Lernzettel

Abstrakte Datentypen und Schnittstellen

Universität: Technische Universität Berlin
Kurs/Modul: Einführung in die Informatik - Vertiefung
Erstellungsdatum: September 20, 2025

Lernzettel: Abstrakte Datentypen und Schnittstellen

(1) Definition. Abstrakte Datentypen (ADTs) beschreiben ein Verhalten bzw. eine Menge von Operationen, ohne die interne Repräsentation festzulegen. Die Schnittstelle des ADT gibt die erlaubten Operationen, deren Namen, Typen und erwartetes Verhalten vor. Die Implementierung verbirgt diese Repräsentation, ermöglicht aber dieselben Operationen.

(2) Abstrakte Datentypen (ADTs). - Ein ADT spezifiziert eine Menge von Operationen mit gültigen Invarianten, z.B.
Stack mit push, pop, top, isEmpty, size.
- Repräsentationsunabhängigkeit: der Code, der das ADT nutzt, kennt nur die Schnittstelle.
- Vorteile: Austauschbarkeit von Implementierungen, bessere Wartbarkeit und Tests.

(3) Schnittstellen. - Eine Schnittstelle (Interface) definiert eine Menge von Operationen ohne Implementierung.
- In Java z.B. `public interface Stack<T>` mit Methoden `void push(T x)`, `T pop()`, `T peek()`, `boolean isEmpty()`, `int size()`.
- Vertrag (Design by Contract): Pre- und Postbedingungen beschreiben das erwartete Verhalten.

(4) Implementierung vs Abstraktion. - ADT beschreibt das Verhalten; konkrete Klassen liefern die Repräsentation (z.B. ArrayStack, LinkedStack).
- Wechsel der Implementierung ohne Änderung der Nutzerschnittstelle ist möglich.
- Typische Implementierungen: verkettete Strukturen, Arrays,Deque-basierte Strukturen.

(5) Generische ADTs. - Generische ADTs arbeiten typunabhängig, z.B. `interface Stack<T>`, `class ArrayStack<T>`.
- Typensicherheit und Wiederverwendbarkeit steigen, da dieselbe Schnittstelle mit jedem Typ genutzt werden kann.

(6) Iteration über ADTs. - Iteratoren ermöglichen sequentiellen Zugriff, ohne die interne Repräsentation offenzulegen.
- Typische Vertreter: `Iterator<T>` oder `Iterable<T>` in Java.

(7) Beispiel-ADTs. - Stack-ADT: `push(T x), T pop(), T peek(), boolean isEmpty(), int size()`.
- Queue-ADT: `enqueue(T x), T dequeue(), T front(), boolean isEmpty(), int size()`.
- Liste-ADT: `insert(int index, T x), T get(int index), T remove(int index), int size()`.

(8) Schnittstellen-Design in Java (Beispiel).

```
public interface Stack<T> {
  void push(T x);
  T pop();
  T peek();
  boolean isEmpty();
  int size();
}
```

(9) Generische Implementierungen (Beispiel). - `class ArrayStack<T> implements Stack<T>` implementiert die Operationen mit einem generischen Array als zugrundeliegende Struktur.
- Vorteil: Typsicherheit, Wiederverwendbarkeit, einfache Austauschbarkeit der Repräsentation.

(10) Kapselung und API-Design. - Die Implementierung ist verborgen; Nutzer arbeiten mit der API (Schnittstelle)
und ihren Verträgen.
- Guter API-Entwurf minimiert Abhängigkeiten, sorgt für klare Invarianten und einfache Wartung.

(11) Praxis-Tipps. - Beginne mit einer kleinen, klaren Schnittstelle (Single-Responsibility).
- Dokumentiere Pre-/Postbedingungen, invarianten Fälle.
- Nutze Generics, um Wiederverwendbarkeit zu erhöhen.
- Schreibe Tests, die nur die Schnittstelle verwenden (Black-Box-Tests).

(12) Fazit. Abstrakte Datentypen definieren das Verhalten, Schnittstellen spezifizieren das Vertragsverhältnis. Durch Generics und Iteration wird die Abstraktion robuster, wartbarer und flexibel gegenüber Implementierungswechseln.