Linee di prodotti software
(Software Product Lines)

Ferruccio Damiani

www.di.unito.it/~damiani

Tre mattine all'Universita'

Torino, 27 febbraio 2012

System Diversity

Systems exist in different variants to adapt to their application context.

Nokia 5310 XpressMusic

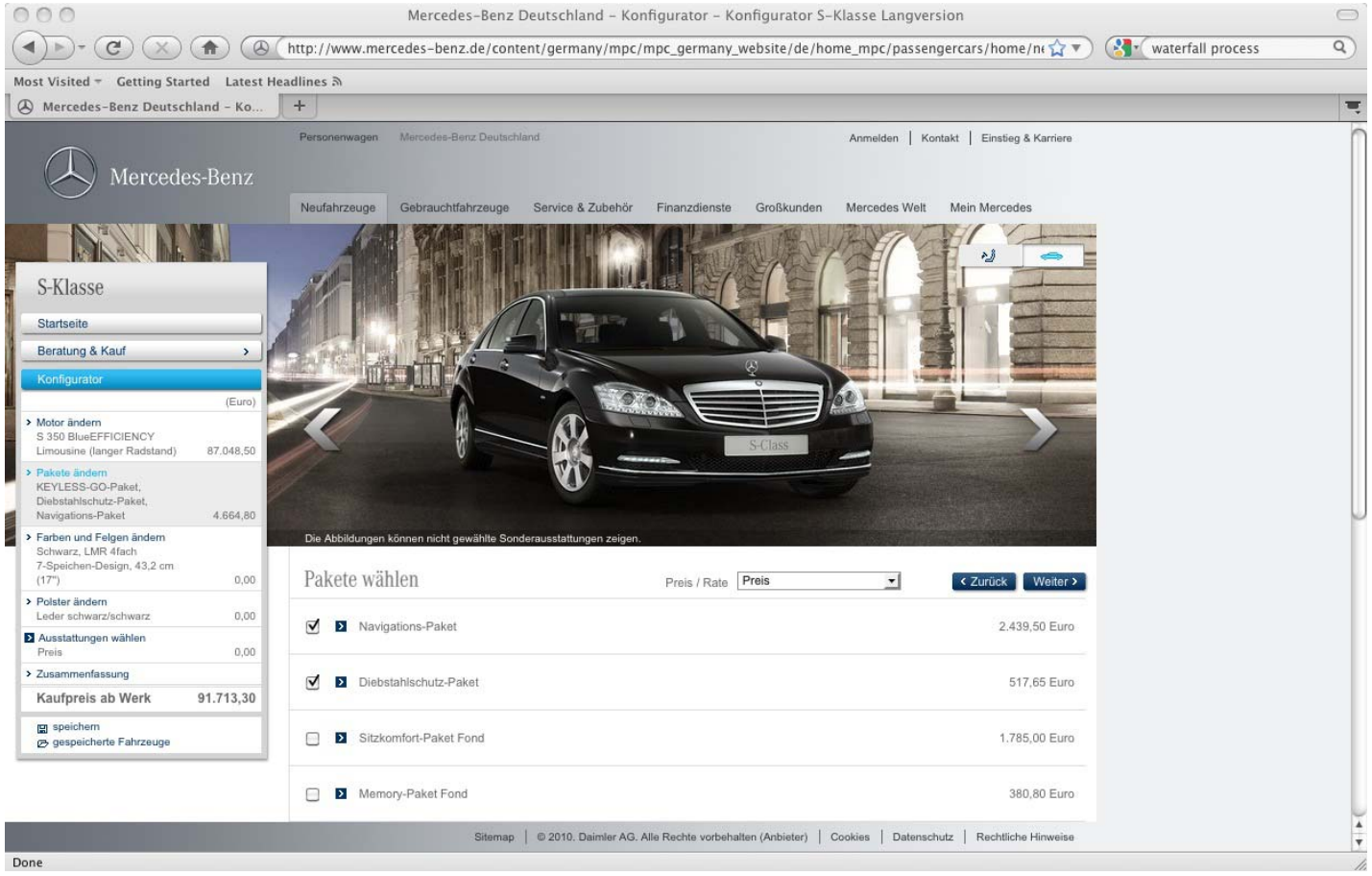
Nokia 5610 XpressMusic

Nokia N95 8GB

Nokia N81 8GB

System Diversity

Systems exist in different variants to adapt to their application context.



Variety of Variants

Variants are a significant key to business success.
- Franz Decker, Head of Program Variant Management, BMW Group

Example: BMW X3
3.000 different doors, 324 different rear axles, ...

What is a "Software Product Line"?

Product Line

A family of products designed to take advantage of their common aspects and predicted variabilities.

1. Don't want to build these programs from scratch
2. Want to automate the design and development of SPL programs
   - requires: a design for a program family, not a design for a single program
   - very common (and becoming more common) problem

What is a "Software Product Line"?

Product Line

A family of products designed to take advantage of their common aspects and predicted variabilities.

- Industrial-scale Approach for Reuse in Software Engineering
- Members of Product Line Hall of Fame: HP, Ericsson, Nokia, GM Powertrain, Boeing, Bosch, Lucent, Philips,Toshiba, ...
- Commercially successful, e.g., HP Owen Firmware Cooperative: 1/4 of the staff, 1/3 of the time, and 1/25 of the bugs (compared to previous single application engineering)

Outline

- Features
- Feature Oriented Software Development (FOSD)
- Software Product Line Engineering (SPLE)
- Delta Oriented Programming (DOP)
- Future Work

Feature Model (FM)

1. It defines the products of a (software) product line
   - Kyo Kang, et al 1990
   - hierarchically arranged set of features
   - particular (S)PL products are expressed as unique sets of features
2. It is the most important representation of a product-line
   - it is the master plan or master design of an (S)PL
3. FMs allow users to specify products declaratively
   - when coupled with implementations, products can be synthesized and downloaded to customers in minutes

Examples of On-Line PLs

- Build your own car

  http://www.fiat.it/modelli

- Build your own desktop

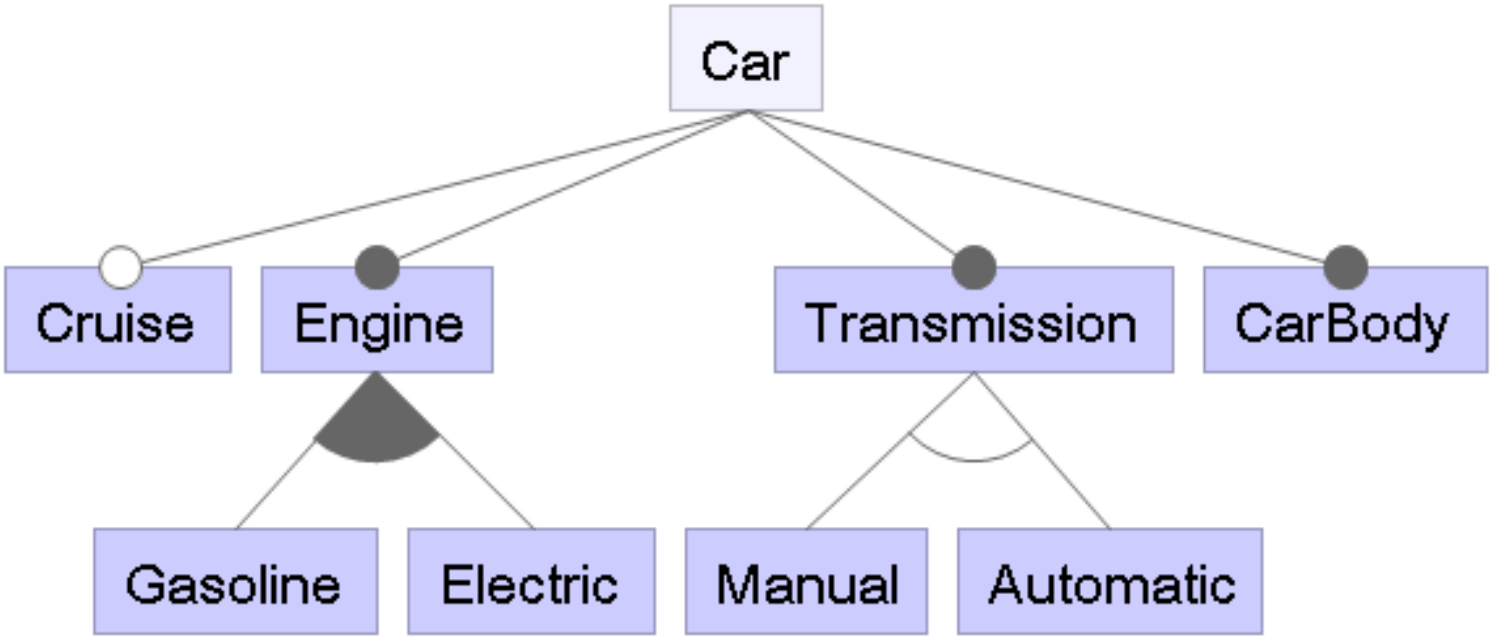
  http://www.dell.com/it/p/desktop-deals

- ...

Feature Diagram (FD)

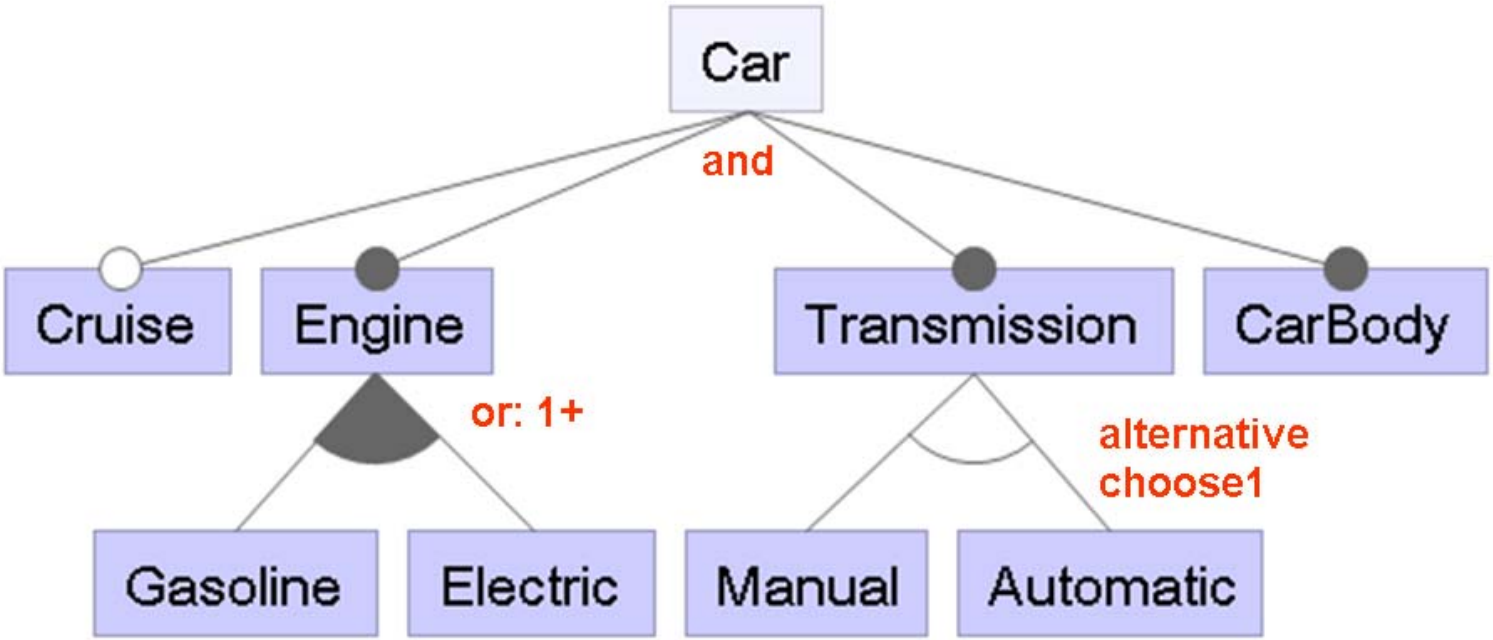
It is a standard notations for a feature model

- a FD is a tree
- leaves are primitive features
- internal nodes are compound features
- parent-child are containment relationships



Feature Diagram (FD)

- mandatory: features that are required •
- optional: features that are optional ◦
- and: all subfeatures (children) are selected
- alternative: only 1 subfeature can be selected
- or: at least 1 subfeatures must be selected



Why Feature Oriented Software Development (FOSD) ?

- Present functionality in an understandable way (good marketing)
- Controls complexity
- Allows customization
- Economical to realize
- Amortize development costs for additional functionality

Methodology for Construction

What Comp. Sci. methodology builds systems by incrementally adding details?

Methodology for Construction

What Comp. Sci. methodology builds systems by incrementally adding details?

- Stepwise Refinement
  - Dijkstra, Wirth early 1970s
  - abandoned in early 1980s as it didn't scale...
    - had to compose hundreds or thousands of transformations to produce admittedly small programs

Methodology for Construction

What Comp. Sci. methodology builds systems by incrementally adding details?

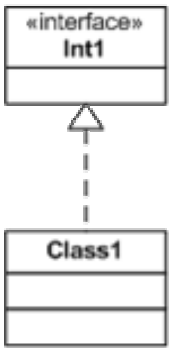
- Stepwise Refinement
  - Dijkstra, Wirth early 1970s
  - abandoned in early 1980s as it didn't scale...
    - had to compose hundreds or thousands of transformations to produce admittedly small programs

- Stepwise Development
  - program transformations correspond to adding (or removing) features
    - so that composing a few of them yields an entire system

Terminology disclaimer!

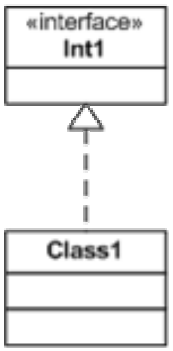
Terminology disclaimer!

Refinement: process of adding implementation detail without changing semantics

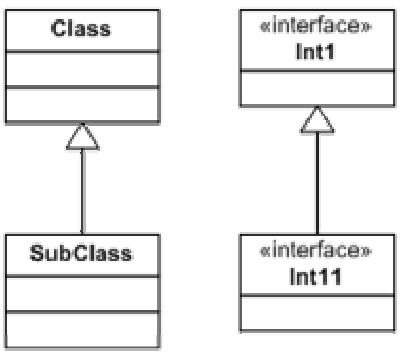


Terminology disclaimer!

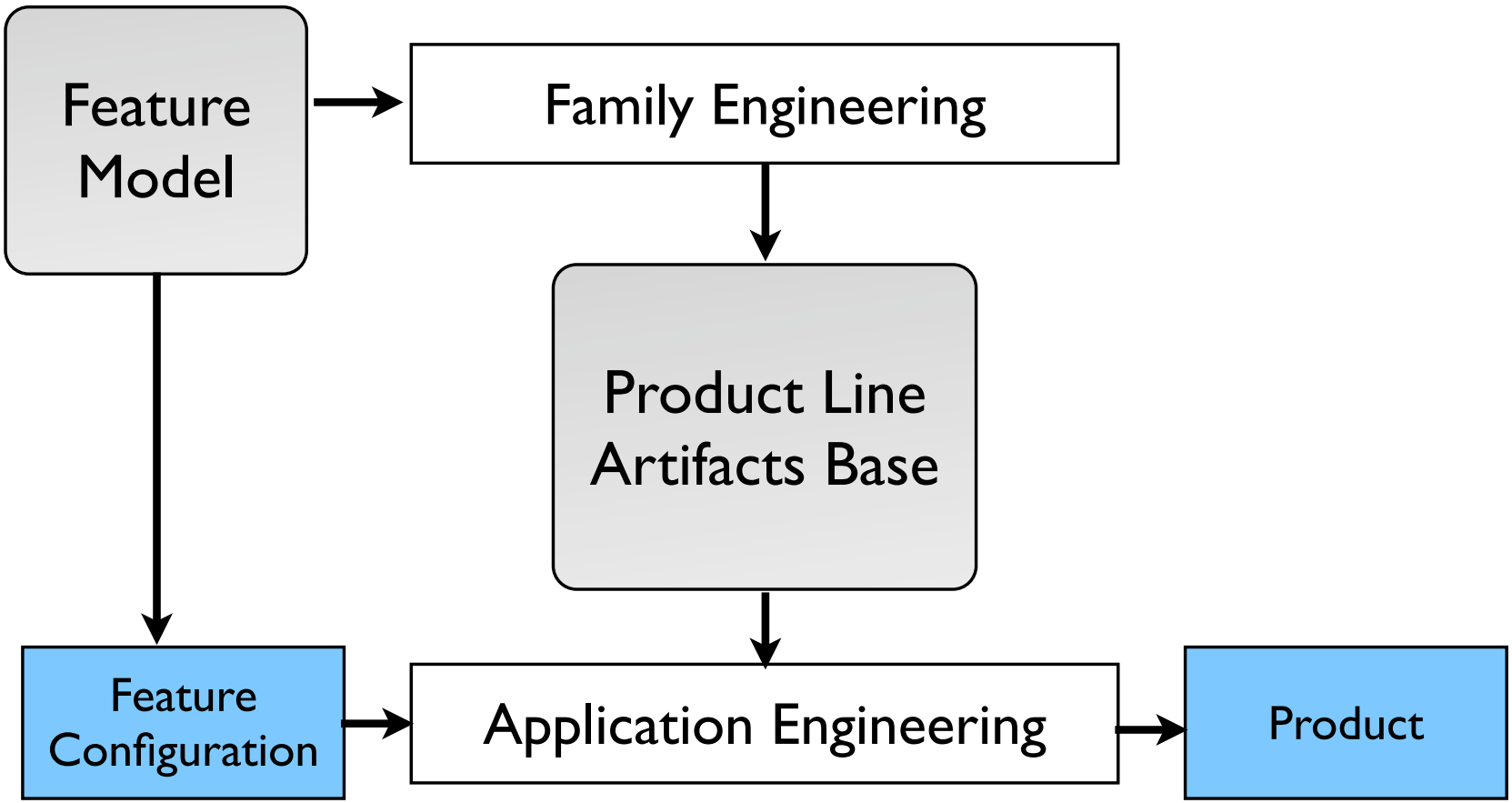
Refinement: process of adding implementation detail without changing semantics



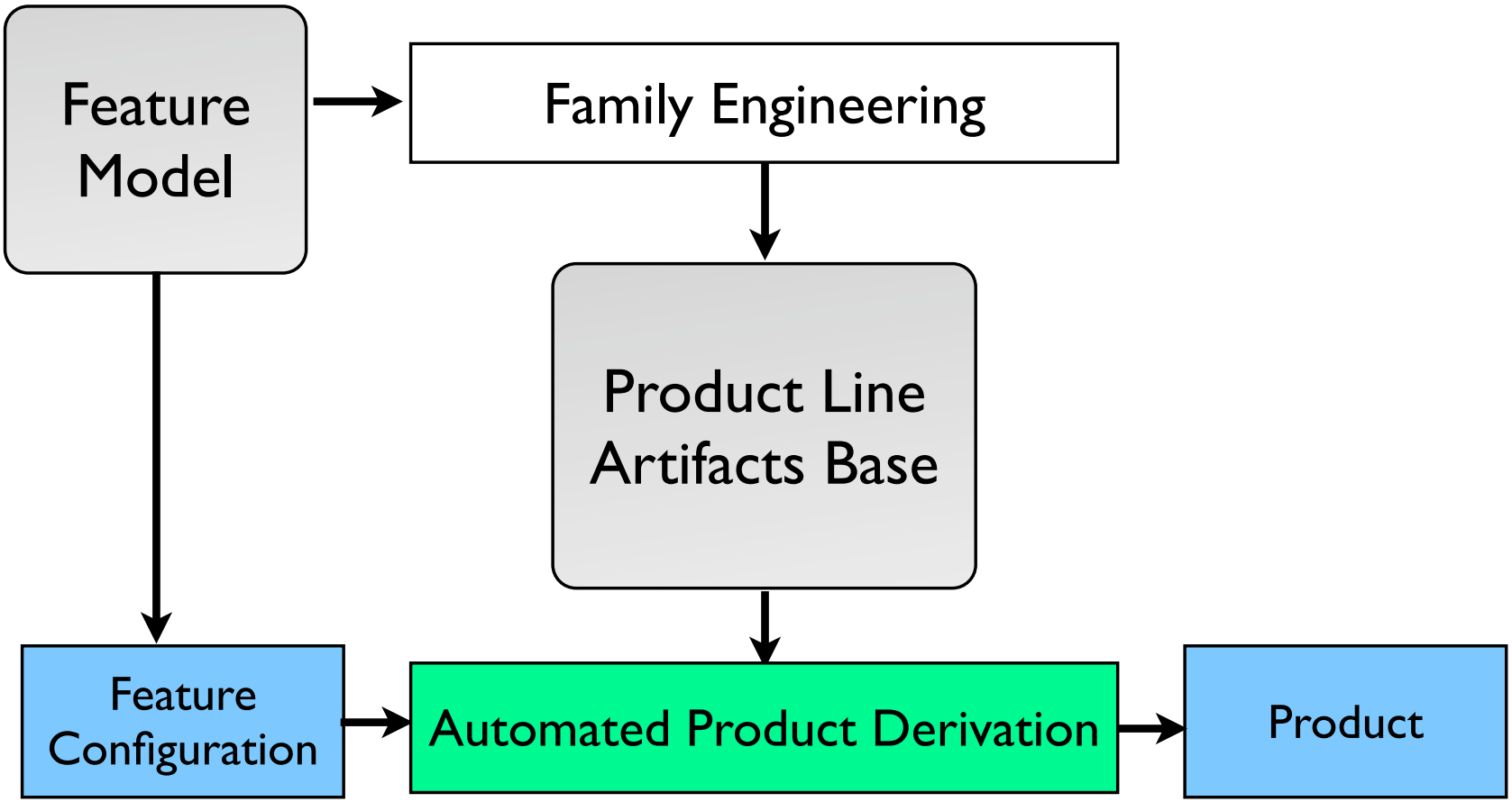
Extension: process of adding details to augment or extend semantics/capabilities



Software Product Line Engineering (SPLE)



Automatic SPLE



Single and multiple class-based inheritance inappropriate as a reuse mechanism

Single and multiple class-based inheritance inappropriate as a reuse mechanism

- Class-based inheritance does not support low-coupling (*Fragile Base Class Problem* [Mikhajlow and Sekerinsky, ECOOP 1998])

Single and multiple class-based inheritance inappropriate as a reuse mechanism

- Class-based inheritance does not support low-coupling (*Fragile Base Class Problem* [Mikhajlow and Sekerinsky, ECOOP 1998])
- Classes play two competing roles ([Schärli et al., ECOOP 2003, TOPLAS 2006])

Single and multiple class-based inheritance inappropriate as a reuse mechanism

- Class-based inheritance does not support low-coupling (*Fragile Base Class Problem* [Mikhajlow and Sekerinsky, ECOOP 1998])
- Classes play two competing roles ([Schärli et al., ECOOP 2003, TOPLAS 2006])

Generator of instances

Must provide a *complete* set of basic features

Single and multiple class-based inheritance inappropriate as a reuse mechanism

- Class-based inheritance does not support low-coupling (*Fragile Base Class Problem* [Mikhajlow and Sekerinsky, ECOOP 1998])

- Classes play two competing roles ([Schärli et al., ECOOP 2003, TOPLAS 2006])

Generator of instances

Must provide a *complete* set of basic features

Unit of reuse

Should provide a *minimal* set of basic features which can sensibly be combined together

Annotative vs Compositional approaches

- Annotative approaches
  - Conditional compilation (e.g., C preprocessor's #ifdef)
  - Frames
  - Colored Integrated Development Environment (CIDE)
  - ...

Annotative vs Compositional approaches

- Annotative approaches
  - Conditional compilation (e.g., C preprocessor's #ifdef)
  - Frames
  - Colored Integrated Development Environment (CIDE)
  - ...

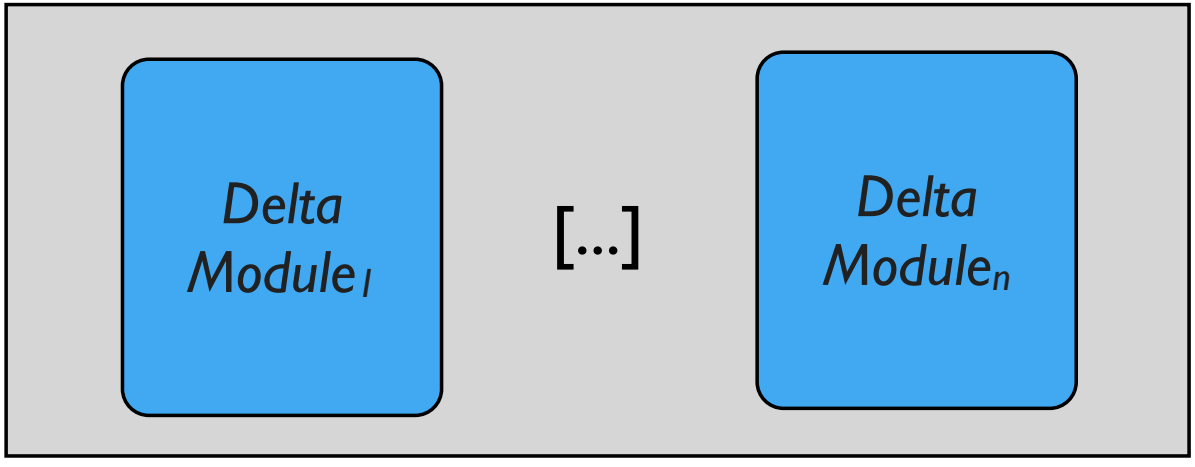
- Compositional approaches
  - Mixins
  - Traits
  - Aspects
  - Feature modules
  - Delta modules
  - ...

Delta-oriented Programming (DOP)

Product Line Declaration

- Connection between Delta Modules and Product Features
- Order of Delta Module Application

Code Base



[Schaefer et al., SPLC 2010; Schaefer and Damiani, FOSD 2010; Schaefer et al., AOSD 2011]

Product Generation in Delta-oriented Product Lines

Given a given feature configuration:

1. determine delta modules with valid application condition
2. apply the changes specified by delta modules
   - to the empty program
   - according to the delta module application ordering

Product Generation in Delta-oriented Product Lines

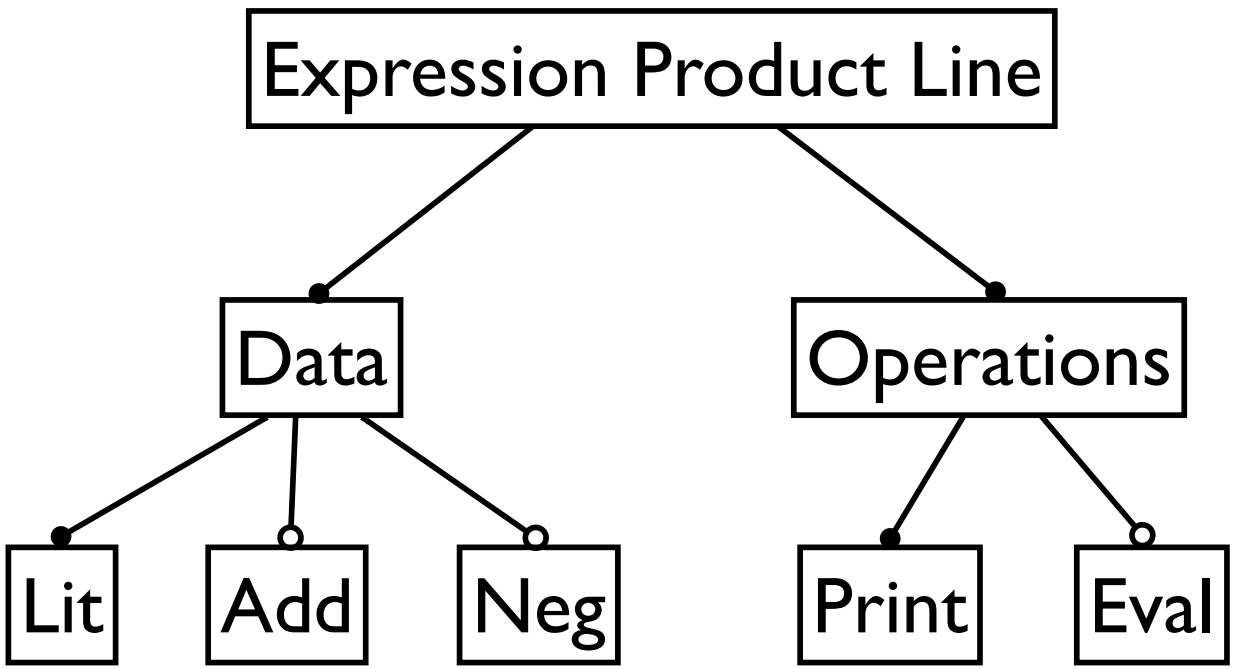
Given a given feature configuration:

1. determine delta modules with valid application condition
2. apply the changes specified by delta modules
   - to the empty program
   - according to the delta module application ordering

DeltaJ, a language for DOP (deltaj.sourceforge.net/)

Example: Expression Product Line (EPL)

Feature Model of EPL:



Delta Modules for EPL

```
delta DLit{
   adds interface Exp {
   }
   adds class Lit implements Exp {
      int value;
      Lit(int n) { value = n; }
   }
}

delta DLitPrint{
   modifies interface Exp { adds String toString();
   }
   modifies class Lit {
      adds String toString() { return value; }
   }
}

delta DLitEval{
   modifies interface Exp { adds int eval();
   }
   modifies class Lit {
      adds int eval() { return value; }
   }
}
```

Delta Modules for EPL (2)

```
delta DAdd {
   adds class Add implements Exp {
      Exp expr1;
      Exp expr2
      Add(Exp a, Exp b) { expr1 = a; expr2 = b;}
   }
}


delta DAddPrint{
   modifies class Add {
      adds String toString() { return expr1 + " + " + expr2; }
}


delta DAddEval{
   modifies class Add {
      adds int eval() { return expr1.eval() + expr2.eval(); }
   }
}
```

Delta Modules for EPL (3)

```
delta DNeg{
   adds class Neg implements Exp {
      Exp expr;
      Neg(Exp a) { expr = a; }
   }
}

delta DNegPrint{
   modifies class Neg {
      adds String toString() { return "-" + expr; }
   }
}

delta DNegEval{
   modifies class Neg {
      adds int eval() { return -1 * expr.eval(); }
   }
}

delta DAddNegPrint {
   modifies class Add {
      modifies toString { return "(" +  original + ")"; }
   }
}
```

Product Line Declaration for EPL

```
features Lit, Add, Neg, Print, Eval

configurations Lit & Print

deltas
    [ DLit,
      DAdd when Add,
      DNeg when Neg ]

    [ DLitPrint,
      DLitEval when Eval,
      DAddPrint when Add,
      DAddEval when (Add & Eval),
      DNegPrint when Neg,
      DNegEval when (Neg & Eval) ]

    [ DAddNegPrint when (Add & Neg) ]
```

Product for Features Lit, Add, Neg, Print

```
interface Exp { adds String toString();
}

class Lit implements Exp {
   int value;
   Lit(int n) { value = n; }
   String toString() { return value; }
}

class Add implements Exp {
   Exp expr1;
   Exp expr2
   Add(Exp a, Exp b) { expr1 = a; expr2 = b;}
   String toString() { return "(" + expr1 + " + " + expr2 + ")"; } }
}

class Neg implements Exp {
   Exp expr;
   Neg(Exp a) { expr = a; }
   String toString() { return "-" + expr; }
}
```

Software Product Line Engineering (SPLE)

Delta-oriented Programming supports

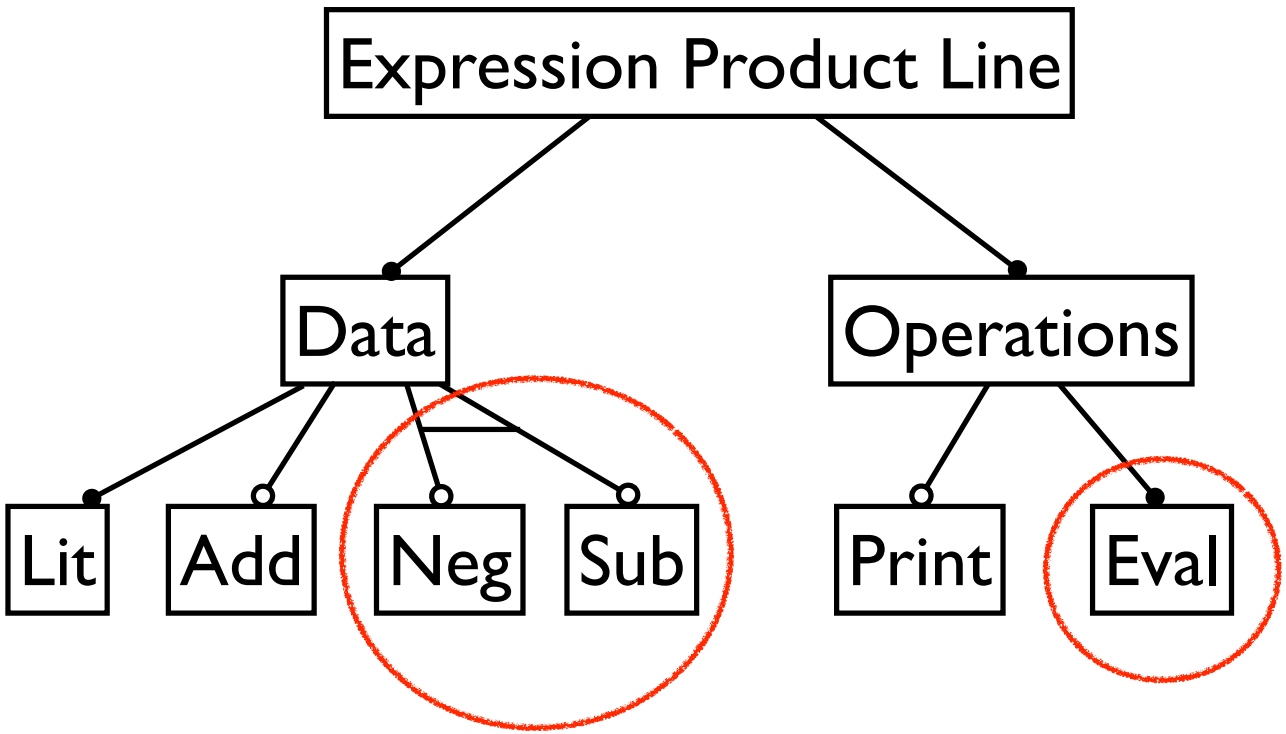
- Proactive SPLE: All products are planned in advance
- Extractive SPLE: Start from existing products
- Reactive SPLE: Evolve product line, when new features arise

Extractive Development of EPL

```
features Lit, Add, Neg, Print, Eval

configurations Lit & Print

deltas
   [ DLitNegPrint when (!Add & Neg) ] /* Existing product */

   [ DLitAddPrint when (Add | !Neg) ] /* Existing product */

   [ DNeg when (Add & Neg),
     DremAdd when (!Add & !Neg) ]     /* Feature removal */

   [ DNegPrint when (Add & Neg),
     DLitEval when Eval,
     DAddEval when (Add & Eval),
     DNegEval when (Neg & Eval) ]

   [ DAddNegPrint when (Add & Neg) ]
```

Evolution of EPL

Feature model for Evolved EPL:



Reactive Development of EPL

```
features Lit, Add, Neg, Sub, Print, Eval

configurations Lit & Eval & choose1(Neg,Sub)

deltas
   [ DLit,
     DAdd when Add,
     DNeg when Neg,
     DSub when Sub  /* new delta module */ ]

   [ DLitPrint when Print,
     DLitEval,
     DAddPrint when (Add & Print),
     DAddEval when Add,
     DNegPrint when (Neg & Print),
     DNegEval when Neg,
     DSubPrint when (Sub & Print),  /* new delta module */
     DSubEval when Sub   /* new delta module */ ]

   [ DAddNegPrint when (Add & (Neg | Sub) & Print) ]
```

Type-checking of Delta-oriented SPLs

Unambiguity

A SPL is unambiguous if for each valid feature configuration exactly one product is generated.

Type-safety

A SPL is type safe if all its products are well-typed programs.

Naive approach:

- Generate all the products
- Type check each product separately

Problems:

- Infeasible for large product lines
- Difficult to trace errors to delta modules

Requirements for DOP Type System

1. Check type safety without generating the products
2. Report errors in code of delta modules
3. Analyze each delta module in isolation (reusability)

Future work

- Case studies
- Extending the theory
  - SPLs testing
  - SPLs of verified programs
  - Dynamic SPLs
  - ...