Forderungen an gute Bibliotheken

- systematische Taxonomie für Bibliothek

  $\longrightarrow$ sonst Erlernen der Bibliothek zu teuer

  $\longrightarrow$ Verhalten besser einschätzbar (Zusicherungen, Namen)

- so generisch wie möglich

  $\longrightarrow$ Anwendbar in vielen Bereichen

- so effizient wie möglich

  $\longrightarrow$ sonst nicht verwendet

- Bibliotheken müssen verständlich sein

  $\longrightarrow$ sonst nicht verwendet

Persistente Objekte (einfache Form)

Objekte persistent, wenn in einer Datei/Datenbank gespeichert und länger als Programm existent

Einfachste Form: Objekt zu (char $*$) konvertiert, als String gespeichert, beim Einlesen zurückkonvertiert

- einfach, aber funktioniert nicht mit Zeigern
  $\rightarrow$ vorher auflösen
- nicht portabel, da Speicherformat von Hardware, Compiler, Programmversion (Bibliotheksversion) abhängig
- fehleranfällig weil oft nicht im Detail verstanden

einfache Persistenzmechanismen von Standardbibliotheken funktionieren nach diesem Schema $\rightarrow$ kaum besser.

Persistente Objekte

meist sinnvoll, externe von interner Darstellung zu trennen:

- internes Format leichter änderbar
- leichter portierbar
- Zeiger (Refernzen) explizit darstellbar

Funktionen zur Konvertierung zwischen interner und externer Darstellung meist kontextabhängig

gemeinsame Oberklasse aller Objekte, die persistent gemacht werden können, nur begrenzt sinnvoll

einfache Datei — XML — Datenbank

Konzept versus Datenstruktur

Beispiel: Implementierung einer Liste

Variante A: Liste als Konzept; Listenelemente privat

Variante B: Klasse beschreibt Listenelement (Datenstruktur)

- B einfacher zu implementieren
- A empfohlen, da schwächere Objektkopplung
  (B führt leicht zu zusätzlichen Parametern)
- Probleme bei der Verwendung von B werden normalerweise früh erkannt, und Refaktorisierung führt zu A

$\rightarrow$ Denken in Konzepten, Verlass auf Refaktorisierung

Globale und statische Variablen

- globale Variablen bewirken, dass Änderungen beliebige andere Programmteile beeinflussen können
- statische Variablen von vielen Objekten manipulierbar
- statische Variablen in „templates“ fehleranfällig
  (homogene versus viele Arten heterogener Generizität)
- Konstante bilden oft Ausnahme
  (Vermeidung würde Änderbarkeit einschränken)
- statische Variablen sinnvoll, wenn tatsächlich als Variablen der Klassen selbst gesehen

Zugriff auf Variablen

- public Variablen machen Implementierungsdetails sichtbar – nachträgliche Codeänderungen schwierig

  → public Variablen vermeiden

- nichtlokale Variablen in der Regel nur über „this“ schreiben

- Vermeidung von public Variablen ist durch „get“- und „set“-Methoden immer möglich

  → Änderungen leichter als bei Variablen, da Methoden in Unterklassen überschreibbar

  → starke Objektkopplung bleibt

  → besseres Design überlegen

Typumwandlung

- „static cast“ besonders gefährlich, da unüberprüfte Annahmen fix verdrahtet
- „dynamic cast“ (überprüfte Typumwandlung) auch gefährlich, da falsche Annahmen zu spät erkannt
- Typumwandlungen setzen fixe, bekannte Typhierarchie voraus, was nachträgliche Änderungen erschwert

Bedingte Anweisungen, Typvergleiche

- scheinen harmlos weil häufig verwendet
- Typvergleiche setzen fixe Typhierarchie voraus

  $\rightarrow$ nachträgliche Änderungen erschwert (nicht lokal)
- case-Statements setzen fixe Menge von Fallunterscheidungen voraus – Änderungen der Menge nicht lokal
- Bedingte Anweisungen und Typvergleiche durch dynamisches Binden fast immer ersetzbar

  $\rightarrow$ Änderungen lokal und überprüft

Binäre Methoden – Lösungsvergleiche

- dynamischer Typvergleich
  $\rightarrow$ akzeptabel, da nur eigener Typ vorkommt
- Kombination Überladen und dynamischer Typvergleich
  $\rightarrow$ effizient wenn Typ statisch bekannt
- Multimethoden bzw. Visitor Pattern $\rightarrow$ meist overkill
- einfache Generizität
  $\rightarrow$ selten eine Lösung, da Typ statisch unbekannt
- gebundene Generizität
  $\rightarrow$ Lösung, wenn von Sprache und Klassen unterstützt
  (aber Probleme bei tiefen Hierarchien)

Arten gebundener Generizität

- F-gebundene Generizität: $S \leq T\langle S \rangle$

  Beispiel: Integer $\leq$ Comparable$\langle$Integer$\rangle$

  kann mit binären Methoden umgehen

  funktioniert nur eingeschränkt über mehrere Ebenen

- higher-order subtyping (matching):

  Def.: $S <\# T$ wenn $S\langle U \rangle \leq T\langle U \rangle$ für alle U

  Matching unterstützt binäre Methoden

  als Schranke bei Generizität typsicher

  allgemein muss U für statische Typsicherheit bekannt sein

Wann Generizität, wann Subtyping

- Mechanismen nicht immer austauschbar. Daher:
  Subtyping für heterogene Datenstrukturen
  Generizität zur Vermeidung kovarianter Probleme

- Erwartete Änderungen abfangen:
  Generizität wenn sich Parametertypen ändern
  Subtyping für neue Versionen mit gemeinsamem Obertyp

- Subtyping für Kompatibilität zu gegebener Klasse

- Subtyping um case-Statements, if-Statements und dynamische Typvergleiche zu vermeiden

- kombinieren, oder natürlicheren Mechanismus verwenden

Rückgabewerte von Funktionen

- Ergebnistyp „void“ wenn kein Rückgabewert nötig

  → unbedeutend bessere Fehlererkennung

- „this“ als Rückgabewert wenn nichts anderes nötig

  → manchmal schönere Syntax

  Beispiel: Collection x = f(); x.add(y); x.add(z); g(x)

  ersetzt durch

  g(f().add(y).add(z));

Aspekte

Komponente: alles was sauber in Prozedur/Objekt gekapselt werden kann

Aspekt: alles andere —

Eigenschaften, die die Effizienz oder Semantik einer Komponente auf systematische Weise beeinflussen, wie z.B.

- Muster der Speicherzugriffe
- Synchronisation bei Nebenläufigkeit
- Netzwerkbelastung
- Verschmelzung von Schleifen
- Speicherung von Zwischenergebnissen

Ebenen der Interprozesskommunikation

- Transaktionen
- atomare Aktionen
- serialisierbare Kommunikation
- „mutual exclusion“
- einfacher Input/Output – unsynchronisiert

wenn mehr als „mutual exclusion“ gefordert, muss sich Programmierer selbst darum kümmern

Aspekte der Kommunikation

Für die Ebenen der Interprozesskommunikation und der Sicherheit vor unerwünschten Zugriffen gelten:

- hohe Ebene: teuer und ineffizient
- niedrige Ebene: fehleranfällig, oft inakzeptabel
- Optimum hängt von der Anwendung ab
- Programmierer/Anwender sollen wählen können

Reflektive Programmierung

Reflektion: Sprache wird dynamisch analysiert oder verändert

- lässt dem Programmierer viele Freiheiten
- sehr fehleranfällig
- Basis für aspektorientierte Programmierung

Änderungen der Semantik eines Programms möglich durch

- Änderungen des Programms
- Änderungen der Sprachen bzw. Standardbibliotheken
- Parameterisierte Sprachen bzw. Bibliotheken

OO Arten der Reflektion

- interne Klassenstruktur sichtbar machen (Metadaten)
- Klassen zur Laufzeit hinzufügen, Typinfo. verwenden
- Klassen dynamisch erzeugen bzw. verändern
- „message passing mechanism“ oder andere Sprachkonzepte der untersten Stufe sichtbar machen und verändern

Alternativen zur Reflektion

Precompiler vor Programmübersetzung ausführen:

- effizient zur Laufzeit
- einfache Änderungen mit kleinem Aufwand
- größere Analysen können teuer sein
- Laufzeitinformation nicht verfügbar

Programmiersprache ändern oder erweitern

- nur für große Änderungen sinnvoll
- interne Informationen im Compiler wiederverwendbar
- nicht portabel (mit Ausnahmen)
- einfach, wenn Erweiterung nur mit Bibliotheken

Attribute in C#

- Attribut = Objekt, das mit Programmelementen verknüpfte Daten darstellt
- verknüpfbar mit Klassen, Methoden, Parametern, etc.:

```
[Serializable]
class MySerializableClass { ... }
```

- intrinsic (in .NET integriert) versus benutzerdefiniert
- Beispiel für Verwendung eines benutzerdef. Attributs:

```
[BugFixAttribute("Stefan", "23.3.2007", Comment="OK")]
class MyTestClass { ... }
```

- über Reflection zur Laufzeit zugänglich

Beispiel für Attributdefinition

```
[AttributeUsage(AttributeTargets.Class,AllowMultiple=true)]
public class BugFixAttribute : System.Attribute {
  public BugFixAttribute (string programmer, string date) {
    this.programmer = programmer;  this.date = date);
  }
  public string Comment {
    get { return comment; }
    set { comment = value; }
  }
  public string Programmer { get { return programmer; } }
  public string Date { get { return date; } }
  private string programmer, date, comment;
}
```

Annotationen in Java

- entsprechen ungefähr Attributen von C# – Beispiel:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface BugFix {
  String prog(); String date(); String comment();
}
...
@BugFix(prog="Stefan", date="23.3.2007", comment="OK")
class MyTestClass { ... }
```

- SOURCE Annotationen ersetzen javadoc Annotationen

- eigentlich mächtiger als Attribute, aber zur Laufzeit derzeit kaum verwendet (nicht in Standardbibliotheken)

Dynamische Erweiterung von Klassen

- dynamische Sprachen unterstützen Klassenerweiterung zur Laufzeit, statische Sprachen nicht direkt – Auswirkungen?
- Decorator erlaubt dynamische Erweiterung
- durch „final“ nur ohne Ersetzbarkeit verhinderbar
  $\rightarrow$ Erweiterbarkeit auch in statischen Sprachen Normalfall
- direkte Klassenänderung in dyn. Sprachen etwas effizienter als Decorator (Laufzeit + Programmieraufwand)
- statisches Konzept effizienter wenn keine Klassenänderung nötig (auch in dynamischen Sprachen)