented Programming (AOP) very popular for implementing variability.

Use AOP to:

- influence Tree factory creation; either array or list factory
- add Visitor design pattern; Trees should implement Visitable interface

Implementing optional visitor functionality

```
package tree;
import visitors.*;
public aspect Visitability {
  declare parents: Tree extends Visitable;
  public void Tree.accept(Visitor v) {
    v.visit(this);
  }
  public Visitable[] getKids() { ... }
}
```

Questions

Tree component has 4 variants:

- array with visiting, array without visiting
- list with visiting, list without visiting

How to configure this component family?

- How to prevent invalid configurations?
- Do we call AspectJ by hand?
- Which components go into the final `jar` when?

Moreover: propagation of variability

- What if `visitors` component is configurable as well?
- How to verify inter-component configuration?

Towards automatic delivery of compositions

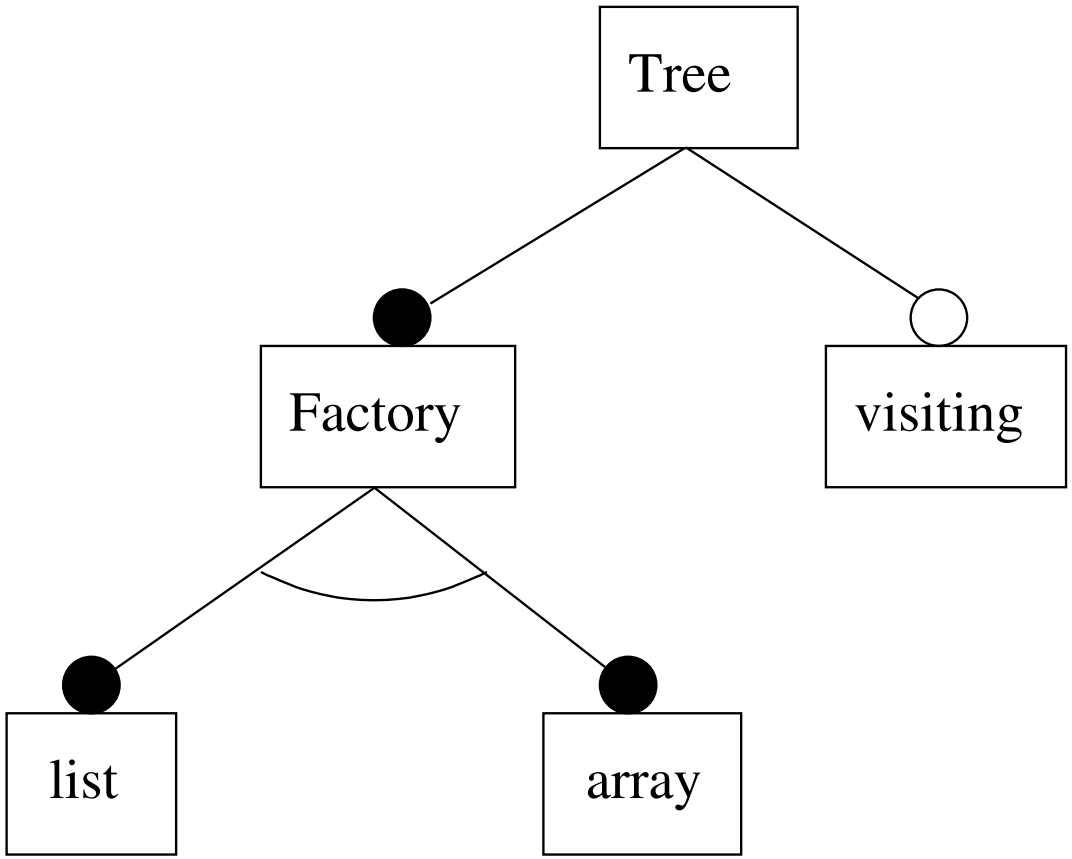
Goals:

- configuration user interface
- checking consistency of configuration
- automatically derive compositions

Component Description Language (CDL):

- configuration interfaces: specification of variability
- binding interfaces: mapping of variability to implementation

Feature diagrams



Configuration interfaces

Feature description:

- Composite features (e.g. `Factory`)
- Atomic features (e.g. `visiting`)
- Connectives: **`all`**, **`one-of`**, **`more-of`**, **`?`**
- Constraints: a **`requires`** b, **`include`** a

Feature description for Tree

```
Tree: all(Factory, visiting?)
Factory: one-of(list, array)
```

Binding interfaces

The interface between configuration and variation

- If-statements:
  - conditionals on atomic features
- Binding statements, e.g.:
  - set a property
  - generate code
  - weave an aspect
- Composition statements:
  - require another component

Example binding interface

Binding interface for `tree`

```
if (array) weave(UseArrayTrees);
if (list) weave(UseListTrees);
if (visiting) {
  require(visitors);
  weave(Visitability);
  if (list) weave(list.Visitability);
  if (array) weave(array.Visitability);
}
```

Dependency configuration

Dependencies may be passed atomic features.

Example

Assume `visitors` had alternative features `top-down` and `bottom-up`.

```
require(visitors, [top-down]);
```

Require only the top-down variant of the visitors component.

Variability inheritance

Dependencies may be configured partially.

Example

Assume `visitors` had an optional `logging` feature.

```
if (visiting) {
  require(visitors, [top-down]);
}
```

Now, `tree` inherits an optional `logging` feature *iff* the `visiting` feature is enabled.

NB: composition is recursively dependent on configuration.

Deriving compositions

A composition follows from the configuration of composed configuration interfaces:

- Union of configuration interfaces
- Constraints to respect **`requires`** relation

After configuring:

- Atomic features induce bindings
- Top composite features induce the composition

Example composition

- Union of configuration interfaces:

```
Tree: all(Factory, visiting?)
Factory: one-of(array, list)
Visitors: all(Strategy, logging?)
Strategy: one-of(top-down, bottom-up)
```

- Composition constraints:

```
visiting requires Visitors
visiting requires top-down
```

Consistency checking

Check composed configuration interface as boolean formula:

Features	Logic
feature description	boolean formula
atomic and composite features	atoms
configurability	satisfiability
configuration	valuation
validity of a configuration	satisfaction

Binary Decision Diagrams (BDDs) used to check satisfiability.

Valid configurations

Configuration	Composition
top-down	Visitors
top-down, logging	Visitors
bottom-up	Visitors
bottom-up, logging	Visitors
array	Tree
list	Tree
array, visiting, top-down	Tree, Visitors
list, visiting, top-down	Tree, Visitors
array, visiting, top-down, logging	Tree, Visitors
list, visiting, top-down, logging	Tree, Visitors

Component description language (CDL)

```
package tree {
  Tree: all(Factory, visiting?)
  Factory: one-of(list, array)
  if (array) weave(UseArrayTrees);
  if (list) weave(UseListTrees);
  if (visiting) {
    require(visitors);
    weave(Visitability);
    if (list) weave(list.Visitability);
    if (array) weave(array.Visitability);
  }
}
```

Summary

- Configurable Java Components
- Composition becomes complex:
  - Configuration correctness
  - Binding of features
- CDL for automation:
  - Feature descriptions
  - Binding actions

End

- Deliver project: `http://www.cwi.nl/projects/deliver`
- Technical report available
- More info: `http://www.cwi.nl/~storm`

Thank you!