Linhas de Produtos de Software

Arquitetura de Software

Marco Túlio Valente
DCC - UFMG

Definição

- Surgimento do termo: Workshop do SEI, 1996
  - Ou então: On the Design and Development of Program Families, David Parnas, 1976

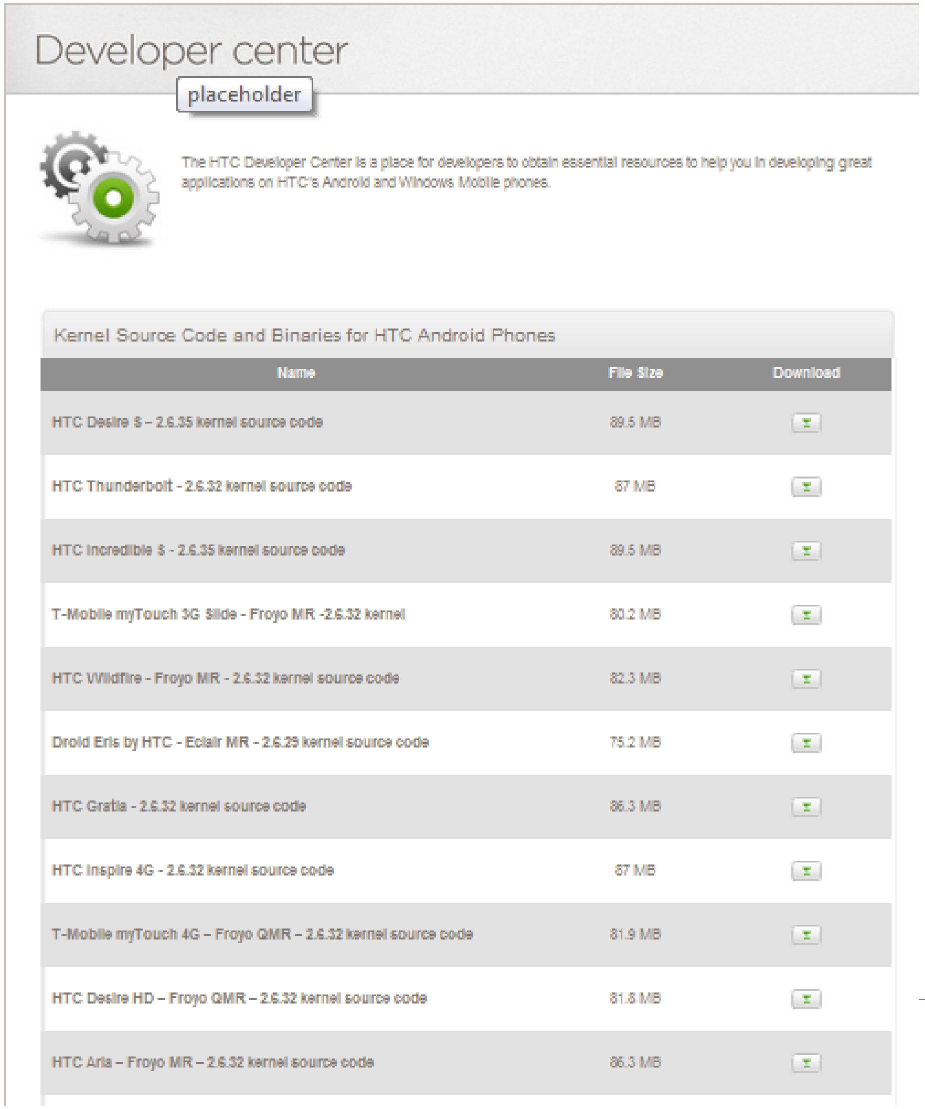
- SEI: “A SPL is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.”

- Inspiração: linhas produtos industriais (customização em massa)
  - Exemplo: indústria automobilística
  - Plataforma de carro comum; a partir dessa plataforma são fabricados diversos carros; que possuem diversos itens opcionais

Motivação: HTC Android

htc
quietly brilliant

http://developer.htc.com/



Developer center

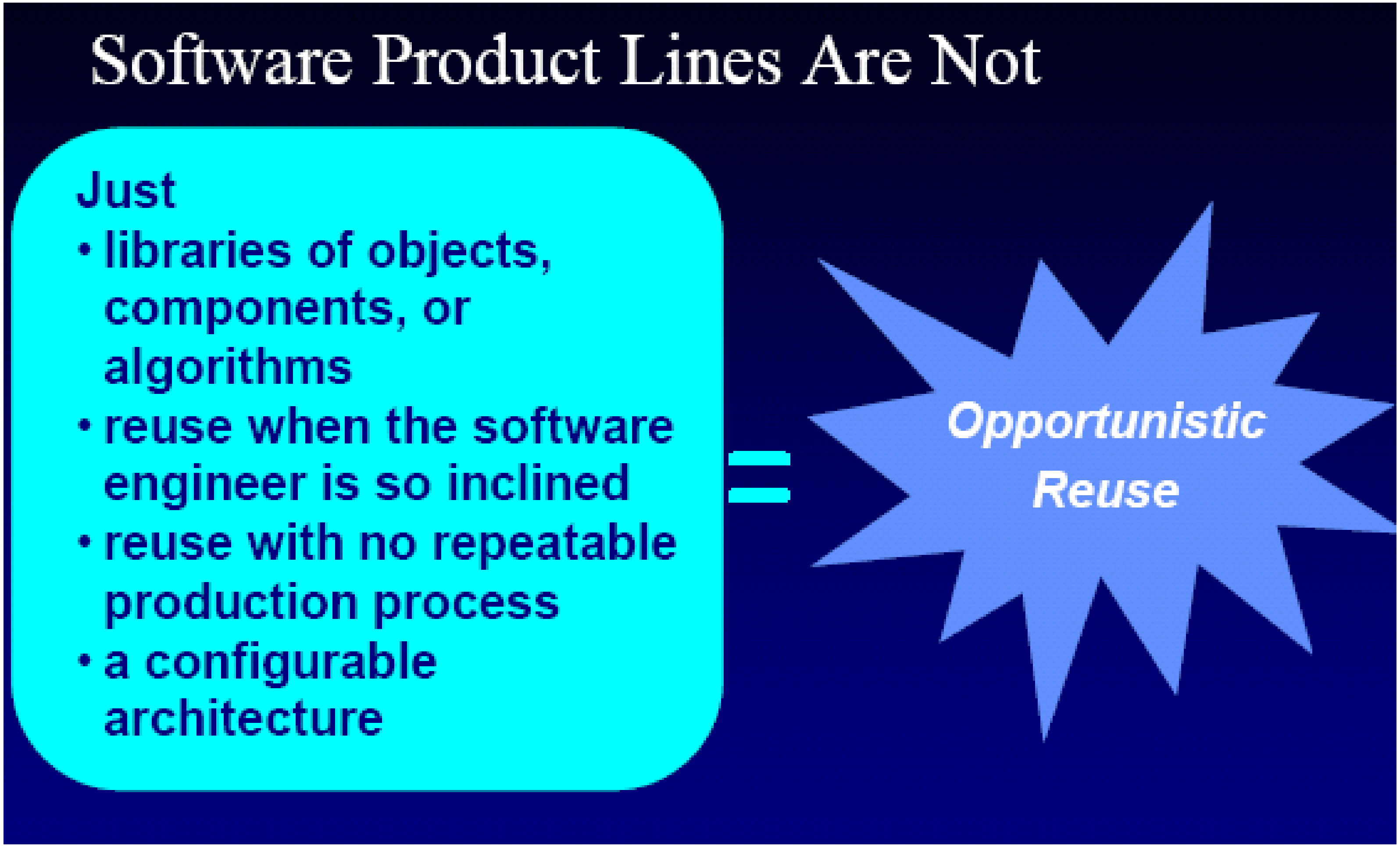
placeholder

The HTC Developer Center is a place for developers to obtain essential resources to help you in developing great applications on HTC's Android and Windows Mobile phones.

Kernel Source Code and Binaries for HTC Android Phones

Name	File Size	Download
HTC Desire S – 2.6.35 kernel source code	89.5 MB	⤓
HTC Thunderbolt - 2.6.32 kernel source code	87 MB	⤓
HTC Incredible S - 2.6.35 kernel source code	89.5 MB	⤓
T-Mobile myTouch 3G Slide - Froyo MR -2.6.32 kernel	80.2 MB	⤓
HTC Wildfire - Froyo MR - 2.6.32 kernel source code	82.3 MB	⤓
Droid Eris by HTC - Eclair MR - 2.6.29 kernel source code	75.2 MB	⤓
HTC Gratia - 2.6.32 kernel source code	86.3 MB	⤓
HTC Inspire 4G - 2.6.32 kernel source code	87 MB	⤓
T-Mobile myTouch 4G – Froyo QMR – 2.6.32 kernel source code	81.9 MB	⤓
HTC Desire HD – Froyo QMR – 2.6.32 kernel source code	81.8 MB	⤓
HTC Aria – Froyo MR – 2.6.32 kernel source code	86.3 MB	⤓

LPS ≠ Reúso Oportunista



Software Product Lines Are Not

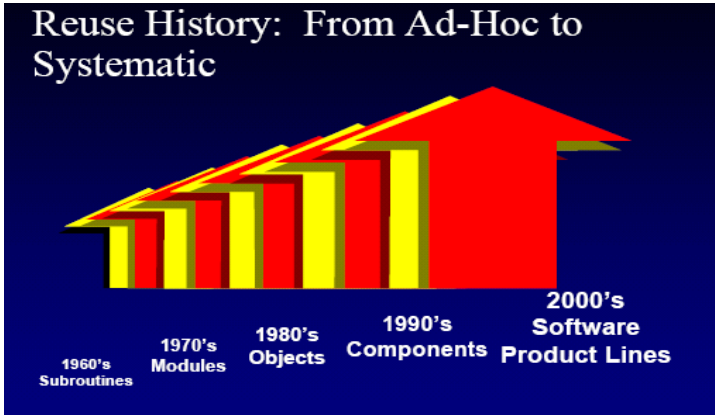
Just

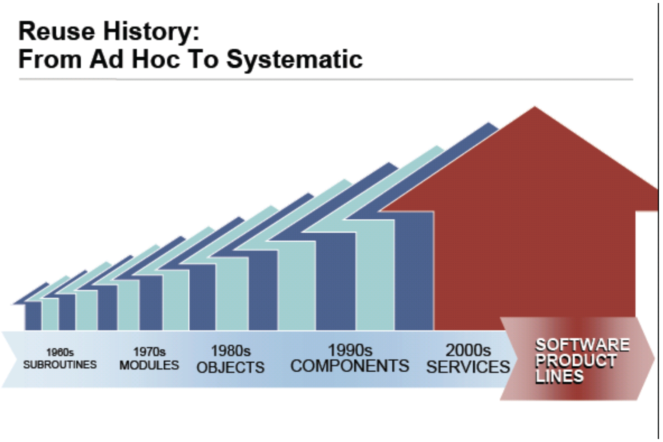
- libraries of objects, components, or algorithms
- reuse when the software engineer is so inclined
- reuse with no repeatable production process
- a configurable architecture

=

Opportunistic Reuse

LPS = Reúso Sistemático


Reuse History: From Ad-Hoc to Systematic
1960's Subroutines
1970's Modules
1980's Objects
1990's Components
2000's Software Product Lines



Desenvolvimento baseado em LPS

- Objetivo: identificar os aspectos comuns e as diferenças entre os artefatos de software ao longo do processo de desenvolvimento

- Pontos de **variabilidade**: pontos em que as características dos produtos podem se diferenciar

Desenvolvimento baseado em LPS

- Two key-phases:

  1. Domain Engineering involves creating a set of reusable assets for building systems in a particular problem domain.

  2. These reusable assets are then assembled to customer-specific systems in the complementary application engineering phase.

- The notion of a Feature Model, resulting from the Domain Analysis phase, is the most important contribution of domain engineering.

Domain-Engineering

- Domain engineering covers all the activities for building software core assets.

- These activities include:
  - Identifying one or more domains
  - Capturing the variation within a domain (Domain Analysis)
  - Constructing an adaptable design (Domain Design)
  - Defining the mechanisms for translating requirements into systems created from reusable components (Domain Implementation)

- The products of these activities are domain model(s), design model(s), code generators, and code components.

On the Notion of Variability in Software Product Lines

Jilles Van Gurp, Jan Bosch, Mikael Svahnberg

WICSA 2001

Introduction

- Newer approaches to software design: many design decisions are delayed to later stages

- A typical example of such delayed design decisions is provided by software product lines

- SPL: a software architecture and set of components is defined to match the requirements of a family of software products
  - Rather than deciding on what product to build beforehand

The Variability Funnel

Desenvolvimento
“Tradicional”

Desenvolvimento
Baseado em LPS

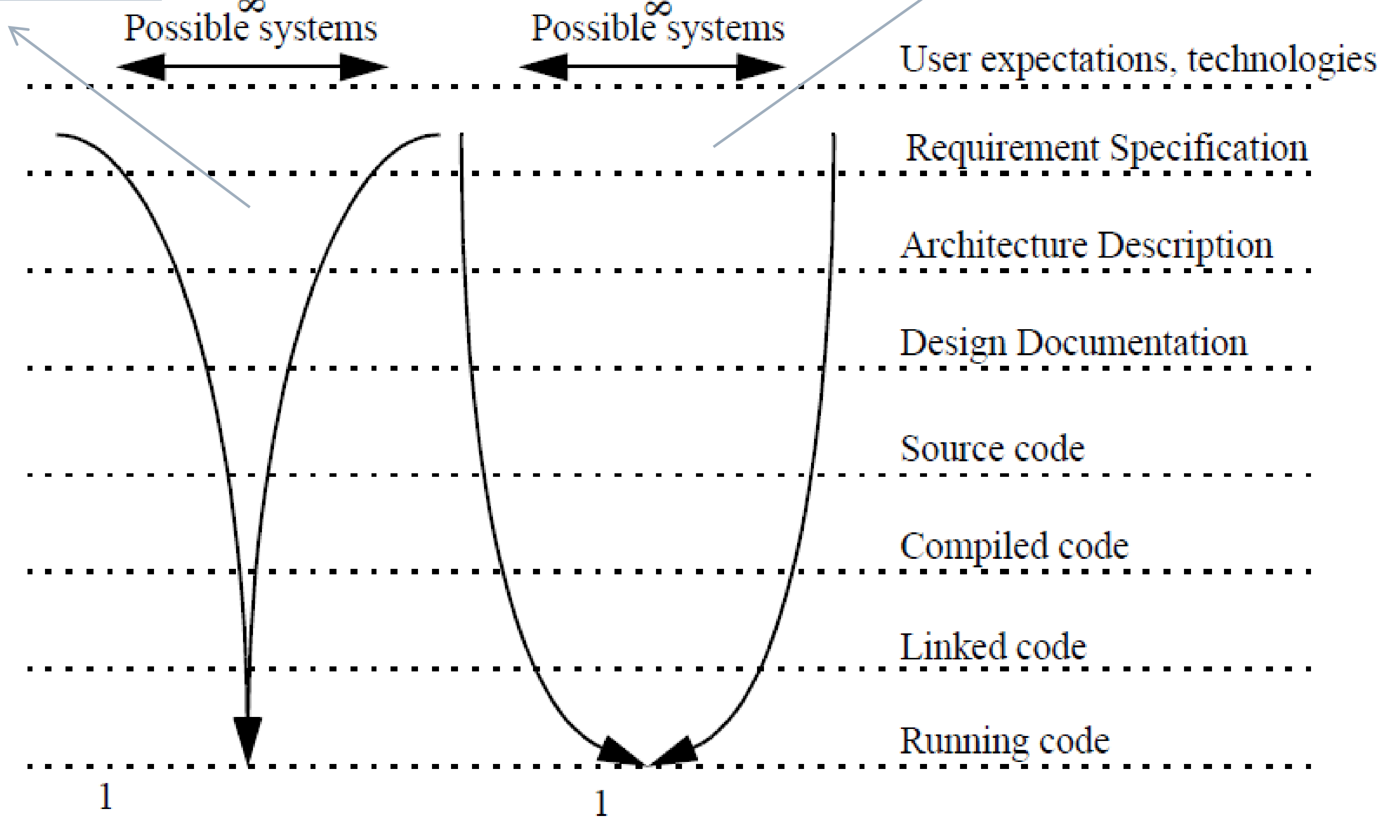


FIGURE 1. The Variability Funnel with early and delayed variability

The Variability Funnel

- Figure 1 illustrates how the variability of a software system is constrained during development

- When the development starts, there are no constraints on the system (i.e. any system can be built).

- During development the number of potential systems decreases until finally at run-time there is exactly one system
  - i.e. the running and configured system

- At each step in the development, design decisions are made.
- Each decision constrains the number of possible systems.

Software Product Lines

- The goal of a SPL:
  - To minimize the cost of developing and evolving software products that are part of a product family

- A SPL captures commonalities between software products for the product family

- When SPLs are considered, it is beneficial to delay some decisions so that products implemented using the shared product line assets can be varied.

- We refer to these delayed design decisions as variability points.

Feature Types

- External Features
- Mandatory Features
- Optional Features
- Variant Features

External Features

- Features offered by the target platform of the system
  - They are not directly part of the system
  - But they are important because the system uses them and depends on them.

- E.g. in an email client, the ability to make TCP connections to another computer is essential but not part of the client

- Instead the functionality for TCP connections is typically part of the OS on which the client runs

Mandatory and Optional Features

- **Mandatory Features:**
  - These are the features that identify a product.
  - E.g. the ability type in a message and send it to the SMTP server is essential for an email client application

- **Optional Features:**
  - These are features that, when enabled, add some value to the core features of a product
  - Example: the ability to add a signature to each message
  - It is in no way an essential feature and not all users will use it but it is nice to have it in the product

Feature Graph

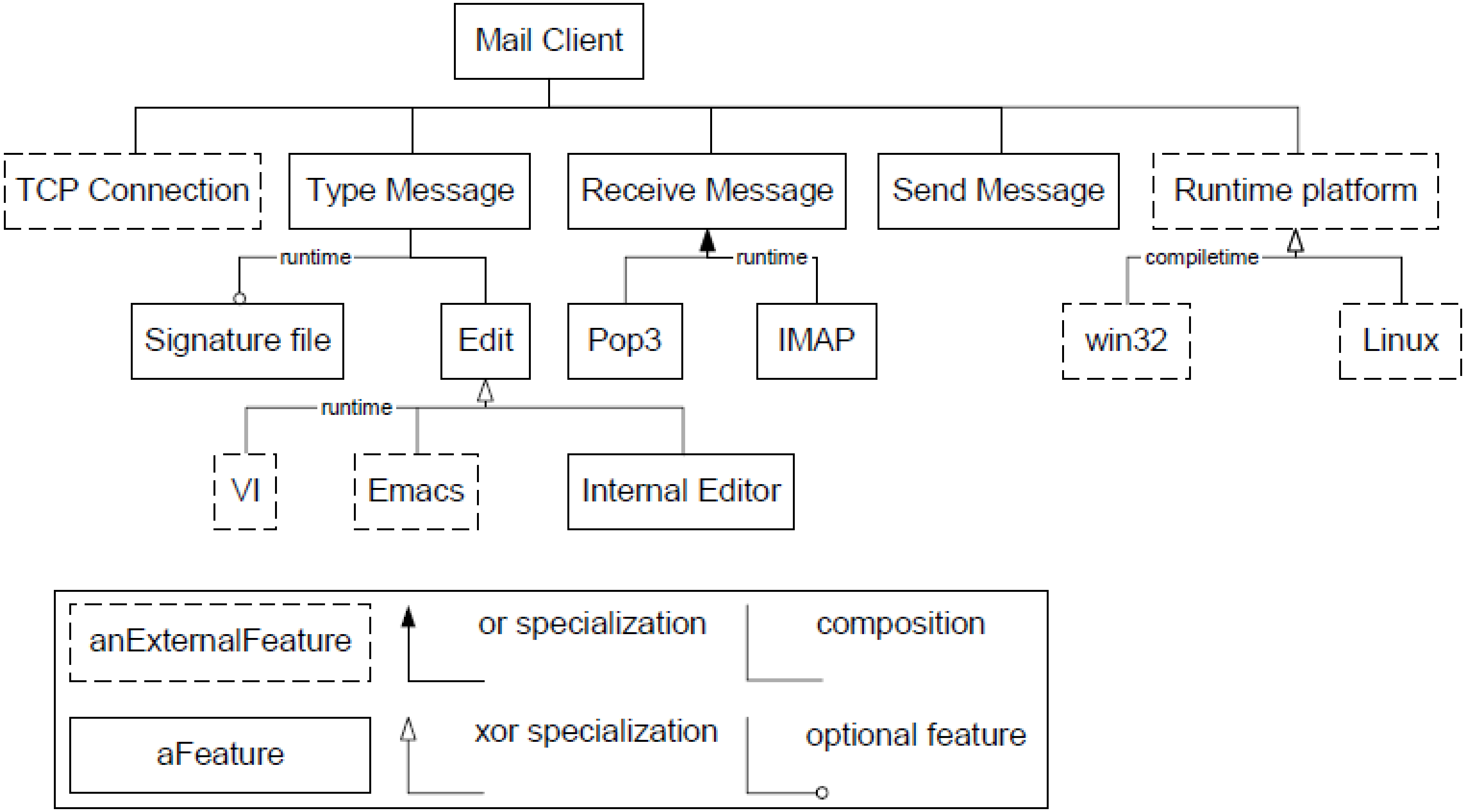


FIGURE 2. Example feature graph

Feature-Oriented Domain Analysis Feasibility Study

Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson

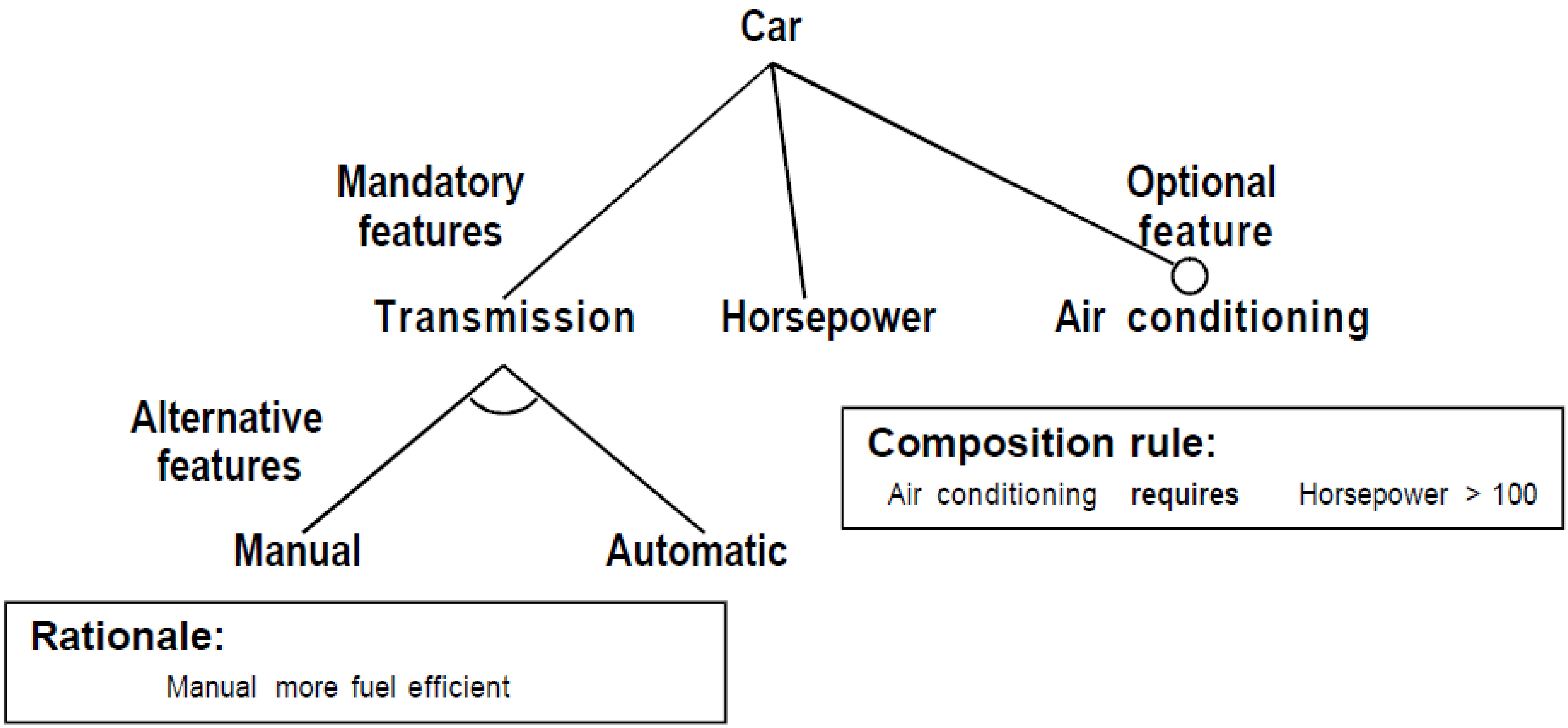
Technical Report CMU/SEI-90-TR-21, 1990

Feature Model

- Feature model: should capture the common features and differences of the applications in the domain

- Communication medium between users and developers.
  - To the users, it shows what the standard features are, what other features they can choose, and when they can choose them
  - To the developers, it indicates what needs to be parameterized in the other models and the software architecture.

Feature Model



Car

Mandatory features

Optional feature

Transmission

Horsepower

Air conditioning

Alternative features

Manual

Automatic

Composition rule:

Air conditioning **requires** Horsepower > 100

Rationale:

Manual more fuel efficient

Binding Time

- Compile-time features:
  - Features that result in different packaging of the software
  - It is better to process this class of features at compile time for efficiency reasons (time and space)
- Load-time features:
  - Features that are selected or defined at the beginning of execution but remain stable during the execution.
  - Examples of this class of features are the features related to the operating environment (e.g., terminal types)
- Runtime features:
  - Features that can be changed interactively or automatically during execution

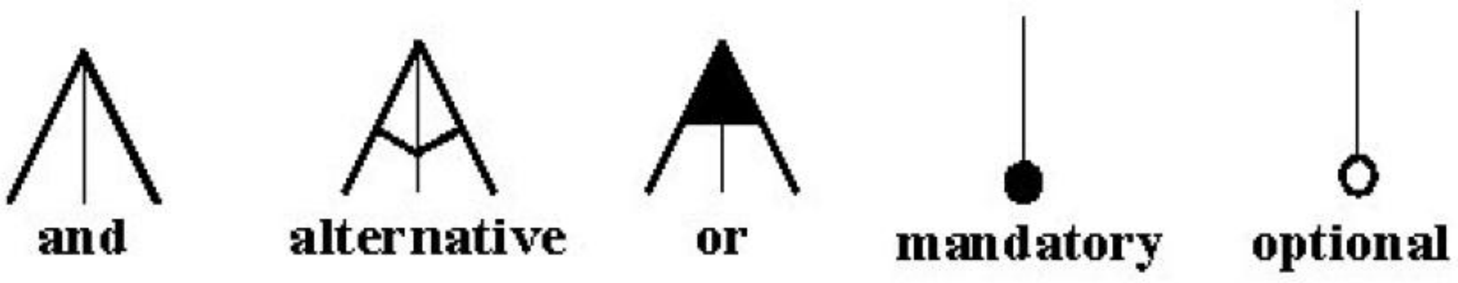
Feature Models, Grammars, and Propositional Formulas

Don Batory,

SPLC 2005

Feature Model

- A feature model is a hierarchically arranged set of features.
- Relationships between a parent (or *compound*) feature and its child features (or subfeatures) are categorized as:
  - *And:* all subfeatures must be selected
  - *Alternative:* only one subfeature can be selected
  - *Or:* one or more can be selected
  - *Mandatory:* features that are required
  - *Optional:* features that are optional.



Example

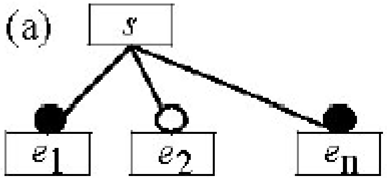
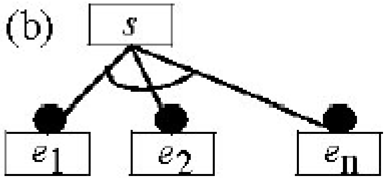
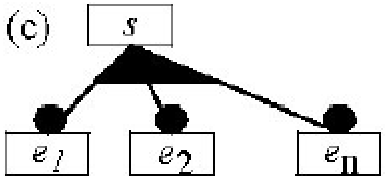




Fig. 3. Parent-Child Relationships in FDs

FeaturePlugin: Feature Modeling Plug-In for Eclipse

Michał Antkiewicz, Krzysztof Czarnecki

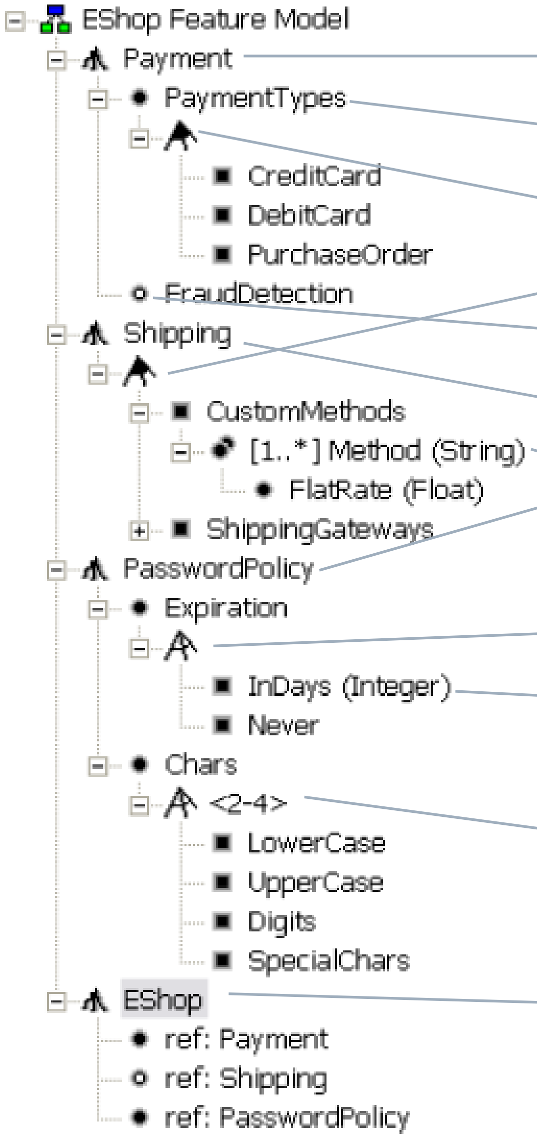
OOPSLA Workshop on Eclipse Technology eXchange, 2004

Introduction

- FeaturePlugin is an Eclipse plug-in for feature modeling
- Feature modeling in an IDE is attractive for two reasons:

1. It helps to optimally support modeling variability in different artifacts, including implementation code, models, documentation, development process guidance, languages, libraries, and more.
2. FeaturePlugin uses the Eclipse Modeling Framework (EMF), which significantly reduced our development effort. This was possible because feature modeling follows a tree structure.

EShop Feature Model



```
EShop Feature Model
  Payment
    ● PaymentTypes
        ■ CreditCard
        ■ DebitCard
        ■ PurchaseOrder
    ○ FraudDetection
  Shipping
      ■ CustomMethods
        [1..*] Method (String)
          ● FlatRate (Float)
      ■ ShippingGateways
  PasswordPolicy
    ● Expiration
        ■ InDays (Integer)
        ■ Never
    ● Chars
      <2-4>
        ■ LowerCase
        ■ UpperCase
        ■ Digits
        ■ SpecialChars
  EShop
    ● ref: Payment
    ○ ref: Shipping
    ● ref: PasswordPolicy
```

root feature

cardinality [1..1] (mandatory)

cardinality [1..k], k is the group size

cardinality [0..1] (optional)

root feature

method can be selected na unbounded number of times

cardinality [1,1]

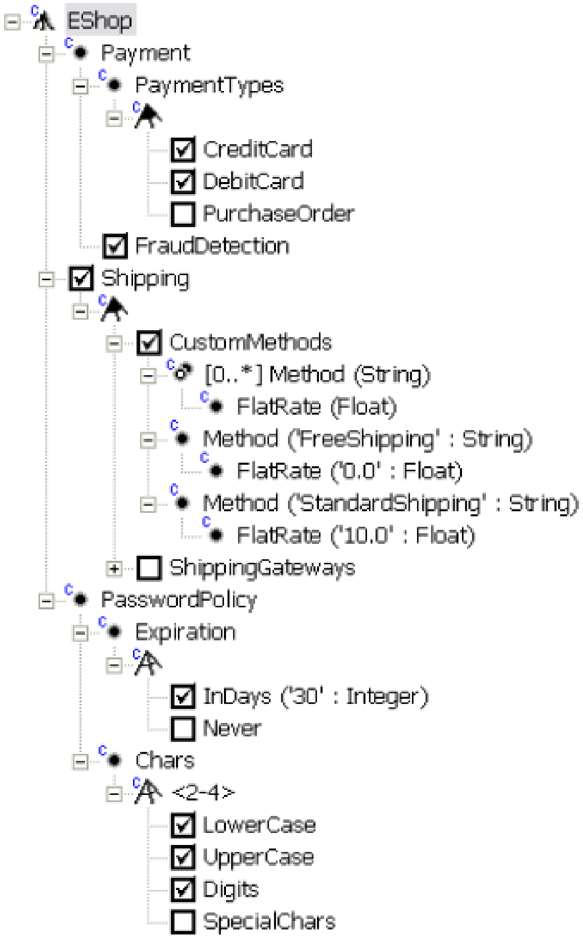
feature attribute (of type integer)

at least two differente char types must be used

E-Shop has references to 3 diagrams
ref:shipping is opcional

Feature-based Configurattion

- Configuration is the process of <u>deriving a concrete configuration</u> conforming to a feature diagram by selecting and cloning features, and specifying attribute values



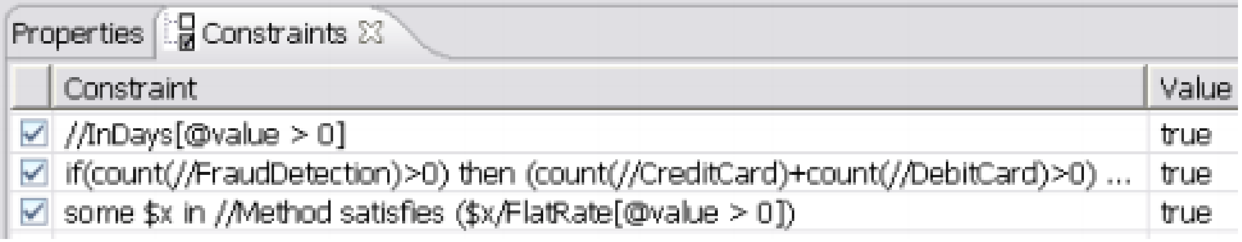
- EShop
  - Payment
    - PaymentTypes
      - ☑ CreditCard
      - ☑ DebitCard
      - □ PurchaseOrder
    - ☑ FraudDetection
  - ☑ Shipping
    - ☑ CustomMethods
      - [0..*] Method (String)
        - FlatRate (Float)
      - Method ('FreeShipping' : String)
        - FlatRate ('0.0' : Float)
      - Method ('StandardShipping' : String)
        - FlatRate ('10.0' : Float)
    - □ ShippingGateways
  - PasswordPolicy
    - Expiration
      - ☑ InDays ('30' : Integer)
      - □ Never
    - Chars
      - <2-4>
        - ☑ LowerCase
        - ☑ UpperCase
        - ☑ Digits
        - □ SpecialChars

Method was cloned two times

Attribute values for the two clones of Method and FlatRate as well as for InDays were specified.

The resulting configuration can be used as input to code generators or it can be accessed by a system as a runtime configuration.

Constraints



	Constraint	Value
☑	//InDays[@value > 0]	true
☑	if(count(//FraudDetection)>0) then (count(//CreditCard)+count(//DebitCard)>0) ...	true
☑	some $x in //Method satisfies ($x/FlatRate[@value > 0])	true

- Expressed using Xpath
- #1: InDays is positive
- #2: selecting FraudDetection implies that CreditCard and/or DebitCard are selected
- #3: at least one custom shipping method should have a rate of more than 0

SPL Extraction

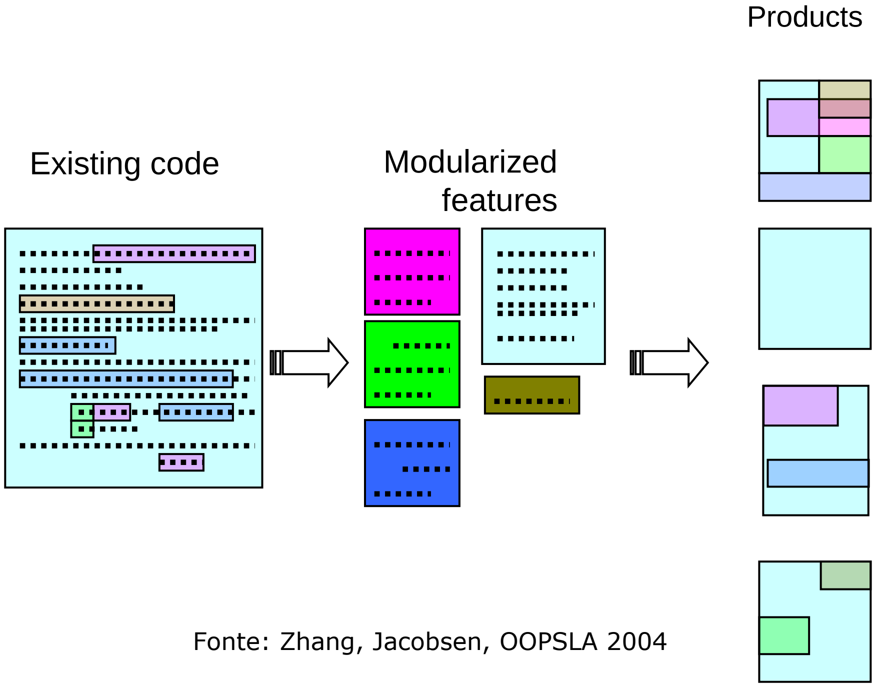
Motivation

- SPL extraction is a time-consuming task

- Two approaches for extracting SPL:
  - Compositional-based
  - Annotation-based

Compositional-based Approaches

- Aspects (AspectJ)



Fonte: Zhang, Jacobsen, OOPSLA 2004

- Physical “modularization”
- Slow adoption

Annotation-based Approaches

- Example: preprocessors

```
 boolean push(Object o) {
   Lock lk = new Lock();
   if (lk.lock() == null) {
      Log.log("lock failed");
      return false;
   }
   elements[top++]= o;
   size++;
   lk.unlock();
   if ((size % 10) == 0)
      snapshot("db");
   if ((size % 100) == 0)
      replicate("db","srv2");
   return true;
}
```

```
boolean push(Object o) {
   #ifdef MULTITHREADING
   Lock lk = new Lock();
   if (lk.lock() == null) {
      #ifdef LOGGING
      Log.log("lock failed");
      #endif
      return false;
   }
   #endif
   elements[top++]= o;
   size++;
   #ifdef MULTITHREADING
   lk.unlock();
   #endif
   #ifdef SNAPSHOT
   if ((size % 10) == 0)
      snapshot("db");
   #endif
   #ifdef REPLICATION
   if ((size % 100) == 0)
      replicate("db","srv2");
   #endif
   return true;
}
```

- Widely adopted
- Problems: annotation hell; code pollution

Visual Annotations

- CIDE: Colored IDE (Eclipse + background colors)

```
boolean push(Object o) {
  Lock lk = new Lock();
  if (lk.lock() == null) {
    Log.log("lock failed");
    return false;
  }
  elements[top++]= o;
  size++;
  lk.unlock();
  if ((size % 10) == 0)
    snapshot("db");
  if ((size % 100) == 0)
    replicate("db","srv2");
  return true;
}
```

```
boolean push(Object o) {
  Lock lk = new Lock();
  if (lk.lock() == null) {
    Log.log("lock failed");
    return false;
  }
  elements[top++]= o;
  size++;
  lk.unlock();
  if ((size % 10) == 0)
    snapshot("db");
  if ((size % 100) == 0)
    replicate("db","srv2");
  return true;
}
```

- Less code poluttion than #ifdefs
- Problem: colors assigned manually (repetitive, error-prone etc)

Extracting Software Product Lines: A Case Study Using Conditional Compilation

Marcus Vinícius Couto
Marco Tulio Valente
Eduardo Figueiredo

15th CSMR - March, 2011 – Oldenburg, Germany

Software Product Lines

- Goal: variable software systems
- Systems: core components + features components
- Product: core + specific set of features

Motivation

- Several papers about SPLs
  - Google Scholar: allintitle: "software product lines“ → 868 papers
- Most reported public, source-code-based SPLs are trivial systems
- Examples:
  - Expression Product Line (2 KLOC), Graph Product Line (2 KLOC), Mobile Media Product Line (4 KLOC)
- Our claim:
  - SPL targets reuse-in-the-large
  - To assess SPL-based technology, we need large systems, with complex features

Our Solution: ArgoUML-SPL

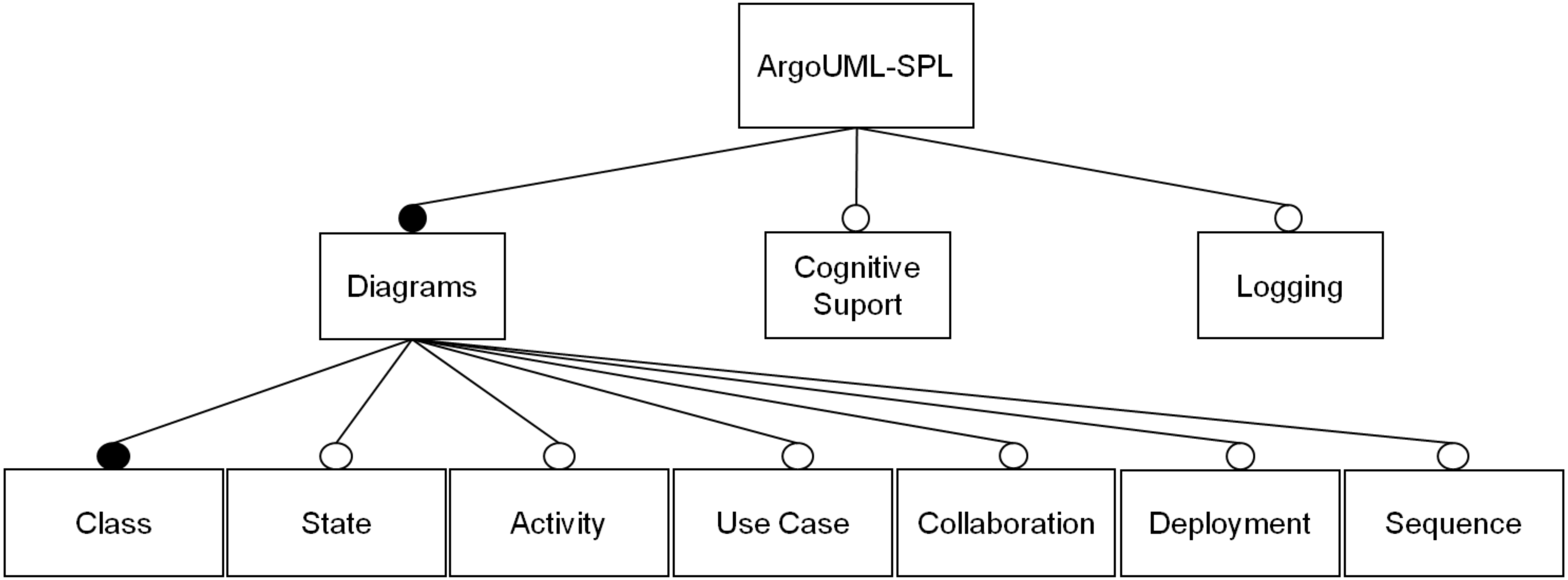
- We decided to extract our own -- complex and real -- SPL
- Target system: ArgoUML modelling tool (120 KLOC)
- Eight features (37 KLOC ~ 31%)
- Technology: conditional compilation
- Baseline for comparison with tools (e.g. CIDE+) and languages (e.g. aspects) for SPL implementation

In this CSMR Paper/Talk

- We report our experience extracting a SPL for ArgoUML
  - ArgoUML-SPL
  - Extraction Process
  - Characterization of the Extracted SPL

ArgoUML-SPL

Feature Model



Feature Selection Criteria

- Relevance:
  - Typical functional requirements (diagrams)
  - Typical non-functional concern (logging)
  - Typical optional feature (cognitive support)

- Complexity:
  - Size
  - Crosscutting behavior (e.g. logging)
  - Feature tangling
  - Feature nesting

Extraction Process

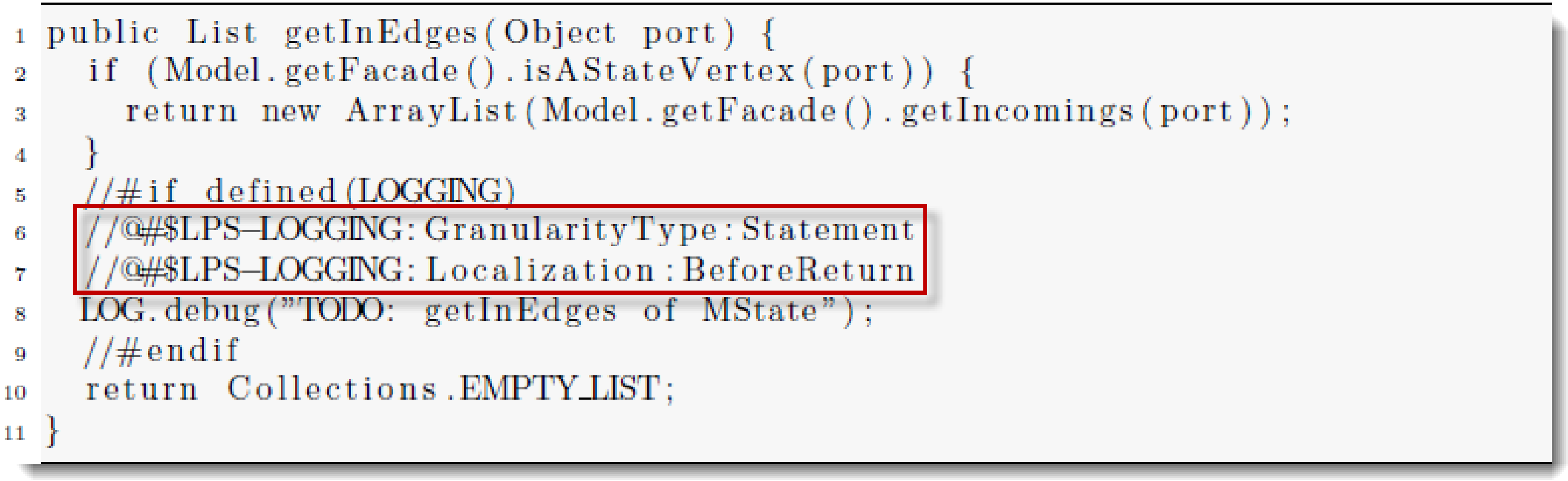
Extraction Process

- Pre-processor: `javapp`
  - http://www.slashdev.ca/javapp

- Extraction Process:
  - ArgoUML’s documentation:
    - Search for components that implement a given feature
    - E.g.: package org.argouml.cognitive
  - Eclipse Search:
    - Search for lines of code that reference such components
  - Delimit such lines with #ifdefs and #endifs

- Effort:
  - 180 hours for annotating the code
  - 40 hours for testing the various products

Example



```
public List getInEdges(Object port) {
  if (Model.getFacade().isAStateVertex(port)) {
    return new ArrayList(Model.getFacade().getIncomings(port));
  }
  //#if defined(LOGGING)
  //@#$LPS–LOGGING:GranularityType:Statement
  //@#$LPS–LOGGING:Localization:BeforeReturn
  LOG.debug(”TODO: getInEdges of MState”);
  //#endif
  return Collections.EMPTY_LIST;
}
```

Characterization

Metrics

- Metric-suite proposed by Liebig et al. [ICSE 2010]

- Four types of metrics:
  A. Size
  B. Crosscutting
  C. Granularity
  D. Location

(A) Size Metrics

- How many LOC have you annotated for each feature?
- How many packages?
- How many classes?

Size Metrics

Product	LOC	NOP	NOC
Original, non-SPL based	120,348	81	1,666
Only Cognitive Support disabled	104,029	73	1,451
Only Activity Diagram disabled	118,066	79	1,648
Only State Diagram disabled	116,431	81	1,631
Only Collaboration Diagram disabled	118,769	79	1,647
Only Sequence Diagram disabled	114,969	77	1,608
Only Use Case Diagram disabled	117,636	78	1,625
Only Deployment Diagram disabled	117,201	79	1,633
Only Logging disabled	118,189	81	1,666
All the features disabled	82,924	55	1,243

LOC: Lines of code; NOP: Number of packages; NOC: Number of classes

Size Metrics

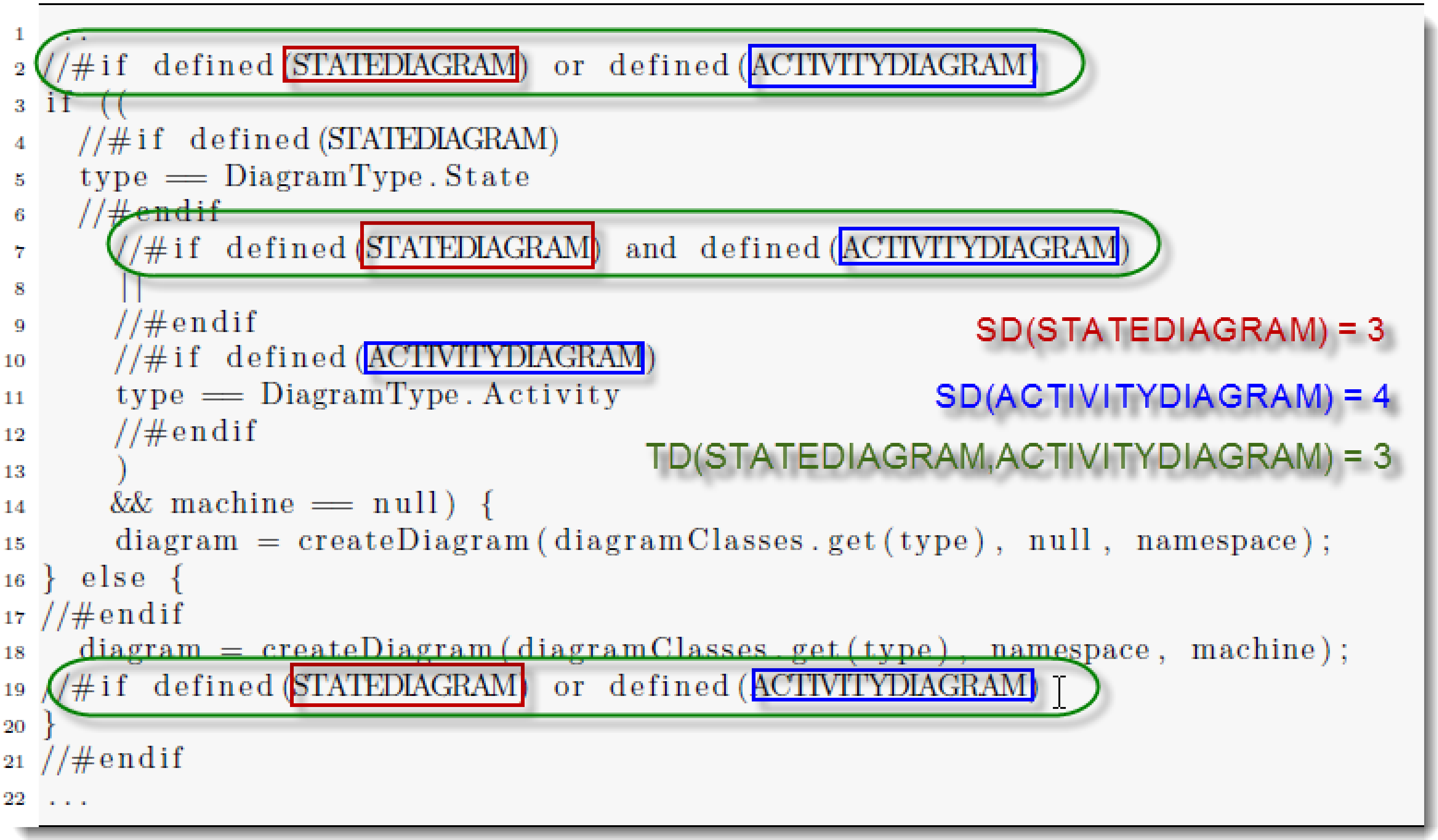
Feature	LOF	
Cognitive Support	16,319	13.59%
Activity Diagram	2,282	1.90%
State Diagram	3,917	3.25%
Collaboration Diagram	1,579	1.31%
Sequence Diagram	5,379	4.47%
Use Case Diagram	2,712	2.25%
Deployment Diagram	3,147	2.61%
Logging	2,159	1.79%
Total	37,424	31.10%

LOF: Lines of Feature code

(B) Crosscutting Metrics

- How are the #ifdefs distributed over the code?
- How many #ifdefs are allocated for each feature?
- Are “boolean expressions” common (e.g. #ifdef A && B)?

Crosscutting Metrics (Example)



```
...
//#if defined(STATEDIAGRAM) or defined(ACTIVITYDIAGRAM)
if ((
  //#if defined(STATEDIAGRAM)
  type == DiagramType.State
  //#endif
    //#if defined(STATEDIAGRAM) and defined(ACTIVITYDIAGRAM)
    ||
    //#endif
    //#if defined(ACTIVITYDIAGRAM)
    type == DiagramType.Activity
    //#endif
    )
   && machine == null) {
    diagram = createDiagram(diagramClasses.get(type), null, namespace);
} else {
//#endif
  diagram = createDiagram(diagramClasses.get(type), namespace, machine);
//#if defined(STATEDIAGRAM) or defined(ACTIVITYDIAGRAM)
}
//#endif
...
```

SD(STATEDIAGRAM) = 3

SD(ACTIVITYDIAGRAM) = 4

TD(STATEDIAGRAM,ACTIVITYDIAGRAM) = 3

SD: Scattering Degree; TD: Tangling Degree

Scattering Degree (SD)

Feature	SD	LOF/SD
Cognitive Support	319	51.16
Activity Diagram	136	16.78
State Diagram	167	23.46
Collaboration Diagram	89	17.74
Sequence Diagram	109	49.35
Use Case Diagram	74	36.65
Deployment Diagram	64	49.17
Logging	1287	1.68

Tangling Degree (TD)

Pairs of Features	TD
(State Diagram, Activity Diagram)	66
(Sequence Diagram, Collaboration Diagram)	25
(Cognitive Support , Sequence Diagram)	1
(Cognitive Support , Deployment Diagram)	13

(C) Granularity Metrics

- What is the granularity of the annotated lines of code?
  - How many full packages have been annotated?
  - And classes?
  - And methods?
  - And just method bodies?
  - And just single statements?
  - And just single expressions?

Granularity Metrics

Feature	Package	Class	Interface Method	Method	Method Body
Cognitive Support	11	8	1	10	5
Activity Diagram	2	31	0	6	6
State Diagram	0	48	0	15	2
Collaboration Diagram	2	8	0	5	3
Sequence Diagram	4	5	0	1	3
Use Case Diagram	3	1	0	1	0
Deployment Diagram	2	14	0	0	0
Logging	0	0	0	3	15

Granularity Metrics

Feature	ClassSignature	Statement	Attribute	Expression
Cognitive Support	2	49	3	2
Activity Diagram	0	59	2	6
State Diagram	0	22	2	5
Collaboration Diagram	0	40	1	1
Sequence Diagram	0	31	2	3
Use Case Diagram	0	22	1	0
Deployment Diagram	0	13	1	3
Logging	0	789	241	1

(D) Localization Metrics

- Where are the #ifdefs located?
  - In the beginning of a method
  - In the end of a method
  - Before a return statement

- Important for example to evaluate a migration to composition-based approaches (e.g. aspects)

Localization Metrics

Feature	StartMethod	EndMethod	BeforeReturn	NestedStatement
Cognitive Support	3	5	0	10
Activity Diagram	2	20	2	19
State Diagram	2	19	3	12
Collaboration Diagram	1	10	3	3
Sequence Diagram	0	9	3	7
Use Case Diagram	0	2	0	1
Deployment Diagram	0	0	0	3
Logging	127	21	89	336

Conclusions

Importance

- What´s the importance of a “realistic” PL like ArgoUML?
  - SPL targets reuse-in-the-large
  - Evaluating SPL tools and languages only in “small scenarios” can lead to misleading conclusions