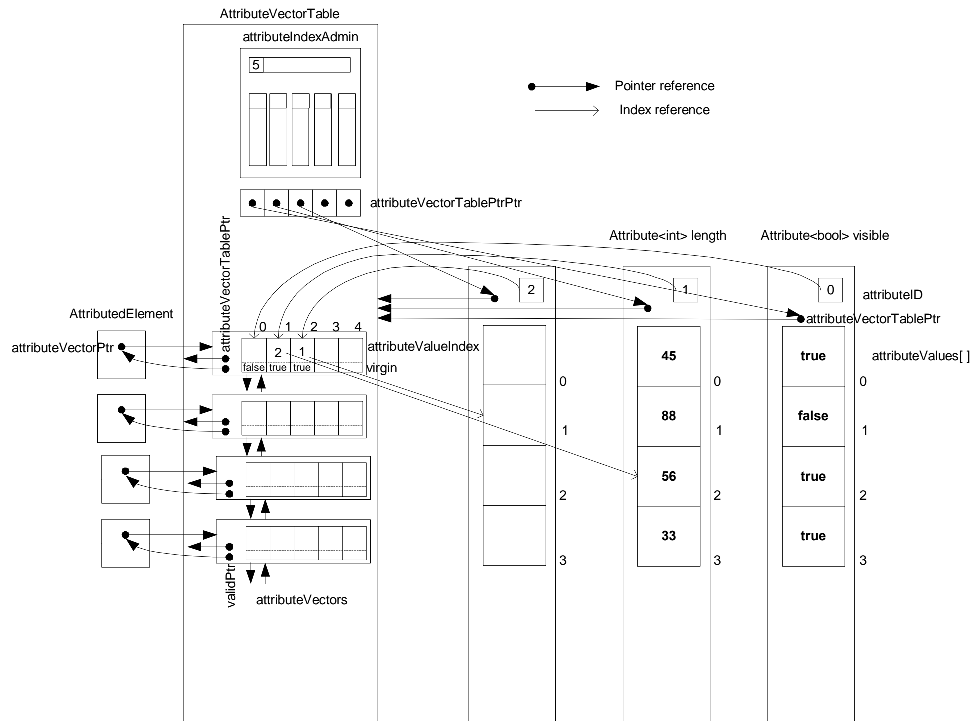
Structure + Software = System



Indizierte Attributverwaltung

Version 1.5

Autor: Josef Hübl

Erstellt am: 01.08.2003

Geändert am: 01.06.2006

Von: Josef Hübl (Triple-S GmbH)

INHALTSVERZEICHNIS

Seite

1. ALLGEMEINES 3
1.1 ZUSAMMENFASSUNG 3
1.2 DAS GRUNDPROBLEM 3
1.3 ENTWICKLUNGSSCHRITTE 4
2. ATTRIBUTE 5
3. INDIZIERTE ATTRIBUTVERWALTUNG 6
3.1 EINFACHE GRUNDVERSION 6
3.2 OPTIMIERUNG SPEICHERPLATZBEDARF 7
3.3 SETZEN VON DEFAULT-WERTEN 8
3.4 FINALE VERSION 8
3.5 WEITERE SPEICHERPLATZOPTIMIERUNG 10
4. KLASSENDEFINITIONEN 11

© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf
Autor: Josef Hübl

1. Allgemeines

1.1 Zusammenfassung

Dieses Dokument befasst sich mit der Implementierung einer indizierten Attributverwaltung. Es werden Sinn und Zweck einer indizierten Attributverwaltung erklärt, sowie deren konkrete Implementierung in C++ aufgezeigt.

1.2 Das Grundproblem

Im Fokus stehen Elemente denen Attributwerte zugeordnet werden sollen. Wird zum Beispiel auf einem Graphen eine Tiefensuche durchgeführt, so erhalten die Knoten für das Attribut „`visited`“ den Wert `true` oder `false` zugewiesen. Natürlich könnte man die Datenstruktur für ein Element so festlegen, dass alle Attributwerte darin abgelegt werden können. Dies setzt aber voraus, dass bei der Festlegung dieser Datenstruktur alle Attribut-Datentypen bekannt sein müssen, die jemals verwendet werden sollen. Bezogen auf das Beispiel, der auf einem Graphen arbeitenden Algorithmen, ist dies schlicht unmöglich, da immer wieder neue Algorithmen entwickelt werden, die meist ihre eigenen Attribute benötigen. Sollen mehrere dieser Algorithmen auf demselben Graphen ausgeführt werden, so müsste der Graph von einer Datenstruktur in eine andere kopiert werden. Dies kann einerseits sehr zeitaufwendig sein, andererseits löst es auch nicht das Problem, dass ein Algorithmus die von einem anderen Algorithmus erstellten Attribute wiederverwenden könnte.

Das Ziel ist es also, die Datenstruktur(en) für die Attribute von denen der zu attributierenden Elemente soweit zu trennen, dass ein Algorithmus zu einen Element die Menge der vorhanden Attribute beliebig um eigene Attribute erweitern kann und der uneingeschränkte Zugriff auf die einem Element zugeordneten Attributwerte möglich ist, obwohl zur Compile-Zeit nicht bekannt ist, welchen Datentyp die Attributwerte haben werden.

Da die Attributwerte losgelöst von den Elementen gespeichert werden sollen, ist klar, das ein Referenzierungsmechanismus zu Einsatz kommen muss. Die normale Referenzierung über Pointer scheidet aus, das der Datentyp des Attributwertes und damit des Pointers zur Compile-Zeit noch nicht bekannt ist. Das Arbeiten mit `(void *)`-Pointern wäre naheliegend, erfordert aber innerhalb eines Algorithmus' bei Zugriff auf die Attributwerte eines Elements eine häufige Pointer-Typenkonvertierung. Um dies zu vermeiden, erfolgt die Referenzierung über Indizes, wobei vorausgesetzt wird, dass die Attributwerte in einem Vector bzw. Array abgelegt sind, da nur so ein indizierter Zugriff darauf möglich ist.

Da beim Löschen von Elementen in den Werte-Arrays der Attribute Lücken bzw. unbenutzte Werte entstehen, die bei unterbleibender Wiederverwendung zur Verschwendung von

© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf

Speicherplatz führen würden, ist ein Hauptaugenmerk darauf gelegt, eine effiziente Index-Verwaltung zu implementieren, die es zulässt frei gewordenen Speicherplatz wiederzuverwenden. Weitere Verkomplizierungen des Problems entstehen dadurch, dass zum Einen den Elenenten Default-Attributwerte zugewiesen werden sollen und zum anderen nicht jedem Element alle Attribute zugewiesen sein müssen.

1.3 Entwicklungsschritte

Da eine indizierte Attributverwaltung in ihrer vollen Ausprägung sehr komplex ist, wird in mehreren Schritte vorgegangen, wobei jeweils eine neue Anforderung hinzukommt und die dadurch notwendigen zusätzlichen Mechanismen dargestellt werden.

Die Entwicklungsstufen sind:

1. Verwenden von Attributwerte-Indizes statt Pointern.
2. Zusätzliche Indexverwaltung zur Speicherplatzwiederverwendung
3. Automatisches setzen von Defaultwerten und dazu notwendige AttributeID-Verwaltung.
4. Speicherplatzreduzierung für nicht vergebene Attribute und dazu notwendige erweiterte Attributwerte-Indizierung

© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf

2. Attribute

Im folgenden wird zwischen einem Attribut und einem Attributwert dahingehend unterschieden, dass der Begriff „Attribut“ eine Eigenschaft aller betroffenen Elemente kennzeichnet, der „Attributwert“ dagegen die konkrete Ausprägung für ein bestimmtes Element. Zum Beispiel kann „Schuhgröße“ das Attribut sein und „42“ ein spezifischer Attributwert.

Da die Attributwerte getrennt von den Elementen gespeichert werden, werden Attribute als Klassen implementiert, die intern ein Array bzw. einen Vektor von Attributwerten verwalten. Um den Attributwert für eine Element setzen bzw. lesen zu können sind im Prinzip zwei Funktionen der Art
`MyAttribute.SetValue( element, value )` und `MyAttribute.GetValue( Element )` notwendig. Damit ist jedoch keine Zuweisung der Art „a = b“ möglich. Aus diesem Gunde werden die Operatoren „=“, „[ ]“, „==“ und „!=“ der Attributklasse überladen. Damit sind Anweisungen in gewohnter Weise möglich. Zum Beispiel:

`MyAttribute[ element ] = value;` oder `value = MyAttribute[ element ];`

Da alle Attributklassen bis auf den Datentyp der Attributwerte identisch sind wird eine Templateklasse zur Verfügung gestellt. Dabei ist `Attribute<Valuetype, Elementtype>` der Typ einer Attributklasse, die Attributwerte vom Typ Valuetype verwaltet, welche wiederum Elementen vom Typ `Elementtype` zugewiesen werden können. Die Templateklasse selbst idt von der Klasse `AttributeBase` abgeleitet in der das von den Attributwerten unabhängige Zusammenspiel mit der Attributvektortabelle implementiert ist.

Für den einfachen internen Zugriff muss die Klasse für die Elemente von der Klasse `AttributedElement` abgeleitet sein.

Beispiel:

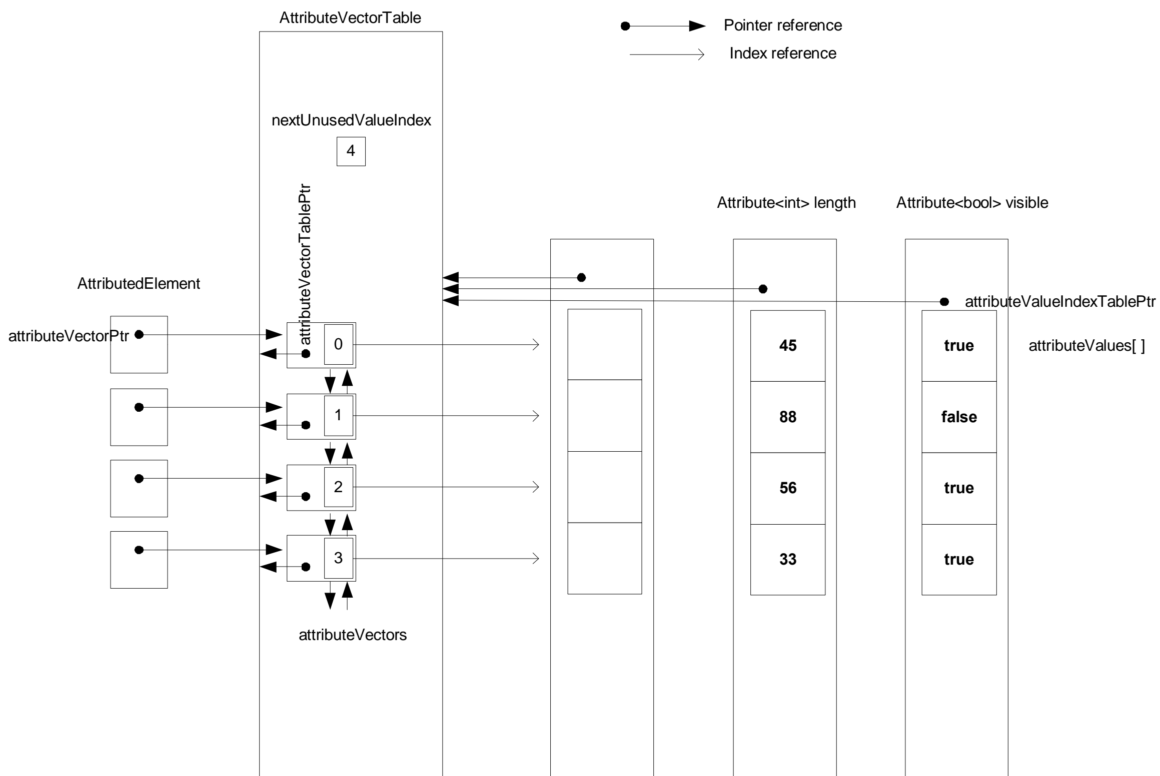
```
Class Node : AttributedElement
{ … }

Attribute< bool, Node > visible;
Node node;

visible[ node ] = true;

Attribute< String, Node > name;

name[ node ] = "Ingolstadt";
```


© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf Autor: Josef Hübl

3. Indizierte Attributverwaltung

3.1 Einfache Grundversion

In der einfachsten Form der indizierten Attributverwaltung verweist ein attributiertes Element nicht direkt auf den Attributwert, sondern auf einen stellvertretenden Attributvektor, der den Attibutwertindex kennt. Der Attibutwertindex gibt die Position im Attributwertearray an, an der der tatsächliche Attributwert steht. (Um mit der finalen Version der Attributverwaltung kompatibel zu sein, wird der Stellvertreter als `AttributeVector` bezeichnet, obwohl er in der einfachen Grundversion nur einen Attibutwertindex verwaltet.)



Alle Attributvektoren werden in einer Tabelle (`AttributeVectorTable`) zusammen mit dem als nächstes zu vergebenden Index `nextUnusedValueIndex` als verkettete Liste gespeichert. Sowohl die Attributvektoren als auch die Attributklassen haben über den Pointer `attributeVectorTablePtr` Zugriff auf diese Tabelle. Sie benötigen diesen Pointe insbesondere um bei ihrer Zerstörung durch den Destruktor unabhängig von den attributierten Elementen bzw. Attributen den `AttributeVectorTable` zu informieren.

© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf

Wird ein neues attributiertes Element erstellt, so wird die Liste der Attributvektoren um ein Element erweitert, im Attributwertindex der Wert von `nextUnusedValueIndex` eingetragen und dem attributierten Element ein Zeiger auf den neuen Attributvektor zugewiesen. Der Wert von `nextUnusedValueIndex` wird um +1 erhöht.

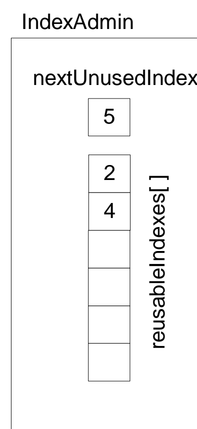
Achtung! Für ein konkretes Attribut zeigt der neue Index zunächst ins Leere. Deshalb muss bei jedem Zugriff auf den Attributwert eines attributierten Elements zuerst geprüft werden, ob die Größe des Attributwertearrays groß genug ist. Falls dies nicht der Fall ist, ist als Erstes der Speicher für das Attributwertearray mit neuer Größe zu reallokieren, wobei beim lesenden Zugriff den neuen Attributwertefeldern ein Default-Wert zugewiesen wird.

Wird ein neues Attribut angelegt, so ist zunächst nicht bekannt welchen Elementen das Attribut zugeordnet sein soll, d.h. der Pointer `attributeVektorTablePtr` hat der Wert `NULL`. In diesem Fall wird beim ersten Zugriff auf dieses Attribut dieser Pointer vom Attributvektor des attributierten Elementes übernommen und Speicherplatz für alle Attributwerte allokiert und mit Default-Werten belegt. Dabei ist die Anzahl der zu allokierenden Felder über den Tabellen-Pointer durch den Wert von `nextUnusedValueIndex` bekannt.

Beim Löschen von attributierten Elementen kann zwar sowohl das Element als auch der zugeordnete Attributvektor zerstört werden. Jedoch verbleiben im Attributwertearray ungenutzte Lücken.

3.2 Optimierung Speicherplatzbedarf

Um den Speicherplatz für die Attributwerte gelöschter Elemente nicht ungenutzt zu verschwenden werden im nächsten Optimierungsschritt die Indizes dieser Attributwerte zu eine zusätzliche Indexverwaltung „recycelt". Dazu wird beim Löschen eines Elements sein Attributwertindex in eine Liste `reusableIndezes[ ]` eingetragen. Wird Speicherplatz für den Attributwert eines neuen Elements benötigt, so werden zuerst die Indizes aus der Liste bzw. der Speicherplatz für die zugehörigen Attributwerte wiederverwendet, bevor neuer Speicher allokiert wird.



© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf Autor: Josef Hübl

Soweit der erste Zugriff auf den Attributwert eines attributierten Elements immer schreibend ist, wird es mit den recycelten Wertefeldern keine Probleme geben.

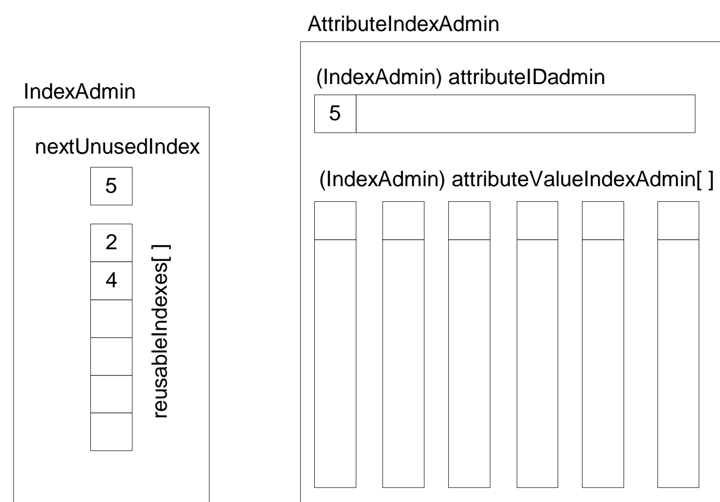
3.3 Setzen von Default-Werten

Wird im Falle einer optimierten Speicherplatzverwaltung mit recycelten Attributwerteindizes gestattet, dass der erste Zugriff auf einen Attributwert auch lesend sein kann, so muss jedem Attributwerteindex ein zusätzliches Flag `virgin` zugeordnet werden, welches im Falle, dass es gesetzt ist besagt, dass beim ersten lesenden Zugriff auf den Attributwert zuvor der Defaultwert gesetzt werden muss. Bei Verwendung recycelter Attributwerteindizes ist dieses neue Flag unbedingt notwendig, da aus der Größe des Attributwerteindexes nicht mehr geschlossen werden kann, ob das Wertefeld zum ersten Mal benutzt wird.

Da der Default-Wert für ein Attribut nur der Attributklasse bekannt ist und von dieser gesetzt werden kann, muss das Flag `virgin` für jede Klasse bzw. jeden Index der in das Wertearray einer solchen Klasse verweist, getrennt verwaltet werden. Dazu werden diese Flags in einem Vektor angeordnet, wobei in der Attributklasse eine Attribute ID auf die Position in diesem Vektor verweist.

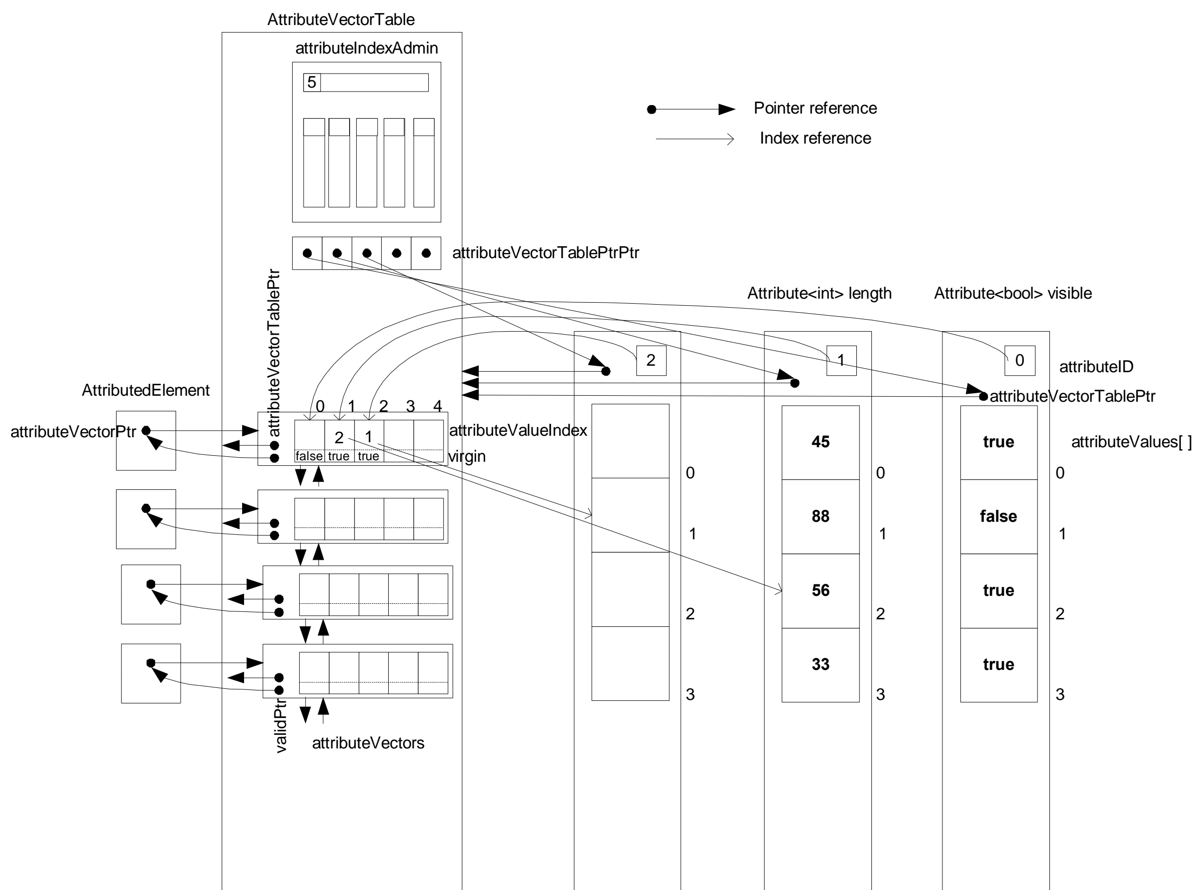
3.4 Finale Version

Werden Attribute bzw. die Instanzen von Attributklassen gelöscht, entstehen in den `virgin`-Vektoren Speicherplatzlücken, die ebenso wie Speicherplatz für Attributwerte recycelt werden können, in dem für die Attribut ID’s eine analoge Indexverwaltung eingesetzt wird, die diese recycelt.



© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf
Autor: Josef Hübl

Geht man davon aus, dass grundsätzlich nicht jedem attributierten Element die gleiche Anzahl von Attributen zugewiesen wird, kann der Speicherplatzbedarf weiter optimiert werden in dem die Attributwertindizes für jedes Attribut getrennt verwaltet werden. Es wird dann für das Wertearray eines Attributs nur soviel Speicher allokiert wie es der größte Index für genau dieses Attribut verlangt. D.h. im Attributvektor eines attributierten Element ist für jedes Attribut ein eigener Attributwerteindex zu verwalten. Dies geschieht zusammen mit dem Flag `virgin` in einem gemeinsamen `attributIndexVector`. Jedes Attribut erhält zur Identifikation einen eindeutigen Index als `AttributeID` zugewiesen. Die `AttributeID` kennzeichnet auch die Postion in jedem `attributIndexVextor` an der zu genau diesem Attribut der Attributwertindex mit dem Flag `virgin` zu finden ist.



Die Vergabe der `AttributeID` und deren Speicherung beim Attribut erfolgt beim ersten lesenden oder schreibenden Zugriff auf den Attributwert eines attributierten Elementes. Das Speichern der `AttributeID` beim Attribut ist unter anderem notwendig, damit bei der Zerstörung des Attributs durch dessen Destruktor die Attributindexverwaltung (`attributeIndexAdmin`) über die Freigabe der `AttributeID` informiert werden kann. Als

© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf

Folge davon stehen die `AttributeID` und alle sich darauf beziehenden AttributValueIndexes zur Wiederverwendung zur Verfügung.

Da beim Zerstören des AttributeVektorTables und den darauf verweisendeen Attributen die Reihenfolge beliebig sein kann, speichert der `AttributeVektorTable` in Vektor `attributeVectorTablePtrPtr` einen Zeiger auf die Stelle das Attributs, an der dieses den Zeiger auf den Table abgelegt. Wird der Table vor den Attribut zerstört, kann so die Referenz bzw. der Inhalt von `attributeVectorTablePtrPtr` auf `NULL` gesetzt werden.

Eine analoge Funktion wie sie `attributeVectorTablePtrPtr` für das Zusammenspiel von `AttributeVektorTable` und den Attributen übernimmt, wird über den Zeiger `validPtr` für das Zusammenspiel von jedem `AttributVector` zu seinem `AttributedElement` realisiert.

Für Attribute der Zuweisungsoperator „=“ überladen. Wird ein Attribut auf ein anderes zugewiesen, so wird die Absicht unterstellt, dass die Attributwerte kopiert werden sollen. Entsprechend werdenalle Indizes der alten `AttributeID` gelöscht eine neue `AttributeID` erzeugt und die Indizes neu vergeben.

3.5 Weitere Speicherplatzoptimierung

Im Prinzip ließe sich der Speicherplatzbedarf weiter reduzieren in dem bei freiwerden eines maximalen Indexes dieser nicht in die Liste der wiederverwendbaren Indizes übernommen würde, sonder der entsprechende Speicherplatz tatsächlich freigegeben werden würde. Da dies jedoch im allgemeinen Fall weiter zu Lasten der Perfomance geht, wurde dies nicht näher untersucht.

© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf

4. Klassendefinitionen

```
// Josef Huebl, Triple-S GmbH, www.sss.de 5.8.2003

#ifndef INDEXATTR_H
#define INDEXATTR_H

#ifdef WIN32
#pragma warning( disable : 4786 )  // Disable warning message C4786
#endif

#include <vector>
#include <list>

typedef size_t Index;
typedef Index AttributeID;
typedef Index AttributeValueIndex;

const Index                                   NONE_VALID_INDEX = 0xFFFF;
const AttributeID                     NONE_VALID_ATTRIBUTE_ID = NONE_VALID_INDEX;
const AttributeValueIndex     NONE_VALID_ATTRIBUTE_VALUE_INDEX = NONE_VALID_INDEX;

// ****************************** IndexAdmin ************************************

class IndexAdmin
{
public:
        IndexAdmin() : m_nextUnusedIndex(0), m_reusableIndexes(0) {};
        ~IndexAdmin() { m_reusableIndexes.clear(); };
        Index NewIndex( void );
        Index MaxIndex( void );
        void DelIndex(Index index);
        void ReInitialize(Index nextIndexInUse );
        void ClearIndexes( void );

private:
        Index m_nextUnusedIndex;
        std::vector<Index> m_reusableIndexes;                      // indexes for recycling
};

// ****************************** AttributeIndexAdmin ************************************

class AttributeIndexAdmin
{
public:
        AttributeIndexAdmin() : m_attributeValueIndexAdmin(0) {};
        ~AttributeIndexAdmin() { m_attributeValueIndexAdmin.clear(); };
        AttributeID NewAttributeID( void );
        AttributeID CopyAttributeID( AttributeID originalID );
        AttributeID MaxAttributeID( void );
        void DelAttributeID( AttributeID id );
        AttributeValueIndex NewAttributeValueIndex( AttributeID id );
        void DelAttributeValueIndex( AttributeID id, AttributeValueIndex valueIndex );
        void ReInitializeAttributeValueIndexes( AttributeID id, AttributeValueIndex
nextValueIndexInUse );
        void ClearAttributeValueIndexes( AttributeID id );
        void Clear( void );
private:
        IndexAdmin m_attributeIDadmin;
        std::vector<IndexAdmin> m_attributeValueIndexAdmin;
};


struct IndexedAttribute
{
```


© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf   Autor: Josef Hübl

```
        AttributeValueIndex attributeValueIndex;            // zur Speicherplatz-Optimierung
        bool virgin;   // == true, if never gots a real attribute value
};

class AttributeVectorTable;

// ****************************** AttributeVector ************************************

class AttributeVector
{
public:
        AttributeVector( bool *validPointerPtr ) : m_attributeVectorTablePtr( NULL ),
m_validPointerPtr( validPointerPtr )  {};
        IndexedAttribute &GetIndexedAttribute( AttributeID id ) { return m_indexedAttributes[
id ]; };
        void NewIndexedAttribute( AttributeID id );
        void CopyIndexedAttribute( AttributeID idOriginal, AttributeID idCopy );
        void DelIndexedAttribute( AttributeID id );
        void ResetIndexedAttribute( AttributeID id ) {m_indexedAttributes[ id ].virgin =
true;};
        void UseIndexedAttribute( AttributeID id ) {m_indexedAttributes[ id ].virgin = false;};
        void AddAllAttributes( AttributeVectorTable *table );
        void DelAllAttributes( void );
        AttributeVectorTable *GetAttributeVectorTablePtr( void ) { return
m_attributeVectorTablePtr; };

private:
        std::vector<IndexedAttribute> m_indexedAttributes;
        AttributeVectorTable *m_attributeVectorTablePtr;
        bool *m_validPointerPtr; // for destructor
};

// ****************************** AttributeVectorTable ************************************

class AttributeVectorTable
{
public:
        virtual ~AttributeVectorTable();
        std::list<AttributeVector>::iterator NewAttributeVector( bool *validPointerPtr );
        void DelAttributeVector( std::list<AttributeVector>::iterator );
        void DelAllAttributeVectors( void );
        AttributeID NewAttribute( AttributeVectorTable **attributeVectorTablePtrPtr );
        AttributeID CopyAttribute( AttributeID originalID, AttributeVectorTable
**copyAttributeVectorTablePtrPtr );
        void DelAttribute( AttributeID id );
        void ReinitialiseAttribute( AttributeID id );
        AttributeIndexAdmin &GetAttributeIndexAdmin(void) { return m_attributeIndexAdmin; };

        Index GetNumberOfElements( void ) { return m_attributeVectors.size(); };
        bool SetToFirstAttributeVector( void ) { m_it = m_attributeVectors.begin(); if ( m_it
== m_attributeVectors.end()) return false; else return true; };
        bool SetToNextAttributeVector( void ) { if ( ++m_it == m_attributeVectors.end()) return
false; else return true; };
        IndexedAttribute GetCurrentIndexedAttribute( AttributeID id ) { return (m_it-
>GetIndexedAttribute( id )); };
        void TouchCurrentIndexedAttribute( AttributeID id ) { m_it->GetIndexedAttribute( id
).virgin = false; };

private:
        std::vector<AttributeVectorTable **> m_attributeVectorTablePtrPtr; // for independent
destruction
        AttributeIndexAdmin m_attributeIndexAdmin;
        std::list<AttributeVector> m_attributeVectors;
        std::list<AttributeVector>::iterator m_it;
};


// ****************************** AttributedElement ************************************
// Elements that use indexed attributs must be derived from AttributedElement
```


© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf
Autor: Josef Hübl

```
class AttributedElement
{
public:
        AttributedElement( ) : m_validPointer(false) {};
        void NewAttributeVector( AttributeVectorTable *ptr );
        void DelAttributeVector( void );
        IndexedAttribute &GetIndexedAttribute( AttributeID id ) { return m_attributeVectorPtr-
>GetIndexedAttribute( id ); };
        std::list<AttributeVector>::iterator GetAttributeVectorPtr( void ) { return
m_attributeVectorPtr; };
        void SetAttributeVectorPtr( std::list<AttributeVector>::iterator ptr ) { m_validPointer
= true; m_attributeVectorPtr = ptr; };

private:
        bool m_validPointer;
        std::list<AttributeVector>::iterator m_attributeVectorPtr;
};

// ****************************** AttributeBase ***********************************
// Attibute classes must be derived from AttributeBase

class AttributeBase
{
protected:
        AttributeBase() : m_attributID( NONE_VALID_ATTRIBUTE_ID ), m_attributeVectorTablePtr(
NULL ) {};
        virtual ~AttributeBase() { if (m_attributeVectorTablePtr != NULL)
m_attributeVectorTablePtr->DelAttribute( m_attributID ); };
        AttributeValueIndex CheckAttributeValue( std::list<AttributeVector>::iterator vectorPtr
);
        IndexedAttribute GetIndexedAttribute( std::list<AttributeVector>::iterator vectorPtr );
        virtual void ResetToDefaultValue( AttributeValueIndex index ) = 0;  // pure virtual;
must be implemented by the attribute class

public:
        void ReinitialiseValues( void ){ if (m_attributeVectorTablePtr != NULL)
m_attributeVectorTablePtr->ReinitialiseAttribute( m_attributID ); };

protected:
        AttributeID m_attributID;
        AttributeVectorTable *m_attributeVectorTablePtr; // for destructor
};

/// ********************** Template class Attribute *********************************

template< typename ElementType, typename AttributeType >
class Attribute : public AttributeBase
{
public:
        Attribute(): m_attributeDefaultValue( AttributeType() ){};
        Attribute( AttributeType defaultValue ) : m_attributeDefaultValue(defaultValue) {};
        Attribute( const Attribute &a ) { CopyValues( a ); };
        Attribute& operator=(const Attribute &a) { CopyValues( a ); return *this; };
        friend bool operator==(const Attribute &a1, const Attribute &a2) {return a1.Equal( a2
); };
        friend bool operator!=(const Attribute &a1, const Attribute &a2) { return !(a1 == a2);
};
        void SetDefaultValue( ElementType value );
        AttributeType& operator [](ElementType element);

private:
        AttributeType GetValue( const IndexedAttribute &ia) const { if (ia.virgin) return
m_attributeDefaultValue; else return m_attributeValues[ ia.attributeValueIndex ]; };
        void ResetToDefaultValue( AttributeValueIndex index );
        void CopyValues(const Attribute &attribute); // deletes old attribute from table,
creates a new attribute id and copies all values
        bool Equal( const Attribute &a) const;
        AttributeType          m_attributeDefaultValue;
        std::vector<AttributeType>    m_attributeValues;
};
```


© Triple-S GmbH, Zur Drexlerschleife 1, 93152 Nittendorf Autor: Josef Hübl