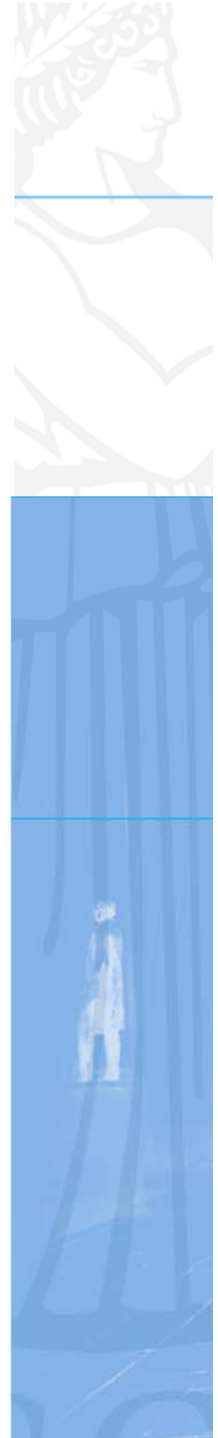
Dynamic Composition with Package Templates

Fredrik Sørensen, Eyvind W. Axelsen, and Stein Krogdahl
The SWAT-project

Composition&Variability 2010 15 march 2010

Overview

- Package Templates
  - Mechanism for reuse with name changes and re-typing
  - Existing static version
- Templates with template parameters
- Dynamic loading in a typed language
  - Different from Groovy PT
- Current and future work



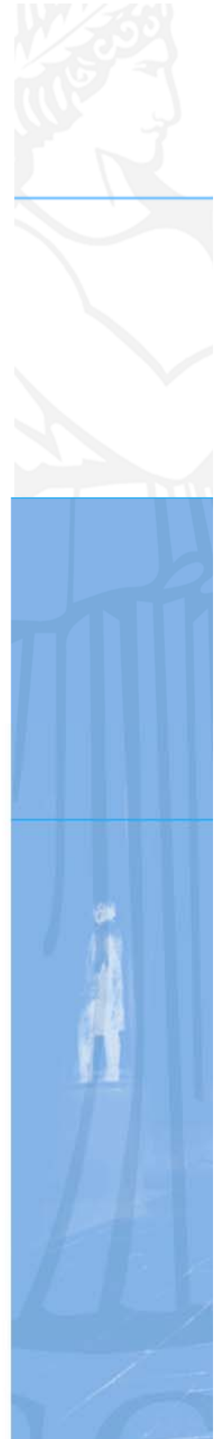
“Static” Package Templates

- Described in article from 2009 (JOT)
- Write reusable templates with cooperating classes
  - Frameworks
  - Libraries
- Statically and separately type checked
- Each use of a template is kept separate from the others
- Adapted to use with merging of classes and name changing
  - The hope is that a package template for each use can be tailored so that it seems to have been written especially for that use, concerning types, names, methods etc.
- Re-typed to types in context
  - Avoids casting (as for generics in general) and some problematic issues with co- and contra-variance
- Creates regular Java packages and works with Java
  - or other object oriented language
- Static PT can be translated to standard Java
  - Heterogeneous
  - Homogeneous

Graph template

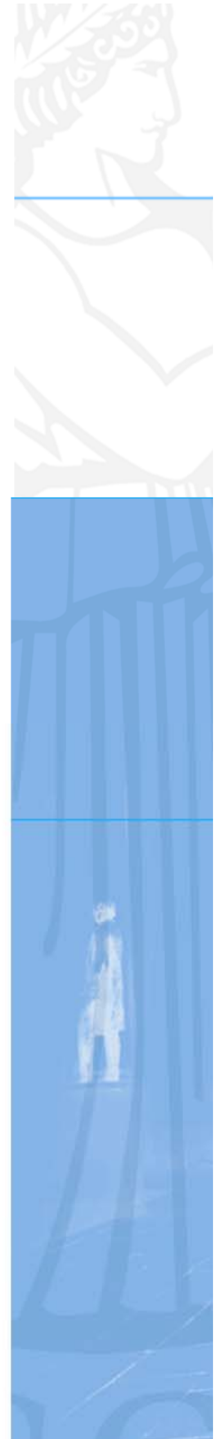
```
template Graph {
  class Node {
    Edge[] edges;
    Edge insertEdgeTo(Node to) { ... }
  }
  class Edge {
    Node from, to;
    void delete( ) { ... }
  }
}
```

- A reusable template
- Regular classes and inheritance inside
- Can not be used without creating an instance
- Statically create an instance and re-type for use
- We will not discuss private/public modifiers and type parameters



Using the template

```
package Program {
  inst Graph with Node => City, Edge => Road;
  class City adds {
    String name;
  }
  class Road adds {
    int length;
  }
  class Main {  // Node and Edge are not visible!
    static void main(String args[]){
      ...
      City r = new City(); r.name = "Rennes";
      City s = new City(); s.name = "Saint-Malo";
      Road n137 = r.insertEdgeTo(s);
      System.print(n137.to.name);
                                //Outputs 'Saint-Malo'
    } } }
```



Merging classes

```
template GeoData{
  class CityData {
    String name;
    void visit( ) { ... }
  }
  class RoadData{
    int length;
    void travel( ) { ... }
  }
}
```

```
package MyProgram {
  CRGraph: inst Graph with Node => City,        // Merged with CityData
                           Edge => Road;        // Merged with RoadData
  CRGeoData: inst GeoData with CityData=> City,  // Merged with Node
                               RoadData => Road; // Merged with Edge
  class City adds { ... more attributes? ... }
  class Road adds { ... more attributes? ... }
  class MyProgram {
    static void main(String args[]){
    ...
    System.print(n137.to.name);}
  }
}
```

First: Template parameters

- And classes with template/instance parameters

- Use template as interface or abstract template
  - Template interface (or abstract template) is not a language construct
  - Can have abstract declarations (a few or all) at the template level (tabstract)
  - Can be used (e.g. as parameter bound) without knowing the “implementation”.

- Hierarchies of templates
  - Only single inheritance, we talk about sub/super templates
  - Rules as for sub/super classes in bounded generic parameters

Graph template “interface”

```
template Graph {
  class Node {
    tabstract Edge insertEdgeTo(Node to); // Some abstract
    void deleteEdgeTo (Node to) { ... }   // Some implemented
  }
  class Edge {
    tabstract Edge[] edges();
    tabstract void delete ( );
  }
}
```

Extensions (implementations)

```
template ArrayGraph extends Graph {
  class Node adds { ... Edge insertEdgeTo(){ ... }; ... }
  class Edge adds { ... Edge[] edges(){...}; void delete() {...} ... }
}

template LinkedGraph extends Graph {
  class Node adds { ... Edge insertEdgeTo(){ ... }; ... }
  class Edge adds { ... Edge[] edges(){...}; void delete() {...} ... }
}
```

UNIVERSITETET I OSLO

Parameterized template

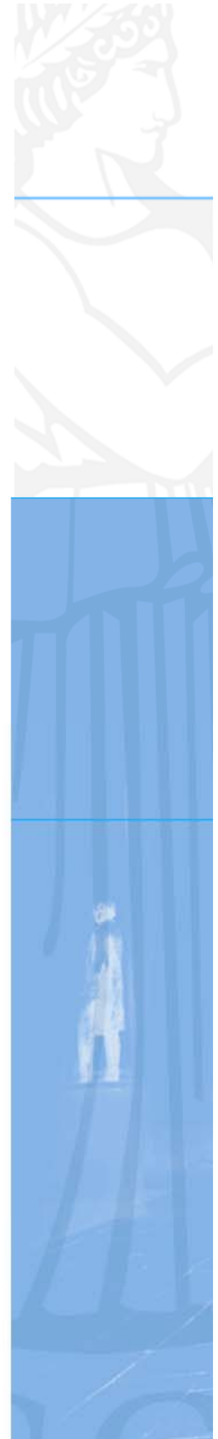
```
template Geography <G extends Graph> {
  inst G with Node=>City, Edge=>Road;
  inst GeoData with CityData=>City, RoadData=>Road;
  class City adds { ... }
  class Road adds { ... }
}
```

And it can be used like this

```
package Program {
  inst Geography<ArrayGraph>;
  class City adds { ... }
  class Road adds { ... }
}
```

Three kinds of template use

- A template instantiating another
  - Instatiating template has full control over adaption and use
  - Can allow elements from instantiated template to be hidden
  - Can instantiate any number of templates
- A template extending another
  - It is known that it has all the elements of the one it extends
  - The super-template can be used as bound on formal parameter
  - Can extend only one template (single inheritance)
- Template parameter
  - Does not need to know exact implementation
  - Can make several instances of (formal) parameter



Dynamic loading

- Add code to a running system
  - Code written after the system was started
  - Postponing combining and merging
- Obtain the same flexibility as (some) other systems
  - Like J&, gbeta, etc
  - Achieve full family polymorphism
- Regular classes are parameterized by a template in similar ways as for a type

Instance parameterized classes

- Java code parameterized by an instance (not template) type
- Each template *instance* gets a new type
- The types of the classes from the template depend on a shared type (T)
- Instance class

```
class Instance <T instance> { ... }
```

```
class GraphClient <T instance Graph> {
  public GraphClient<T>(){  }
  public void doIt(){
    T.Node n1 = new T.Node();
    T.Node n2 = new T.Node();
    T.Edge e = n1.insertEdgeTo(n2);
  }
}
```

And it can be used like this

```
class Program {
  public static void main(String[] args){
    Instance<? instance Graph> l = TLoader.load("...");
    GraphClient<?> gc = createGC(l);
    gc.doIt();
  }
  private <T> GraphClient<T> createGC(Instance<T> ins){
    return new GraphClient<T>();
  }
}
```

With parameterized template

```
class GeoClient <T instance Geography> {
  public GeoClient<T>(){  }
  public void doIt(){
    T.City c1 = new T.City(); c1.name = "Rennes";
    T.City c2 = new T.City(); c2.name = "Saint-Malo";
    T.Road n137 = c1.insertEdgeTo(c2);
  } }
```

And it can be used like this

```
class Program {
  public static void main(String[] args){
    Instance<? instance Geography> l =
        TLoader.load("... Geography with ArrayGraph ...");
    GeoClient<?> gc = createGC(l);
    gc.doIt();
  }
 private <T> GeoClient<T> createGC(Instance<T> ins){
    return new GeoClient<T>();
  }}
```

Translation to Java

- Instance parameterized can be translated to Java Generics
  - A type ties together the template classes
  - A new set of classes for each instance
  - Type safety through generics

- Factory pattern
  - Hidden parameter passed around for allowing “new”

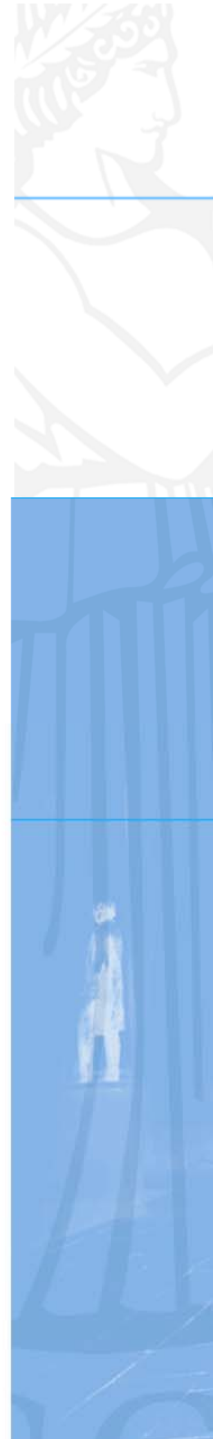
Challenges and further work

- Details of the language
  - Inheritance within template
  - tsuper/tabstract
  - Name changes
  - private/public
  - Reflection, debugging and runtime errors
  - Refactoring
- Statically checked composition
  - Load dynamically
  - Instantiate with composition, merging additions and changes
- Multiple inheritance at template level
- Type safety
- Compiler and efficient implementation

Summary

- Keeps the properties of PT, but allows dynamic loading and instantiation
- Statically checked even with the dynamic features
- Flexible way of composing and instantiating
- Different from classes with inner classes

Thanks!



Selected related work

- Traits
- Mixins
- Virtual Classes
  - Beta
  - J&
  - Caesar
- AOP

Translation to Java

```
abstract class Graph <T> {
    abstract Node<T> newNode();
    abstract Edge<T> newEdge();
}
class Node<T>{
    private Graph<T> secret;
    public Edge<T> insertEdgeTo(Node<T> other){
         return secret.newEdge(); }
}
class Edge<T>{
    private Graph<T> secret;
}
class GraphClient<T>{
    private Graph<T> secret;
    public GraphClient(Graph<T> secret)
       { this.secret=secret; }
    void test(){
       Node<T> n1 = secret.newNode();
        Node<T> n2 = secret.newNode();
       Edge<T> e = n1.insertEdgeTo(n2);
    }  }
```

Translation to Java

```
class Graph_0 extends Graph<Graph_0>{
    Node<Graph_0> newNode(){return new Node_0(); }
    Edge<Graph_0> newEdge(){return new Edge_0(); }
}
class Node_0 extends Node<Graph_0>{}
class Edge_0 extends Edge<Graph_0>{}

class Main{
   public static <T> GraphClient<T>
     createGraphClient(Graph<T> g) {
        return new GraphClient<T>(g);
   }

    public static void main(String[] args){
        Graph<?> g = new Graph_0();
        GraphClient<?> gc = createGraphClient(g);
        gc.test();
    }
}
```

tsuper

```
template Figures {
  abstract tabstract class Figure{
    abstract draw();
  }
  class Square extends Figure{}
  class Circle extends Figure{}
}
```

And it can be used like this

tabstract

And it can be used like this