Generative Programming in C++
Methods, Tools, and Applications

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

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

Overview

An Appetizer

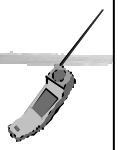
Motivation

In-Depth Example

Motivation

➔ Challenges
- Single System Engineering
- System Family Engineering
- Programming Languages
- Extendible Programming Environments

Complexity



- Software systems are deployed for new application domains
- Software systems support increasingly complex tasks
- Requirements concerning functionality and efficiency are constantly growing

Quality



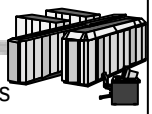
- Software is deployed in life-critical application areas
- Software malfunction causes economic losses
- Usability is a major acceptance criterion
- Users expect a specific quality that is determined by varying criteria

Productivity



- Growing demand for new systems and variants
- Increase of productivity and development staff cannot close the gap between demand and supply
- Standard software becomes more and more important, but it cannot be adapted to every specific requirement

Maintenance & Legacy Systems



- Long lifetime of software systems (sometimes 20 years and more)
- Software must be continuously adapted to new and changing requirements
- Maintenance of legacy systems absorbs development capacity to a high degree
- Software systems of today are the legacy of tomorrow!

Example "Object Technology"

- Theory
  - Productivity and quality should increase thanks to reuse
- Reality
  - Classes are too small as units of reuse
  - Frameworks are sufficiently large as units of reuse, but frameworks from different vendors do not integrate well
  - Design patterns are pieces of reusable knowledge, but they do not exist as compilable or executable code

Example "Component Technology"

- Standards and platforms improve interoperability and the distributed execution of components
- Many components provide a special mode for customization
- Reusing small components does not have a large impact on software development while large components require high customization efforts

Causes

- Currently dominating methods and tools are designed for developing *single systems*
- Software development is mostly focused on using *one particular fixed programming language*

Motivation

- Challenges

➔ Single System Engineering

- System Family Engineering
- Programming Languages
- Extendible Programming Environments

Single System Development

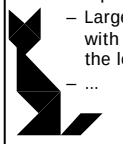
- Focus on analyzing, modeling, and implementing a single system
- Important goal: decomposition into components, i.e. modularization
  - Individual components are easier to maintain, to replace, and to reuse
  - What are the criteria for decomposition?

Motivation

- Challenges
- Single System Engineering
- ➔ System Family Engineering
- Programming Languages
- Extendible Programming Environments

How To Order Systems?

- By Components
  - Downward arrow on top
  - Large triangle below, with its 90° angle at the lower left point
  - ...





- By Specification
  - A cat
  - Sitting upright
  - Waiving its tail



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.

From Order To Product



Creating the order
- Expert language
- Form, catalogue
- By example



Processing the order
- Completion
- Validation
- Construction rules
- Optimization



Product generation
- Assembling elementary, reusable components



Two Processes

- Domain Engineering
  - Analysis: Domain scoping and defining a set of reusable, configurable requirements for the systems in the domain
  - Design: Developing a common architecture for the systems in the domain and devising a production plan
  - Implementation: Implementing the reusable assets, for example, reusable components, domain-specific languages, generators, a reuse infrastructure, and a production process
- Application Engineering
  - producing concrete systems using the reusable assets developed during Domain Engineering.

Motivation

- Challenges
- Single System Engineering
- System Family Engineering
- ➔ Programming Languages
- Extendible Programming Environments

"Ich bin ein Berliner"

Two thousand years ago the proudest boast was "civis Romanus sum." Today, in the world of freedom, the proudest boast is "Ich bin ein Berliner."

John F. Kennedy



Rhetoric

- Alienation by switching between languages
  - English - Latin - English - German
- Historical comparison
  - Two thousand years ago ... Today ...
- Actual comparison
  - ... "civis Romanus sum." ... „Ich bin ein Berliner.“ ...
- Hidden allusion
  - Citizen of the United States of America
- Appraisal
  - ... in the world of freedom ...
- Pleonasm
  - ... proudest boast ...
- Anaphora
  - ... proudest boast ... proudest boast ...

The Computer Science Perspective

- The speaker interlaces three languages
- Different aspects, that is rhetorical figures are used intertwinedly
- Each part in a different language has to be translated in some common representation
- Finally the contents of the different parts can be related and the meaning can be interpreted

Programming Today

- Loss of design information
  - Problems and their solutions are described by means of a programming language and not in an appropriate expert language
  - Domain specific optimizations are note applied and domain specific errors are not diagnosed
- Code-tangling
  - Mixing different aspects in one source
- Evolution of programming languages is hard to achieve
- Different programming languages don't cooperate in compilation

Conclusion

- Generative programming combines languages for
  - specifying systems,
  - analyzing system specifications,
  - implementing components
  - assembling components
- Nowadays programming language only partially serve these different purposes

Motivation

- Challenges
- Single System Engineering
- System Family Engineering
- Programming Languages

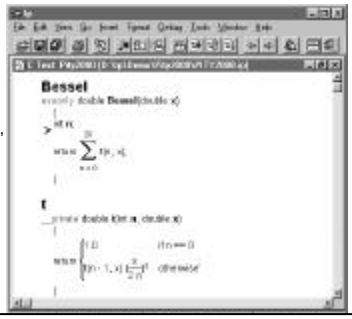
➔ Extendible Programming Environments

Programming Tomorrow

- Modularly extensible programming environments
  - enable the development of active libraries
  - can be extended by loading active libraries
- Active libraries contain domain specific
  - language extensions
  - rendering and editing support
  - optimizations
  - debugging support
  - support for testing, profiling, refactoring, and many more
- Legacy programming languages can be preserved as active libraries and thus allowing further use of legacy systems and their incremental reengineering

Intentional Programming System

- Modularly extensible programming envirnment
- Brainchild of Charles Simonyi, chief-architect at Microsoft Research
- Current status: being developed as a product, *www.research.micosoft.com*



Vision

- Generative programming is not limited to implementing application code, testing, and creating GUIs
- System family engineering also includes
  - documentation
  - system installation
  - user training
  - maintenance, and many more
- Vision for the future: Vendors offering the production of customer-specific components and systems via Internet

Overview

Motivation

An Appetizer

In-Depth Example

An Appetizer

➔ Vision
- Single Systems vs. System Families
- Feature Models
- Implementation Components
- System Architecture
- Generator
- Applications
- New Exciting Technologies
- Summary and Further Information

Development with Components

generating the needed components and automatic assembly

Generative Programming

current practice

searching for and adapting components and manual assembly

The Programmer States ...

... *what* she needs

```
CAR_GENERATOR<EcoLine,TrailerCoupling>::RET
LIST_GENERATOR<Person,cp,poly,with_counter>::RET
ACCOUNT_GENERATOR<credit<>,EUR>::RET
MATRIX_GENERATOR<matrix<double,structure
                 <up_triang<> >,speed<> > >::RET
```

and gets the desired component!

An Appetizer

- Vision
- ➔ Single Systems vs. System Families
- Feature Models
- Implementation Components
- System Architecture
- Generator
- Applications
- New Exciting Technologies
- Summary and Theses

"One of a Kind" Development

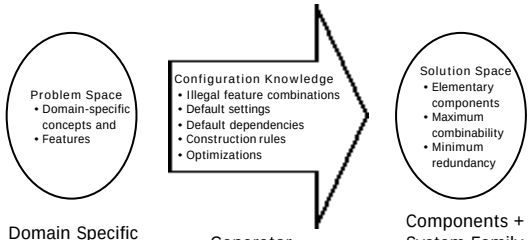
Object-Oriented Technology

- Virtually no support for development for and with reuse
- No domain scoping (domain = the set of existing and potential systems)
- no differentiation between variability within one application and variability between several applications
- no implementation-independent means of variability modeling

System Family Approach

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

Generative Domain Model



Domain Specific Language (DSL)

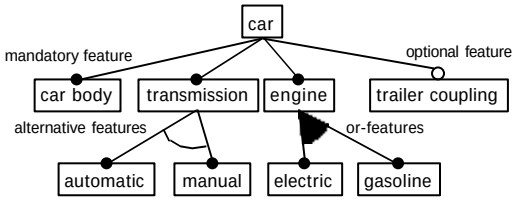
Generator

Components + System Family Architecture

An Appetizer

- Vision
- Single Systems vs. System Families
- ➔ Feature Models
- Implementation Components
- System Architecture
- Generator
- Applications
- New Exciting Technologies
- Summary and Theses

Feature Diagram



No decision regarding the mechanism for implementing variability!

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

An Appetizer

- Vision
- Single Systems vs. System Families
- Feature Models

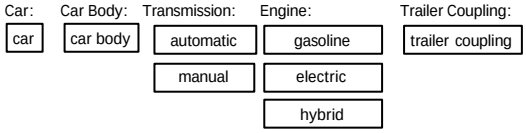
➔ Implementation Components

- System Architecture
- Generator
- Applications
- New Exciting Technologies
- Summary and Theses

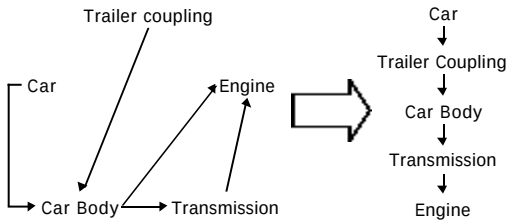
Domain Engineering: Design

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

Implementation Components



Dependency Analysis



An Appetizer

- Vision
- Single Systems vs. System Families
- Feature Models
- Implementation Components

➔ System Architecture

- Generator
- Applications
- New Exciting Technologies
- Summary and Theses

GenVoca Layered Architecture

Car: car
Trailer Coupling: trailer coupling
Car Body: car body
Transmission: automatic | manual
Engine: gasoline | electric | hybrid
Configuration: engine, transmission, car body, car

Home Page of the GenVoca Project,
www.cs.utexas.edu/users/schwartz

GenVoca Grammar

Car:	Car[CarBodyWithOptTC]
CarBodyWithOptTC:	CarBodyWithTC[CompleteCarBody] | CompleteCarBody
CompleteCarBody:	CarBody[TransmissionWithEngine]
TransmissionWithEngine:	ManualTransmission[Engine] | AutomaticTransmission[Engine]
Engine:	GasolineEngine[Config] | ElectricEngine[Config] | HybridEngine[Config]
Config	
Engine:	GasolineEngine | ElectricEngine | HybridEngine
Transmission:	ManualTransmission[Engine] | AutomaticTransmission[Engine]
CarBody:	CarBody[TransmissionWithEngine]
...	

Car Components

```
template<class Config_> struct GasolineEngine
{ typedef Config_ Config;
  GasolineEngine() { cout << "GasolineEngine "; }
};
template < class Engine> struct AutomaticTransmission
{ typedef typename Engine::Config Config;
  enum { speeds = Config::speeds };
  Engine e;
  AutomaticTransmission()  { cout << speeds << "-SpeedAutomatic "; }
};
template < class Transmission> struct CarBody
{ typedef typename Transmission::Config Config;
  Transmission t;
  CarBody() { cout << "CarBody "; }
};
```


Configuration

```
template <class CarBody_>
struct Car
{ typedef typename CarBody_::Config Config;
  CarBody_ cb;
  Car() { cout << "Car " << endl; }
};
struct C1
{ enum { speeds = 5 };
  typedef GasolineEngine <C1> Engine;
  typedef ManualTransmission<Engine> Transmission;
  typedef CarBody<Transmission> CarBody;
  typedef Car<CarBody> Car;
};
```


Manual Configuration

- Manually writing 12 configuration repositories is tedious and error prone
- Only 8 configuration repositories are valid
- How to automatically produce valid configurations?

➔Configuration generators

An Appetizer

- Vision
- Single Systems vs. System Families
- Feature Models
- Implementation Components
- System Architecture
- ➔ Generator
- Applications
- New Exciting Technologies
- Summary and Theses

Ordering Cars

According to the catalog, ordering invalid configurations is not possible (e.g. electric engine with manual transmission)

	BaseLine	CityLine	EcoLine
Engine			
gasoline	●		
electric		●	
hybrid			●
Transmission			
five-speed manual	●		
four-speed automatic	❍	●	●
Options			
trailer coupling	❍	❍	❍

● standard equipment
❍ options available for a surcharge

Domain Specific Language

```
// implementation of the catalog in a domain-specific
// language (DSL)
enum Transmission { fiveSpeed, fourSpeedAutomatic };
enum Line { baseline, cityline, ecoline }; //implementation detail

template <int transmission_ = fiveSpeed>
struct BaseLine
{   enum {   transmission = transmission_,
             line = baseline
         };
};
// ...
enum Options    { none, trailerCoupling };
```


Configuration Generator

```
// specification in a domain-specific language
template <class Line = BaseLine< >,int options_ = none>
struct CAR_GENERATOR
{   // declare a shorthand for the generator
    typedef CAR_GENERATOR<Line,options_> Generator;
    // parse the specification
    enum {   line_ = Line::line,
             transmission_ = Line::transmission
         };
```


Configuration Generator

```
// buildability not checked; DSL gives no chance to make a mistake
 enum { speeds_ = (transmission_ == fiveSpeed) ? 5 : 4 };
 typedef SWITCH<line_,
          CASE<baseline,GasolineEngine<Generator>,
          CASE< cityline,ElectricEngine<Generator>,
          CASE< ecoline,HybridEngine<Generator> > > > >::RET Engine_;
 typedef IF<(transmission_ == fiveSpeed),
          ManualTransmission<Engine_>,
          AutomaticTransmission<Engine_> >::RET Transmission_;
 typedef IF<(options_ == trailerCoupling),
          CarBodyWithTC<CarBody<Transmission_> >,
          CarBody<Transmission_> >::RET CarBody_;
 typedef Car<CarBody_> RET; // complete car type
```


Template Metaprogramming

- C++ templates define a Turing-complete, functional language executed at compile time (➔ Erwin Unruh)
- C-like control structures IF, SWITCH, DO, WHILE, FOR available
- Czarnecki, Eisenecker: Generative Programming - Methods, Techniques, and Applications. Addison Wesley, 2000 (in print)

Configuration Generator

```
// generating configuration repository
struct Config
    {   typedef Engine_ Engine;
        typedef CarBody_ CarBody;
        // the remaining information as enumerators
        enum {  speeds       = speeds_,
                line         = line_,
                transmission = transmission_,
                options      = options_,
             };
        typedef RET Car;
    };
// the configuration repository is a part of each product; i.e., it
// can be accessed by other generators (e.g. generating
// matching appliances)
```


Usage

```
typedef
    AUTO_GENERATOR<>::RET
    C1; // give me a default one
typedef
    AUTO_GENERATOR<BaseLine<>,TrailerCoupling>::RET
    C2;
typedef
    AUTO_GENERATOR<BaseLine<fourSpeedAutomatic> >::RET
    C3;

// ...

C1 c1; C2 c2; C3 c3;

// ...
```


An Appetizer

- Vision
- Single Systems vs. System Families
- Feature Models
- Implementation Components
- System Architecture
- Generator

➔ Applications

- New Exciting Technologies
- Summary and Theses

Applications

- Generative Matrix Computation Library (GMCL)
  - generates 1.840 highly-optimized matrix types
  - contains a separate generator for optimizing matrix expressions
  - code size of ca. 7.500 lines of C++
  - *http://nero.prakinf.tu-ilmenau.de/~czarn/gmcl/*
- Generative Matrix Factorization Library (GMFL)
  - generates algorithms for LU factorization of matrices
  - uses GMCL
  - code size of ca. 3.800 lines of C++
  - *http://nero.prakinf.tu-ilmenau.de/~czarn /generate/stja98/knaupp.zip*
- Statistics Library in Postal Automation
  - generates different kinds of counters, timers and statistical algorithms (part of a commercial product of Siemens EC)

An Appetizer

- Vision
- Single Systems vs. System Families
- Feature Models
- Implementation Components
- System Architecture
- Generator
- Applications

➔ New Exciting Technologies

- Summary and Theses

Debugging

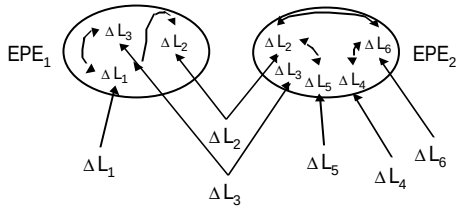
```
MultiplicationExpression<class LazyBinaryExpression<class AdditionExpression<class MatrixICCL::Matrix<class
MatrixICCL::BoundsChecker<class MatrixICCL::ArrFormat<class MatrixICCL::StatExt<struct
MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>,class MatrixICCL::Rect<class
MatrixICCL::StatExt<struct MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>>,class
MatrixICCL::Dyn2DCContainer<class MATRIX_ASSEMBLE_COMPONENTS<class
MATRIX_DSL_ASSIGN_DEFAULTS<class MATRIX_DSL_PARSER<struct MatrixDSL::matrix<int,struct
MatrixDSL::structure<struct MatrixDSL::rect<struct MatrixDSL::stat_val<struct MatrixDSL::int_number<int,7>>,struct
MatrixDSL::stat_val<struct MatrixDSL::int_number<int,7>>,struct MatrixDSL::unspecified_DSL_feature>,struct
MatrixDSL::dense<struct MatrixDSL::unspecified_DSL_feature>,struct MatrixDSL::dyn<struct
MatrixDSL::unspecified_DSL_feature>>,struct MatrixDSL::speed<struct MatrixDSL::unspecified_DSL_feature>,struct
MatrixDSL::unspecified_DSL_feature,struct MatrixDSL::unspecified_DSL_feature,struct
MatrixDSL::unspecified_DSL_feature,struct MatrixDSL::unspecified_DSL_feature>>::DSLConfig>::DSLConfig>>>>,class
MatrixICCL::Matrix<class MatrixICCL::BoundsChecker<class MatrixICCL::ArrFormat<class MatrixICCL::StatExt<struct
MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>,class MatrixICCL::Rect<class
MatrixICCL::StatExt<struct MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>>,class
MatrixICCL::Dyn2DCContainer<class MATRIX_ASSEMBLE_COMPONENTS<class
MATRIX_DSL_ASSIGN_DEFAULTS<class MATRIX_DSL_PARSER<struct MatrixDSL::matrix<int,struct
MatrixDSL::structure<struct MatrixDSL::rect<struct MatrixDSL::stat_val<struct MatrixDSL::int_number<int,7>>,struct
MatrixDSL::stat_val<struct MatrixDSL::int_number<int,7>>,struct MatrixDSL::unspecified_DSL_feature>,struct
MatrixDSL::dense<struct MatrixDSL::unspecified_DSL_feature>,struct Ma...
```

Part of the type name produced by the C++ compiler for the expression (A+B)*C

Active Libraries

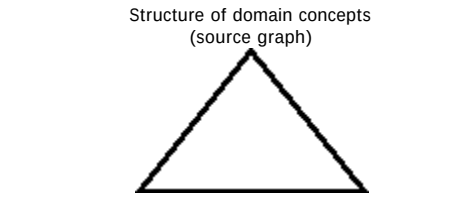
- Problems with template metaprogramming
  - No debugging (cannot debug the compilation process!)
  - Long compilation times (compilers not optimized for that kind of "strange" template programming!)
- Nevertheless it demonstrates the idea of putting compile-time metacode into domain-specific libraries
- Active libraries
  - in addition to classes and functions also contain metacode for configuration, generation, optimization, error reporting, debugging and profiling, editing and visualization of code, code refactoring, versioning, ...
- Extensible programming environments
  - e.g. Intentional Programming (Microsoft Research)

Extensible Programming Environments

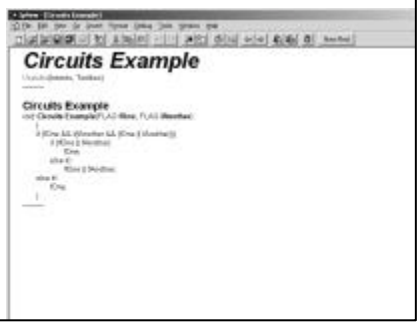


EPE = Extensible Programming Environment
ΔL = Active Library (Δ compilation, Δ debugging, Δ rendering, Δ type-in & editing, Δ refactoring...)

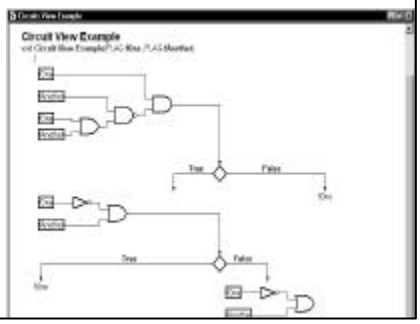
Active Source



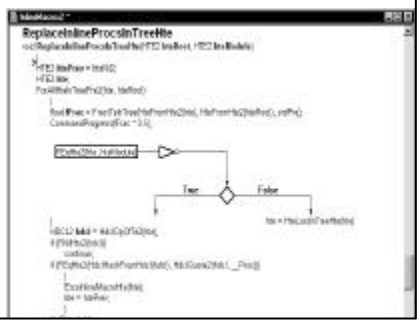
Domain-Specific Notation



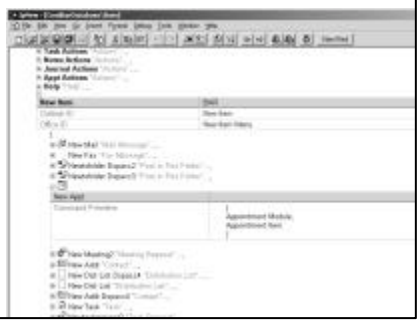
Domain-Specific Notation



Domain-Specific Notation



Domain-Specific Notation



Embedding Bitmaps



GMCL in Intentional Programming



GMCL in Intentional Programming



An Appetizer

- Vision
- Single Systems vs. System Families
- Feature Models
- Implementation Components
- System Architecture
- Generator
- Applications
- New Exciting Technologies

➔ Summary and Theses

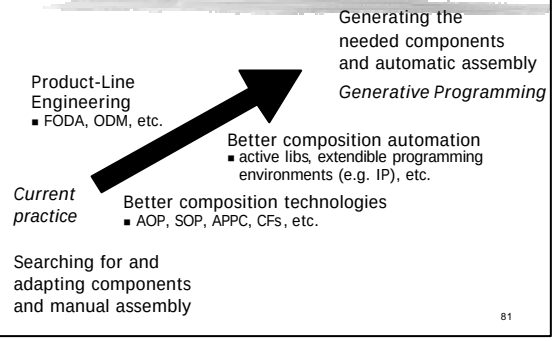
Industrial Revolution

1826 interchangeable parts
successfully introduced by John Hall (after 25 years of unsuccessful attempts!)

1901 assembly lines
introduced by Ransom Olds; popularized and refined by Henry Ford in 1913

1980s automated assembly lines
first industrial robot installed in 1961 at General Motors; 1970's advance of microchips

Industrialization Software Production



Searching for and adapting components and manual assembly

Theses

- A component is always a part of a well-defined production process
- If you can assemble the components manually, you can also automate it using a generator
- Generation can be performed both statically and dynamically
- Criteria such as binary format, interoperability, language independence, etc., are always relative to the production process



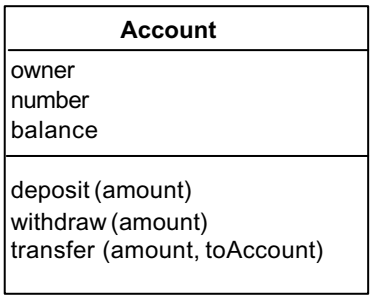
In-Depth Example

➔ Example "Personal Account"

- Feature Model
- Dependency Analysis
- System Family Architecture
- Implementation Components
- Domain Specific Language
- Generator

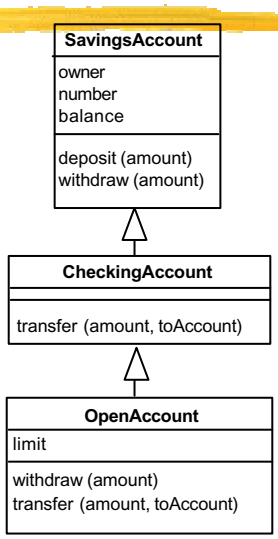
Example "Personal Account"

Simple account class
(trivial textbook example)



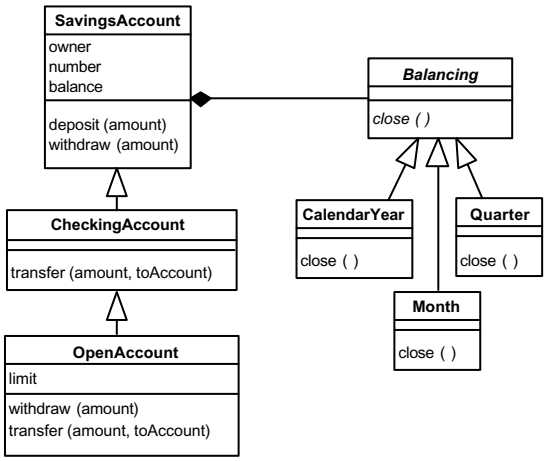
Example "Personal Account"

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!)



In-Depth Example

- Example "Personal Account"
- ➔ Feature Model
- Dependency Analysis
- System Family Architecture
- Implementation Components
- Domain Specific Language
- Generator

Feature Model

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

Features and Psychology

- The Human mind processes lots of information
- Important function of human information processing: reducing the amount of information
  - Classifying a perceptional unit as an instance of a concept, e.g. Aunt Martha's bird
  - Describing concepts by features, e.g. a bird has wings, a bill, and can fly
- Acquisition and evolution of concepts will not be pursued further in this course

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

Feature Model

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

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

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 Model

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

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

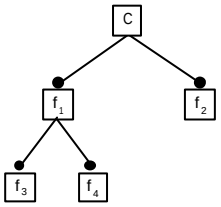


Feature Model

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

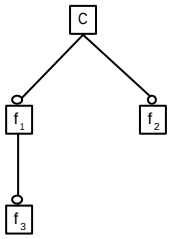
Mandatory Feature

- 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}



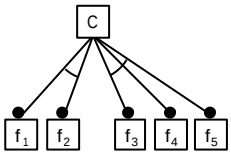
Optional Feature

- 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}



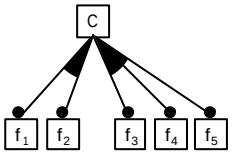
Alternative Features

- Exactly one from a set of alterntative 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}



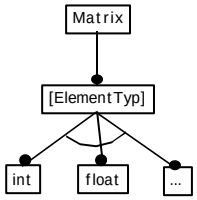
Or-Features

- 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



Feature Model

- Background
- Feature Model
- Feature Diagram
- Feature Types

➔Normalized Feature Diagrams

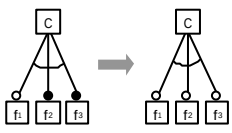
- Commonality and Variability
- FAQs
- Feature Diagram "Personal Account"

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

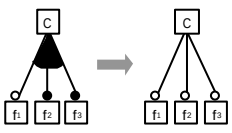
Normalized Feature Diagrams

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.



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.



Feature Model

- Background
- Feature Model
- Feature Diagram
- Feature Types
- Normalized Feature Diagrams

➔Commonality and Variability

- FAQs
- Feature Diagram "Personal Account"

Commonality in Feature Diagrams

- From the perspective of a concept
  - What are the common features of all concept instances?
  - ➡Common Features
- From the perspective of a feature
  - What are the common features of all concept instances that have the special feature?
  - ➡Common Subfeatures

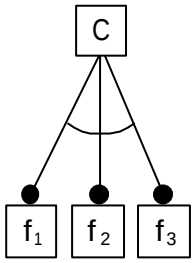
Variability in Feature Diagrams

- 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

Kinds of Variation Points

Dimension

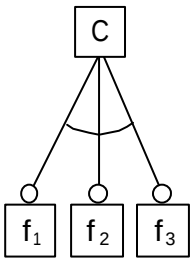
- Feature or concept with all direct subfeatures or features being alternative features only



Kinds of Variation Points

Dimension with optional features

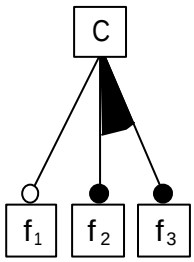
- Feature or concept with direct subfeatures or features being alternative-optional features only



Kinds of Variation Points

Extension point

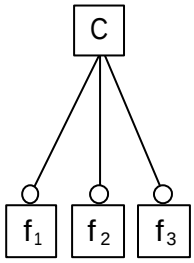
- Concept or feature
  - with at least one set of direct or-features or or-subfeature
  - with at least one direct optional feature or subfeature



Kinds of Variation Points

Extension point with optional features

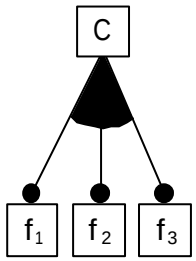
- Concept or feature with all direct features or subfeatures being optional features only



Kinds of Variation Points

Extension point with or-features

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



Feature Model

- Background
- Feature Model
- Feature Diagram
- Feature Types
- Normalized Feature Diagrams
- Commonality and Variability

➔FAQs

- Feature Diagram "Personal Account"

FAQs

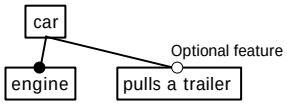
What are the semantics of the edges of a feature diagram?

- 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

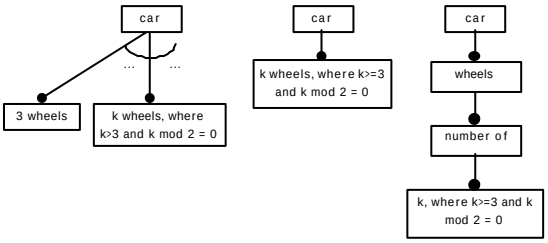
Isn't a feature diagram just a simple part-of hierarchy?

- 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

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



FAQs

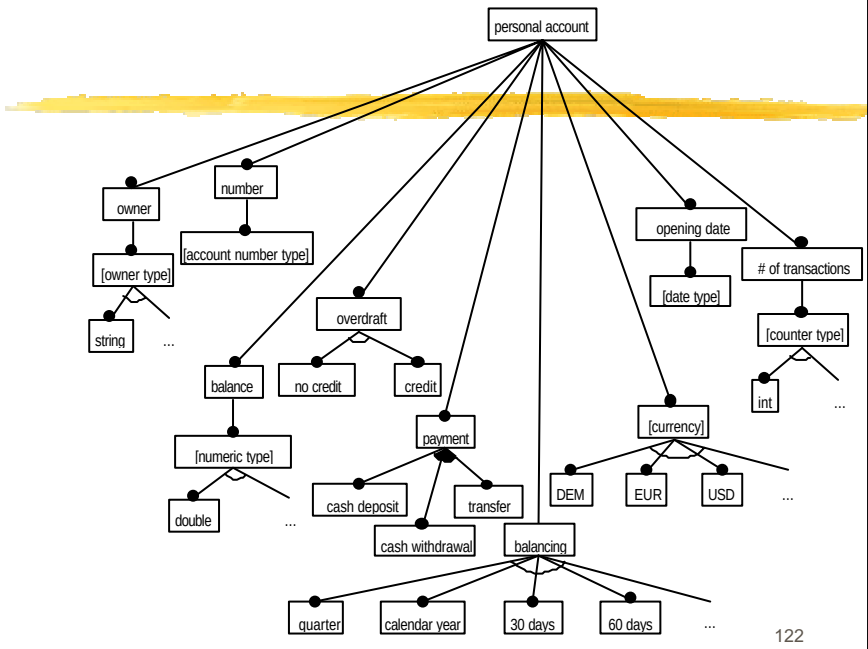
A feature diagram forms a tree - isn't that too restrictive?

- 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

Feature Model

- Background
- Feature Model
- Feature Diagram
- Feature Types
- Normalized Feature Diagrams
- Commonality and Variability
- FAQs

➔Feature Diagram "Personal Account"



In-Depth Example

- Example "Personal Account"
- Feature Model
- ➔Dependency Analysis
- System Family Architecture
- Implementation Components
- Domain Specific Language
- Generator

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!

Dependency Analysis

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

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

Dependency Analysis

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

Equivalence Categories

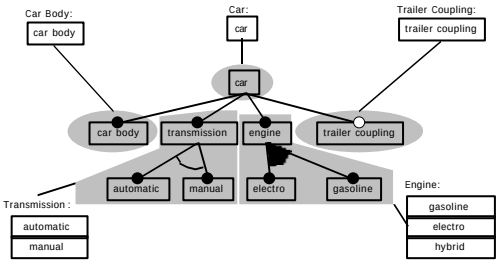
- The next step is to define equivalence categories
- An equivalence category comprises features that are mapped to implementation components which implement a common category interface
- The dependency analyses is then done at the level of equivalence categories
  - More efficient that examining each 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

Engine:

gasoline
electro
hybrid

Dependency Analysis

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

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)

Dependency Analysis

- Goal
- Identifying Implementation Components
- Defining Equivalence Categories
- Analyzing Dependencies

➔Results

- Notation
- Dependency Analysis of "Personal Account"

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

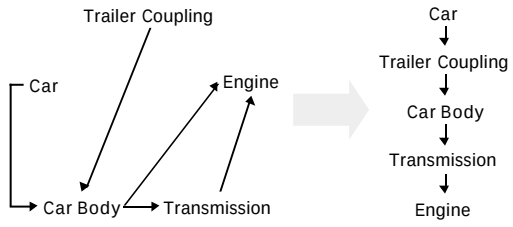
Dependency Analysis

- Goal
- Identifying Implementation Components
- Defining Equivalence Categories
- Analyzing Dependencies
- Results

➔Notation

- Dependency Analysis of "Personal Account"

Notation

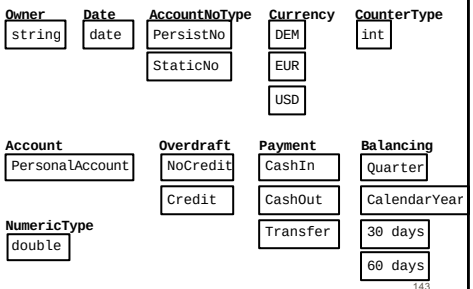


Dependency Analysis

- Goal
- Identifying Implementation Components
- Defining Equivalence Categories
- Analyzing Dependencies
- Result
- Notation

➔Dependency Analysis of "Personal Account"

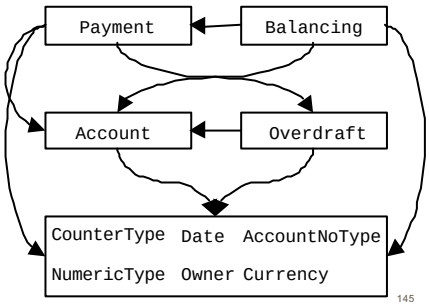
Implementation Components



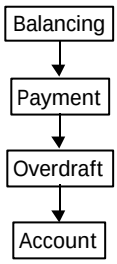
Analyzing Dependencies

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



In-Depth Example

- Example "Personal Account"
- Feature Model
- Dependency Analysis

➔System Family Architecture

- Implementation Components
- Domain Specific Language
- Generator

System Family Architecture

➔Goal

- GenVoca Architecture
- Notation
- Usage
- GenVoca Grammar
- Configuration Repository
- GenVoca Model for "Personal Account"

Goal

- A system family architecture describes the structure of all members of a system family
- The following architecture model relies on composition
  - It describes successive refinements that yield a concrete system
  - Each refinement can add several components in a coordinated manner

System Family Architecture

- Goal
- ➔GenVoca Architecture
- Notation
- Usage
- GenVoca Grammar
- Configuration Repository
- GenVoca Model for "Personal Account"

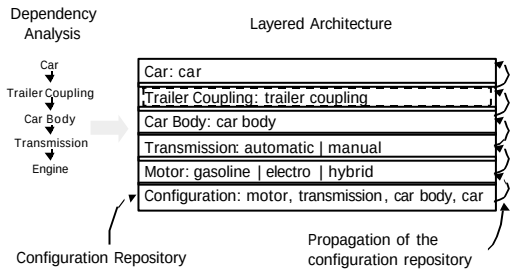
GenVoca Architecture

- The result of the dependency analysis is a dependency graph of the equivalence categories
- GenVoca architectures are layered architectures
  - They follow the composition principle
  - Components of a lower layer parameterize the components of the layer above
- GenVoca architectures may also form trees

GenVoca Architecture

- The dependency hierarchy corresponds to the structure of a GenVoca hierarchy
- In the following, the GenVoca model is complemented by a special bottom layer, the configuration repository
  - Repository for any configuration information for some or all components
  - Repository for any configuration information exchanged between components
  - Each layer exports the configuration repository to the layer above

GenVoca Architecture



System Family Architecture

- Goal
- GenVoca Architecture
- ➔Notation
- Usage
- GenVoca Grammar
- Configuration Repository
- GenVoca Model for "Personal Account"

Notation

- The architecture diagram shows the layers that are stacked on each other
- The name of a layer corresponds to the equivalence category and is placed before the colon
- The names following the colon are the names of the implementation components in the category
- Alternative components are separated by “|”

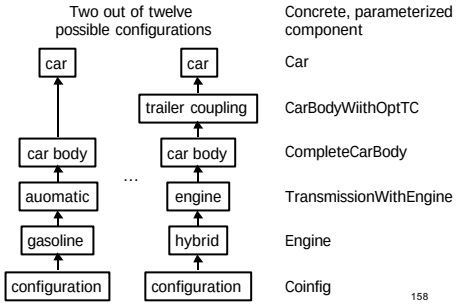
Notation

- The dashed line marks an optional layer
  - Optional layers can be omitted in parameterization
  - In general, an optional layer can be replaced by a nonoptional layer with an additional component with an empty implementation
- The configuration repository displays the available information separated by commas

System Family Architecture

- Goal
- GenVoca Architecture
- Notation
- ➔Usage
- GenVoca Grammar
- Configuration Repository
- GenVoca Model for "Personal Account"

Usage



System Family Architecture

- Goal
- GenVoca Architecture
- Notation
- Usage
- ➔GenVoca Grammar
- Configuration Repository
- GenVoca Model for "Personal Account"

GenVoca Grammar

- A GenVoca grammar is an equivalent representation of a GenVoca architecture
- Parameters are enclosed in brackets []
- Alternatives are separated by vertical bars |
- The name to be defined is placed before the colon, its definition after the colon

GenVoca Grammar

```
Car:                    Car[CarBodyWithOptTC]
CarBodyWithOptTC:       CarBodyWithTC[CompleteCarBody] |
                        CompleteCarBody
CompleteCarBody:        CarBody[TransmissionWithEngine]
TransmissionWithEngine: ManualTransmission[Engine] |
                        AutomaticTransmission[Engine]
Engine:                 GasolineEngine[Config] |
                        ElectricEngine[Config] |
                        HybridEngine[Config]
Config
  Engine:               GasolineEngine | ElectricEngine |
                        HybridEngine
  Transmission:         ManualTransmission[Engine] |
                        AutomaticTransmission[Engine]
  CarBody:              CarBody[TransmissionWithEngine]
  ...
```


GenVoca Grammar

- Sentences of a GenVoca grammar describe type equations, e.g. car[car_body[automatic[gasoline[C1]]]]
- Not every syntactically correct type equation represents a semantically valid configuration
- A GenVoca grammar can be easily mapped to C++ templates, e.g.
  template <class CarBodyWithOptTC>
  class car;

System Family Architecture

- Goal
- GenVoca Architecture
- Notation
- Usage
- GenVoca Grammar

➔Configuration Repository

- GenVoca Model for "Personal Account"

Configuration Repository

- Repository for any information which shall be accessible by a larger number of components; mostly horizontal parameters
- It can contain the name of the complete resulting type; hence the second lowest layer can access the name of the type of completely parameterized, uppermost layer
- Any information that introduces a cyclic dependency will be generally handled by the configuration repository

Configuration Repository

Structure and Usage in C++

```
template <class Config_>
struct GasolineEngine
{ typedef Config_ Config;
  // ...
};
struct C1
{ enum { speeds = 5 };
  typedef GasolineEngine<C1> Engine;
  typedef ManualTransmission<Engine> Transmission;
  typedef CarBody<Transmission> CarBody;
  typedef Car<CarBody> Car;
};
```

Exporting config

Using config

Using config recursively

Complete car type

System Family Architecture

- Goal
- GenVoca Architecture
- Notation
- Usage
- GenVoca Grammar
- Configuration Repository

➔GenVoca Model for "Personal Account"

GenVoca Architecture

```
Balancing      : Quarter | CalendarYear | 30 days | 60 days
Payment        : CashIn, CashOut, Transfer
Overdraft      : NoCredit | Credit
Account        : PersonalAccount
Config
        Owner          : string | ...
        Currency       : DEM | EUR | USD | ...
        AccountNoType  : StaticNo | PersistentNo | ...
        Date           : date | ...
        NumericType    : double | ...
        CounterType    : int | ...
```


GenVoca Grammar

```
AccountWithBalancing : Qarter[Payment] | CalendarYear[Payment] |
                       30 days[Payment] | 60 days[Payment] ...
Payment              : CashIn[Overdraft | Payment] |
                       CashOut[Overdraft | Payment] |
                       Transfer[Overdraft | Payment]
Overdraft            : NoCredit[Account] |
                       Credit[Account]
Account              : PersonalAccount[Config]
Config               :
    Owner            : string | ...
    Currency         : DEM | EUR | USD | ...
    AccountNoType    : StaticNo | PersistentNo | ...
    Date             : date | ...
    NumericType      : double | ...
    CounterType      : int | ...
```


Type Equations

- Specifying a savings account
  CalendarYear[CashIn[CashOut[NoCredit[PersonalAccount[C1]]]]]
- Specifying an open account
  Quarter[Transfer[CashIn[CashOut[Credit[PersonalAccount[C2]]]]]]
- C1 and C2 are two different configuration repositories

In-Depth Example

- Example "Personal Account"
- Feature Model
- Dependency Analysis
- System Family Architecture

➔Implementation Components

- Domain Specific Language
- Generator

Implementation Components

➔Requirements

- Some C++ Idioms
- Conclusions
- Selected Implementation Components from "Personal Account"

Requirements

Implementation components may be anything

- Assembler-code
- Source code fragments of high-level programming languages
- Generic functions or classes
- Compiled classes or any binary components
- Interoperable, distributed components, e.g. CORBA components
- Binary components of proprietary standards such as COM or JavaBeans
- For now, let's take a closer look at C++ class templates as implementation components

Implementation Components

- Requirements
- ➔Some C++ Idioms
- Conclusions
- Selected Implementation Components from "Personal Account"

Dynamic "Framework"

```
#include <iostream>
using namespace std;

class Application
{   public:
        virtual void init () = 0;
        virtual void main() = 0;
        virtual void end() = 0;
        void run()
        {   init();
            main();
            end();
        }
};
```

```
class MyApp: public Application
{   public:
        void init()
        { cout << "init" << endl; }
        void main()
        { cout << "main" << endl; }
        void end()
        { cout << "end" << endl; }
};

void main()
{   MyApp app;
    app.run();
}
```

Static "Framework"

```
#include <iostream>
using namespace std;

template <class Concretization>
class Application: public Concretization
{   public:
    void run()
    {   typedef
            Application<Concretization>
            self;
        self::init();
        self::main();
        self::end();
    }
};
```

```
class MyApp
{   public:
        void init()
        { cout << "init" << endl; }
        void main()
        { cout << "main" << endl; }
        void end()
        { cout << "end" << endl; }
};

void main()
{   Application<MyApp> app;
    app.run();
}
```


Parameterizing Binding Time

```
#include <iostream>
using namespace std;

class Static {};

class Dynamic
{ public:
      virtual ~Dynamic() {};
      virtual void action1() = 0;
      virtual void action2() = 0;
};
```

```
template <class BindingSite = Static>
class Demo: public BindingSite
{ public:
    void action1()
    { cout << "action1" << endl; }
    void action2()
    { cout << "action2" << endl; }
};

void main()
{ Demo<> s; // static binding
  Demo<Dynamic> d; // late binding
}
```

Using Configuration Repositories

```
template<class Config_> struct GasolineEngine
{ typedef Config_ Config;
  GasolineEngine() { cout << "GasolineEngine "; }
};

template <class Engine> struct AutomaticTransmission
{ typedef typename Engine::Config Config;
  enum { speeds = Config::speeds };
  Engine e;
  AutomaticTransmission()
  { cout << speeds << "-SpeedAutomatic "; }
};
```


Refinement Techniques

- Components of a higher level may refine operations of lower-level components
- Two mechanisms for refining operations
  - Forwarding adapter
  - Inhertitance-based adapter
- Both refinements can be implemented statically or dynamically
- Until now only static inheritance adapters were demonstrated

Static Forwarding Adapter

```
template <class Component>
class Adapter
{    public:
        // refine operationA()
        void operationA()
        {    ... // adapter-specific work
             component.operationA();
             ... // adapter-specific work
        };

        // forward operationB()
        void operationB()
        {    component.operationB(); };
    private:
        Component component;
};
```

- Implement components as forwarding adapters only if almost all functions will be refined
- Otherwise too many functions have to be forwarded
- This introduces too many dependencies from the given interface of the component, which impairs maintenance and adaptation

Static Inheritance-Based Adapter

```
template <class Component>
class Adapter : public Component
{    public:
        //    refine operationA()
        void operationA()
        {    ... // adapter-specific work
             Component::operationA();
             ... // adapter-specific work
        };
};
```

- Need no variable of type Component
- Inherited operations that remain unchanged don't need explicit forwarding
- Only refined operations need to be overridden
- Less interface dependencies, hence easier to maintain and to adapt

Implementation Components

- Requirements
- Some C++ Idioms

➔Conclusions

- Selected Implementation Components from "Personal Account"

Conclusions

- Implementation components and the way to use them may be very different from what the problem domain experts would expect

➡ Hence we need a domain specific language for system specification

- The manual assembly of implementation components and their optimization by writing hundreds or thousands of configuration repositories is tedious and error-prone; furthermore, invalid configurations have to be excluded

➡ Hence we should automate the assembly of implementation components and the optimization of their configuration

Implementation Components

- Requirements
- Some C++ Idioms
- Conclusions

➔Selected Implementation Components from "Personal Account"

Currency Components

```
struct DEM
{   static const char * const symbol()
    {   return "DEM"; }
};

struct EUR
{   static const char * const symbol()
    {   return "EUR"; }
};

struct USD
{   static const char * const symbol()
    {   return "USD"; }
};
```

Account Number Component

```
class StaticNo
{   public:
        StaticNo():number_(counter_++) {}
        void print(ostream& os) const
        {   os << number_; }
    private:
        long int number_;
        static long int counter_;
        // prohibit the following operations:
        StaticNo(const StaticNo&) {};
        StaticNo& operator=(const StaticNo&)
        {   return *this; }
};

long int StaticNo::counter_ = 0;

ostream& operator<<(ostream& os,const StaticNo& accNo)
{   accNo.print(os);
    return os;
}
```


A Sample Config

```
struct C1
{   typedef string      Owner;
    typedef DEM         Currency;
    typedef date        Date;
    typedef double      NumericType;
    typedef int         CounterType;
    typedef StaticNo    AccountNoType;
    typedef CalendarYear<CashOut<CashIn<NoCredit
            <PersonalAccount<C1> > > > > Account;
};
```

Personal Account Component

```
template <class Config_>
class PersonalAccount
{   public:
        typedef Config_ Config;
        typedef typename Config ::Owner Owner;
        typedef typename Config ::Currency Currency;
        typedef typename Config ::Date Date;
        typedef typename Config ::NumericType NumericType;
        typedef typename Config ::CounterType CounterType;
        typedef typename Config ::AccountNoType AccountNoType;

        PersonalAccount(const Owner& own):  owner_(own),
                                balance_(0),
                                transactions_(0)
        {}
        const Owner& owner() const
        {   return owner_; }
        const AccountNoType& number() const
        {   return AccountNo_; }
        const NumericType& balance() const
        {   return balance_; }
        const CounterType& transactions() const
        {   return transactions_; }
        const Date& openedOn() const
        {   return openedOn_; }
```


Overdraft Components

```
template <class Base>
class NoCredit: public Base
{   public:
        typedef typename Base::Config Config;
        typedef typename Config ::Owner Owner;
        NoCredit (const Owner& own):Base(own)
        {}
        typedef typename Base::NumericType NumericType;
        void debit(const NumericType& amount)
        {   assert(amount > 0);
            if (Base::balance() - amount < 0)
            throw "No credit available";
            else Base::debit (amount);
        }
};
```

Payment Component

```
template <class Base>
class CashIn: public Base
{   public:
        typedef typename Base::Config Config;
        typedef typename Config::NumericType NumericType;
        typedef typename Config::Owner Owner;
        CashIn(const Owner& own):Base(own)
        {}
        void deposit(const NumericType& amount)
        {   Base::credit(amount); }
};
```


Balancing Components

```
template <class Base,int numberDays>
class AfterNoOfDays: public Base
{   public:
        typedef typename Base::Config Config;
        typedef typename Config::Date Date;
        typedef typename Config::Owner Owner;
        AfterNoOfDays(const Owner& own):Base(own),
                lastBalancing_(Base::openedOn()) {}
        void close(ostream& os = cout)
        {   Date d(lastBalancing_ + numberDays);
            os  << "Statement on " << d << endl
                << "For account no. "
                << Base::number() << endl
                << "Owner " << Base::owner() << endl
                << "Transactions " << Base::transactions() << endl
                << "Balance " << Base::balance()
                << ' ' << Base::currency() << endl;
            Base::transactions_ = 0;
            lastBalancing_ = d;
        }
    private:
        Date lastBalancing_;
};
```

Balancing Components

```
//now define ThirtyDays and SixtyDays using AfterNoOfDays
template <class Base>   // 1 banking month has 30 days
class ThirtyDays: public AfterNoOfDays<Base,30>
{   public:
        typedef typename Base::Config Config;
        typedef typename Config::Owner Owner;
        ThirtyDays(const Owner& own):
                AfterNoOfDays<Base,30>(own)
        {}
};

template <class Base>  // 2 banking months, each 30 days
class SixtyDays:  public AfterNoOfDays<Base,60>
{   public:
        typedef typename Base::Config Config;
        typedef typename Config::Owner Owner;
        SixtyDays(const Owner& own):
                AfterNoOfDays<Base,60>(own)
        {}
};
```


In-Depth Example

- Example "Personal Account"
- Feature Model
- Dependency Analysis
- System Family Architecture
- Implementation Components

➔ Domain Specific Language

- Generator

Domain Specific Language

➔Goal
- DSL Design
- DSL for "Personal Account"

Goal

- A domain specific language (DSL) is used to specify systems or higher-order components; a DSL is used for "ordering" systems
- A DSL provides the user with domain specific concepts and features; it hides the complexities of the solution space and its specific components
- A DSL is based on the language of users or experts
- Different groups of users or experts can use different languages; hence we may need to provide several alternative DSLs

Domain Specific Language

- Goal

➔DSL Design
- DSL for "Personal Account"

DSL Design

Designing DSLs is a science in itself; some important criteria for designing DSLs are
- Usability
  - Simplicity: easy to understand, learn, and use
  - Efficiency: using a few language elements for ordering many systems; default values are used for unspecified features
  - Safety: the likelihood of ordering an invalid system is should be minimized
  - Openness: new language elements are easy to introduce
- Technical qualities
  - modular structure
  - easy to extend
  - easy to parse

Sources for DSL Design

- Feature diagram
  - Features that do not need to be specified are eliminated
  - Rearranging dimensions and optional features to avoid the possibility of specifying invalid systems
  - Rearranging features so that frequently used features occur first
  - Introducing additional abstract features
- Using existing DSLs, e.g. catalogues for ordering cars
- Designing innovative DSLs by asking users and experts and applying the aforementioned usability criteria

Techniques

- GenVoca grammars are well suited to describe DSLs without specifying implementation details
- Default values should be written down separately
- Implementing the DSL in C++ using
  - enum types and constants
  - traits classes
  - traits class templates
  - implementation components as direct DSL features

Domain Specific Language

- Goal
- DSL Design

➔DSL for "Personal Account"

GenVoca Grammar "Personal Account“ DSL

AccountSpecification	:	Account, CurrencyType, AccNoType, NumType, CountType, OwnerType, DateType
Account	:	fixedTermDeposit[Balancing] | shortNoticeDeposit[Balancing] | checking | openAccount | savings
Balancing	:	thirtyDays | sixtyDays | quarter | calendarYear
CurrencyType	:	DEM | EUR | USD | ...
AccNoType	:	StaticNo | PersistentNo | ...
NumType	:	double | ...
CountType	:	int | ...
InhaberTyp	:	string | ...
DateType	:	date | ...

Default Settings of "Personal Account" DSL

Account	:	Kontokorrent
CurrencyType	:	DEM
AccNoType	:	PersistentNo
NumType	:	double
CountType	:	int
OwnerType	:	string
DateType	:	date

Implementation of "Personal Account" DSL

```
// for internal use only
enum AccountTypes
{   fixedTermDeposit,
    shortNoticeDeposit,
    checking,
    openAccount,
    savings
};
```

```
enum BalancingStrategies
{   thirtyDays,
    sixtyDays,
    quarter,
    calendarYear
};

// for internal use only
enum CreditStrategies
{   noCredit,
    credit
};
```

Implementation of "Personal Account" DSL

```
template <BalancingStrategies Balancing = thirtyDays>
struct FixedTermDeposit
{   enum
    {   accountType     = fixedTermDeposit,
        balancing       = Balancing,
        creditStrategy  = noCredit
    };
};

template <BalancingStrategies Balancing = thirtyDays>
struct ShortNoticeDeposit
{   enum
    {   accountType     = shortNoticeDeposit,
        balancing       = Balancing,
        creditStrategy  = noCredit
    };
};
```


Implementation of "Personal Account" DSL

```
struct Checking
{   enum
    {   accountType     = checking,
        balancing       = quarter,
        creditStrategy  = noCredit
    };
};

struct OpenAccount
{   enum
    {   accountType     = openAccount,
        balancing       = quarter,
        creditStrategy  = credit
    };
};
```

Implementation of "Personal Account" DSL

```
// The last part of the DSL corresponds to the
// parameter list of the generator:

template<   class Account = OpenAccount,class CurrencyType = DEM,
            class AccNoType = PersistentNo, class NumType = double,
            class CountType = int,class OwnerType = string,
            class DateType = date
        >
struct ACCOUNT_GENERATOR
{   //...
```


In-Depth Example

- Example "Personal Account"
- Feature Model
- Dependency Analysis
- System Family Architecture
- Implementation Components
- Domain Specific Language

➔Generator

Generator

➔Goal
- Structure
- Hints
- "Personal Account" Generator

Goal

- Invalid configurations have to be detected (optional)
- Manually assembling components is expensive and error-prone
- Manual optimization of configurations is difficult
- A generator automates assembly: It takes a specification in DSL, parses the specification, checks buildability, and assembles the final product (system or higher-order component) from implementation components

Generator

- Goal

➔Structure
- Hints
- "Personal Account" Generator

Structure

- A generator has to perform five tasks
  - Parsing the input DSL specification
  - Completing the DSL specification (by assuming or computing defaults for the unspecified parameters)
  - Checking buildability (if the DSL allows specifying invalid configurations)
  - Assembling implementation components
  - Creating the configuration repository
- In complex generators these tasks can be implemented as separate functions

Template Metaprogramming

- If the generator should be executed at compile time, static metaprogramming is required
- The C++ compiler executes template metafunctions at compile-time
- Fundamental C/C++ control structures are available as template metafunctions
  - IF - simple branch
  - SWITCH - multiple branch
  - EVAL_TABLE - evaluating extended decision tables
  - DO, WHILE, and FOR - classical looping constructs

Generator

- Goal
- Structure

➔Hints
- "Personal Account" Generator

Hints

- Manual assembly: the configuration repository parameterizes the components of the lowest layer
- Automatic assembly: the generator replaces the configuration repository as the parameter of the components of the lowest layer
- Hence the components of the lowest layer need to be adapted
  - *Generator* replaces *Config* as template-parameter
  - *Config* needs to be retrieved from *Generator*
- The configuration repository is part of every product
  - All available metainformation can be retrieved from each product
  - The metainformation itself can serve as a specification for another generator (e.g., for creating higher-order products)

Generator

- Goal
- Structure
- Hints

➔"Personal Account" Generator

Error Checking - Outside the Generator

```
struct NoError {};
struct WrongBalancingStrategy {};

template <class ErrorType>
struct Error
{   enum { everythingOK };
};

template<>
struct Error<WrongBalancingStrategy>
{}; // no member everytingOK!
```


Parsing DSL Specification

```
template<   class Account = OpenAccount,class CurrencyType = DEM,
            class AccNoType  = PersistentNo, class NumType    = double,
            class CountType  = int,          class OwnerType  = string,
            class DateType   = date
        >
struct ACCOUNT_GENERATOR
{   // define a short name for the complete generator
    typedef ACCOUNT_GENERATOR<  Account,CurrencyType,AccNoType,
                                NumType,CountType,OwnerType,
                                DateType> Generator;
    // parse DSL
    enum {
        accountType    = Account::accountType,
        withCredit     = (CreditStrategies)Account::creditStrategy == credit,
        withCashOut    = 1, // Cash withdrawal always possible
        withCashIn     = 1, // Cash deposit always possible
        withTransfer   = (AccountTypes)accountType == checking ||
                         (AccountTypes)accountType == openAccount,
        withBalancing  = Account::balancing
    };
```

Checking Buildability

```
// check buildability
   enum{ errorCode = (((BalancingStrategies)withBalancing == quarter ||
                       (BalancingStrategies)withBalancing == calendarYear) &&
                       ((AccountTypes)accountType == fixedTermDeposit ||
                       (AccountTypes)accountType == shortNoticeDeposit)) ?
                       wrongBalancingStrategy : noError
         };

   typedef SWITCH<errorCode,
             CASE<noError,                  Error<NoError>,
             CASE<wrongBalancingStrategy, Error<WrongBalancingStrategy>
          > > >::RET ErrorChecking;
   enum   {  everythingOK = ErrorChecking::everythingOK
          };
```


Component Assembly

```
// assemble components
   typedef PersonalAccount<Generator> BaseAccount;
   typedef IF<withCredit,
              Credit<BaseAccount>,
              NoCredit<BaseAccount> >::RET
           OptionalOverdraft;
   typedef IF<withCashOut,
              CashOut<OptionalOverdraft>,
              OptionalOverdraft>::RET
           OptionalCashOut;
   typedef IF<withCashIn,
              CashIn<OptionalCashOut>,
              OptionalCashOut>::RET
           OptionalCashIn;
// continuated on next slide
```

Component Assembly

```
// continuation from previous slide

typedef IF<withTransfer,
              Transfer<OptionalCashIn>,
              OptionalCashIn>::RET
        OptionalTransfer;

typedef SWITCH<withBalancing,
             CASE<thirtyDays,ThirtyDays<OptionalTransfer>,
             CASE<sixtyDays,SixtyDays<OptionalTransfer>,
             CASE<quarter,Quarter<OptionalTransfer>,
             CASE<calendarYear,CalendarYear<OptionalTransfer>
          > > > > >::RET RET;
```


Creating Configuration Repository

```
// create config
  struct Config
  {   typedef OwnerType     Owner;
      typedef CurrencyType Currency;
      typedef DateType       Date;
      typedef NumType        NumericType;
      typedef CountType     CounterType;
      typedef AccNoType    AccountNoType;
      enum    {  AccountType      = Account::accountType,
                 Balancing        = withBalancing,
                 Credit           = Account::creditStrategy,
                 CashDeposit      = withCashIn,
                 CashWithdrawal   = withCashOut,
                 Transfer         = withTransfer
              };
  };
};
```

Final Work and Usage

```
// Adapting the one and only base layer component
template <class Generator>
class PersonalAccount
{   public:
        typedef typename Generator::Config Config;
        //...

// Usage
// create an account for a fixed-term deposit in euro
ACCOUNT_GENERATOR<FixedTermDeposit<sixtyDays>, EUR>::RET a1;

// using quarterly balancing for a deposit causes a compile time error
ACCOUNT_GENERATOR<FixedTermDeposit<quarter>, EUR>::RET a1;
```


Further Information

- Czarnecki, Eisenecker: Generative Programming - Methods, Tools, and Applications, Addison-Wesley, 2000
- *www.generative-programming.org*
- GI working group "Generative and Component-Based Software-Engineering", *www.prakinf.tu-ilmenau.de/~czarn/generate*
- GCSE 2000, October 10 -12, 2000, Erfurt, Germany, *www.netobjectdays.org/gcse*

Acknowledgements

- Mehmet Aksit, University of Twente, NL
- Don Batory, Univ. of Texas at Austin, USA
- Sholom Cohen, SEI/CMU, USA
- Jim Coplien, Bell Labs, Lucent Technologies, USA
- Gregor Kiczales, Xerox PARC, USA
- Johannes Knaupp, Heiler Software AG
- Karl Lieberherr, Northeastern Univ., Boston, MA, USA
- Tobias Neubert, Systemhaus Chemnitz
- Charles Simonyi, Microsoft Research, USA
- Mark Simos, Synquiry Ltd., USA
- ... and many, many more!

And ... Watch Out For These Cats!

