Generative Programming
Methods, Tools, and Applications

Dr. Krzysztof Czarnecki
DaimlerChrysler Research and Technology
Software Technology Lab, Ulm, Germany
czarnecki@acm.org

Prof. Dr. Ulrich W. Eisenecker
University of Applied Sciences Kaiserslautern,
Zweibrücken, Germany
ulrich.eisenecker@t-online.de

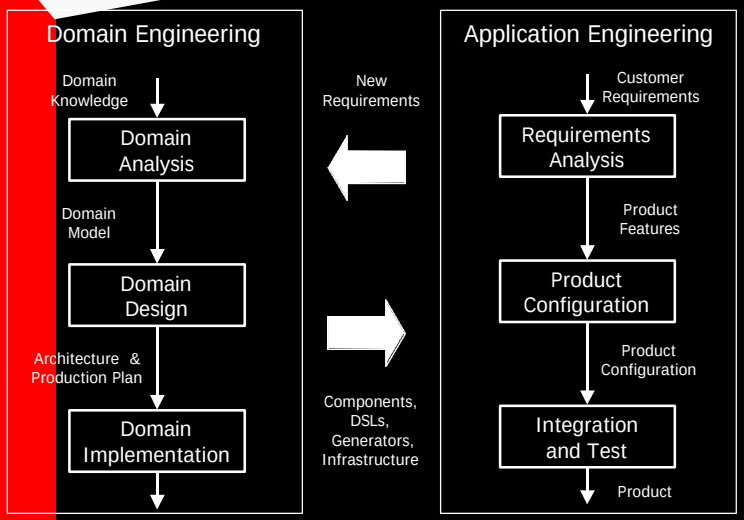
Single System Development

- Focus on analyzing, modeling, and implementing a single system
- Important goal: decomposition into components, i.e. modularization
  - Individual components are easier to maintain, to replace, and to reuse
  - What are the criteria for decomposition?

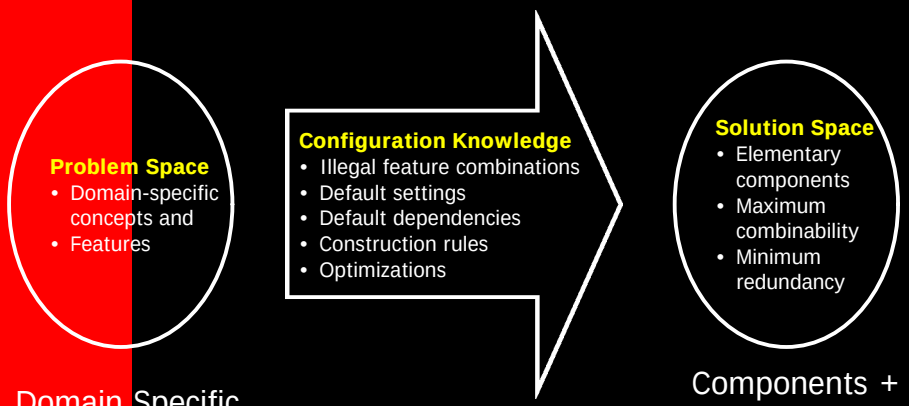
System Family Approach

- Domain Engineering
  - analysis: scoping, common and variable features, feature dependencies (feature diagrams)
  - design: common architecture for the system family and a production plan
  - implementation: reusable components, domain-specific languages (DSLs), configuration generators
- Application Engineering
  - production of concrete, highly customized systems and components using the above-mentioned results

System Family Approach



Generative Domain Model



5

Generative Programming ...

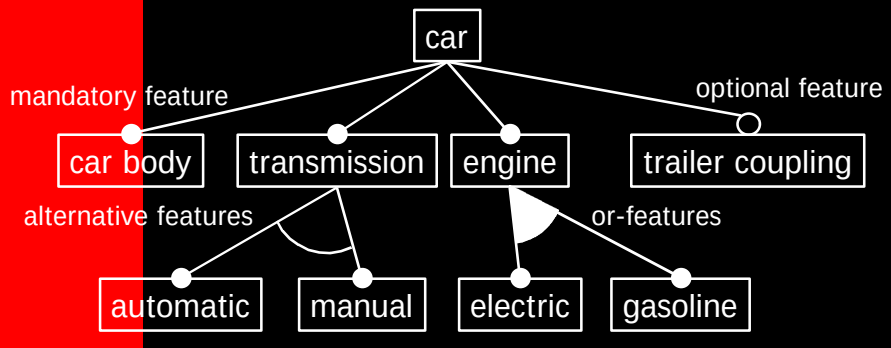
... is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.

6

Domain Engineering: Analysis

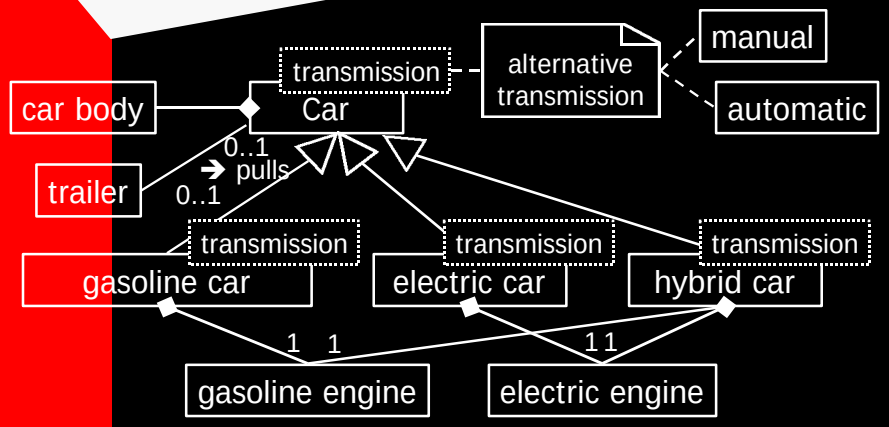
- Stakeholder analysis
- Domain scoping
- Analysis of common and variable features and their dependencies
- Documentation using feature models (FODA)
  - Feature diagrams representing commonalities and variabilities in a implementation-independent and easy to understand manner
  - Tables and free text documenting feature dependencies, short descriptions, origin, rationale, defaults, binding time, usage examples, etc.

Feature Diagram Example



No decision regarding the mechanism for implementing variability!

To Compare: a UML Class Diagram



Variability is implemented using concrete mechanisms, e.g. inheritance, aggregation, associations, and templates

9

Feature Combinations

- The feature diagram contains
  2 (transmission)
  x 3 (engine)
  x 2 (trailer coupling)
  = 12 combinations
- Constraint (separately documented): electric engine requires automatic transmission

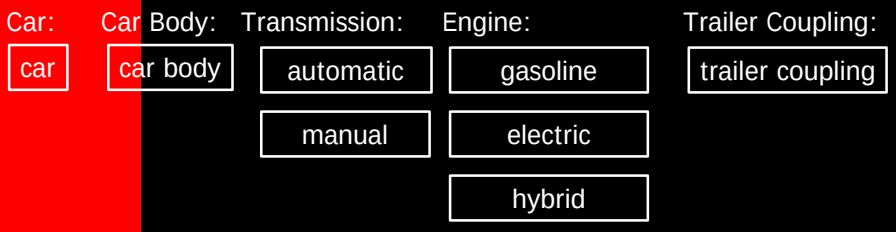
➡ i.e., we have 8 valid combinations

10

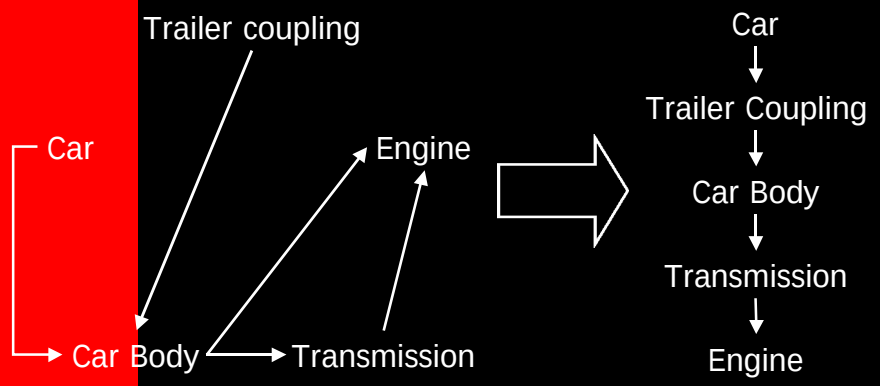
Domain Engineering: Design

- Identifying categories of implementation components
- Dependency analysis
- Architecture for the system family
  - GenVoca layered architecture
  - GenVoca grammar

Implementation Components



Dependency Analysis



Feature Model

- ➔ Background
- Feature Model
- Feature Diagram
- Feature Types
- Normalized Feature Diagrams
- Commonality and Variability
- FAQs
- Feature Diagram "Personal Account"

Concepts and Classes

Aren't these equations generally true?

- Concept = class
- Object = instance of a concept

... No!

- Objects have predefined semantic properties such as identity, state, behavior
- Instances of concepts do not have any predefined semantics; they could be virtually anything
- There are so many variants of instances and concepts that they can't be described using a single class only

15

Feature Modeling

- Feature modeling is about identifying
  - common and
  - variable features of concepts and their
  - dependencies

  and documenting them in a coherent model, called a feature model
- Feature modeling is a core activity of important domain-engineering methods, such as ODM and FODA

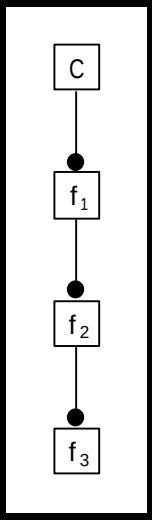
16

A Feature Model Consists of

- Feature diagram(s) and
- additional information
  - Short semantic description of each feature
  - Rationale for the relevance of each feature
  - Stakeholders and systems that are interested in a feature
  - Examples of systems with that feature
  - Restrictions
  - Dependencies and default values
  - Availability (where, when, and to whom is the feature available?)
  - Binding (where, when, and who binds the feature?)
  - Binding mode (dynamic, static)
  - Open/closed (is it possible to add sub-features later?)
  - Priority (how important is the feature?)

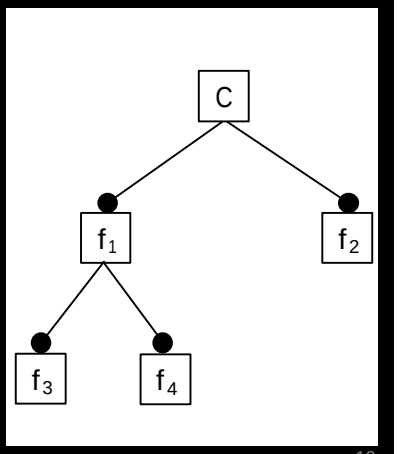
Feature Diagram

- The root node C represents the concept C
- All other nodes are features
- f_1 is a direct feature of C
- f_2 and f_3 are indirect features of C
- f_2 is a direct subfeature of f_1
- f_3 is an indirect subfeature of f_1



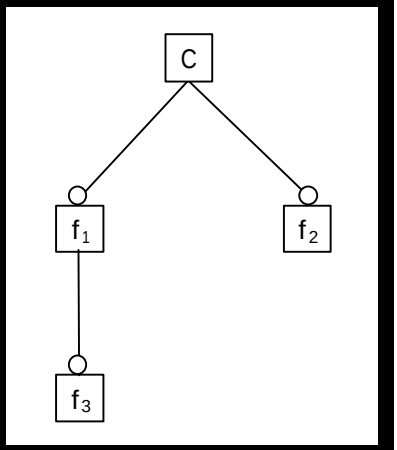
Mandatory Feature

- A mandatory feature is part of a concept instance description only if its parent is also part of the description
- Mandatory features are pointed to by edges with a filled circle (e.g., f_1,f_2,f_3, and f_4)
- All instances of C are described by the feature set {C,f_1,f_2,f_3,f_4}



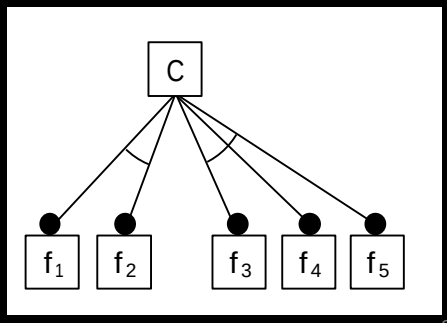
Optional Feature

- An optional feature can be part of a concept instance description only if the parent node is also part of the description
- Optional features are pointed to by edges with an empty circle (e.g., f_1,f_2, and f_3)
- The following sets describe instances of C: {C}, {C,f_1}, {C,f_1,f_3}, {C,f_2}, {C,f_1,f_2}, {C,f_1,f_3,f_2}



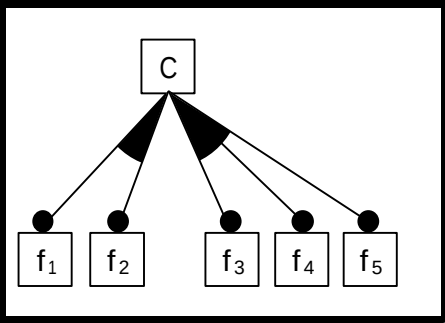
Alternative Features

- Exactly one from a set of alternative features is part of a concept instance description if its parent node is also part of the description
- Edges pointing to alternative features of one set are connected by an empty arc
- The following sets describe instances of C:
  $\{C,f_1,f_3\}$, $\{C,f_1,f_4\}$, $\{C,f_1,f_5\}$, $\{C,f_2,f_3\}$, $\{C,f_2,f_4\}$, $\{C,f_2,f_5\}$



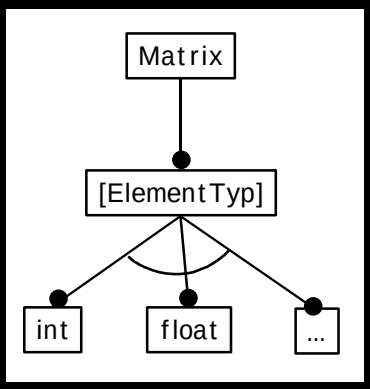
Or-Features

- Any non-empty subset from a set of optional features can be part of a concept instance description if the parent node is also part of it
- Edges pointing to or-features of one set are connected by a filled arc
- The diagram denotes 21 different concept instances



Open/Closed Features

- An open feature is expected to be refined with further subfeatures
- In a feature diagram, brackets [] are used to indicate openness
- We can also show selected examples of subfeatures

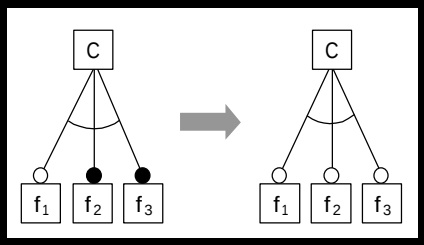


Normalized Feature Diagrams

- Besides mandatory, optional, and or-features, there are also
  - optional alternative features and
  - optional or-features
- This results in redundant representations
- Each feature diagram can be transformed such
  - that it does not contain any optional or-features
  - and any set of alternative features contains either alternative or optional-alternative features only

Normalized Feature Diagrams

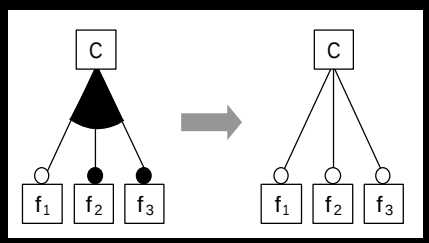
If one or more of the features in a set of alternative features is optional, it has the same effect as if all the alternative features in this set were optional.



25

Normalized Feature Diagrams

If one or more of the features in a set of or-features is optional, it has the same effect as if all the features in this set were optional features.



26

Variability in Feature Diagrams

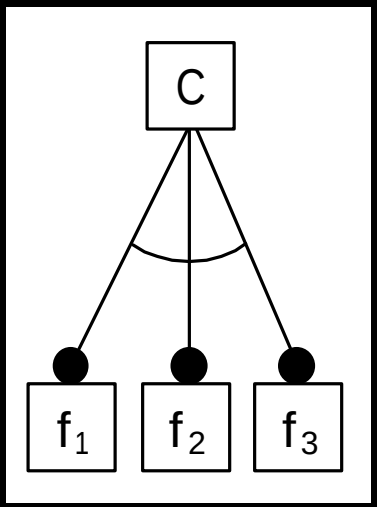
- Variable features describe variability
  - optional, alternative, optional-alternative, and or-features
- The nodes having variable subfeatures are so-called *variation points* (VP)
  - homogeneous vs. inhomogeneous VPs: all direct subfeatures are of the same vs. different kinds
  - singular vs. nonsingular VPs: none or one vs. many direct subfeatures can be selected

27

Kinds of Variation Points

Dimension

- Feature or concept with all direct subfeatures or features being alternative features only

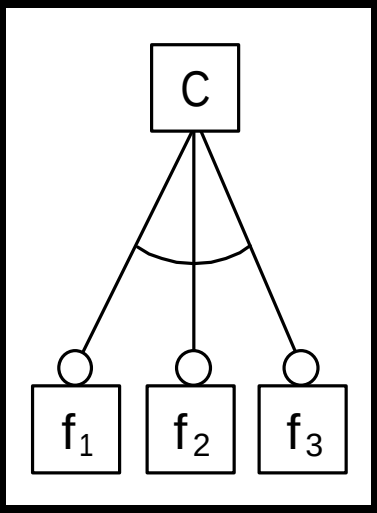


28

Kinds of Variation Points

Dimension with optional features

- Feature or concept with direct subfeatures or features being alternative-optional features only

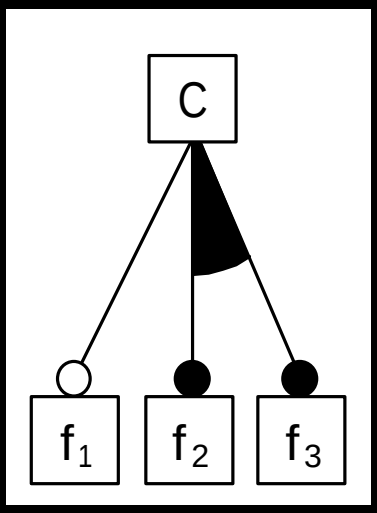


29

Kinds of Variation Points

Extension point

- Concept or feature
  - with at least one set of direct or-features or or-subfeature
  - with at least one direct optional feature or subfeature

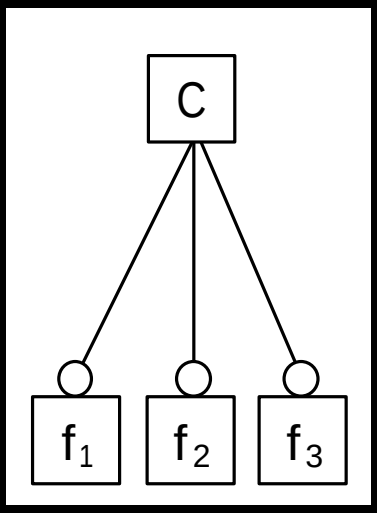


30

Kinds of Variation Points

Extension point with optional features

- Concept or feature with all direct features or subfeatures being optional features only

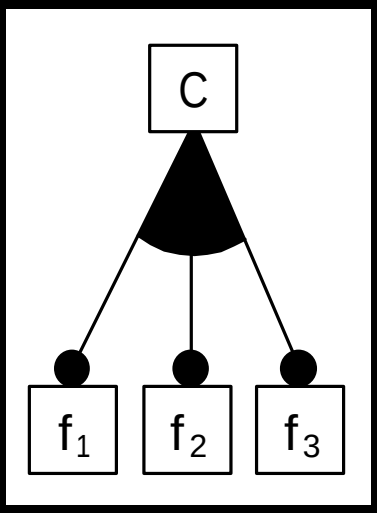


31

Kinds of Variation Points

Extension point with or-features

- Concept or feature with all direct features or subfeatures being or-features only



32

FAQs

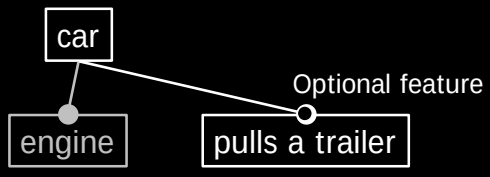
What are the semantics of the edges of a feature diagram?

- Edges don‘t have their own full semantics
- The decoration of the edges completes the semantic: a certain feature must, can, or can not be part of a concept instance description
- Edges don‘t represent part-of or inheritance relationships!
- Edges don‘t have cardinalities!

FAQs

How to describe the configurability aspect of a relation if edges don‘t possess relationship semantics?

- If a feature diagram contains a variable relation it should be modeled as a feature
- For example, the option "pulls a trailer" is represented as an optional feature



FAQs

Isn't a feature diagram just a simple part-of hierarchy?

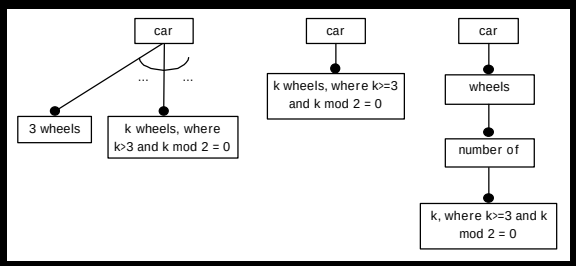
- No; features can represent part-of relations, but they can carry any other meaning, too
- Features can be implemented quite differently, for example
  - concrete features as concrete components
  - aspectual features, e.g. synchronization, are often woven into other components
  - abstract features, such as memory usage optimization, determine the configuration of other components

35

FAQs

Wouldn't it be useful to include cardinalities in a feature diagram?

- No; if the feature "wheel" would be annotated with the cardinality of 4, this would only mean to assert the feature "wheel" for the concept "car" 4 times
- Important cardinalities should be modeled as features



36

FAQs

A feature diagram forms a tree - isn't that too restrictive?

- No; a feature diagrams shows only the most important dependencies between features
- Dependencies that can't be explicitly represented, e.g. constraints between variable features from different locations in the diagram, are documented separately
- Being strongly hierarchical makes feature diagrams easier to understand

Dependency Analysis

- ➔Goal
- Identifying Implementation Components
- Defining Equivalence Categories
- Analyzing Dependencies
- Results
- Notation
- Dependency Analysis of "Personal Account"

Goal

- Dependency analysis identifies the dependencies between implementation components
- It is the basis for defining a common system family architecture
- For performing dependency analysis the implementation components have to be organized in so-called "equivalence categories"
- Doing this does not require the actual implementation of implementation components!

From Features to Components

- Feature diagrams is the primary source for identifying implementation components
- Please remember
  - concrete features are implemented as concrete components
  - aspectual features, e.g. synchronization, are often woven into other components
  - abstract features, such as memory optimization, determine the configuration of other components

From Features to Components

- A concrete feature is usually implemented by one concrete component
- Depending on the kind of an aspectual feature, we have various options for implementing them
  - The aspect can be encapsulated in an aspectual component
  - Each concrete implementation component requires its specific aspectual component; commonalities can be captured in one aspectual component
- Abstract features contribute to the configuration knowledge

Equivalence Categories

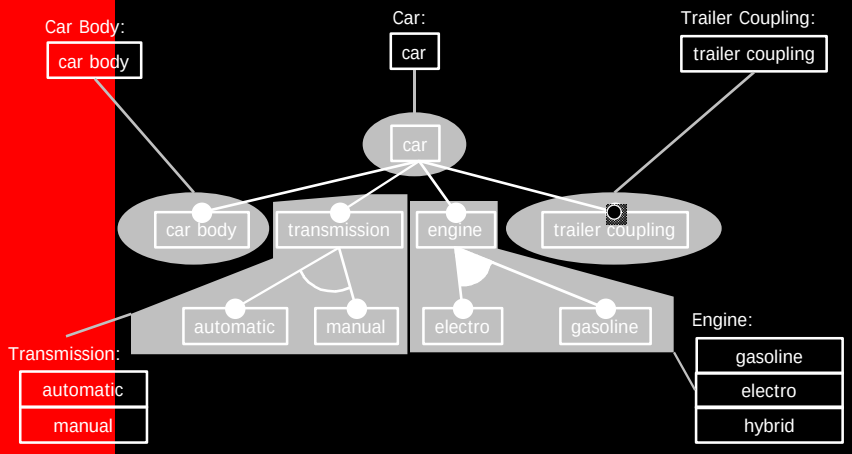
- The next step is to define equivalence categories
- An equivalence category comprises features that are mapped to implementation components which implement a common category interface
- The dependency analysis is then done at the level of equivalence categories
  - More efficient that examining each implementation component
  - Nevertheless component specific dependencies can be captured; they are propagated up to the equivalence category level

Equivalence Category

The usual procedure is as follows

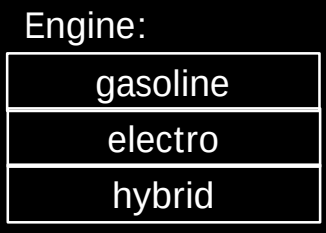
- A stand-alone concrete feature or subfeature forms an equivalence category
- Concrete alternative features or subfeatures form an equivalence category
- Concrete or-features or subfeatures form an equivalence category

Equivalence Categories



Implementation Components

- At this stage the concrete implementation of the components it still not relevant
  - The components can be implemented arbitrarily
  - Hybrid engine could inherit from electro- and gasoline-engine
  - Hybrid engine could aggregate electro- and gasoline engine and forward messages to them



Analyzing Dependencies

These questions drive the identification of dependencies

- Do components of one equivalence category use components of another category?
- Do components of one equivalence category parameterize those of another category?
- Does the assembly of one component of an equivalence category require that a component of another category is already present?

Cyclic Dependencies

- Cyclic dependencies may emerge during dependency analysis
  - In the simplest case, components of two equivalence categories are mutually dependent
- Cyclic dependencies will be broken at first
  - More or less arbitrarily or driven by experience
  - Mutually dependent components will become part of the configuration repository (more about this later)

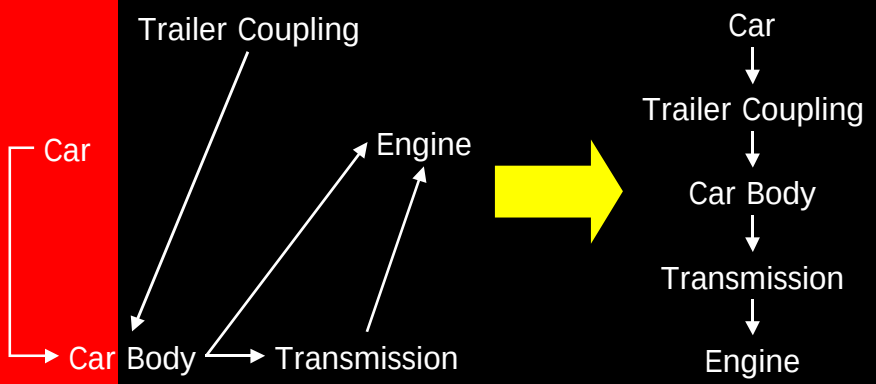
Results of Dependency Analysis

- Equivalence categories are sorted hierarchically according to the identified dependencies
- The equivalence category without dependencies go to the bottom (later, we'll add the configuration repository at the very bottom)
- Then we successively put the remaining categories on top of the hierarchy such that each category mostly depends on those below it
- The equivalence category with the most dependencies will be at the top

Notes

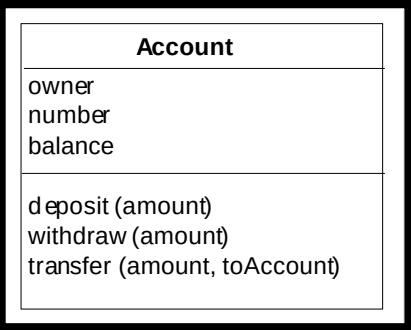
- The dependency analysis must not necessarily yield a linear sequence
- It may also form a tree
- Cyclic dependencies will be handled by the configuration repository

Example



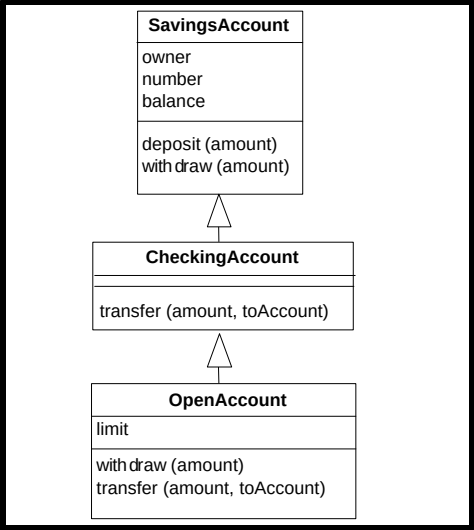
Example "Personal Account"

Simple account class
(trivial textbook example)



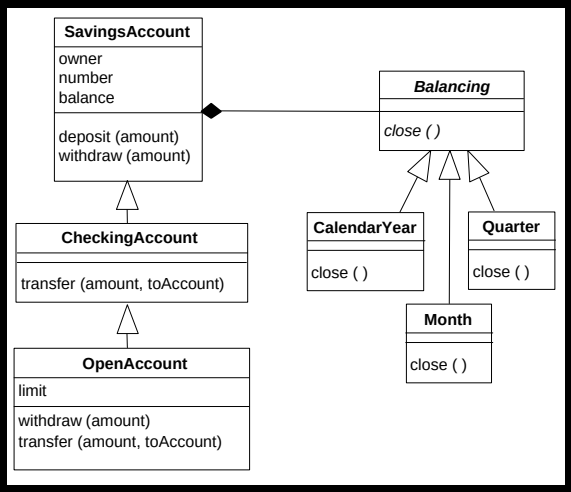
Example "Personal Account"

Distinction
between
- savings account
- accounts
  for cashless
  payments
  (checking
  account,
  open account)

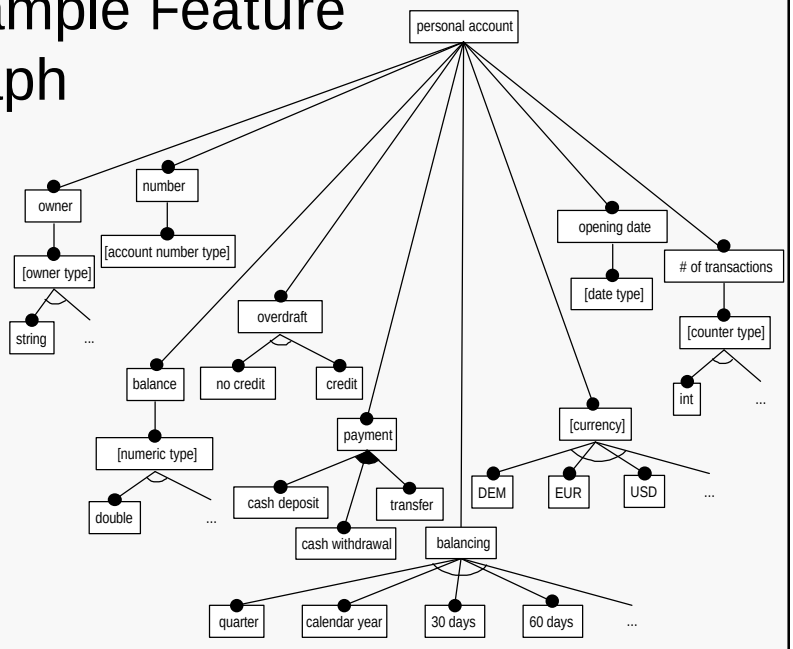


Example "Personal Account"

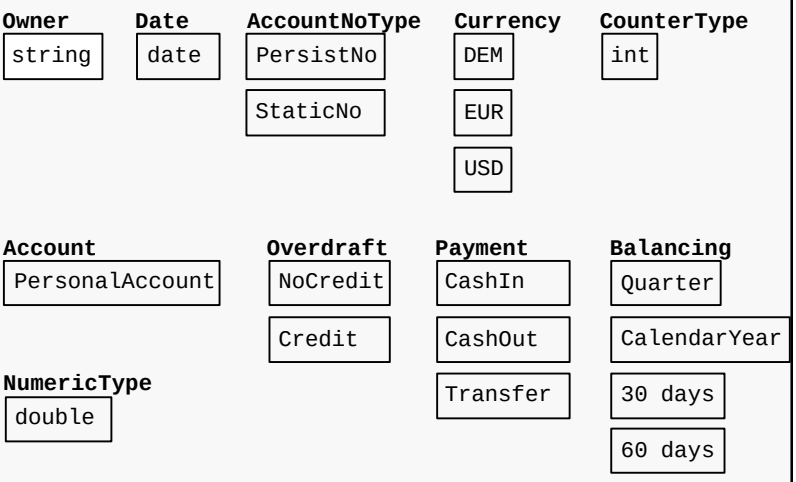
Adding different balancing periods (and many more variation points are still missing!)



Example Feature Graph



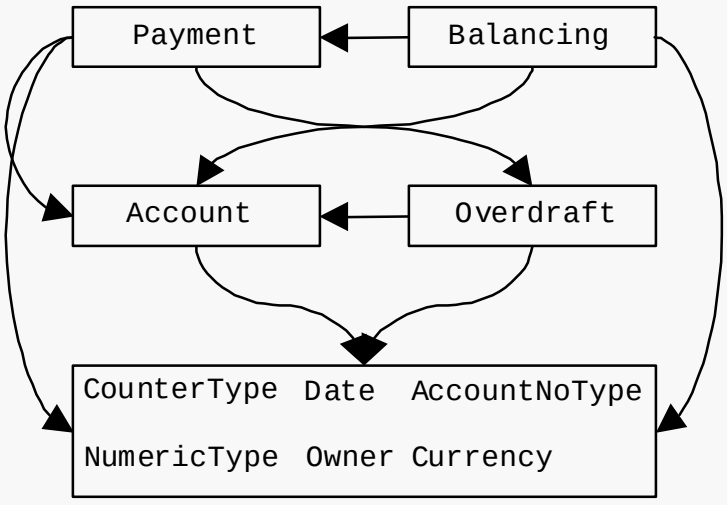
Implementation Components



Analyzing Dependencies

According to the feature diagram, every account has an owner, an opening date, a number, and a balance. Furthermore, it keeps track of the number of transactions and its currency. There are no dependencies between these basic categories. But all other categories depend on them. Therefore, we collect the basic categories into a configuration repository. Overdraft, payment, and balancing all need an account to do anything useful. Thus, they all depend on `Account`. Whether you can perform a payment or not, depends on overdraft, which results in an arrow from `Payment` to `Overdraft`. `Payment` keeps track of each transaction, and `Balancing` needs this information. Therefore, `Balancing` has access to `Payment`.

Documenting Dependencies



Dependency Hierarchy