Generative Programming

based on the book by
K.Czarnecki and
U.Eisenecker

Maja D' Hondt
26/7/2000

CONTENTS

Titel (see contents)

subtitel (see contents)

subsubtitel

structure of the slides

slide contents

clarifications,
side remarks, ...
about contents
on the right ->

chapter.section... - page number (in GP book)

page number of slide

CONTENTS

Principles

current practice

domain engineering	product-line architectures: from producing single systems to family of systems
generic programming AOP COTB C++	building a library of implementation components, a framework, … designed to fit the product-line architecture

contribution of Generative Programming

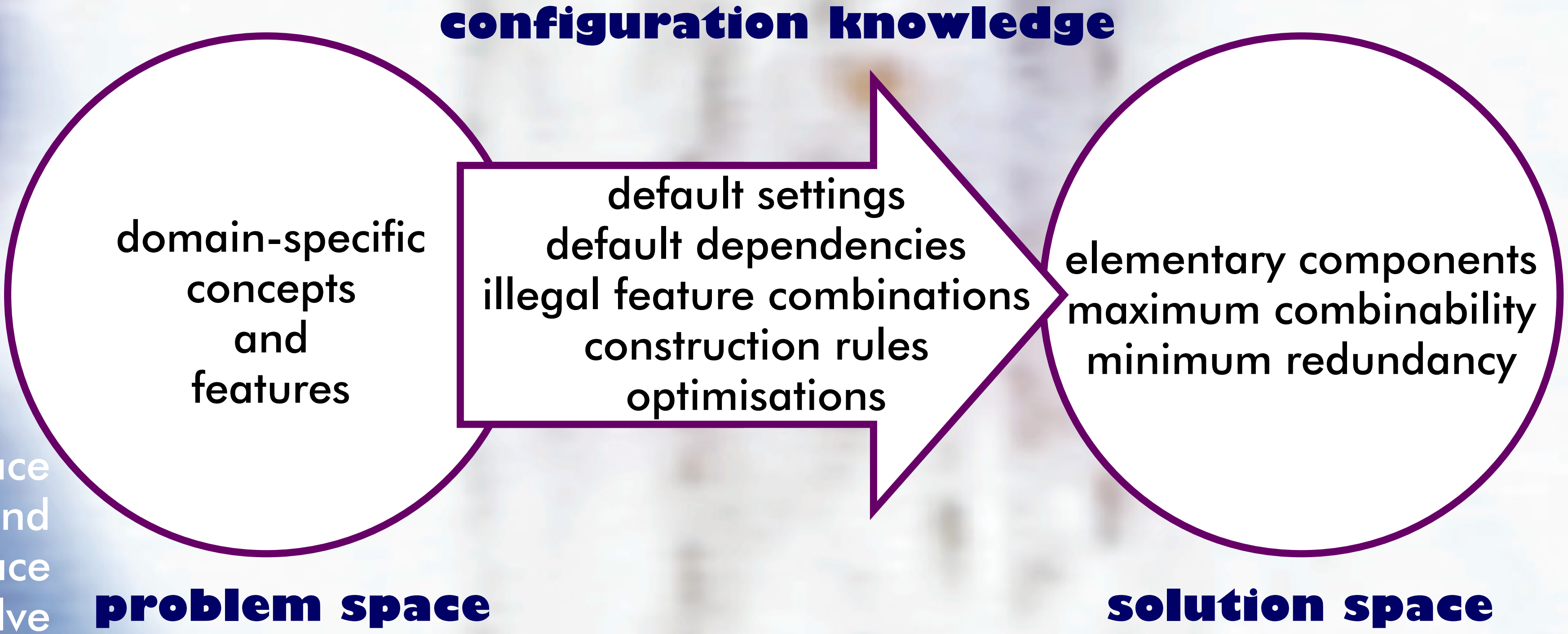
model configuration knowledge	programmers “order” (specify) family member in abstract way using domain-specific languages
specify translation from abstract specification to concrete configuration IP static MP C++	generators automatically produce family member from abstract specification using library of implementation components

Principles

generative domain model

minimal coupling
between
problem space
and
solution space

problem space
and
solution space
can evolve
independently



configuration knowledge

domain-specific
concepts
and
features

default settings
default dependencies
illegal feature combinations
construction rules
optimisations

elementary components
maximum combinability
minimum redundancy

problem space

solution space

1.2 - p.5 & 5.2 - p.131

Principles

development steps

- domain scoping
- feature modeling and concept modeling

- designing a common architecture and identifying the implementation components
  - specifying domain-specific notations for ordering (specifying) systems
    - specifying the configuration knowledge

- implementing the implementation components
  - implementing the domain-specific notations
    - implementing the configuration knowledge using generators

CONTENTS

Domain Engineering

definitions concepts

domain

2 definitions :

domain as the real world OO, AI, KE
= knowledge about a problem area

domain as set of systems DE
= software automating and supporting the processes in this problem area

2 kinds :

- vertical domain is characterised by the business area and the tasks it supports, contains classes of systems
- horizontal domain contains parts of software systems, characterised by (for example) their common functionality

domain engineering

definition :

capturing past experience in building (part of) systems in a domain as reusable assets (i.e. work products or artifacts)

and

providing a means for reusing them (adaptation, assembly)

properties

used to develop
domain-specific frameworks
component libraries
domain-specific languages
generators

tailored for modeling different kinds of systems

Domain Engineering

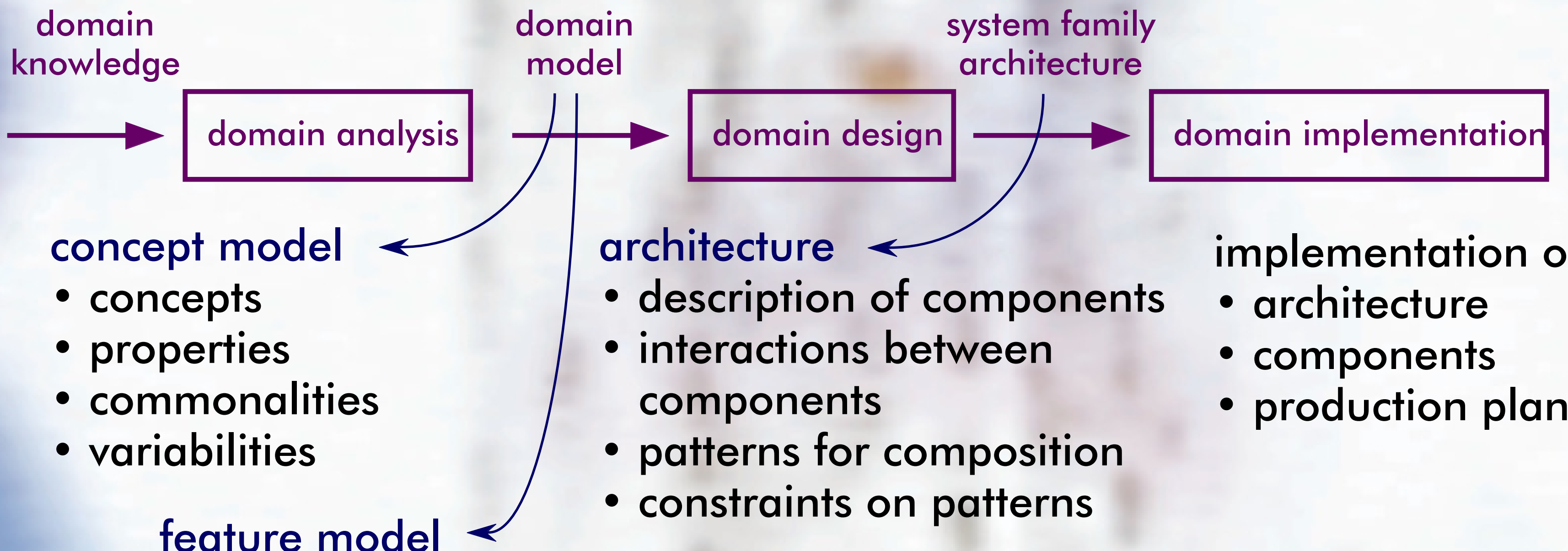
process

features are
reusable and
configurable
requirements

- manual assembly
- automated assembly support
- automatic assembly

suitable
implementation
technologies
discussed later

domain engineering



concept model
- concepts
- properties
- commonalities
- variabilities

feature model
- configuration of concept models

architecture
- description of components
- interactions between components
- patterns for composition
- constraints on patterns

production plan
- provides interface to customers for ordering concrete system
- process of assembling concrete system
- process of custom development

implementation of
- architecture
- components
- production plan

2.3,2.4 - p.23

Domain Engineering

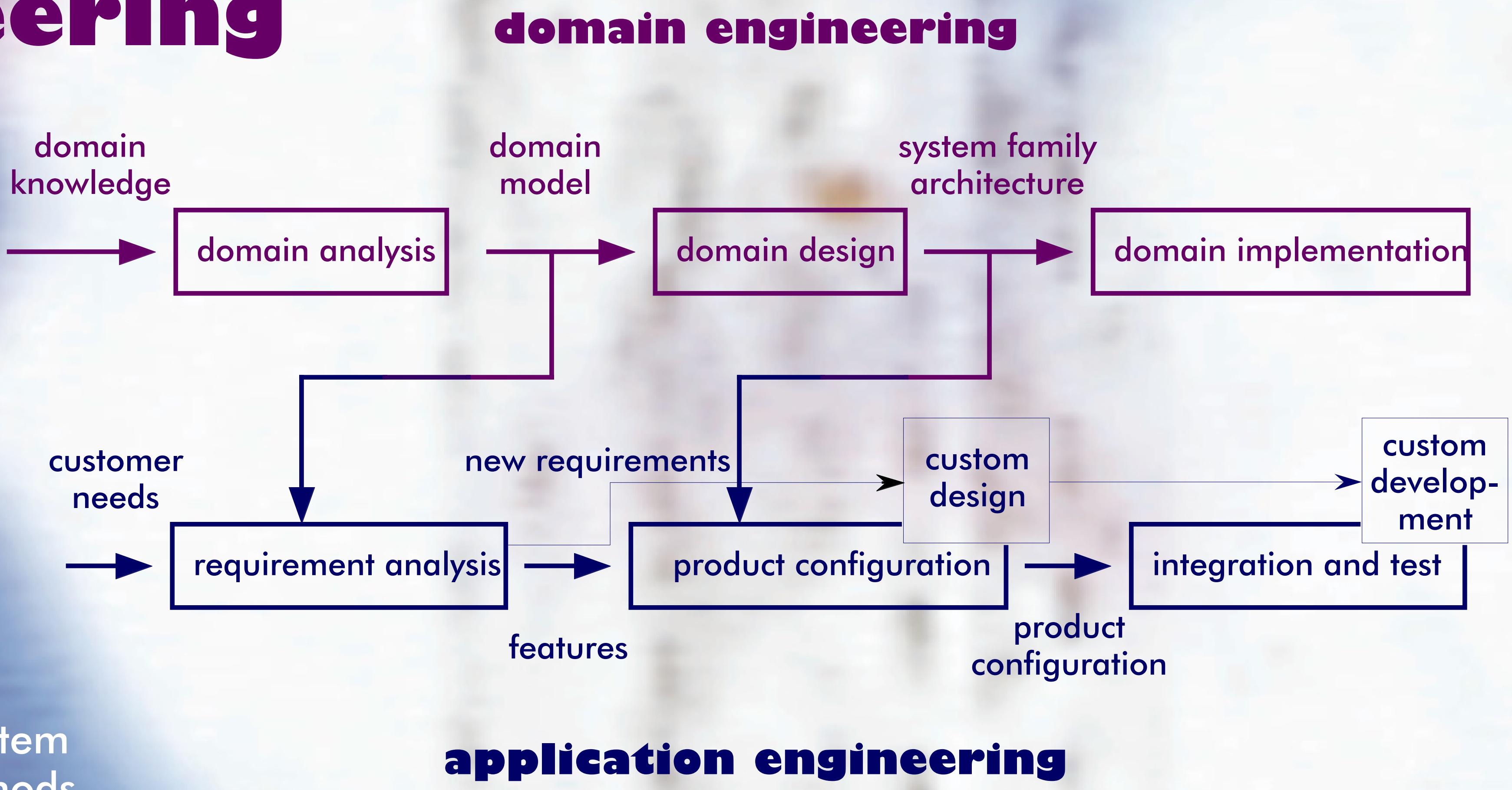
relation to application engineering

results of DE are reused during AE

AE produces concrete systems

AE has single-system engineering methods

DE is multi-system development



Domain Engineering

adaptations for generative programming

Domain Engineering provides basis for **Generative Programming**

- focus on system families
  - model problem space
    - find implementation components
      - model configuration knowledge

but **Generative Programming** also needs:

- appropriate means for “ordering” concrete family members
  - modeling of configuration knowledge to a level of detail that is amenable to automation

CONTENTS

Solution Space

generic programming

representing domains as collections of highly general and abstract components, which can be combined in vast numbers to yield very efficient, concrete programs

C++ Standard Template Library of container data structures and algorithms

generic programming is a technique to organise the solution space of generative programming, i.e. implementation components that are orthogonal, reusable, nonredundant

principles and techniques of generic programming are

Solution Space

generic programming

type parameters

- write code that works with different types
- avoid code duplication in statically typed languages
- in C++: function templates and class templates

Smalltalk:

```
sqr: x
^x*x
```

C++:

```
int sqr(int x)
{ return x*x; }
double sqr(double x)
{ return x*x; }
...
```

⬇

```
template <class T>
T sqr(T x)
{ return x*x; }
```

Solution Space

generic programming

different kinds of polymorphism

universal	bounded or not	binding mode of type of parameters	available in languages
unbounded polymorphism	unbounded	dynamic	Smalltalk
unbounded parametric polymorphism unconstrained genericity	unbounded	static	C++
sybtype polymorphism	bounded	dynamic	C++, Java, Eiffel
bounded parametric polymorphism constrained genericity	bounded	static	Eiffel, Ada

6.5,6.6,6.7 - p.173...

Solution Space

generic programming

different kinds of polymorphism

ad hoc	binding mode of implementation	#arguments to distinguish	available in languages
overloading	static	all	C++, Java
partial specialisation	static	some	C++
overriding with single dispatch	dynamic	receiver	C++, Java, Smalltalk
overriding with multiple dispatch multimethods	dynamic	all	CLOS

6.5,6.6,6.7 - p.173...

Solution Space

generic programming

parameterised components

- not only types should be parameterised, but also other variable points
  - design patterns have such variation points
    - template method: some computation steps
    - strategy: part of algorithm
    - bridge: implementation of object
  - mostly dynamic parameterisation is used, allowing parameters to vary at runtime
- BUT, reusable models also have static variation points, i.e. variation from application to application rather than within one application at runtime, which are better implemented using static parameterisation
  - e.g. strategy with static parameterisation:

```
bubblesort<greater>(x,size);
```

```
template<class C,class T>
void bubblesort (T a[ ],unsigned int size)
{...
if (C::compare(a[i],a[j]))
...}
```

```
struct greater
{       template<class T>
        static bool compare(const T& a,const T& b)
        { return a > b; } };
```

Solution Space

generic programming

parameterised programming

- theoretical basis for specifying and building libraries of generic modules (originally in Ada)

- C++ Standard Template Library is an instance of parameterised programming

- modules (components) can be composed with module expressions such as A[B[C,D]] (A<B<C,D> > in STL)

Solution Space

component-oriented template-based C++ programming

component-oriented thinking at class and source level with templates

- complement dynamic configurable designs (such as design patterns) with programming techniques and idioms for static configuration p.15
- support for static configuration in C++ p.16
  - template-based versions of design patterns (bridge, wrapper, adapter, strategy, template method)
    - parameterising binding mode for switching between dynamic and static configuration
      - different static interactions between components, and configuration knowledge in configuration repositories. . . . p.17
        - first hint at automatic configuration

Solution Space

component-oriented template-based C++ programming

types of configuration

		selection of configuration	
		static	dynamic
creation of components and configurations	static	post-runtime optimisations	reflective systems
	dynamic	static typing, templates, static binding, inlining, preprocessing, configuration management	dynamic polymorphism

Smalltalk, CLOS

virtual methods in C++

Solution Space

component-oriented template-based C++ programming

static configuration in C++

static wrapper,
template method

short names for
template instantiations

propagate information
between components,
configuration repositories

uniformly parameterise
number of classes,
name scopes in
configuration repositories

C++ supports static configuration with following features:

- static typing
- static binding
- inlining
- templates
- parameterised inheritance
- typedefs
- member types
- nested classes

```
template<class Superclass>
class SomeClass : public Superclass { ... };
```

```
typedef Vector<Vector<double, 10>, 10 > MyMatrix;
```

```
class SomeClass
{ public: typedef int MyNumberType; ... };
```

```
SomeClass::MyNumberType
```

```
class DerivedClass : public SomeClass
{ public: typedef short MyNumberType; ... };
```

```
class Outer
{ class Inner
   { class MostInner
      { ... }; ... }; ... };
```

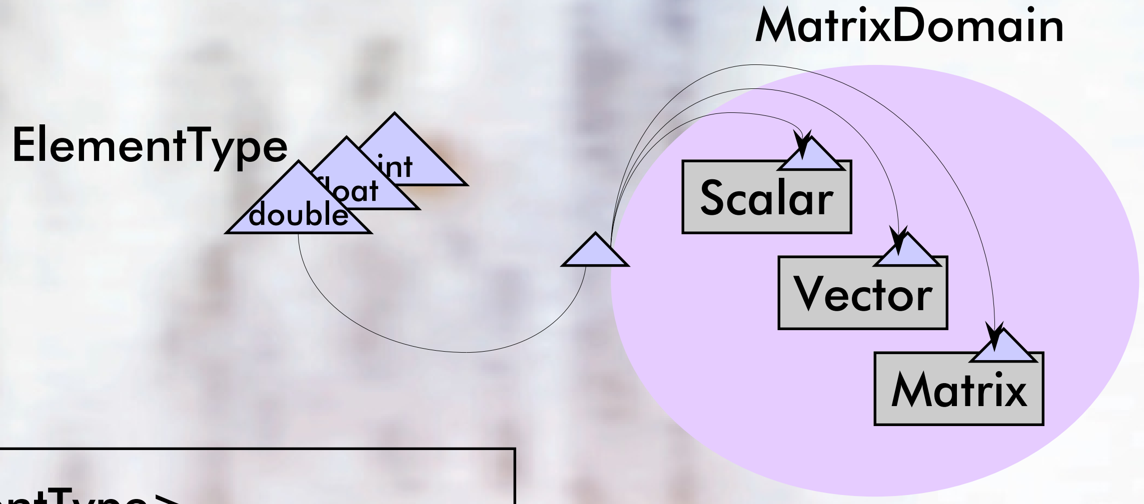
Solution Space

component-oriented
template-based
C++ programming

static
component
interactions

matrix domain: scalars, vectors and matrices

1. consistent parameterisation of the matrix domain



variations of
ElementType are
as local as possible

```
template <class ElementType>
class MatrixDomain
{ public:
      typedef ElementType Scalar;
      class Matrix
      { public: ... };
      class Vector
      { public: Scalar vectorProduct(...);
                  Matrix matrixProduct(...); ... }; };
```

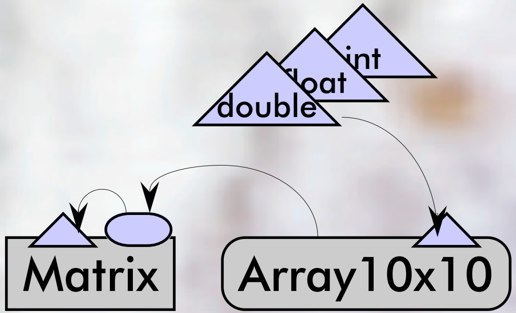
```
typedef MatrixDomain<double> Domain;
Domain::Vector v; //initialising elements of v
Domain::Scalar s = v.vectorProduct(v);
Domain::Matrix m = v.matrixProduct(v);
```

Solution Space

component-oriented
template-based
C++ programming

static
component
interactions

2. components with influence:
advertise properties with member types
(and member constants)



ElementType
is advertised in
component Array10x10
for environment Matrix

```
template <class ElementType_>
class Array10x10
{ public:
      typedef ElementType_ ElementType;
      ... };
```

```
template <class Rep>
class Matrix
{ public:
      typedef typename Rep::ElementType ElementType;
      ... };
```

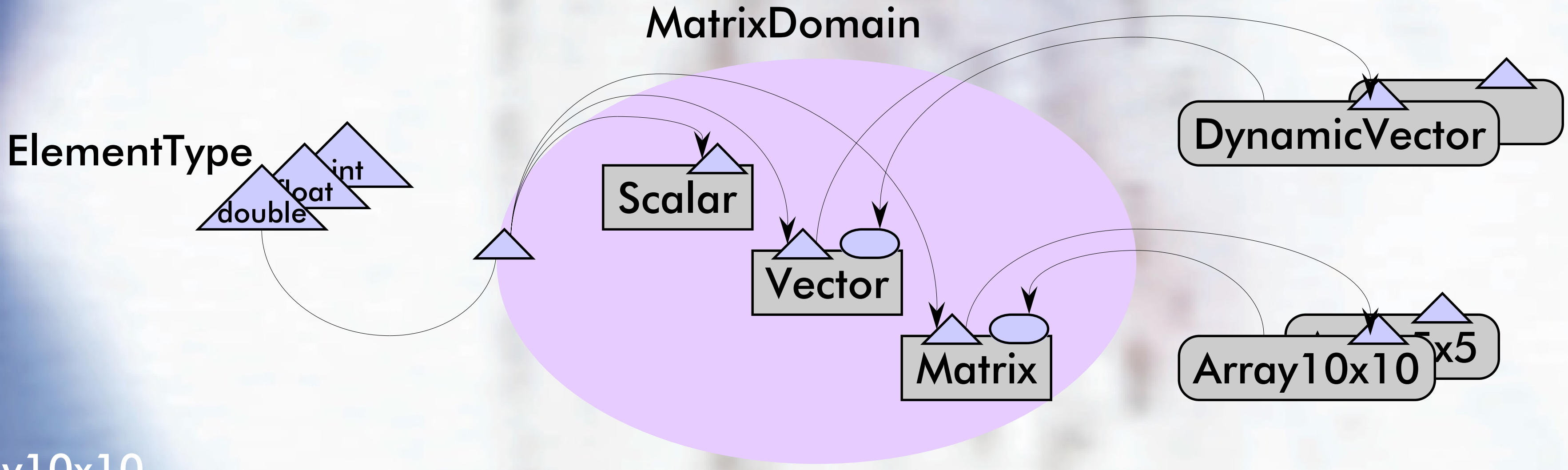
```
Matrix<Array10x10<double> >
```

Solution Space

component-oriented
template-based
C++ programming

static component interactions

3. components under influence:
retrieve properties



ElementType
is retrieved by
component Array10x10
from environment Matrix
(same for component
DynamicVector)

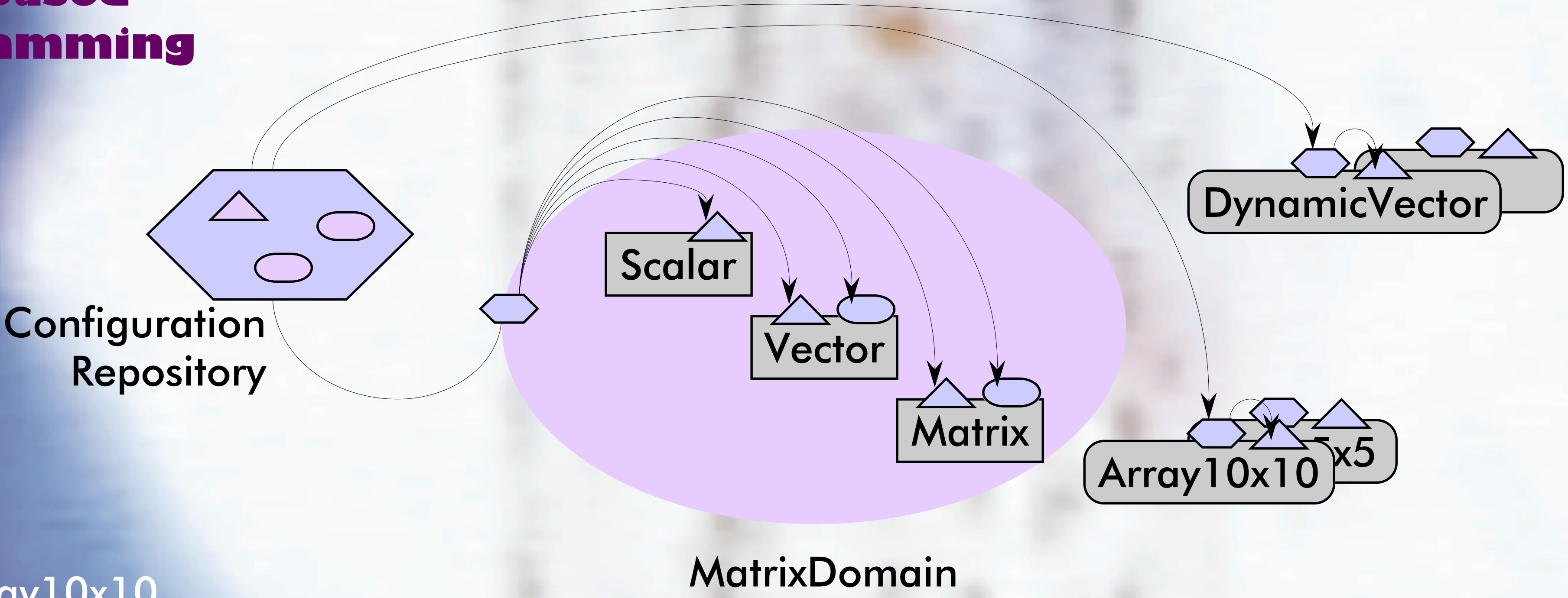
too complicated !!!

Solution Space

component-oriented
template-based
C++ programming

static
component
interactions

3bis. components under influence:
with configuration repositories



ElementType
is retrieved by
component Array10x10
and environment
from configuration repository

Solution Space

component-oriented
template-based
C++ programming

static
component
interactions

3tris. components under influence:
implementation

```
struct Config
{       typedef double Scalar;
        struct ForContainer
        { typedef Scalar ElementType; }
        struct ForVector
        { typedef DynamicVector<Config> Rep; };
        struct ForMatrix
        { typedef Array10x10<Config> Rep; };
        typedef MatrixDomain<Config> Domain;
        typedef Domain::Matrix Matrix;
        typedef Domain::Vector Vector; };
```

```
template <class Config>
class MatrixDomain
{ public:
        typedef typename Config::Scalar Scalar;
        class Matrix
        { ...
          private: typedef typename Config::ForMatrix::Rep Rep; ... };
        class Vector
        { ...
          private: typedef typename Config::ForVector::Rep Rep; ... }; };
```

using nested structs
for structuring
configuration repositories

```
template <class Config> class Array10x10
{ public: typedef typename Config::ForContainer::ElementType ElementType; ... };

template <class Config> class DynamicArray
{ ... };
```

7.11 - p.236

Solution Space

component-oriented template-based C++ programming

static component interactions

4. components influences other component: model dependencies between parameters

```
template <int r, int c> struct Config
{ typedef SelectSquareIfTrue<r==c>::Base Base; … };
```

```
template <bool cond> struct SelectSquareIfTrue
{ typedef NotSquare Base; };
template <> struct SelectSquareIfTrue<true>
{ typedef Square Base };
```

```
template <class Config>
class Matrix : public typename Config::Base
{ … };

class Square
{ public: double determinant() const { … };
};
class NotSquare {};
```

first hint at automatic configuration

SelectSquareIfTrue uses template specialisation

```
Matrix<Config<3,3> > m;
m.determinant() //OK
Matrix<Config<3,4> > m;
m.determinant() //compiler reports error
```

Solution Space

aspect-oriented programming

domain concepts are represented by well-localised modular units, expressed by object-oriented, generic and component-oriented programming

sometimes these traditional modularisation constructs fall short...

aspects

implementing some domain concepts requires
- inserting code fragments in many existing components
- changing component structure to get reasonable performance

- aspect-oriented decomposition
    - subject-oriented programming
    - composition filters
    - adaptive programming
    - relation to domain engineering
- how aspects arise
- composition mechanismes
- expressing aspects in programming languages
- implementation technologies for AOP

CONTENTS

Configuration Knowledge

domain-specific languages

DSL is:
- specialised problem-oriented language
- used to "order" concrete members of a system family

in general several DSLs are used to specify a complete application

modular DSLs are most reusable, scaleable, flexible

different kinds of DSLs, depending on the implementation technology:
- fixed, separate DSLs (such as TEX and SQL):
  - need a language translator, costly to develop from scratch!
- embedded DSLs:
  - embedded in general-purpose language;
  - implemented using constructs of this language;
  - meta-programming is needed to express domain-specific optimisations, error reporting, syntax;
- modularly composable DSLs:
  - each DSL is a component, can be composed;
  - all modular DSLs should be implemented in common language platform with infrastructure to implement language plug-ins;
  - encapsulated and aspectual DSLs: latter influences semantics of and interacts with other components (modular DSLs)

implemented using preprocessor, meta-programming, modularly extendible compilers and programming environments

Configuration Knowledge

generators

- generators are used to achieve:
  - intentional code
  - high performance
- generators should compute an efficient implementation for a high-level specification
- generators avoid the library scaling problem

tasks of a generator:

- check input specification
- completes specification using default settings
- performs optimisations
- generates implementation

generator tasks = automatic configuration process

generator implements configuration knowledge of domain model

developing domain model provides basis for implementing generators

generators should be modularised
generators are implemented using smaller cooperating generator

Configuration Knowledge

static meta-programming

- meta-programming at compile-time
- two-level languages:
  - distinction between static code and dynamic code
  - static code is executed at compile-time by compiler
  - dynamic code is compiled by compiler and executed at run-time
  - static code can manipulate dynamic code
- static code can be used to write generators
- templates in C++ constitute a Turing-complete sublanguage to write static code
- dynamic code in C++ is imperative, object-oriented and procedural
- static code in C++ is functional:
  - class templates as functions
  - no assignments
  - recursion instead of iteration
  - ...

```
template <int> struct Factorial
{ enum { RET = Factorial<n-1>::RET * n }; };

template<> struct Factorial<0>
{ enum { RET = 1 }; };

cout << “fact(7)= ” << Factorial<7>::RET << endl;
```

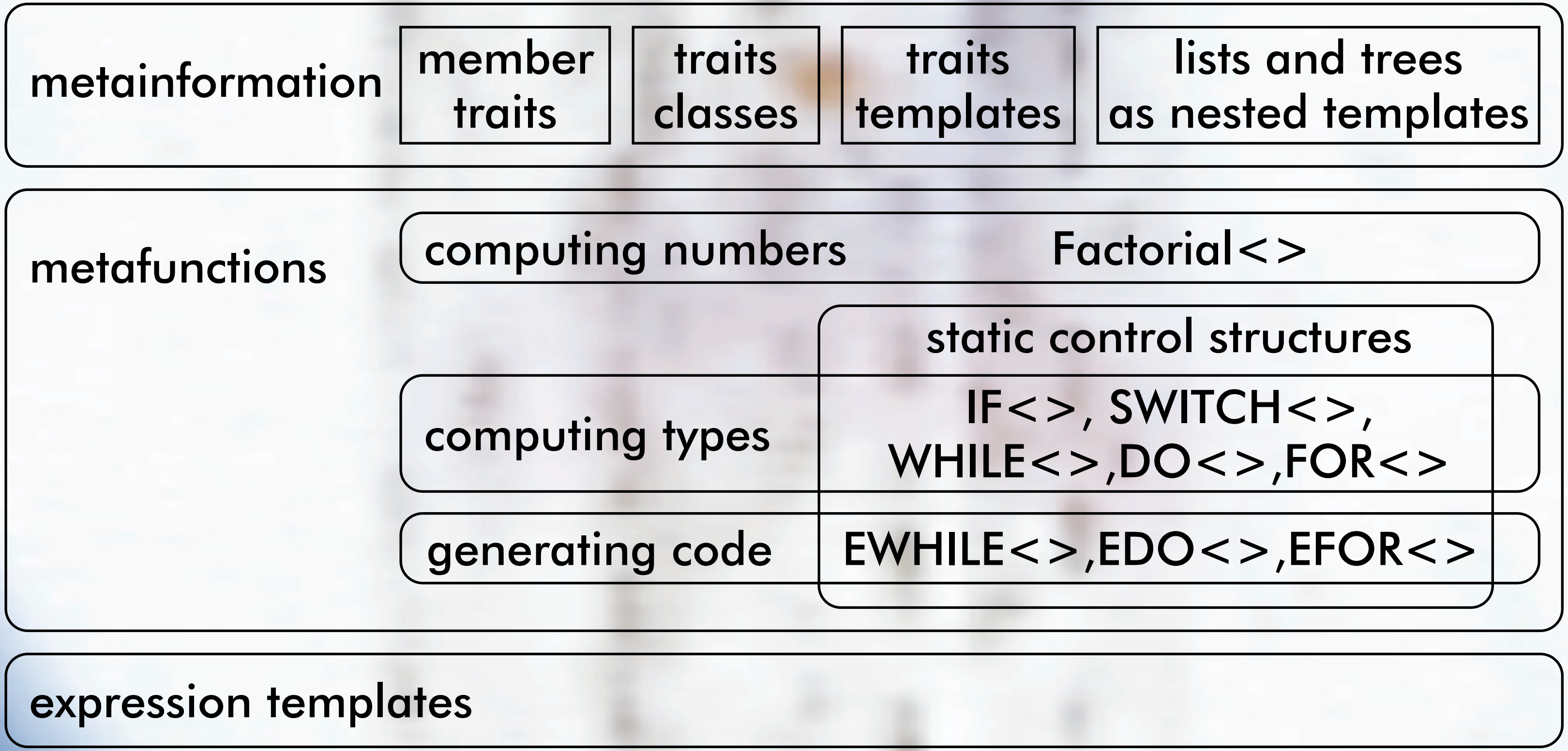
code generated by C++ compiler is the same as the code generated for

```
cout << “fact(7)= ”
      << 5040
        << endl;
```

Configuration Knowledge

static meta-programming

template meta-programming



metainformation

- member traits
- traits classes
- traits templates
- lists and trees as nested templates

metafunctions

- computing numbers — Factorial<>
- static control structures
  - computing types — IF<>, SWITCH<>, WHILE<>,DO<>,FOR<>
  - generating code — EWHILE<>,EDO<>,EFOR<>

expression templates

10.7, ... - p.410

Configuration Knowledge

static meta-programming

compile-time control structures

static IF<>
with templates

```
namespace intimate
{ struct SelectThen
     { template<class ThenType, class ElseType>
        struct Result
        { typedef ThenType RET; }; };
  struct SelectElse
     { template<class ThenType, class ElseType>
        struct Result
        { typedef ElseType RET; }; };
  template<bool condition>
  struct ChooseSelector
     { typedef SelectThen RET; };
  template<false>
  struct ChooseSelector
     { typedef SelectElse RET; }; };
```

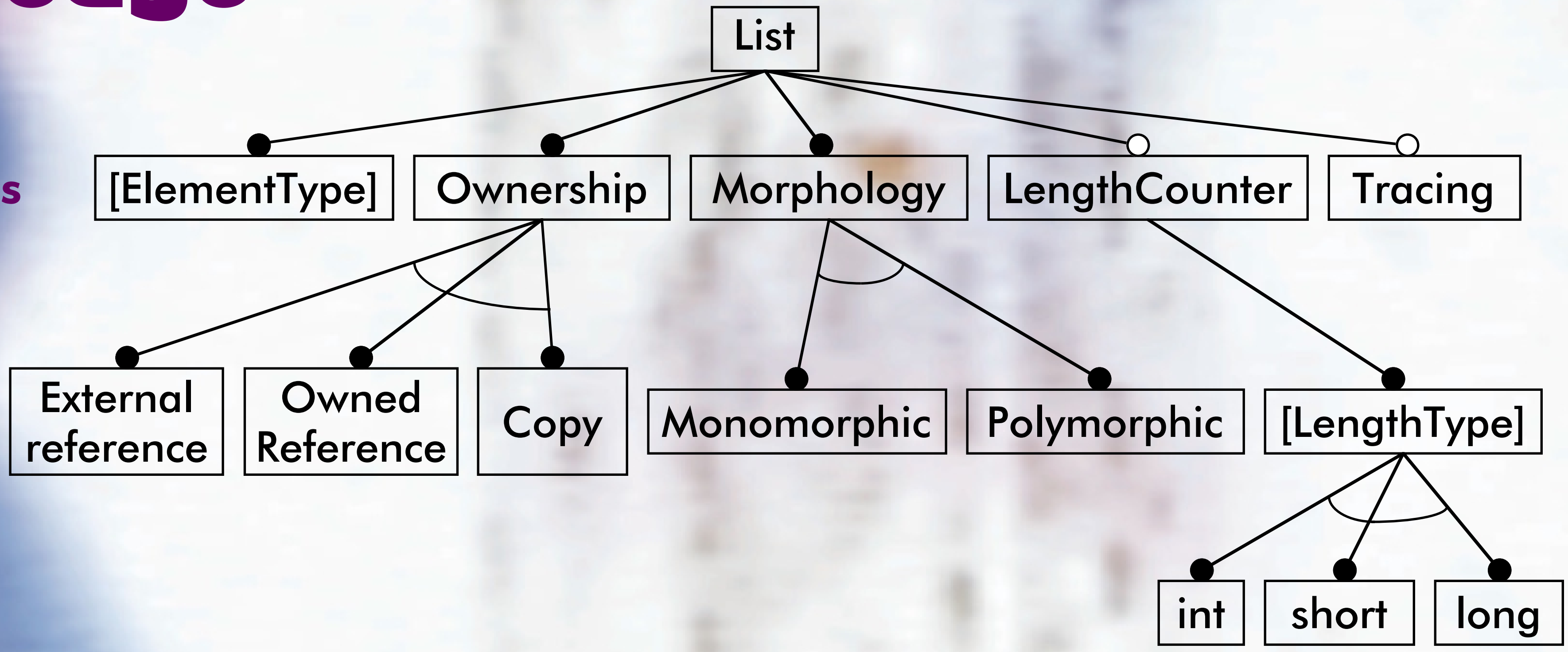
```
template<bool condition, class ThenType, class ElseType>
class IF
{ typedef typename
     intimate::ChooseSelector<condition>::RET Selector;
  public: typedef typename
     Selector::template Result<ThenType,ElseType>::RET RET; };
```

Configuration Knowledge

example

results of domain analysis

in the domain of list containers...



concepts
and features
of the domain
of list containers

Configuration Knowledge

example

results of domain design

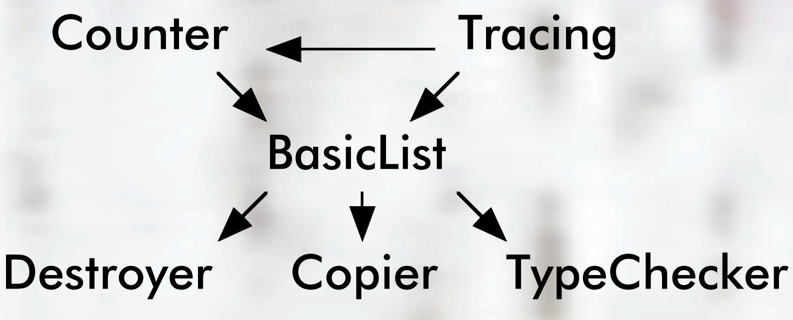
BasicList:	Destroyer:	Copier:	TypeChecker:
PtrList	ElementDestroyer	PolymorphicCopier	DynamicTypeChecker
	EmptyDestroyer	MonomorphicCopier	EmptyTypeChecker
		EmptyCopier	

Counter:	Tracing:
LengthList	TracedList

categories of responsabilities

components listed per category

dependencies between categories



Configuration Knowledge

example

results of domain design

```
List                : TracedList[OptCounterList] | OptCounterList
OptCounterList      : LenghtList[BasicList] | BasicList
BasicList           : PtrList[Config]
Config              :
        ElementType : [ElementType]
        Destroyer   : ElementDestroyer | EmptyDestroyer
        Copier      : PolymorphicCopier | MonomorphicCopier | EmptyCopier
        TypeChecker : DynamicTypeChecker | EmptyTypeChecker
        LengthType  : int | short | long
        ReturnType  // final list type
```

GenVoca grammar

Configuration Knowledge

example

results of domain implementation

basic list component

```
template<class Config_>
class PtrList {
public: typedef Config_ Config;
private:
  typedef typename Config::ElementType ElementType;
  typedef typename Config::SetHeadElTpe SetHeadElTpe;
  typedef typename Config::ReturnType ReturnType;
  typedef typename Config::Destroyer Destroyer;
  typedef typename Config::TypeChecker TypeChecker;
  typedef typename Config::Copier Copier;
public:
  void setHead(SetHeadElementType& h)
  { TypeChecker::check(h);
     head_ = Copier::copy(h); }
...
private:
  ElementType* head_;
  ReturnType* tail_;
```

a type checker

```
template<class ElementType>
struct EmptyTypeChecker
{ static void check(const ElementType& e) {} };
```

Configuration Knowledge

example

configuration knowledge

vocabulary

```
enum Ownership {ext_ref, own_ref, cp};
enum Morphology {mono, poly};
enum CounterFlag {with_counter, no_counter};
enum TracingFlag {with_tracing, no_tracing};
```

default settings

```
Ownership    = cp;
Morphology   = mono;
CounterFlag       = no_count
TracingFlag  = no_tracing;
LengthType   = int
```

for ordering
(specifying)
list containers

```
LIST_GENERATOR<Person, cp, poly, with_counter, with_tracing>::RET
```

```
LIST_GENERATOR<Person, ext_ref, poly>::RET
```

Configuration Knowledge

example

generator

```
template      <
        class           ElementType_,
        Ownership    ownership     = cp,
        Morphology  morphology  = mono,
        CounterFlag        counterFlag  = no_count
        TracingFlag  tracingFlag    = no_tracing,
        class           LengthType_ = int >

class LIST_GENERATOR
{ public:
        typedef LIST_GENERATOR<
                ElementType_,
                ownership,
                morphology,
                counterFlag,
                tracingFlag,
                LengthType_> Generator;
...
```

Configuration Knowledge

example

generator

```
private:
        enum {
                isCopy          = ownership       == cp,
                isOwnRef        = ownership       == own_ref,
                isMono          = morphology      == mono,
                hasCounter      = counterFlag     == with_counter,
                doesTracing     = tracingFlag     == with_tracing };

        typedef
                IF<isCopy || isOwnRef,
                        ElementDestroyer<ElementType_>,
                        EmptyDestroyer<ElementType_> >::RET Destroyer_;
...
public: ...
        struct Config
        {       typedef ElementType_ElementType;
                typedef Destroyer_ Destroyer;
                ... };
};
```