Ada mit Stil

Tim Schönleber
Universität Stuttgart
Institut für Softwaretechnologie
Breitwiesenstraße 20-22
D-70565 Stuttgart
tim@sw-technik.net

Zusammenfassung

Mit Hilfe von Ada soll qualitativ hochwertige, zuverlässige, wiederverwendbare und portable Software hergestellt werden. Da dies allerdings keine Programmiersprache vollständig garantieren kann, existieren Programmierrichtlinien, sogenannte Style-Guides, welche zusätzliche Regeln und Richtlinien zur Erstellung von Programmen aufstellen, und spezielle Entwurfsmethoden bzw. Design Patterns.

Dieser Artikel befaßt sich zu Beginn mit den generellen Vor- und Nachteilen von Style-Guides. Im folgenden werden dann die wichtigsten Regeln anhand des Style-Guides „Ada 95 Quality and Style: Guidelines for Professional Programmers“ zusammengefasst und an positiven und negativen Beispielen erläutert. Am Ende des Artikels wird noch kurz auf Entwurfsmethoden und Design Patterns im Kontext mit Ada 95 eingegangen.

1. Style-Guides allgemein

Style-Guides enthalten Regeln und Richtlinien zur Erstellung von Programmen über die Programmiersprache hinaus, das heißt ein Style-Guide legt fest, wie bestimmte Sachverhalte beim Programmieren zu handhaben sind.

Der Einsatz von Style-Guides bringt eine Reihe von enormen Vorteilen mit sich, denen nur wenige Nachteile gegenüber stehen. Diese Nachteile verschwinden allerdings nahezu völlig, wenn die Mitarbeiter von den Vorzügen eines Style-Guides überzeugt sind bzw. wenn der Style-Guide den Anforderungen der Firma oder des Projektes entspricht.

Der wohl wichtigste Vorteil durch den Einsatz von Style-Guides ist die größere Wahrscheinlichkeit, weniger Fehler von Beginn an im Code zu haben. Vor allem subtile Fehler schleichen sich nicht so leicht ein. Natürlich lassen sich Fehler auf diese Art und Weise nicht hundertprozentig vermeiden, aber sie können durch den Einsatz von Style-Guides leichter identifiziert und behoben werden. Dies liegt vor allem daran, dass der Code leichter gelesen und schneller verstanden werden kann. Außerdem wird die Wartung durch einen einheitlichen Stil erheblich erleichtert und auch die Anpassung an neue Anforderungen wird erleichtert.

Diesen Vorteilen stehen auch Nachteile gegenüber. Der erste Nachteil, welcher am wahrscheinlichsten für die Ablehnung von Style-Guides ist, ist die anfangs notwendige Zeit-Investition. Zu Beginn ist etwas mehr Zeit notwendig für die Einarbeitung und das eigentliche Programmieren. Hat man diese Zeit allerdings einmal investiert, so existiert dieser Nachteil praktisch nicht mehr, da von nun an durch den Einsatz des Style-Guides sogar Zeit eingespart wird. Eng mit diesem Nachteil verbunden ist auch ein weiterer, nämlich mangelnde Akzeptanz des Style-Guides und das Festhalten am eigenen Stil des Programmierens. Aber auch dieser Nachteil lässt sich meiner Meinung nach mit entsprechender Überzeugungsarbeit aus der Welt schaffen. Der einzige Nachteil, der immer bestehen bleibt, ist die Frage, ob die Richtlinie an einer Stelle nun gebrochen werden darf, oder nicht? An dieser Stelle gilt es die beiden konkurrierenden Eigenschaften Flexibilität und Präzision im Style-Guide entsprechend zu berücksichtigen, so dass sich diese Frage so selten wie möglich stellt. Im Zweifelsfall aber im Sinne des Style-Guides entscheiden und eventuell notwendige Ausnahmen begründen.

Zusammenfassend kann man allerdings festhalten, dass die Vorteile von Style-Guides deutlich überwiegen, wenn alle Beteiligten daran glauben und den Style-Guide einhalten und nicht vorsätzlich sabotieren. Aus diesem Grund sollte ein Style-Guide griffig und nicht zu ausführlich sein, sondern sich auf die wesentlichen Punkte konzentrieren. Umfangreichere Style-Guides, wie der im folgenden betrachtete „Ada 95 Quality and Style“ sollten daher für den spezifischen Einsatz auf das wesentliche gekürzt werden, da diese Richtlinie sonst aufgrund ihres Umfangs von den Entwicklern nicht gelesen und nicht ernst genommen wird.

In vielen Bereichen, wie z.B. der Präsentation des Source Codes steht außerdem nicht der genaue Inhalt der Regeln im Vordergrund, sondern es kommt vielmehr darauf an, dass es definiert und geregelt ist, aber nicht so sehr wie.

Die Umsetzung und Einhaltung von Style-Guides kann auch mit dem Einsatz von Software-Tools unterstützt werden. Die Regeln eines Style-Guides, welche keinen semantischen Inhalt haben, wie z.B. die

Einrückung oder die Beschränkung von 80 Zeichen pro Zeile, lassen sich mit Hilfe von Text-Editoren, Code-Generatoren und Compilern kontrollieren bzw. automatisch umsetzen. Sobald die Regeln jedoch eine Semantik enthalten versagt der Tool-Support. Hierbei gilt es allerdings festzuhalten, dass die Mehrzahl der Regeln eine Semantik besitzen, die nur von Menschen korrekt umgesetzt und angewendet werden können. Somit können Software-Tools nur als Unterstützung betrachtet werden. Die konsequente Anwendung eines Style-Guides direkt beim Programmieren ist daher nicht zu ersetzen.

2. Ada95 Quality and Style: Guidelines for Professional Programmers

Beispielhaft wird nun an dieser Stelle der Style-Guide "Ada95 Quality and Style: Guidelines for Professional Programmers" betrachtet und die wichtigsten Regeln anhand dieses Style-Guides erläutert. Es wurde dieser Style-Guide ausgewählt, da die Entwicklungsphilosophie des Bauhaus Projekts und auch der Bauhaus Coding Style (welcher allerdings noch nicht fertiggestellt wurde) diesem Style-Guide folgt.

Der betrachtete Style-Guide wurde für das „Department of Defense Ada Joint Program Office" durch das „Software Productivity Consortium" angefertigt. Dieser Style-Guide ist in neun kleinere Richtlinien unterteilt, die sich jeweils mit einem bestimmten Aspekt der Programmierung beschäftigen und besteht aus insgesamt über 600 einzelnen Regeln. Diese neun Richtlinien werden nun in den folgenden Kapiteln mit den wichtigsten Regeln und Beispielen kurz vorgestellt und erläutert.

2.1. Source Code Präsentation

Diese Richtlinie beschäftigt sich mit dem Layout des Source Codes, das heißt mit der Formatierung des Source Codes.

In diese Kategorie fallen die Verwendung von Leerzeichen und Leerzeilen, die Definition der Einrückung, die Ausrichtung von Operatoren und Deklarationen, die Anzahl der Anweisungen pro Zeile, sowie die Zeilenlänge.

Bei dieser Formatierung kommt es nicht so sehr auf das wie an, sondern darauf, dass diese Dinge definiert sind und auch von allen konsistent eingehalten werden.

Deshalb sollten einige wenige Beispiele genügen, die die Mehrzahl der Programmierer als selbstverständlich ansehen sollten.

Hierzu zählen zum Beispiel das Einrücken von eingeschachtelten Kontrollstrukturen und fortgeführten Zeilen, die vertikale Ausrichtung von Operatoren und Deklarationen, die Verwendung von Leerzeichen zur Gruppierung logisch zusammengehöriger Abschnitte, die Beschränkung auf eine Anweisung pro Zeile oder auch die Beschränkung der Zeilenlänge auf 80 Zeichen.

Beispiel

```
...
type Second_Of_Day is range 0..86_400
type Noon_Relative_Time is
  (Before_Noon, After_Noon, High_Noon)

subtype Morning   is ...
subtype Afternoon is ...
...
Current_Time := ...

if Current_Time in Morning then
  Time_Of_Day := Before_Noon;
elsif Current_Time in Afternoon then
  Time_Of_Day := After_Noon;
else
  Time_Of_Day := High_Noon;
end if;

case Time_Of_Day is
  when Before_Noon =>
Get_Ready_For_Lunch;
  when High_Noon   => Eat_Lunch;
  when After_Noon  => Get_To_Work;
end case;
...
```

Die Tiefe der Einrückung sollte allerdings so klein wie möglich gehalten werden. Im Allgemeinen ist eine Einrückungstiefe von 5 ein guter Richtwert. Hat man eine größere Einrückungstiefe, so sollte man prüfen, ob man diesen Programmteil nicht anders strukturieren sollte.

Beispiel

```
if not Condition_1 then
  if Condition_2 then
    Action_A;
  else  -- not Condition_2
    Action_B;
  end if;
else -- Condition_1
  Action_C;
end if;
```

Dieses Codestück kann in der folgenden Form klarer und mit geringerer Einrücktiefe geschrieben werden:

```
if Condition_1 then
  Action_C;
elsif Condition_2 then
  Action_A;
else  -- not (Condition_1 or Condition_2)
  Action_B;
end if;
```

2.2. Lesbarkeit

Dieser Teil des Style-Guides regelt die Namenskonventionen, sowie die Verwendung von Typen und Kommentaren.

Ebenso wie im vorherigen Kapitel kommt es hierbei auch primär auf das Vorhandensein und die konsequente Einhaltung dieser Regeln an.

Bezeichner bestehen aus einer Folge von Wörtern, die durch _ voneinander getrennt sind, wobei jedes Wort mit einem Großbuchstaben anfängt und der Rest kleingeschrieben wird. Eine Ausnahme bilden Konstanten, die komplett großgeschrieben werden. Außerdem sollten Bezeichner sinnvoll gewählt (der Anwendung entsprechend) werden und eigene Abkürzungen in der Regel vermieden werden.

Beispiele

```
MAX_CAR_SPEED : constant Positive := 250;
type Car_Speed is range 0..MAX_CAR_SPEED;
Current_Speed : Car_Speed;
```

Als Prozedurnamen sollten Verben gewählt werden, die eine Aktion ausdrücken, für Boolean-Funktionen sollten Prädikatssätze gewählt werden und für andere Funktionen sollte man Substantive verwenden.

Beispiele

```
procedure Get_Next_Token
function  Is_Empty
function  Length
```

Kommentare sollten nur dort verwendet werden, wo sie für das Programmverständnis wichtig sind. Kommentare, die nur die Information aus dem Source Code wiederholen helfen niemandem, sondern blähen das Programm nur unnötig auf. Kommentare für kurze Abschnitte sind dahinter zu schreiben, Kommentare für größere Abschnitte gilt es davor zu platzieren. Kommentare am Ende einer nicht leeren Zeile sollten generell vermieden werden.

Für Kopfkommentare von Dateien und Prozeduren bzw. Funktionen werden Standard-Templates verwendet. Im Bauhaus Projekt werden die Kopfkommentare von Prozeduren und Funktionen allerdings nur in der Spezifikations-Datei (.ads-File) angewandt, um bei Änderungen eine Inkonsistenz zwischen Spezifikations- und Implementierungs-Datei zu vermeiden.

2.3. Programm-Struktur

Diese spezielle Richtlinie beschäftigt sich im Gegensatz zu den beiden Richtlinien zuvor mit der Programm-Struktur im Ganzen und nimmt deshalb Bezug auf grundlegende Prinzipien des Software Engineerings, wie z.B. Information Hiding und Abstraktion. Diese Regeln sind allerdings nicht so eindeutig und offensichtlich wie zum Beispiel die Regeln für die Präsentation des Source Codes bzw. der Namenskonvetionen. Die Umsetzung dieser Regeln setzt daher eine gewisse Erfahrung bzw. ein gewisses Know-How des Programmierers voraus. Die Basis für die Einhaltung dieser Regeln wird meist schon während der Entwurfsphase gelegt. Mit geeigneten Entwurfsmethoden können diese Regeln sichergestellt bzw. unterstützt werden. Mehr zu diesem Thema am Ende dieses Artikels.

Das Programm soll in sinnvolle Pakete unterteilt werden, welche für Information Hiding sorgen. Zudem sollten die Pakete einen hohen Zusammenhalt und eine geringe Kopplung aufweisen.

Generell sollten Unterprogramme verwendet werden, um die Abstraktion zu erhöhen. Dies sollte allerdings in vernünftigem Maße erfolgen, da eine zu tiefe Schachtelung wieder ins Negative umschlagen würde.

2.4. Verwendung von Ada-Konstrukten

Dieser Teil nimmt speziell Bezug auf die Praxis des Programmierens unter Berücksichtigung der Konstrukte von Ada 95.

Die Verwendung optionaler Teile der Ada 95-Syntax können die Lesbarkeit erheblich erleichtern. So sollten eingeschachtelte Schleifen und Blöcke mit Namen versehen und über diese angesprochen werden. (zum Beispiel sollte der exit-Befehl in Schleifen nur in Verbindung mit dem Schleifennamen verwendet werden).

Die formalen Parameter in Prozedur- bzw. Funktionsaufrufen sollten den Namenskonventionen für „normale“ Bezeichner folgen und sprechend gewählt werden, um unnötige Kommentierung zu vermeiden. In Aufrufen mit vielen Parametern oder bei der Instanziierung von generischen Parametern sind sogenannte „Named Associations“ zu verwenden. (gleiches gilt für die Verwendung von Aggregaten bei der Wertzuweisung von Records)

Beispiel

```
procedure Nonsens
  (Factor : in     Real;
   Value  : in out Integer;
   Flag   :    out Boolean)
  is
  ...
end Nonsens;
```

Aufruf mit „Named Associations“

```
...
Nonsens (Factor => X-Factor,
         Value  => Sum,
         Flag   => True);
...
```

anstatt

```
...
Nonsens (X-Factor, Sum, True);
...
```

Auch den Vorteil des sogenannten „Strong Typing“ von Ada 95 sollte man sich zu nutzen machen, d.h. wenn möglich sind Subtypes, abgeleitete Typen (derived types) und private Typen einzusetzen. Der

Einsatz des „Strong Typing“-Konzepts bringt auch fast keinerlei negative Auswirkungen bzgl. der Größe des ausführbaren Codes mit sich, da die Überprüfung der Typen bereits zur Compilezeit erfolgt.

Besondere Vorsicht ist bei dynamischen Daten geboten, welche auf dem Heap erstellt werden. Generell sollten dynamische Datenstrukturen nur verwendet werden, wenn dies unumgänglich ist. Werden solche dynamischen Datenstrukturen allokiert, so ist darauf zu achten, dass diese explizit deallokiert werden und zudem sollte der Einsatz sogenannter „dangling references“ und „aliasing“ minimiert werden.

Beispiel
Die folgenden Zeilen zeigen, wie „dangling references“ entstehen können:

```
P1 := new Object;
P2 := P1;
Unchecked_Object_Deallocation (P2);
...
X := P1.all;
```

In diesem Fall würde die letzte Zeile zu einem Programmabsturz führen, da das zu dereferenzierende Objekt nicht mehr existiert.

Im Umgang mit Arrays und auch Typen sollten stets die Schlüsselwörter `´First`, `´Last` und `´Range` anstatt numerischer Literale verwendet werden.

Die Verwendung des use-Befehls sollte auch vermieden werden. Dies erhöht das Programmverständnis deutlich, da stets klar ist, aus welchem Paket ein Ausdruck stammt und es ermöglicht die gezielte Suche mit Tools wie z.B. Grep. Der rename-Befehl sollte auch vermieden und nur eingesetzt werden, falls ein Paket im Interface die Funktion eines anderen Pakets anbieten soll.

Stattdessen kann und sollte der use-type-Befehl verwendet werden.

Exceptions sind ausschließlich für Fehlersituationen zu verwenden und dürfen in einem korrekt ausgeführten Programm nicht auftreten (z.B. für goto-Befehle), das heißt, dass über Exceptions keine regulären Kontrollflüsse geführt werden dürfen. Prinzipiell sollte man versuchen, die Fehlerfälle bereits im Programm abzufangen und den Einsatz von Exceptions auf ein Minimum zu reduzieren.

2.5. Parallelität

Der Abschnitt Parallelität beschäftigt sich mit der Gleichzeitigkeit von Prozessen und den damit verbundenen Problemen, vor allem der Synchronisation mehrerer Prozesse.

Wenn möglich sollten geschützte Objekte, sogenannte „protected objects“ anstelle von Tasks verwendet werden, da diese besonders leicht handhabbare Mechanismen für die Synchronisation und den exklusiven Datenzugriff zur Verfügung stellen.

Da die Handhabung von Tasks in Ada 95 nicht ganz unproblematisch ist, finden sich noch weitere Regeln und Normen speziell zum Umgang mit Tasks. Da dies allerdings den Umfang dieses Artikels überschreiten würde, werde ich an dieser Stelle nicht darauf eingehen, sondern verweise auf „Ada 95 Quality and Style“.

2.6. Portabilität

Diese Richtlinie enthält Regeln, um die Portierung eines Programms auf unterschiedliche Platformen zu erleichtern.

Prinzipiell sollten wenn nicht unbedingt erforderlich (z.B. um die Kompatibilität mit einem alten Ada 83 Programm sicherzustellen) keine veralteten Besonderheiten aus Ada 83 verwendet werden.

Allgemeine Vorraussetzungen, wie zum Beispiel die Anzahl verfügbarer Bits für Zahlen vom Typ Integer, sollten bereits zu Beginn eines Projektes für alle in Frage kommenden Zielplattformen berücksichtigt werden.

Plattformspezifische Teile der Implementierung sollten in speziellen Paketen isoliert werden, so dass diese Pakete leicht durch angepasste Pakete für andere Plattformen ersetzt werden können.

Sehr wichtig ist auch die spezielle Kommentierung und Kennzeichnung plattformabhängiger Teile der Implementierung, denn nur so können diese Teile bei einer Portierung in kurzer Zeit ausfindig gemacht werden.

2.7. Wiederverwertbarkeit

Wiederverwertbarkeit ist das Ausmaß, in wie fern Code in anderen oder in der gleichen Applikationen mit möglichst minimalem Änderungsaufwand verwendet werden kann. Um Code wieder verwertbar zu machen, muss dieser verständlich, von möglichst hoher Qualität, anpassungsfähig und unabhängig sein. Diese Richtlinie versucht dies aus (programmier-)technischer Sicht zu unterstützen.

Um die Verständlichkeit sicherzustellen, sollten die zuvor erläuterten Namenskonventionen beachtet werden, besonders im Umgang mit generischen Einheiten.

Um eine hohe Qualität zu garantieren, sollten ebenfalls die zuvor betrachteten Regeln beachtet werden, da all diese Regeln als oberstes Ziel eine möglichst hohe Qualität haben. An dieser Stelle sind besonders die Verwendung von Konstanten und die Minimierung und Dokumentation von bestimmten Annahmen zu nennen.

Für eine hohe Anpassungsfähigkeit sei vor allem der Einsatz von generischen Einheiten empfohlen. Auf diese Weise kann auch verhindert werden, dass ähnliche Funktionen mehrfach realisiert werden.

Unabhängigkeit wird vor allem durch entsprechende Strukturierung in der Entwurfsphase erreicht.

2.8. Objektorientierte Besonderheiten

Diese Richtlinie behandelt den Einsatz der objektorientierten Eigenschaften von Ada95 (Vererbung und Polymorphismus).

In Ada 95 gibt es nicht explizit Klassen, wie in anderen Programmiersprachen, zum Beispiel Java. In Ada 95 können die Typen als eine Klasse betrachtet werden und konkrete Instanzen sind dann die Variablen von diesem Typ. Aus diesem und anderen Gründen (zum Beispiel Information Hiding) sollten deshalb abstrakte Datentypen verwendet werden, wenn eine Klassenstruktur modelliert wurde. Die Probleme vor allem in Bezug auf Vererbung lassen sich jedoch aus der reinen objektorientierten Programmierung übernehmen. Mehrfachvererbung wird von Ada 95 nicht unterstützt.

Vererbung, das heißt eine „is-a“-Hierarchie sollte mit Hilfe von erweiterbaren Typen, sogenannten „tagged types“ erfolgen. Bei sehr einfach und primitiven Typen sollte man „tagged types“ vermeiden.

Beispiel

```
type Graphical_Object is tagged
  record
    X_Coordinate : Float;
    Y_Coordinate : Float;
  end record;

type Point is new Graphical_Object with
  null record;

type Circle is new Graphical_Object with
  record
    Radius : Float;
  end record;
```

2.9. Performance-Steigerung

Der letzte Teil des Style-Guides beschäftigt sich mit dem Thema Performance-Steigerung und liefert entsprechende Regeln. Diese Regeln sind allerdings mit Vorsicht zu betrachten, da sie öfters mit den zuvor erläuterten Regeln kollidieren. Um eine verbesserte Geschwindigkeit oder geringeren Speicherverbrauch zu erreichen, benutzt man meist trickreiche Programmiertechniken, so dass meist die Lesbarkeit und das Programmverständnis darunter leiden.

Den höchsten Gewinn an Performance erreicht man in der Regel sowieso mit einem guten Entwurf und nicht mit trickreichen Programmiertechniken. Dennoch kann es in manchen Fällen notwendig sein. Wie dann vorzugehen ist, beschreiben die folgenden Regeln.

Mit Hilfe von Blöcken können späte Initialisierungen vorgenommen werden. Diese eröffnen dem Compiler mehr Möglichkeiten die Verwendung der Register zu optimieren.

Arrays sollten wenn möglich feste Grenzen haben und nur wenn es nicht anders möglich ist dynamisch sein. Zudem sollten Arrays mit dem Index immer bei 0 beginnen, auch wenn dies in Ada 95 nicht zwingend erforderlich ist, da dies einen Vergleich einsparen kann.

Bezüglich Algorithmen kann die Ersetzung von mod- und rem-Operatoren und die Verwendung von case-Anweisungen einen Geschwindigkeitsvorteil bringen.

Auch der richtige Umgang mit Strings kann Vorteile bringen. Die Verwendung von „bounded strings“ gegenüber „unbounded strings“ kann Vorteile bringen, da die „unbounded strings“ eventuell auf dem Heap angelegt werden und dann Geschwindigkeitsnachteile gegenüber sich nicht auf dem Heap befindenden „bounded strings“ haben.

Auch der richtige Umgang mit Pragmas, das heißt mit Anweisungen an den Compiler, kann zu Performancesteigerungen beitragen. Allerdings sollte in der Regel auf Pragmas aufgrund der Portabilität bis auf die Ausnahme von „Assertions“ zur Überprüfung von Vor- und Nachbedingungen sowie Invarianten verzichtet werden.

Wie bereits gesagt, sollten diese Punkte aber alle mit entsprechender Vorsicht angewendet werden, da man sonst auch schnell das Gegenteil, nämlich eine Verschlechterung, erreicht.

3. Entwurfsmethoden mit Ada 95

Da Ada 95 eine prozedurale Programmiersprache mit objektorientierten Erweiterungen ist, kann man nicht sagen, dass es genau eine “richtige” Entwurfsmethode für Ada 95 gibt. Zum einen hat man die funktionale Abstraktion und die Datenabstraktion. Andererseits hat man auch die objektorientierte Sicht. Während des Entwurfs kommt es nun darauf an, die Schwerpunkte - der Anwendung entsprechend - zu setzen. Auch aus dem Aspekt heraus, dass Ada 95 keine reine objektorientierte Programmiersprache wie zum Beispiel Java ist, werden in der Regel die funktionale Abstraktion und die Datenabstraktion überwiegen. Welche Techniken es speziell zur Datenabstraktion und den Umgang mit Design Patterns in Ada 95 gibt, wird in den folgenden Abschnitten kurz erläutert.

3.2. Datenabstraktion

In Programmen müssen meist komplexe Datenobjekte manipuliert werden. Auf diese Datenobjekte wird mit speziellen Prozeduren und Funktionen lesend und schreibend zugegriffen. Es lassen sich nun zwei Grade der Datenabstraktion unterscheiden: abstrakte Datenobjekte und abstrakte Datentypen

3.2.1 Abstrakte Datenobjekte

Ein abstraktes Datenobjekt faßt Datenstrukturen und Zugriffsoperationen auf diese Datenstrukturen zu einer

Einheit zusammen. Der Anwender kann nur über die Zugriffsoperationen auf die Daten zugreifen. Die Abstraktion besteht aus der Sicht des Anwenders darin, dass die Details der Datenstruktur verborgen sind.

Ein Modul, welches ein abstraktes Datenobjekt realisiert, wird Datenobjekt-Modul genannt. Ein Datenobjekt-Modul entspricht weitgehend einem Objekt in der objektorientierten Softwareentwicklung und wird in Ada 95 mit Hilfe eines Package realisiert. Die Daten entsprechen den Attributen und die Zugriffsoperationen den Methoden eines Objekts. Von einem abstrakten Datenobjekt existiert immer nur genau eine Instanz.

Einsatzgebiete eines abstrakten Datenobjekts sind vor allem Sammlungen („collections“) von Einträgen, wie zum Beispiel ein Stack, oder die Verwaltung einzelner komplexer Einträge, wie zum Beispiel eine Hand bei der Robotersteuerung.

3.2.2 Abstrakte Datentypen

Abstrakte Datentypen sind eine Verallgemeinerung abstrakter Datenobjekte. Bei einem abstrakten Datentyp sind Typ- und Variablendeklaration getrennt, und somit können beliebig viele Instanzen eines abstrakten Datentyps gebildet werden. Der abstrakte Datentyp dient sozusagen als Schablone, um abstrakte Datenobjekte zu erzeugen. Ein abstrakter Datentyp entspricht weitgehend einer Klasse in der objektorientierten Softwareentwicklung und wird in Ada 95 durch generische Pakete beschrieben. Da man abstrakte Datentypen oft in leicht modifizierter Form braucht, gibt es auch noch die Möglichkeit abstrakte Datentypen mit formalen Parametern oder auch mit Typen als formalen Parametern auszustatten.

Beispiel

Ads-File:

```
generic
  Max : natural := 100;
  type Element_Typ is private;

package Queue is
  type Queue_Access is private;

  function Create return Queue_Access;

  procedure Insert
    (Queue   : in     Queue_Access;
     Element : in     Element_Typ;
     Flag    :    out Status);
  procedure Remove
    (Queue   : in     Queue_Access;
     Element :    out Element_Typ;
     Flag    :    out Status);

private
  Queue_Type is array (o..Max) of
    Element_Type;
  type Queue_Access is access Queue_Type;

end Queue;
```

Adb-File:

```
package body Queue is
  function Create return Queue_Access is
  Temp_Queue : Queue_Access;

  begin
    Temp_Queue := new Queue_Type;
    return Temp_Queue;
  end Create;

  procedure Insert
    (Queue   : in     Queue_Access;
     Element : in     Element_Typ;
     Flag    :    out Status) is
    ...
  end Insert;

  procedure Remove
    (Queue   : in     Queue_Access;
     Element :    out Element_Typ;
     Flag    :    out Status) is
    ...
  end Remove;

end Queue;
```

Anwendung:

```
procedure Test is
  type Element_Type is
    record
      Vorname  : String;
      Nachname : String;
    end record;

  package Queue_Float is new
    Queue (200, Float);
  package Queue_Name is new
    Queue (50, Element_Type);
  ...
  Demo_Queue : Queue_Float.Queue_Access
    := Queue_Float.Make_Queue ;
  ...
  Queue_Float.Insert (1.25, Status) ;
  ...
end Test ;
```

3.3. Design Patterns und Ada 95

Design Patterns sind Entwurfsmuster, die bewährte, generische Lösungen für immer wiederkehrende Probleme angeben. Da Ada 95 keine rein objektorientierte Programmiersprache ist lassen sich die Design Patterns, wie sie aus der objektorientierten Programmierung bekannt sind, nicht ohne weiteres auf Ada 95 übertragen. Allerdings sind abstrakte Datenobjekte und abstrakte Datentypen auch eine Art Entwurfsmuster, welche man unbedingt anwenden sollte.

Wie und ob sich Design Patterns auf Ada 95 übertragen lassen, muss für jeden Fall individuell geprüft und beurteilt werden. Es kann nämlich praktischer sein, die speziellen Möglichkeiten von Ada 95 auszunutzen, anstatt ein Design Pattern zu übertragen. Am Beispiel des bekannten Singleton-Design-Pattern wird dies deutlich. In Ada 95 kann ein Singleton einfach über ein abstraktes Datenobjekt

realisiert werden, eine Funktion mit entsprechender Abfrage, ob schon eine Instanz existiert, kann man sich somit sparen.

Es sei noch darauf hingewiesen, dass sich im Internet auch einige wenige sehr spezielle Design Patterns finden lassen, die sich speziell auf Ada 95 beziehen (siehe [7]).

4. Fazit

Abschließend kann man festhalten, dass sich die Verwendung und die konsequente Einhaltung eines Style-Guides immer lohnt und auszahlt, besonders in der Zusammenarbeit in einem Team. Auch kommt es bei vielen Regeln nicht auf das genau wie an, sondern nur dass es von allen gleich gemacht wird. Durch den Einsatz von Style-Guides werden vor allem die Lesbarkeit und das Programmverständnis erhöht, was in der Regel eine enorme Zeit- und somit auch Kostenoptimierung mit sich bringt. Werden in der Kombination mit Style-Guides auch Entwurfsmethoden verwendet, so ist dies die Vorraussetzung und der erste Schritt um qualitativ hochwertige, zuverlässige, wiederverwendbare und portable Software zu erhalten.

5. Referenzen

[1] Ada95 Quality and Style, http://www.adaic.org/docs/95style/95style.pdf

[2] J.M. Dautelle, Ada95 Lessons Learned, http://www.adahome.com/articles/1998-02/ar_lessons95.html

[3] Bauhaus Coding Style Guide – DRAFT, http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/technical/bauhaus_coding_style.html

[4] Ralf Reißing, Programmierkurs WS 2000/2001

[5] Stefan Krauß, Richtlinien für die Programmierung in Ada 95 – Version 2.0e

[6] Helmut Balzert, Lehrbuch der Software-Technik – Teil Software-Entwicklung

[7] Design-Patterns in Ada 95, http://www.adapower.com/alg/