# Relationen-Algebra und Persistenz – Teil I

## Implementierungskonzepte für und Anforderungen an Attributdatentypen

Fabio Valdés

FernUniversität Hagen

05. Oktober 2012

# Inhalt

# Inhalt

# Motivation

- bislang nur Verwaltung von einfachen Daten fester Größe möglich, z.B.

# Motivation

- bislang nur Verwaltung von einfachen Daten fester Größe möglich, z.B.
  - integer
  - bool
  - Punkte
  - Rechtecke

# Motivation

- bislang nur Verwaltung von einfachen Daten fester Größe möglich, z.B.
  - integer
  - bool
  - Punkte
  - Rechtecke
- veränderliche Größen oft notwendig, z.B. für

# Motivation

- bislang nur Verwaltung von einfachen Daten fester Größe möglich, z.B.
  - integer
  - bool
  - Punkte
  - Rechtecke
- veränderliche Größen oft notwendig, z.B. für
  - Polygone
  - Videos
  - Straßennetze
  - Indexstrukturen

# Eigenschaften

- FLOB = Faked Large Object

# Eigenschaften

- FLOB = Faked Large Object
- Datenstruktur zur Speicherung beliebiger Datenmengen

# Eigenschaften

- FLOB = Faked Large Object
- Datenstruktur zur Speicherung beliebiger Datenmengen
- eingebauter Persistenzmechanismus

# Eigenschaften

- FLOB = Faked Large Object
- Datenstruktur zur Speicherung beliebiger Datenmengen
- eingebauter Persistenzmechanismus
- Verwaltung unstrukturierter Speicherblöcke (vgl. `malloc`)

# Eigenschaften

- FLOB = Faked Large Object
- Datenstruktur zur Speicherung beliebiger Datenmengen
- eingebauter Persistenzmechanismus
- Verwaltung unstrukturierter Speicherblöcke (vgl. `malloc`)
- Zugriff durch spezielle Funktionen und offset

# Implementierung

## Klasse Flob

```
class Flob {









}
```

# Implementierung

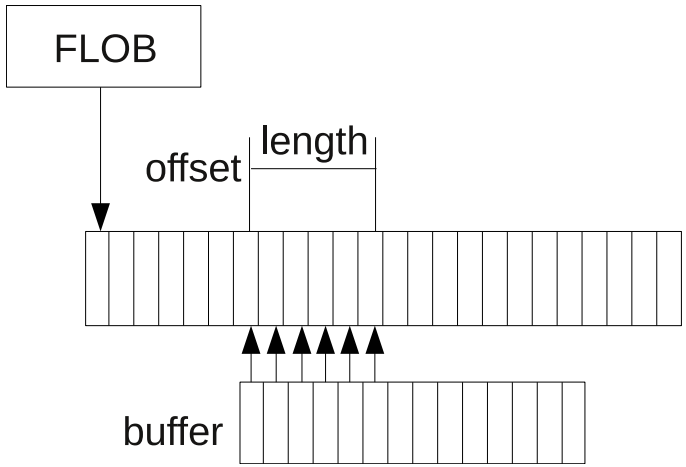
## Klasse Flob

```
class Flob {
  Flob(const SmiSize size);
  SmiSize getSize() const;









}
```

# Implementierung

## Klasse Flob

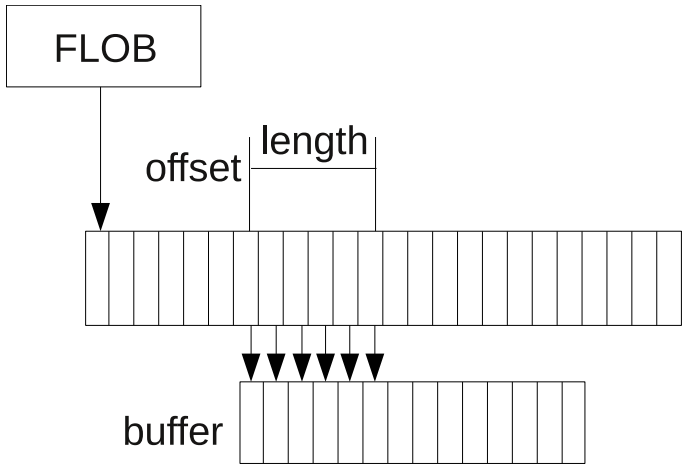
```
class Flob {
  Flob(const SmiSize size);
  SmiSize getSize() const;
  bool write(const char* buffer,
             const SmiSize length,
             const SmiSize offset);









}
```



# Implementierung

## Klasse Flob

```
class Flob {
  Flob(const SmiSize size);
  SmiSize getSize() const;
  bool write(const char* buffer,
             const SmiSize length,
             const SmiSize offset);
  bool read(char* buffer,
            const SmiSize length,
            const SmiSize offset);



}
```



# Implementierung

## Klasse Flob

```
class Flob {
  Flob(const SmiSize size);
  SmiSize getSize() const;
  bool write(const char* buffer,
             const SmiSize length,
             const SmiSize offset);
  bool read(char* buffer,
            const SmiSize length,
            const SmiSize offset);
  bool resize(const SmiSize newSize);
  bool clean();
  bool destroy();
  [...]
}
```

# Achtung:

- Einschränkung: Objekt im Hauptspeicher darstellbar (für einige Operationen)

# Achtung:

- Einschränkung: Objekt im Hauptspeicher darstellbar (für einige Operationen)
- FLOBs werden zusammenhängend auf die Festplatte geschrieben

# Achtung:

- Einschränkung: Objekt im Hauptspeicher darstellbar (für einige Operationen)
- FLOBs werden zusammenhängend auf die Festplatte geschrieben
- FLOBs werden an irgendeine Stelle des Speichers eingelesen

# Achtung:

- Einschränkung: Objekt im Hauptspeicher darstellbar (für einige Operationen)
- FLOBs werden zusammenhängend auf die Festplatte geschrieben
- FLOBs werden an irgendeine Stelle des Speichers eingelesen
- es dürfen keine Zeiger sondern nur offsets verwendet werden

# Inhalt

# Motivation

- dynamisches Array

# Motivation

- dynamisches Array
- verwaltet beliebig viele Einträge fester Größe

# Motivation

- dynamisches Array
- verwaltet beliebig viele Einträge fester Größe
- geeignet zur persistenten Speicherung strukturierter Daten

# DbArrays: Eigenschaften

- Bedingungen an Array-Elemente:

# DbArrays: Eigenschaften

- Bedingungen an Array-Elemente:
  - keine Zeiger

# DbArrays: Eigenschaften

- Bedingungen an Array-Elemente:
  - keine Zeiger
  - keine FLOBs (und damit auch keine DbArrays)

# DbArrays: Eigenschaften

- Bedingungen an Array-Elemente:
  - keine Zeiger
  - keine FLOBs (und damit auch keine DbArrays)
- Template-Klasse `DbArray` abgeleitet von `Flob`

# DbArrays: Eigenschaften

- Bedingungen an Array-Elemente:
  - keine Zeiger
  - keine FLOBs (und damit auch keine DbArrays)
- Template-Klasse `DbArray` abgeleitet von `Flob`
- daher muss die Ein- und Auslagerung von Datenteilen – wie beim FLOB – nicht selbst programmiert werden

# DbArrays: Eigenschaften

- Bedingungen an Array-Elemente:
  - keine Zeiger
  - keine FLOBs (und damit auch keine DbArrays)
- Template-Klasse `DbArray` abgeleitet von `Flob`
- daher muss die Ein- und Auslagerung von Datenteilen – wie beim FLOB – nicht selbst programmiert werden
- binäre Suche möglich bei sortierter Speicherung

# Implementierung

## Klasse DbArray

```
template<class DbArrayElement> class DbArray : public Flob {




}
```

# Implementierung

## Klasse DbArray

```
template<class DbArrayElement> class DbArray : public Flob {
  DbArray(const int n);




}
```

# Implementierung

## Klasse DbArray

```
template<class DbArrayElement> class DbArray : public Flob {
  DbArray(const int n);
  int Size() const;
  bool resize(const int newSize);









}
```

# Implementierung

## Klasse DbArray

```
template<class DbArrayElement> class DbArray : public Flob {
  DbArray(const int n);
  int Size() const;
  bool resize(const int newSize);
  bool Get(const int index, DbArrayElement& elem) const;









}
```

# Implementierung

## Klasse DbArray

```
template<class DbArrayElement> class DbArray : public Flob {
  DbArray(const int n);
  int Size() const;
  bool resize(const int newSize);
  bool Get(const int index, DbArrayElement& elem) const;
  bool Append(const DbArrayElement& elem);









}
```

# Implementierung

## Klasse DbArray

```
template<class DbArrayElement> class DbArray : public Flob {
  DbArray(const int n);
  int Size() const;
  bool resize(const int newSize);
  bool Get(const int index, DbArrayElement& elem) const;
  bool Append(const DbArrayElement& elem);
  bool Put(const int index, const DbArrayElement& elem);









}
```

# Implementierung

## Klasse DbArray

```
template<class DbArrayElement> class DbArray : public Flob {
  DbArray(const int n);
  int Size() const;
  bool resize(const int newSize);
  bool Get(const int index, DbArrayElement& elem) const;
  bool Append(const DbArrayElement& elem);
  bool Put(const int index, const DbArrayElement& elem);
  bool Sort(int (*cmp)(const void *a, const void *b));




}
```

# Implementierung

## Klasse DbArray

```
template<class DbArrayElement> class DbArray : public Flob {
  DbArray(const int n);
  int Size() const;
  bool resize(const int newSize);
  bool Get(const int index, DbArrayElement& elem) const;
  bool Append(const DbArrayElement& elem);
  bool Put(const int index, const DbArrayElement& elem);
  bool Sort(int (*cmp)(const void *a, const void *b));
  bool Find(const void *key,
            int (*cmp)(const void *a, const void *b),
            int& result) const;



}
```

# Implementierung

## Klasse DbArray

```
template<class DbArrayElement> class DbArray : public Flob {
  DbArray(const int n);
  int Size() const;
  bool resize(const int newSize);
  bool Get(const int index, DbArrayElement& elem) const;
  bool Append(const DbArrayElement& elem);
  bool Put(const int index, const DbArrayElement& elem);
  bool Sort(int (*cmp)(const void *a, const void *b));
  bool Find(const void *key,
            int (*cmp)(const void *a, const void *b),
            int& result) const;
  bool TrimToSize();
  bool clean();
  bool Destroy();
}
```

# Inhalt

# Motivation

- Bisher möglich:

# Motivation

- Bisher möglich:
  - Verwaltung einfacher Datentypen

# Motivation

- Bisher möglich:
  - Verwaltung einfacher Datentypen
  - Erzeugung von Objekten

# Motivation

- Bisher möglich:
  - Verwaltung einfacher Datentypen
  - Erzeugung von Objekten
  - Anwendung von Operatoren

# Motivation

- Bisher möglich:
  - Verwaltung einfacher Datentypen
  - Erzeugung von Objekten
  - Anwendung von Operatoren
  - Speicher- und Löschvorgänge auf Datenbank

# Motivation

- Bisher möglich:
  - Verwaltung einfacher Datentypen
  - Erzeugung von Objekten
  - Anwendung von Operatoren
  - Speicher- und Löschvorgänge auf Datenbank
  - Umwandlung von/in Nested List

# Motivation

- Bisher möglich:
  - Verwaltung einfacher Datentypen
  - Erzeugung von Objekten
  - Anwendung von Operatoren
  - Speicher- und Löschvorgänge auf Datenbank
  - Umwandlung von/in Nested List
- Gewünscht:

# Motivation

- Bisher möglich:
  - Verwaltung einfacher Datentypen
  - Erzeugung von Objekten
  - Anwendung von Operatoren
  - Speicher- und Löschvorgänge auf Datenbank
  - Umwandlung von/in Nested List
- Gewünscht:
  - Verwendung als Attribut in Relationen

# Anforderungen

- Mindestfunktionalität

# Anforderungen

- Mindestfunktionalität
  - Implementierung notwendig:
    - `Compare`
    - `SizeOf`
    - `NumOfFLOBs`
    - `GetFLOB`

# Anforderungen

- Mindestfunktionalität
  - Implementierung notwendig:
    - `Compare`
    - `SizeOf`
    - `NumOfFLOBs`
    - `GetFLOB`
  - Implementierung möglich:
    - `SetDefined`
    - `Open`
    - `Save`
    - `Print`

# Anforderungen

- Mindestfunktionalität
  - Implementierung notwendig:
    - `Compare`
    - `SizeOf`
    - `NumOfFLOBs`
    - `GetFLOB`
  - Implementierung möglich:
    - `SetDefined`
    - `Open`
    - `Save`
    - `Print`
  - keine Implementierung
    - `IsDefined`

# Anforderungen

- Mindestfunktionalität
  - Implementierung notwendig:
    - `Compare`
    - `SizeOf`
    - `NumOfFLOBs`
    - `GetFLOB`
  - Implementierung möglich:
    - `SetDefined`
    - `Open`
    - `Save`
    - `Print`
  - keine Implementierung
    - `IsDefined`
- keine Verwendung von dynamischen Objekten / Zeigern als Member

# Implementierung I

## Klasse Polygon

```
class Polygon : public Attribute {
 public:
  Polygon() {}
  Polygon(const int n,
          const int *X = 0,
          const int *Y = 0);
  ~Polygon();

  int NumOfFLOBs() const;
  Flob *GetFLOB(const int i);
  int Compare(const Attribute* rhs) const;
    [...]

 private:
  DbArray<Vertex> vertices;
  PolygonState state;
};
```

# Implementierung I

## Klasse Polygon

```
class Polygon : public Attribute {
 public:
  Polygon() {}
  Polygon(const int n,
          const int *X = 0,
          const int *Y = 0);
  ~Polygon();

  int NumOfFLOBs() const;
  Flob *GetFLOB(const int i);
  int Compare(const Attribute* rhs) const;
    [...]

 private:
  DbArray<Vertex> vertices;
  PolygonState state;
};
```

- Polygon abgeleitet von Attribute
- keine Klasse darf mehrfach von derselben Klasse (hier `Attribute`) erben; sonst Probleme bei Typkonvertierung (`cast`)

# Implementierung I

## Klasse Polygon

```
class Polygon : public Attribute {
 public:
  Polygon() {}
  Polygon(const int n,
          const int *X = 0,
          const int *Y = 0);
  ~Polygon();

  int NumOfFLOBs() const;
  Flob *GetFLOB(const int i);
  int Compare(const Attribute* rhs) const;
    [...]

 private:
  DbArray<Vertex> vertices;
  PolygonState state;
};
```

- Standard-Konstruktor wird vom Persistenzmechanismus verwendet
- kann wegen uninitialisierter Werte zu Fehlern führen
- darf daher nicht verwendet werden
- Verwendung anderer Konstruktoren erforderlich

# Implementierung I

## Klasse Polygon

```
class Polygon : public Attribute {
 public:
  Polygon() {}
  Polygon(const int n,
          const int *X = 0,
          const int *Y = 0);
  ~Polygon();

  int NumOfFLOBs() const;
  Flob *GetFLOB(const int i);
  int Compare(const Attribute* rhs) const;
    [...]

 private:
  DbArray<Vertex> vertices;
  PolygonState state;
};
```

- einziges FLOB-Object der Klasse

# Implementierung I

## Klasse Polygon

```
class Polygon : public Attribute {
 public:
  Polygon() {}
  Polygon(const int n,
          const int *X = 0,
          const int *Y = 0);
  ~Polygon();

  int NumOfFLOBs() const;
  Flob *GetFLOB(const int i);
  int Compare(const Attribute* rhs) const;
    [...]

 private:
  DbArray<Vertex> vertices;
  PolygonState state;
};
```

- gibt die Anzahl der FLOBs zurück

### Funktion NumOfFLOBs

```
int Polygon::
    NumOfFLOBs() const {
  return 1;
}
```

# Implementierung I

## Klasse Polygon

```
class Polygon : public Attribute {
 public:
  Polygon() {}
  Polygon(const int n,
          const int *X = 0,
          const int *Y = 0);
  ~Polygon();

  int NumOfFLOBs() const;
  Flob *GetFLOB(const int i);
  int Compare(const Attribute* rhs) const;
    [...]

 private:
  DbArray<Vertex> vertices;
  PolygonState state;
};
```

- gibt den `i`-ten FLOB zurück

### Funktion GetFLOB

```
Flob *Polygon::
    GetFLOB(const int i){
  assert(i < NumOfFLOBs()
      && i >= 0);
  return &vertices;
}
```

# Implementierung I

## Klasse Polygon

```
class Polygon : public Attribute {
 public:
  Polygon() {}
  Polygon(const int n,
          const int *X = 0,
          const int *Y = 0);
  ~Polygon();

  int NumOfFLOBs() const;
  Flob *GetFLOB(const int i);
  int Compare(const Attribute* rhs) const;
    [...]

 private:
  DbArray<Vertex> vertices;
  PolygonState state;
};
```

- vergleicht `this` mit `rhs`

### falls beide Objekte definiert

$$*this < *rhs \mapsto -1$$
$$*this = *rhs \mapsto 0$$
$$*this > *rhs \mapsto 1$$

### sonst

$$undef \times def \mapsto -1$$
$$undef \times undef \mapsto 0$$
$$def \times undef \mapsto 1$$

# Implementierung II

## weitere zu überschreibende Funktionen

```
bool Adjacent(const Attribute*) const;
Polygon *Clone() const;
void CopyFrom(const Attribute* rhs);
size_t HashValue() const;
```

# Implementierung II

## weitere zu überschreibende Funktionen

```
bool Adjacent(const Attribute*) const;
Polygon *Clone() const;
void CopyFrom(const Attribute* rhs);
size_t HashValue() const;
```

- werden diese Funktionen nicht überschrieben, drohen Speicherfehler

# Implementierung III

## geerbte Funktionen

```
bool IsDefined() const;
void SetDefined(bool defined);
```

# Implementierung III

## geerbte Funktionen

```
bool IsDefined() const;
void SetDefined(bool defined);
```

- `IsDefined()` darf nicht überschrieben werden

# Implementierung III

## geerbte Funktionen

```
bool IsDefined() const;
void SetDefined(bool defined);
```

- `IsDefined()` darf nicht überschrieben werden
- beim Überschreiben von `SetDefined(...)` muss `Attribute::SetDefined(...)` aufgerufen werden

# Fragerunde

Jetzt ist Zeit für Ihre Fragen und Anmerkungen.

# Fragerunde

Vielen Dank für Ihre Aufmerksamkeit.

Um 16 Uhr geht es weiter mit Teil II.