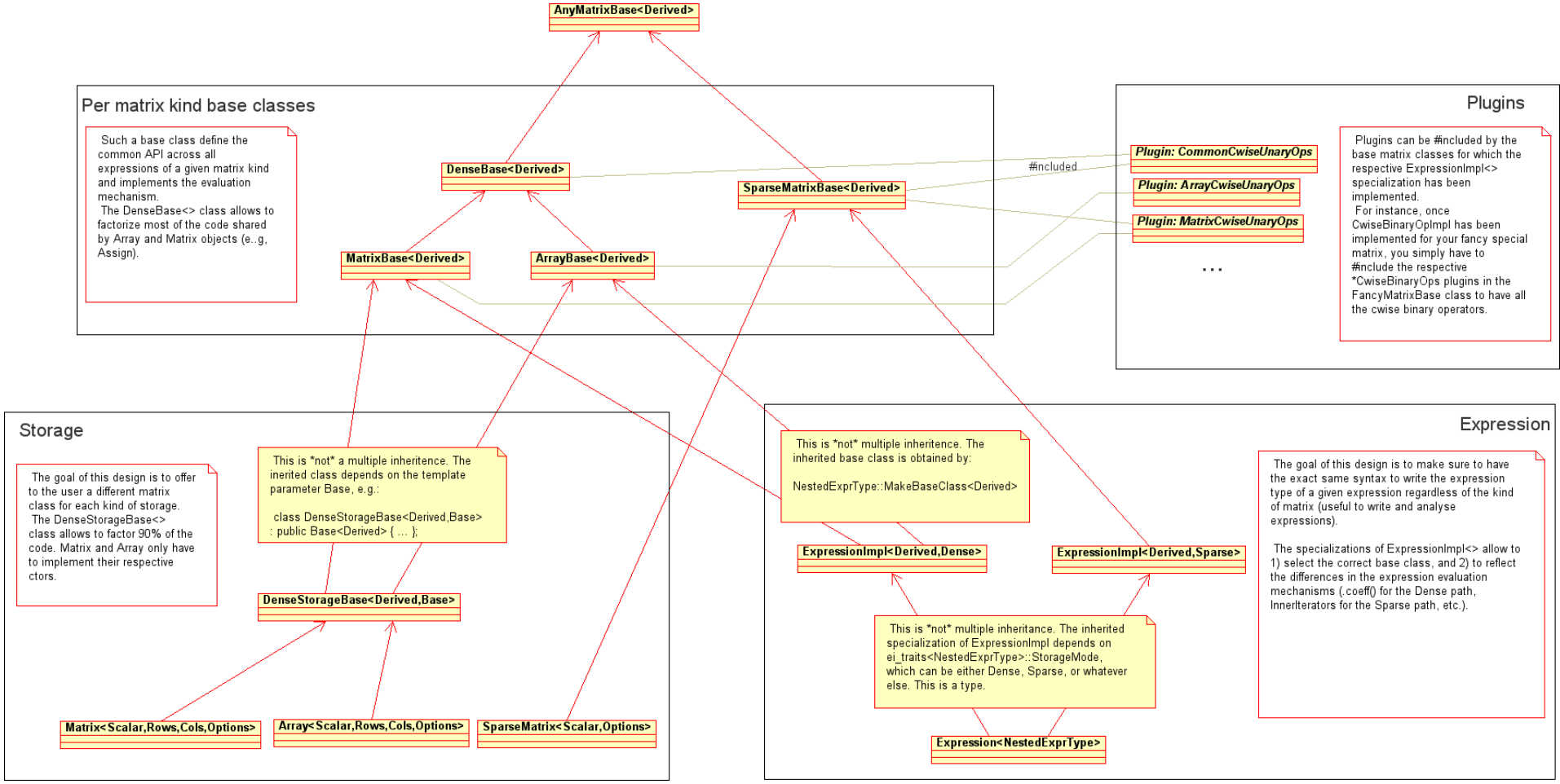
Eigen's class hierarchy


AnyMatrixBase<Derived>
Per matrix kind base classes
Such a base class define the common API across all expressions of a given matrix kind and implements the evaluation mechanism.
The DenseBase<> class allows to factorize most of the code shared by Array and Matrix objects (e..g, Assign).
DenseBase<Derived>
SparseMatrixBase<Derived>
MatrixBase<Derived>
ArrayBase<Derived>
#included
Plugins
Plugin: CommonCwiseUnaryOps
Plugin: ArrayCwiseUnaryOps
Plugin: MatrixCwiseUnaryOps
...
Plugins can be #included by the base matrix classes for which the respective ExpressionImpl<> specialization has been implemented.
For instance, once CwiseBinaryOpImpl has been implemented for your fancy special matrix, you simply have to #include the respective *CwiseBinaryOps plugins in the FancyMatrixBase class to have all the cwise binary operators.
Storage
The goal of this design is to offer to the user a different matrix class for each kind of storage.
The DenseStorageBase<> class allows to factor 90% of the code. Matrix and Array only have to implement their respective ctors.
This is *not* a multiple inheritence. The inerited class depends on the template parameter Base, e.g.:
class DenseStorageBase<Derived,Base> : public Base<Derived> { ... };
DenseStorageBase<Derived,Base>
Matrix<Scalar,Rows,Cols,Options>
Array<Scalar,Rows,Cols,Options>
SparseMatrix<Scalar,Options>
This is *not* multiple inheritence. The inherited base class is obtained by:
NestedExprType::MakeBaseClass<Derived>
ExpressionImpl<Derived,Dense>
ExpressionImpl<Derived,Sparse>
This is *not* multiple inheritance. The inherited specialization of ExpressionImpl depends on ei_traits<NestedExprType>::StorageMode, which can be either Dense, Sparse, or whatever else. This is a type.
Expression<NestedExprType>
Expression
The goal of this design is to make sure to have the exact same syntax to write the expression type of a given expression regardless of the kind of matrix (useful to write and analyse expressions).
The specializations of ExpressionImpl<> allow to 1) select the correct base class, and 2) to reflect the differences in the expression evaluation mechanisms (.coeff() for the Dense path, InnerIterators for the Sparse path, etc.).

Expression templates

- Example:

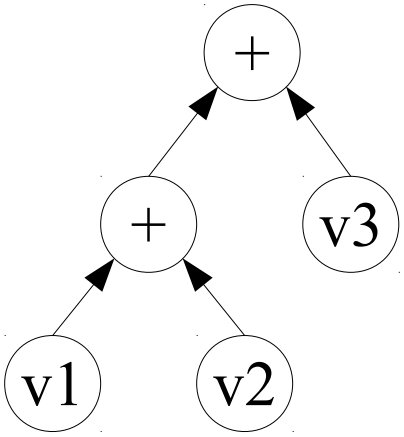
```
v3 = v1 + v2 + v3;
```

- Expression templates:
  - "+" returns an expression

    Sum<typeof(v1),typeof(v2)>

  - expression tree

    Sum<Sum<typeof(v1),typeof(v2),
    typeof(v3)>



  - evaluation entirely performed in the "="

```
for(i=0; i<v3.size(); ++i)
    v3[i] = v1[i] + v2[i] + v3[i];
```

E.T. in Eigen 2.0

- v1 + v2 returns a Sum expression:
  - *Sum<typeof(v1),typeof(v2)>*
- with:
  - *template<Lhs,Rhs> class Sum {*
    const Lhs& m_lhs;
    const Rhs& m_rhs;

    Sum(const Lhs& l, const Rhs& r) : m_lhs(l), m_rhs(r) {}

    *Scalar **coeff**(int i, int j) { return m_lhs.coeff(i,j) + m_rhs.coeff(i,j); }*

    int rows() {…}
    int cols() {...}
    };
- What about: (v1 + v2) + v3 ?

E.T. in Eigen 2.0

- v1 + v2 returns a Sum expression:
  - *Sum<typeof(v1),typeof(v2)>*
- with:
  - *template<Lhs,Rhs> class Sum* ***: MatrixBase<Sum<Lhs,Rhs>>*** *{*

```
const Lhs& m_lhs;
const Rhs& m_rhs;

Sum(const Lhs& l, const Rhs& r) : m_lhs(l), m_rhs(r) {}

Scalar coeff(int i, int j) { return m_lhs.coeff(i,j) + m_rhs.coeff(i,j); }

int rows() {…}
int cols() {...}
};
```

- What about: (v1 + v2) + v3 ?
  - > common base class: MatrixBase

The base class

```
template<Derived> class MatrixBase {
  /* … */

  template<OtherDerived> Sum<Derived,OtherDerived>
  operator+(const MatrixBase<OtherDerived>& other)
  { return Sum<Derived,OtherDerived>(*this, other); }

  template<Src> Derived& operator=(const MatrixBase<Src>& src) {
    foreach coeffs i,j do
      *this.coeff(i,j) = other.coeff(i,j);
    }

};
```

The ei_traits class

```
template<Lhs,Rhs> class Sum : MatrixBase<Sum<Lhs,Rhs>> {
  /* … */
  typedef typename Lhs::Scalar Scalar;
};

template<Derived> class MatrixBase {
  /* … */
  typedef typename Derived::Scalar Scalar;
};
```

! circular dependency !

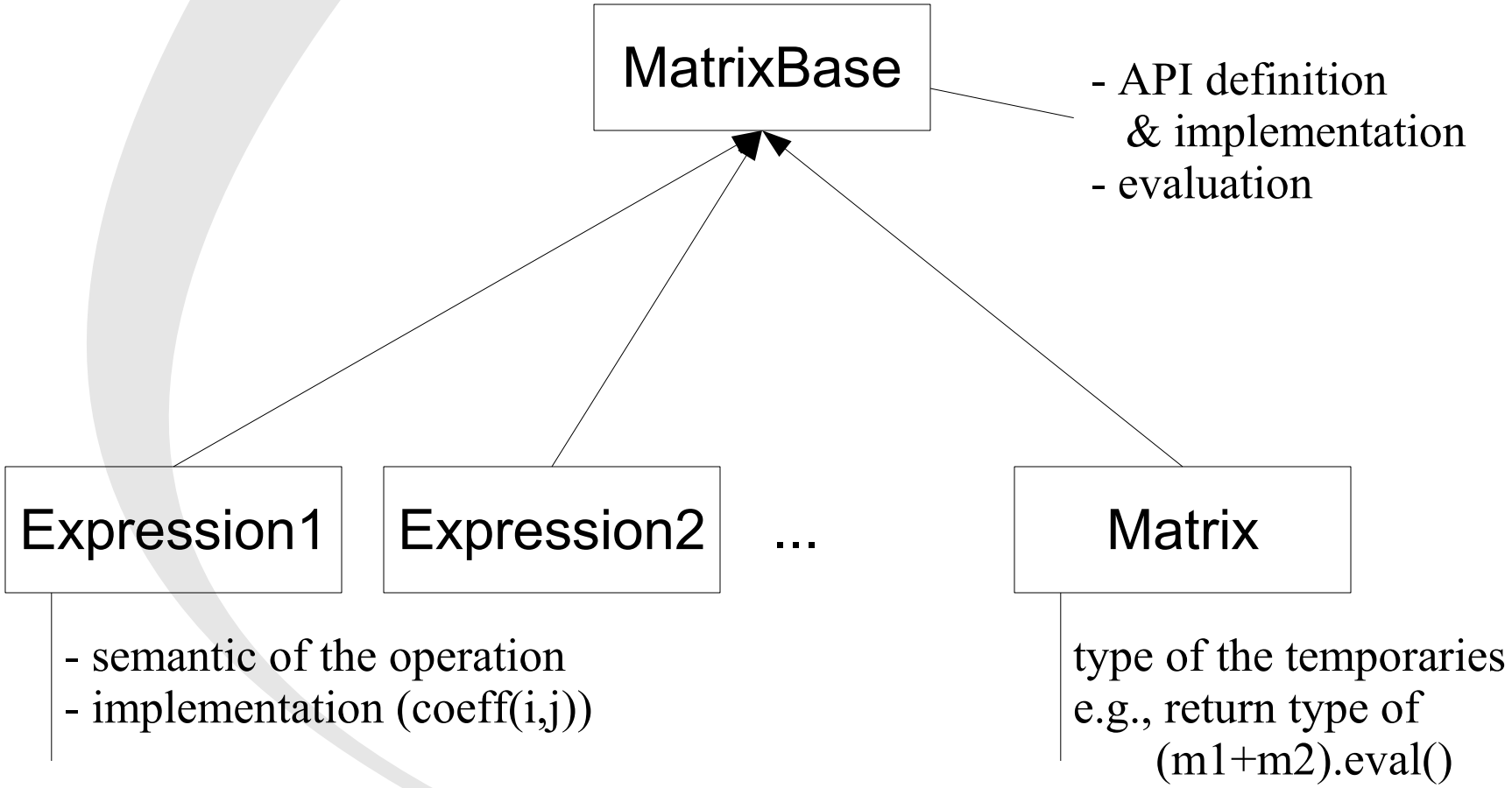
The ei_traits class

```
template<Lhs,Rhs> struct ei_traits<Sum<Lhs,Rhs>> {
  /* .. */
  typedef typename Lhs::Scalar Scalar;
};


template<Lhs,Rhs> class Sum : MatrixBase<Sum<Lhs,Rhs>> {
  /* … */
  typedef typename ei_traits<Sum>::Scalar Scalar;
};

template<Derived> class MatrixBase {
  /* … */
  typedef typename ei_traits<Derived>::Scalar Scalar;
};
```

Eigen 2.0: Class hierarchy

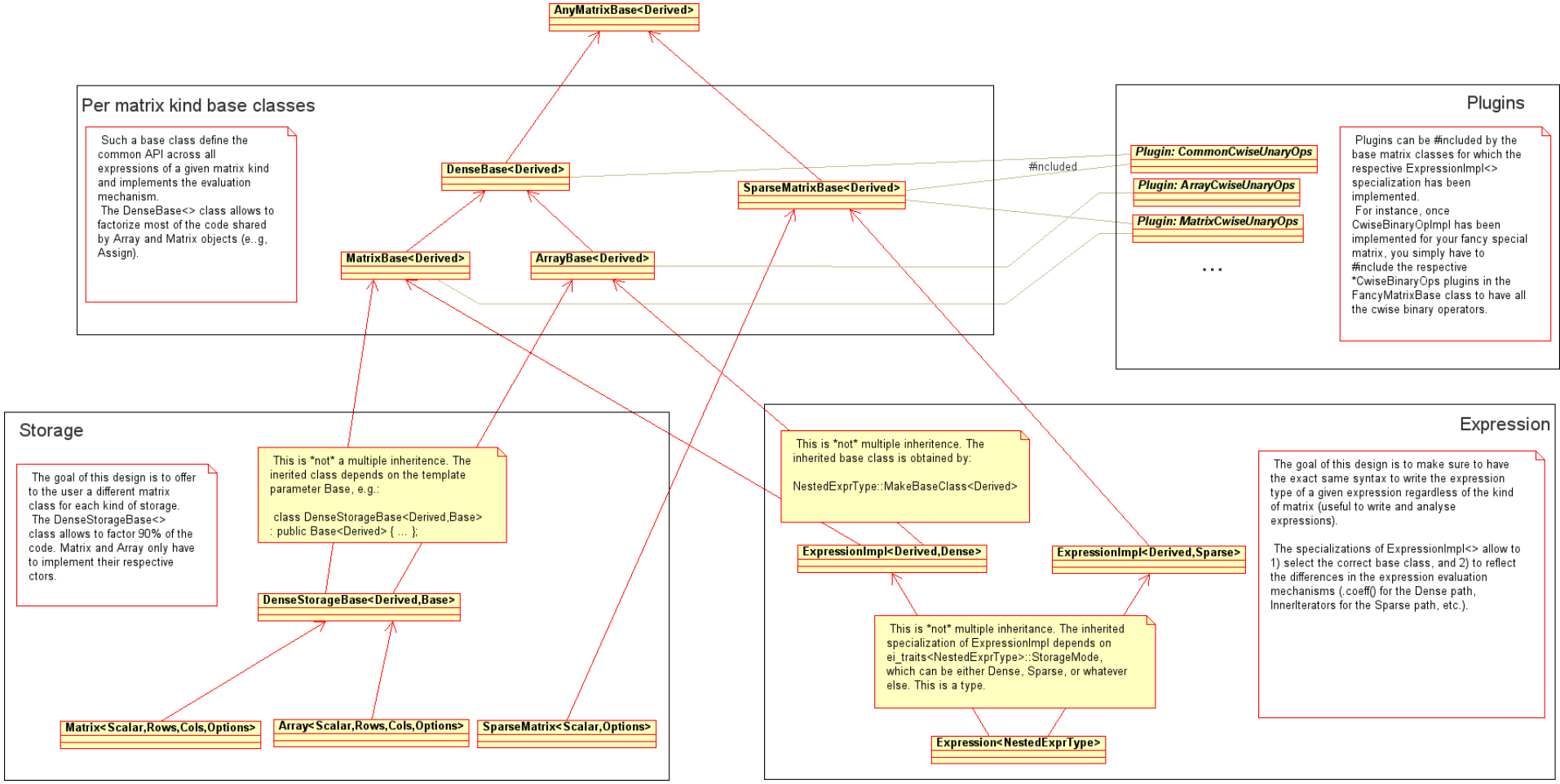


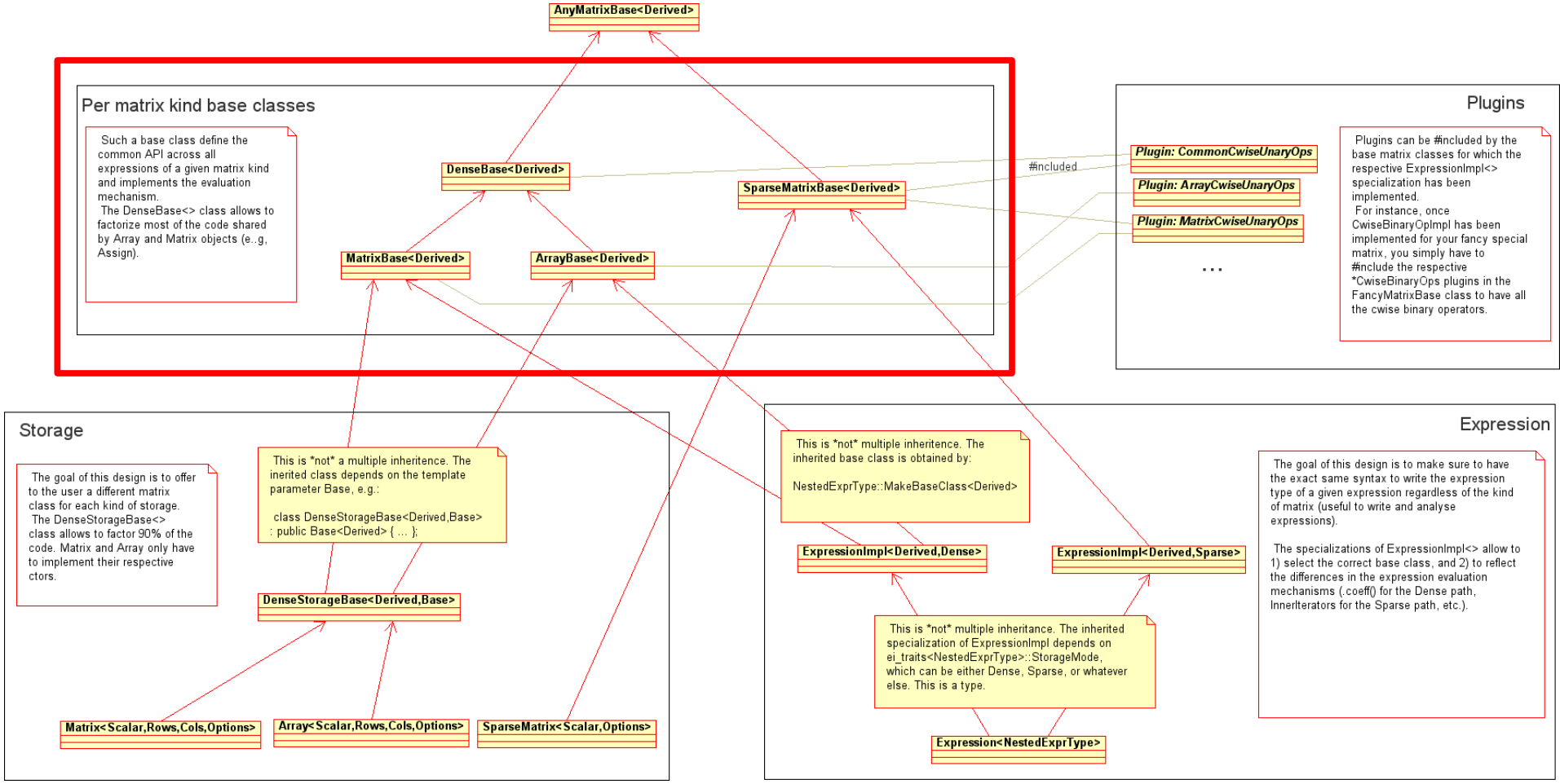
Motivations for 3.0

- Eigen 2.0: limited to a single type of Matrix
  - we want E.T. for all kind of objects
  - different storages
    - dense, sparse matrices, triangular matrices, etc.
      - require different evaluation mechanisms
      - coeffs based, interators, etc.
  - different semantics
    - matrix, array, transform, etc.
      - slightly different API

Additional goals / constraints

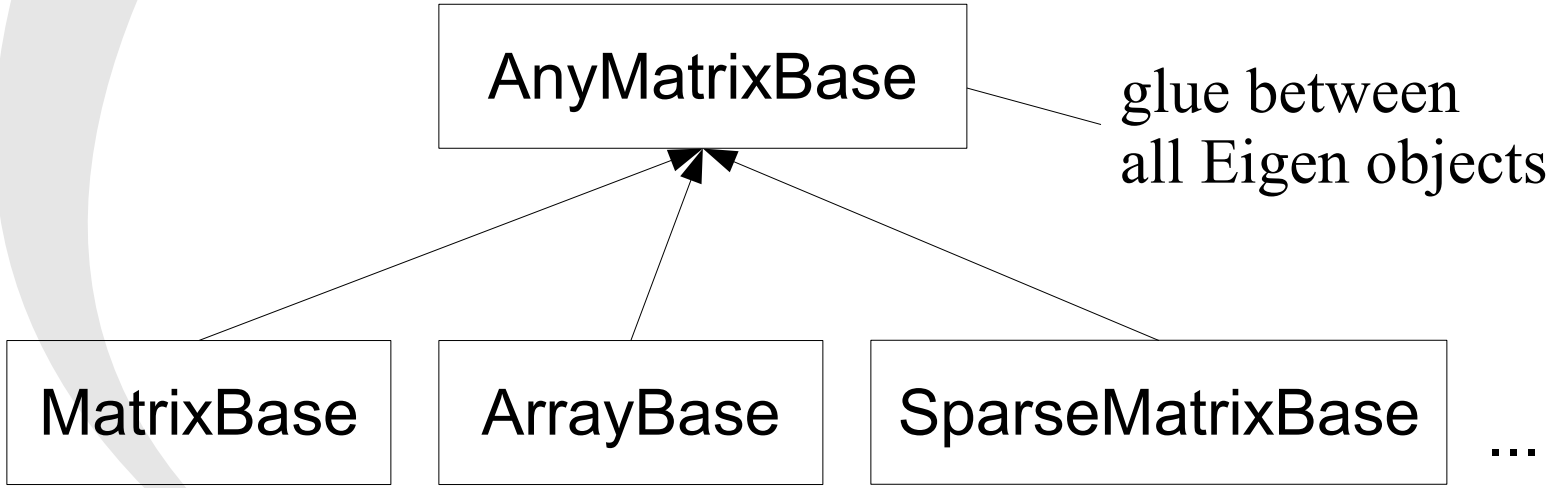
- zero code duplication
  - operator+ declared, documented, implemented only once!
- uniform expression tree
  - typeof(v1+v2) == Sum<typeof(v1),typeof(v2)>
- decouple the semantic and the implementation of the expressions


AnyMatrixBase<Derived>
Per matrix kind base classes
Such a base class define the common API across all expressions of a given matrix kind and implements the evaluation mechanism.
The DenseBase<> class allows to factorize most of the code shared by Array and Matrix objects (e..g, Assign).
DenseBase<Derived>
SparseMatrixBase<Derived>
MatrixBase<Derived>
ArrayBase<Derived>
#included
Plugins
Plugin: CommonCwiseUnaryOps
Plugin: ArrayCwiseUnaryOps
Plugin: MatrixCwiseUnaryOps
...
Plugins can be #included by the base matrix classes for which the respective ExpressionImpl<> specialization has been implemented.
For instance, once CwiseBinaryOpImpl has been implemented for your fancy special matrix, you simply have to #include the respective *CwiseBinaryOps plugins in the FancyMatrixBase class to have all the cwise binary operators.
Storage
The goal of this design is to offer to the user a different matrix class for each kind of storage.
The DenseStorageBase<> class allows to factor 90% of the code. Matrix and Array only have to implement their respective ctors.
This is *not* a multiple inheritence. The inerited class depends on the template parameter Base, e.g.:
class DenseStorageBase<Derived,Base> : public Base<Derived> { ... };
DenseStorageBase<Derived,Base>
Matrix<Scalar,Rows,Cols,Options>
Array<Scalar,Rows,Cols,Options>
SparseMatrix<Scalar,Options>
This is *not* multiple inheritence. The inherited base class is obtained by:
NestedExprType::MakeBaseClass<Derived>
ExpressionImpl<Derived,Dense>
ExpressionImpl<Derived,Sparse>
This is *not* multiple inheritance. The inherited specialization of ExpressionImpl depends on ei_traits<NestedExprType>::StorageMode, which can be either Dense, Sparse, or whatever else. This is a type.
Expression<NestedExprType>
Expression
The goal of this design is to make sure to have the exact same syntax to write the expression type of a given expression regardless of the kind of matrix (useful to write and analyse expressions).
The specializations of ExpressionImpl<> allow to 1) select the correct base class, and 2) to reflect the differences in the expression evaluation mechanisms (.coeff() for the Dense path, InnerIterators for the Sparse path, etc.).

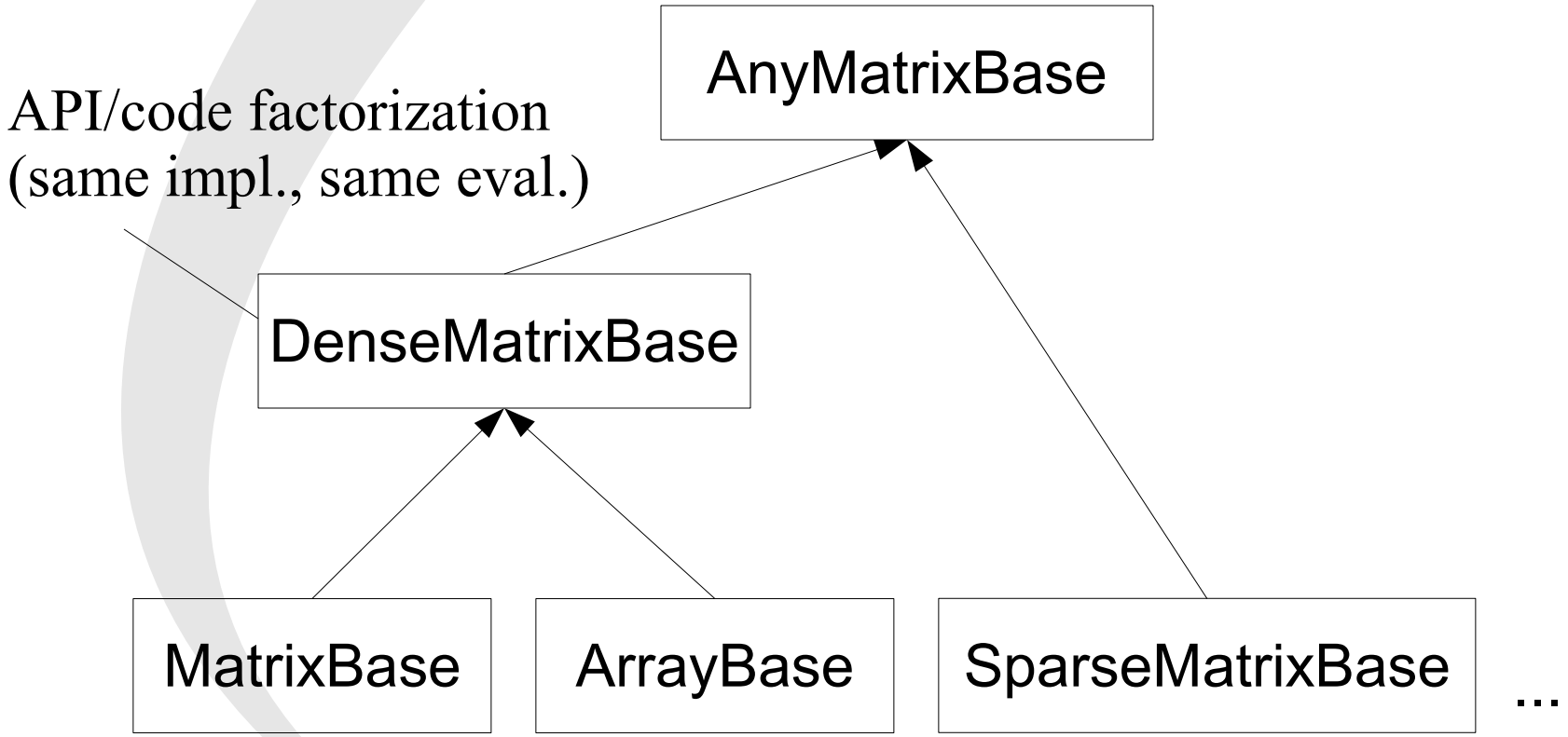

AnyMatrixBase<Derived>
Per matrix kind base classes
Such a base class define the common API across all expressions of a given matrix kind and implements the evaluation mechanism.
The DenseBase<> class allows to factorize most of the code shared by Array and Matrix objects (e..g, Assign).
DenseBase<Derived>
SparseMatrixBase<Derived>
MatrixBase<Derived>
ArrayBase<Derived>
#included
Plugins
Plugin: CommonCwiseUnaryOps
Plugin: ArrayCwiseUnaryOps
Plugin: MatrixCwiseUnaryOps
...
Plugins can be #included by the base matrix classes for which the respective ExpressionImpl<> specialization has been implemented.
For instance, once CwiseBinaryOpImpl has been implemented for your fancy special matrix, you simply have to #include the respective *CwiseBinaryOps plugins in the FancyMatrixBase class to have all the cwise binary operators.
Storage
The goal of this design is to offer to the user a different matrix class for each kind of storage.
The DenseStorageBase<> class allows to factor 90% of the code. Matrix and Array only have to implement their respective ctors.
This is *not* a multiple inheritence. The inerited class depends on the template parameter Base, e.g.:
class DenseStorageBase<Derived,Base> : public Base<Derived> { ... };
DenseStorageBase<Derived,Base>
Matrix<Scalar,Rows,Cols,Options>
Array<Scalar,Rows,Cols,Options>
SparseMatrix<Scalar,Options>
Expression
This is *not* multiple inheritence. The inherited base class is obtained by:
NestedExprType::MakeBaseClass<Derived>
ExpressionImpl<Derived,Dense>
ExpressionImpl<Derived,Sparse>
This is *not* multiple inheritance. The inherited specialization of ExpressionImpl depends on ei_traits<NestedExprType>::StorageMode, which can be either Dense, Sparse, or whatever else. This is a type.
Expression<NestedExprType>
The goal of this design is to make sure to have the exact same syntax to write the expression type of a given expression regardless of the kind of matrix (useful to write and analyse expressions).
The specializations of ExpressionImpl<> allow to 1) select the correct base class, and 2) to reflect the differences in the expression evaluation mechanisms (.coeff() for the Dense path, InnerIterators for the Sparse path, etc.).

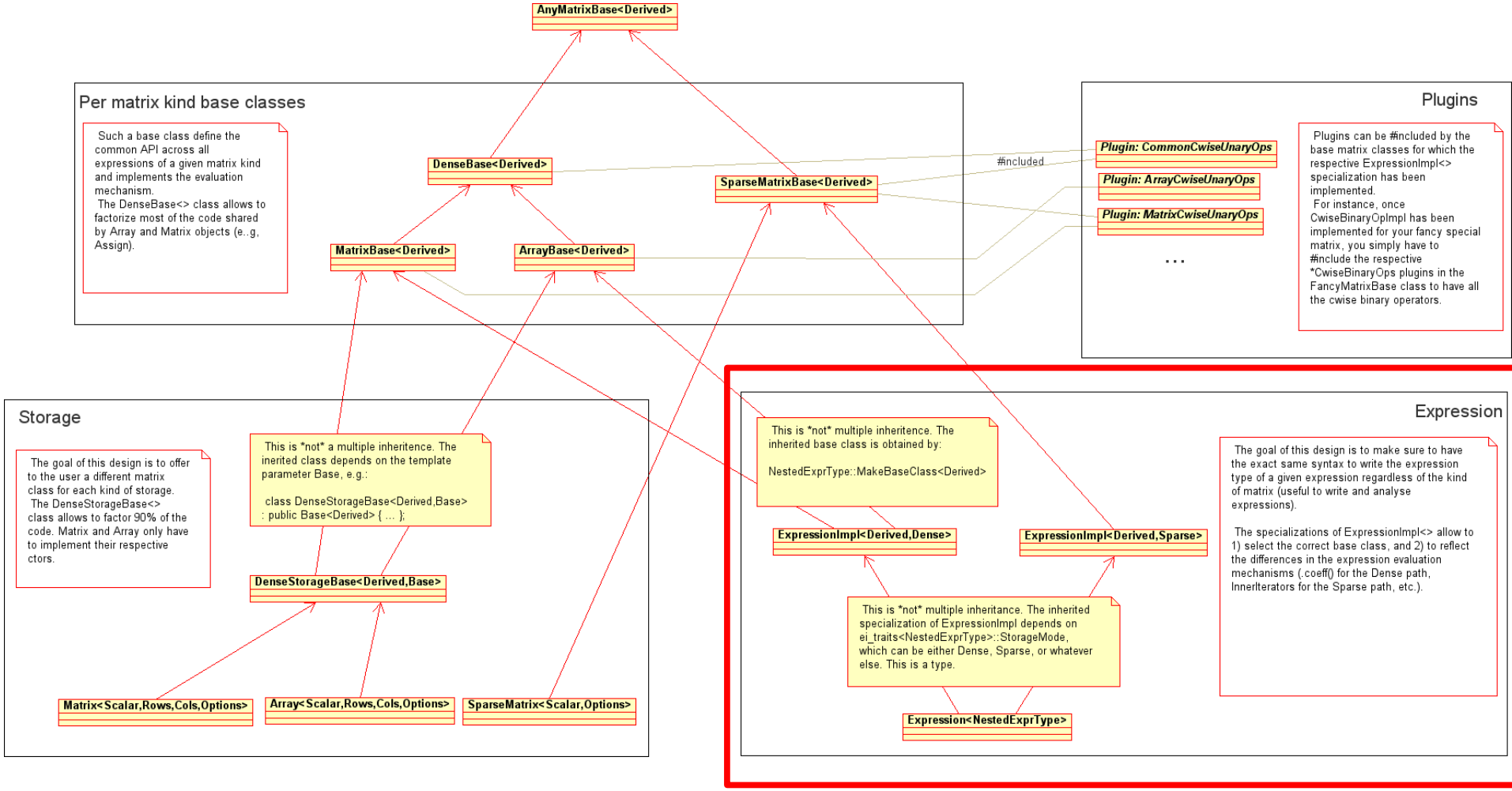
One base class per kind of matrix

- Why?
  - slightly different API
  - different evaluation mechanisms (coeff, iterators, etc.)

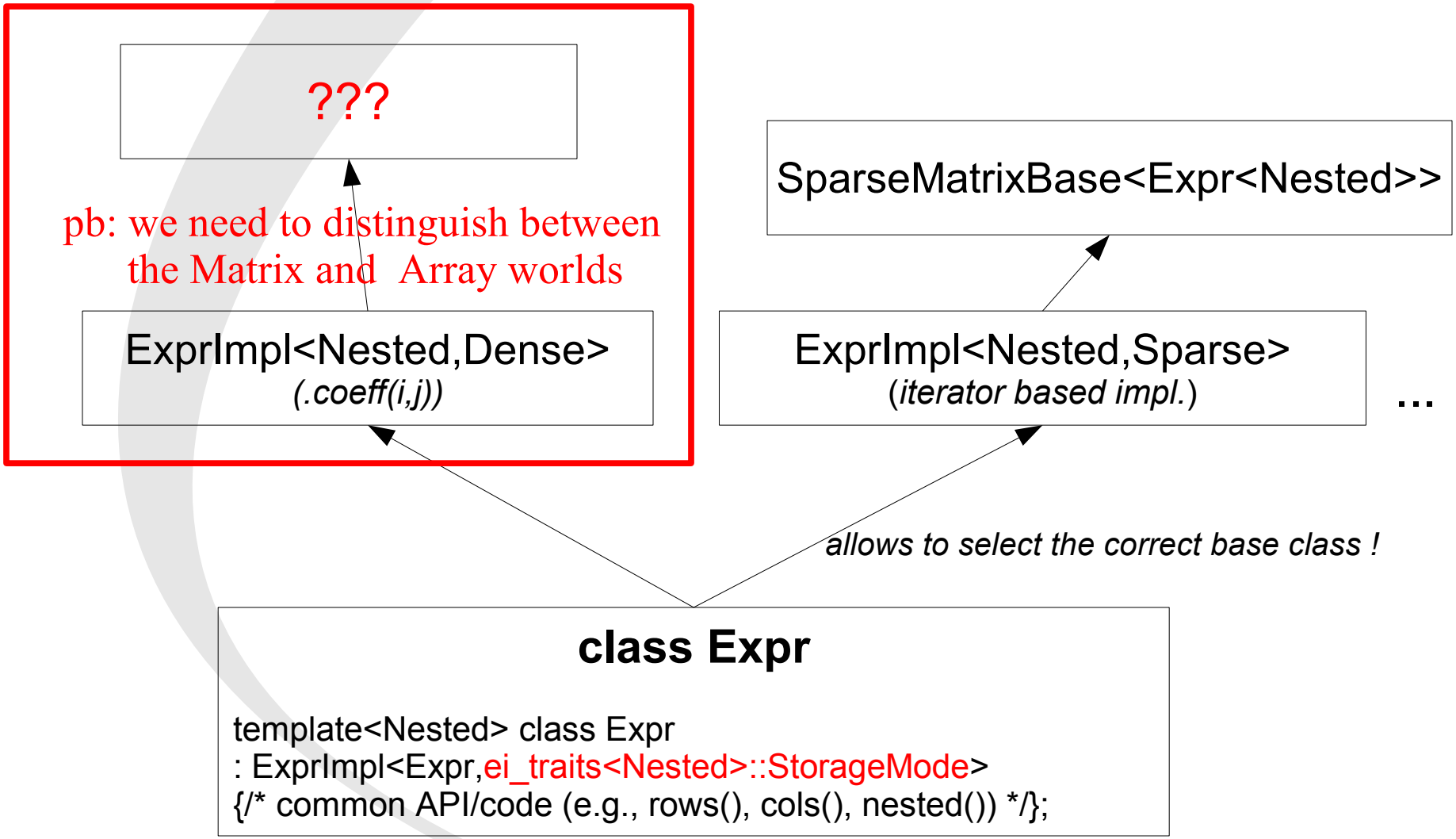


One base class per kind of matrix



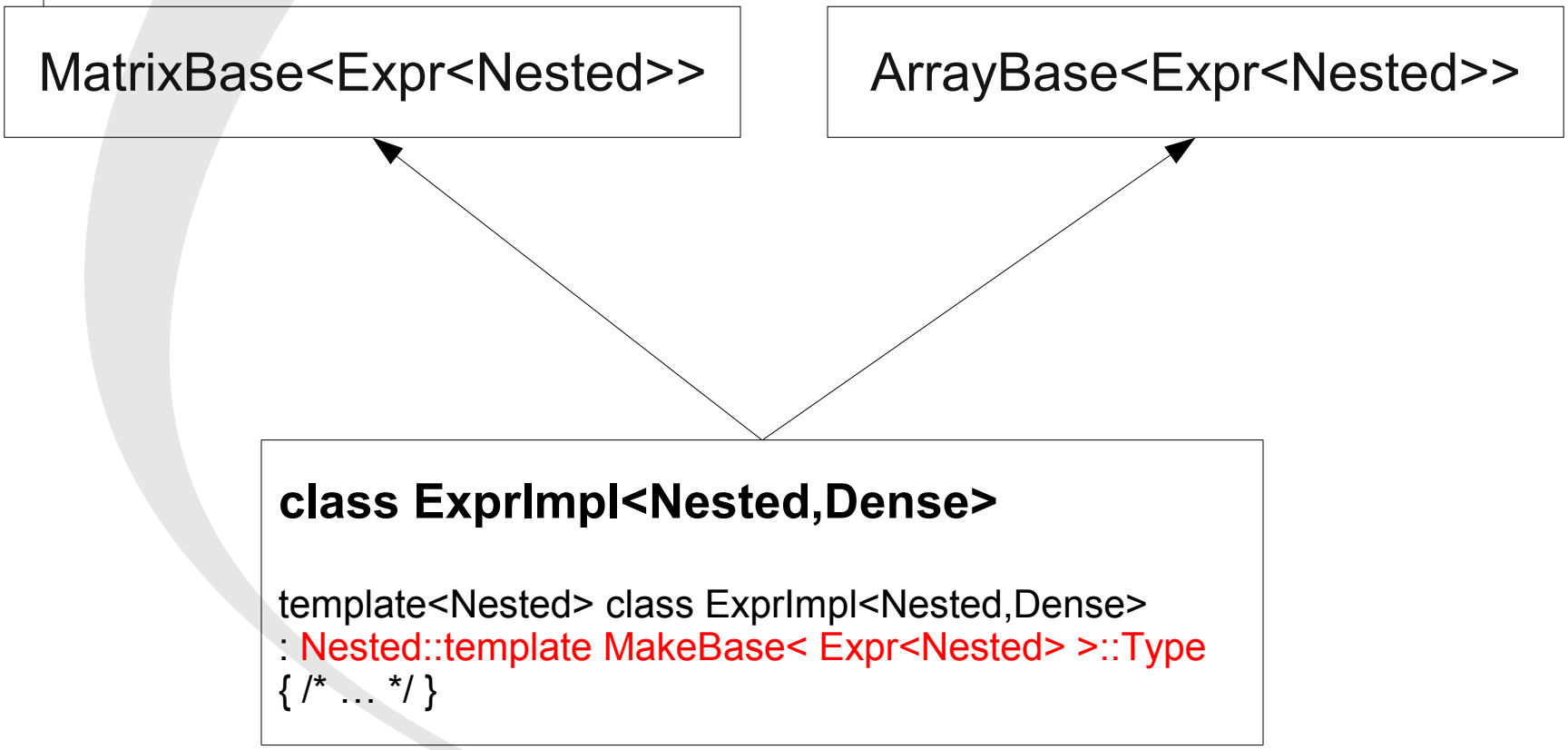

AnyMatrixBase<Derived>
Per matrix kind base classes
Such a base class define the common API across all expressions of a given matrix kind and implements the evaluation mechanism.
The DenseBase<> class allows to factorize most of the code shared by Array and Matrix objects (e..g, Assign).
DenseBase<Derived>
SparseMatrixBase<Derived>
MatrixBase<Derived>
ArrayBase<Derived>
#included
Plugins
Plugin: CommonCwiseUnaryOps
Plugin: ArrayCwiseUnaryOps
Plugin: MatrixCwiseUnaryOps
...
Plugins can be #included by the base matrix classes for which the respective ExpressionImpl<> specialization has been implemented.
For instance, once CwiseBinaryOpImpl has been implemented for your fancy special matrix, you simply have to #include the respective *CwiseBinaryOps plugins in the FancyMatrixBase class to have all the cwise binary operators.
Storage
The goal of this design is to offer to the user a different matrix class for each kind of storage.
The DenseStorageBase<> class allows to factor 90% of the code. Matrix and Array only have to implement their respective ctors.
This is *not* a multiple inheritence. The inerited class depends on the template parameter Base, e.g.:
class DenseStorageBase<Derived,Base> : public Base<Derived> { ... };
DenseStorageBase<Derived,Base>
Matrix<Scalar,Rows,Cols,Options>
Array<Scalar,Rows,Cols,Options>
SparseMatrix<Scalar,Options>
Expression
This is *not* multiple inheritence. The inherited base class is obtained by:
NestedExprType::MakeBaseClass<Derived>
ExpressionImpl<Derived,Dense>
ExpressionImpl<Derived,Sparse>
This is *not* multiple inheritance. The inherited specialization of ExpressionImpl depends on ei_traits<NestedExprType>::StorageMode, which can be either Dense, Sparse, or whatever else. This is a type.
Expression<NestedExprType>
The goal of this design is to make sure to have the exact same syntax to write the expression type of a given expression regardless of the kind of matrix (useful to write and analyse expressions).
The specializations of ExpressionImpl<> allow to 1) select the correct base class, and 2) to reflect the differences in the expression evaluation mechanisms (.coeff() for the Dense path, InnerIterators for the Sparse path, etc.).

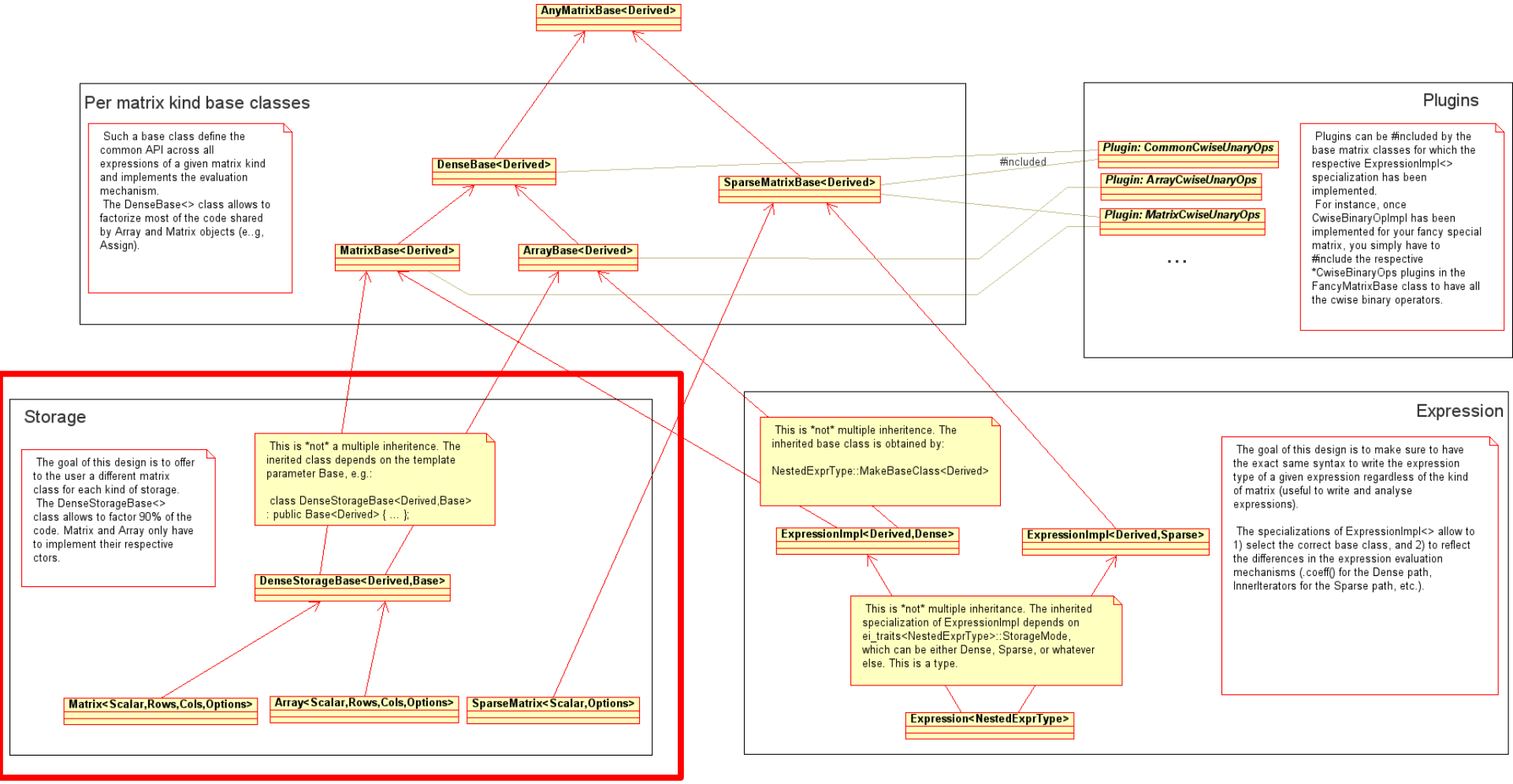
Semantic/implementation decoupling



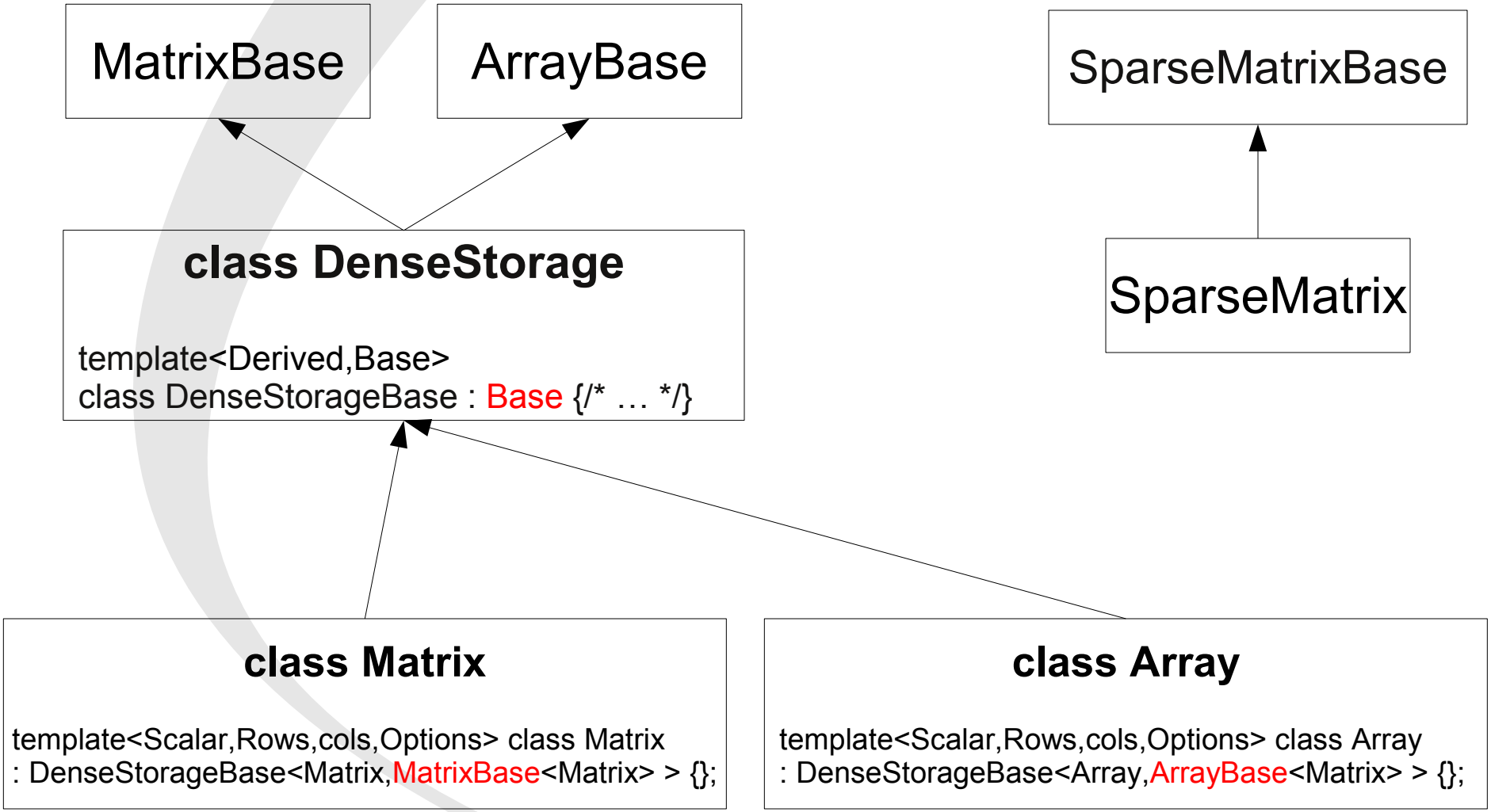
Semantic/implementation decoupling

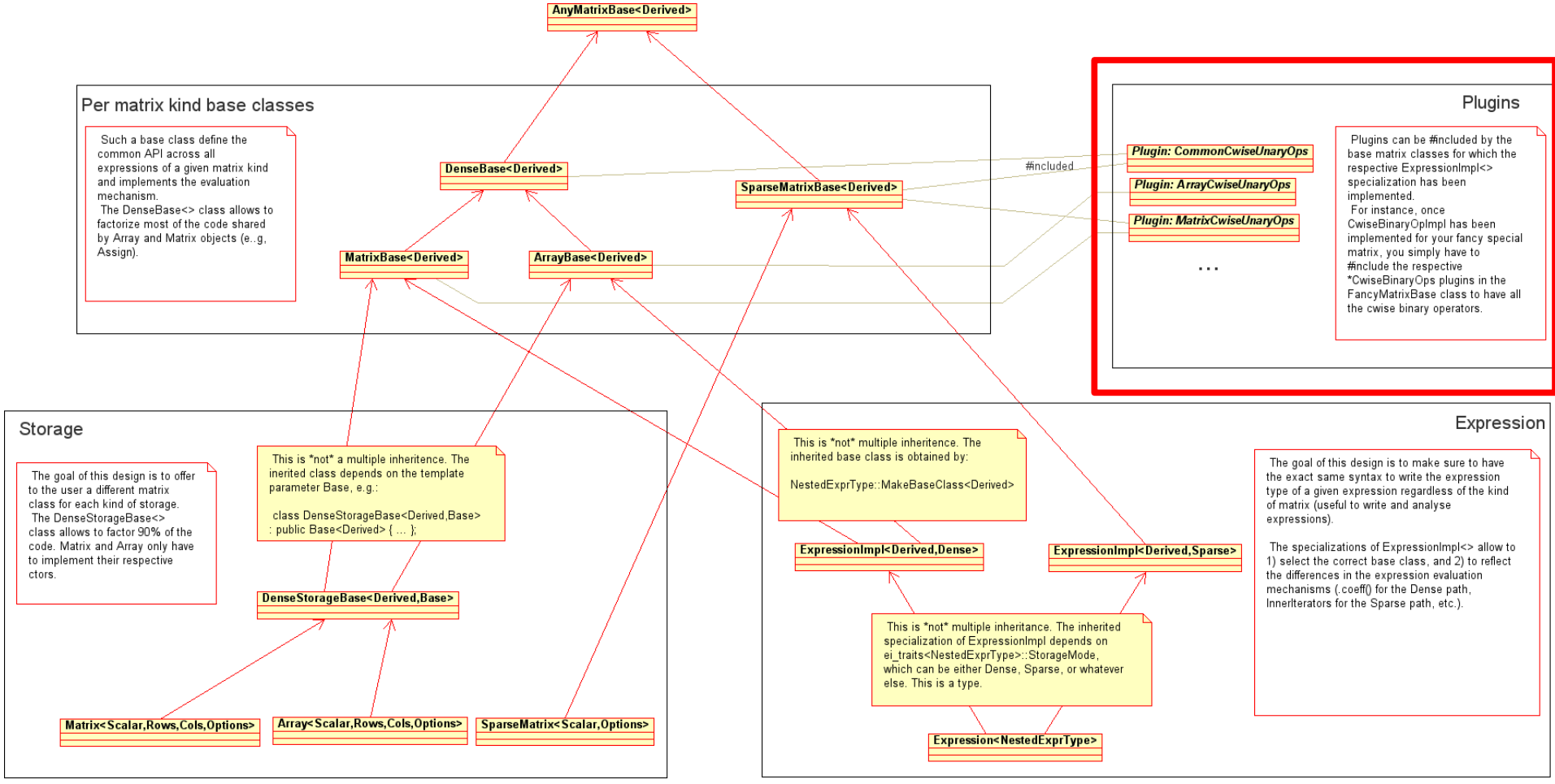
template <typename OtherDerived>
struct MakeBase { typedef MatrixBase<OtherDerived> Type; };




AnyMatrixBase<Derived>
Per matrix kind base classes
Such a base class define the common API across all expressions of a given matrix kind and implements the evaluation mechanism.
The DenseBase<> class allows to factorize most of the code shared by Array and Matrix objects (e..g, Assign).
DenseBase<Derived>
SparseMatrixBase<Derived>
MatrixBase<Derived>
ArrayBase<Derived>
#included
Plugins
Plugin: CommonCwiseUnaryOps
Plugin: ArrayCwiseUnaryOps
Plugin: MatrixCwiseUnaryOps
...
Plugins can be #included by the base matrix classes for which the respective ExpressionImpl<> specialization has been implemented.
For instance, once CwiseBinaryOpImpl has been implemented for your fancy special matrix, you simply have to #include the respective *CwiseBinaryOps plugins in the FancyMatrixBase class to have all the cwise binary operators.
Storage
The goal of this design is to offer to the user a different matrix class for each kind of storage.
The DenseStorageBase<> class allows to factor 90% of the code. Matrix and Array only have to implement their respective ctors.
This is *not* a multiple inheritence. The inerited class depends on the template parameter Base, e.g.:
class DenseStorageBase<Derived,Base> : public Base<Derived> { ... };
DenseStorageBase<Derived,Base>
Matrix<Scalar,Rows,Cols,Options>
Array<Scalar,Rows,Cols,Options>
SparseMatrix<Scalar,Options>
Expression
This is *not* multiple inheritence. The inherited base class is obtained by:
NestedExprType::MakeBaseClass<Derived>
ExpressionImpl<Derived,Dense>
ExpressionImpl<Derived,Sparse>
This is *not* multiple inheritance. The inherited specialization of ExpressionImpl depends on ei_traits<NestedExprType>::StorageMode, which can be either Dense, Sparse, or whatever else. This is a type.
Expression<NestedExprType>
The goal of this design is to make sure to have the exact same syntax to write the expression type of a given expression regardless of the kind of matrix (useful to write and analyse expressions).
The specializations of ExpressionImpl<> allow to 1) select the correct base class, and 2) to reflect the differences in the expression evaluation mechanisms (.coeff() for the Dense path, InnerIterators for the Sparse path, etc.).

Storage classes




AnyMatrixBase<Derived>
Per matrix kind base classes
Such a base class define the common API across all expressions of a given matrix kind and implements the evaluation mechanism.
The DenseBase<> class allows to factorize most of the code shared by Array and Matrix objects (e..g, Assign).
DenseBase<Derived>
SparseMatrixBase<Derived>
MatrixBase<Derived>
ArrayBase<Derived>
#included
Plugins
Plugin: CommonCwiseUnaryOps
Plugin: ArrayCwiseUnaryOps
Plugin: MatrixCwiseUnaryOps
...
Plugins can be #included by the base matrix classes for which the respective ExpressionImpl<> specialization has been implemented.
For instance, once CwiseBinaryOpImpl has been implemented for your fancy special matrix, you simply have to #include the respective *CwiseBinaryOps plugins in the FancyMatrixBase class to have all the cwise binary operators.
Storage
The goal of this design is to offer to the user a different matrix class for each kind of storage.
The DenseStorageBase<> class allows to factor 90% of the code. Matrix and Array only have to implement their respective ctors.
This is *not* a multiple inheritence. The inerited class depends on the template parameter Base, e.g.:
class DenseStorageBase<Derived,Base> : public Base<Derived> { ... };
DenseStorageBase<Derived,Base>
Matrix<Scalar,Rows,Cols,Options>
Array<Scalar,Rows,Cols,Options>
SparseMatrix<Scalar,Options>
Expression
This is *not* multiple inheritence. The inherited base class is obtained by:
NestedExprType::MakeBaseClass<Derived>
ExpressionImpl<Derived,Dense>
ExpressionImpl<Derived,Sparse>
This is *not* multiple inheritance. The inherited specialization of ExpressionImpl depends on ei_traits<NestedExprType>::StorageMode, which can be either Dense, Sparse, or whatever else. This is a type.
Expression<NestedExprType>
The goal of this design is to make sure to have the exact same syntax to write the expression type of a given expression regardless of the kind of matrix (useful to write and analyse expressions).
The specializations of ExpressionImpl<> allow to 1) select the correct base class, and 2) to reflect the differences in the expression evaluation mechanisms (.coeff() for the Dense path, InnerIterators for the Sparse path, etc.).

Objective Zero Code Duplication

- How to factor common API between, e.g., Dense and Sparse objects ?
  - Notion of set of “features”:
    - one feature set per kind of expression (unary, binary, etc.)
    - split feature set wrt matrix/array world
  - create one “plugin” per feature set
    - 1 plugin = 1 header file
    - plugins are #included in the body of the base classes
  - examples:
    - common unary operators (e.g., minCoeff)
    - matrix specific binary operators (matrix product)
    - array specific binary operators (<,>,&,|,etc.)
    - etc.

Objective Zero Code Duplication

plugins/MatrixCwiseUnaryOps.h:

```
/** nice documentation here */
const CwiseUnaryOp<ei_scalar_abs_op<Scalar>,Derived> cwiseAbs() const { return derived(); }
/* … */
```

Core/MatrixBase.h

```
class MatrixBase {
#define EIGEN_CURRENT_STORAGE_BASE_CLASS Eigen::MatrixBase
#   include "../plugins/CommonCwiseUnaryOps.h"
#   include "../plugins/CommonCwiseBinaryOps.h"
#   include "../plugins/MatrixCwiseUnaryOps.h"
#   include "../plugins/MatrixCwiseBinaryOps.h"
#undef EIGEN_CURRENT_STORAGE_BASE_CLASS
};
```

plugins/MatrixCwiseBinaryOps.h:

```
template<typename OtherDerived>
const CwiseBinaryOp<max<Scalar>, Derived, OtherDerived>
cwiseMax(const EIGEN_CURRENT_STORAGE_BASE_CLASS<OtherDerived> &other) const
{
  return CwiseBinaryOp<max<Scalar>, Derived, OtherDerived>(derived(), other.derived());
}
```

Matrix products