Fachhochschule Konstanz
Fachbereich Informatik

Automatische Eintaktung von manuellen Tätigkeiten in der Lackiererei

Rahman Dilsiz

Konstanz, den 28.02.2003

Fachhochschule Konstanz
Fachbereich Informatik

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker (FH)

an der

Fachhochschule Konstanz
Hochschule für Technik, Wirtschaft und Gestaltung

Fachbereich
Informatik/Wirtschaftsinformatik

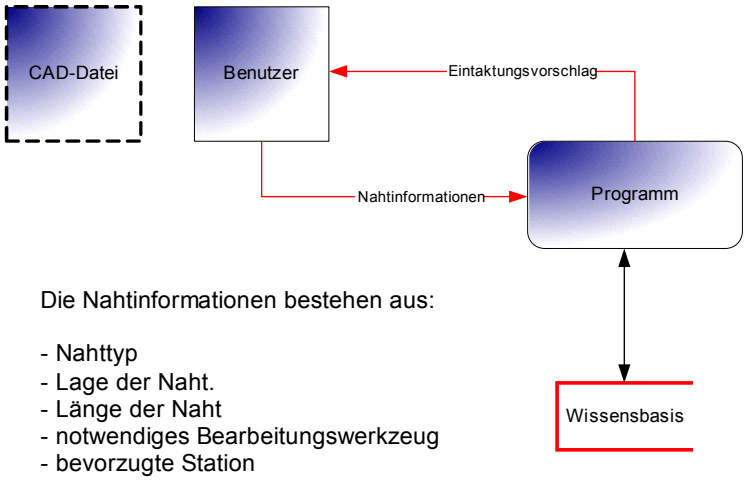
Thema : **Automatische Eintaktung von manuellen Tätigkeiten in der Lackiererei**

Diplomand: Rahman Dilsiz, Ekkehardstr. 88, 78224 Singen

Betreuer : **Professor Dr. Wolfgang Arndt, Herr Dr. Jörg Perschbacher**

Eingereicht : **Konstanz, den 28.02.2003**

ABSTRACT



Die Nahtinformationen bestehen aus:

- Nahttyp
- Lage der Naht.
- Länge der Naht
- notwendiges Bearbeitungswerkzeug
- bevorzugte Station

Die Wissensbasis enthält Informationen über
- die verschiedenen Gruppen
  - Bearbeitungswerkzeuge
  - Bearbeitungshöhe
  - Taktzeit
  - max Anzahl der Werker
  - aktuelle Auslastung der Werker
- die Bearbeitungszeit der Tätigkeiten

Die Wissensbasis ist als Access Datenbank realisiert.

Wenn neue Tätigkeiten in der PVC-Abdichtlinie2 der Lackiererei bearbeitet werden müssen, ist es notwendig die gesamte Produktionslinie zu betrachten, um eine Aussage darüber machen zu können, an welchem Arbeitsplatz diese neuen Tätigkeiten bearbeitet werden können. Die Zuweisung von neuen Tätigkeiten an Arbeitsplätze wird Eintakten genannt.

Ziel der Diplomarbeit, war es ein Konzept für die automatische Eintaktung von manuellen Tätigkeiten an der PVC-Abdichtlinie2 der Lackiererei zu erstellen. Anschließend sollte ein Programm entwickelt werden, womit eine automatische Eintaktung durchgeführt werden kann. Bei der Erstellung dieses Programms sollte darauf geachtet werden, dass das Programm leicht zu handhaben ist. Weiterhin sollte das Programm so gestaltet sein, dass es auch durch die ähnlich aufgebaute Abdichtlinie1 genutzt werden kann.

1 EINFÜHRUNG 1

1.1 AUFBAU DER PRODUKTIONSLINIE „PVC-ABDICHTEN 2“ 1
1.2 TYPISCHE TAKTARBEITSFOLGEN (TAFS) 3
1.2.1 Einfache Tätigkeiten 3
1.2.2 Nahtbezogene Tätigkeiten 4

2 DIE AUTOMATISCHE EINTAKTUNG 5

2.1 BERECHNUNG DER BEARBEITUNGSDAUER 5
2.2 NOTWENDIGE VORAUSSETZUNGEN FÜR DIE AUTOMATISCHE EINTAKTUNG 5
2.2.1 Zeitberechnung in Abhängigkeit von Werkzeug 6
2.2.2 Gliederung der Karosserie in Arbeitsbereiche 8
2.2.3 Hinzufügen von Werkzeuginformationen zu den Taktarbeitsfolgen 10
2.3 DIE BEARBEITUNGSHÖHE DER GRUPPEN 12
2.4 DER ALGORITHMUS ZUR AUTOMATISCHEN EINTAKTUNG 15
2.4.1 Erweiterung der automatischen Eintaktung 20

3 DIE WISSENSBASIS ALS RELATIONALE DATENBANK 22

3.1 RELATIONALE DATENBANK GRUNDLAGEN 22
3.2 AUFBAU DER TABELLEN UND DEREN BEZIEHUNGEN 23
3.3 OPEN DATABASE CONNECTIVITY (ODBC) 27
3.3.1 Die Klasse CRecordset 27

4 AUFBAU DES PROGRAMMS 34

4.1 DAS HAUPTFENSTER 34
4.2 EINGABEFENSTER FÜR NAHTABHÄNGIGE TÄTIGKEIT 36
4.3 EINGABEFENSTER FÜR EINFACHE TÄTIGKEIT 37
4.4 FENSTER FÜR EINTAKTUNG 38
4.5 FENSTER FÜR ADMINISTRATIVE TÄTIGKEITEN 38
4.6 INTELLIGENTES VERSCHIEBEN VON VORHANDENEN ARBEITSSCHRITTEN (TAFS) 41

5 DIE MICROSOFT FOUNDATION CLASSES (MFC) 41

5.1 DIE EREIGNISSBEHANDLUNG 41
5.2 MFC HILFSKLASSEN 42
5.2.1 Behandlung von Listen 43
5.2.2 Behandlung von Koordinaten 54
5.2.3 Behandlung von Zeichenfolgen 62

6 AUSBLICK 68

6.1 SCHNITTSTELLE ZUM CAD-PROGRAMM CATIA 68
6.2 DATENBANKZUGRIFF ÜBER DAS NETZ 69

Vorwort

Bevor ich mit der Beschreibung der Thematik anfange, möchte ich den Vorwort nutzen um mich bei allen zu bedanken, die zur optimalen Lösung der Aufgabe beigetragen haben. Besonders möchte ich mich bei meinen Betreuern bedanken, die diese Diplomarbeit ermöglicht haben. Mein persönliches Ziel war es ein Programm zu entwickeln, welches praxisorientiert ist und flexibel eingesetzt werden kann. Und vor allem nach meiner Diplomarbeit weitergelebt wird.

Rahman Dilsiz Ingolstadt den 20.02.2003

Tabellenverzeichnis

Tabelle 1: Aufbau einer TAF 3
Tabelle 2: Aufbau der Entscheidungstabelle 7
Tabelle 3: Zusatzinformation Bearbeitungsort 10
Tabelle 4: Werkzeugkürzel 11
Tabelle 5: Zusatzinformation Werkzeug(e) 11
Tabelle 6: Beispiel Fremdschlüssel 22
Tabelle 7: Modelle 23
Tabelle 8: Aufbau einer Tabelle für ein Modell 24
Tabelle 9: Konflikt ArbeitsschrittNr 24
Tabelle 10: Platzhalter 25
Tabelle 11: Werkzeuge 25
Tabelle 12: Tätigkeit 26
Tabelle 13: Gruppen 26
Tabelle 14: Typen von Recordsets 29
Tabelle 15: Flags für das öffnen eines Recordsets 29
Tabelle 16: Navigation durch einen Recordset 30
Tabelle 17: Argumentbeschreibung der Funktion Move 31
Tabelle 18: Administrative Funktionen 31
Tabelle 19: Passwortliste 40
Tabelle 20: Vordefinierte Arraybasierte Klassen 44
Tabelle 21: Auflistungsklassen für Listen 46
Tabelle 22: Auflistungsklassen für Tabellen 48
Tabelle 23: Konstruktortypen für die Klasse CPoint 55
Tabelle 24: Konstruktortypen für die Klasse CRect 56
Tabelle 25: Parameterformen für InflateRect und DeflateRect 59
Tabelle 26: Überladene Operatoren für CRect Objekte 60
Tabelle 27: Konstruktortypen für die Klasse CSize 61
Tabelle 28: Überladene Operatoren für CSize 62
Tabelle 29: Mit CString verwendete Konstruktortypen 63
Tabelle 30: Formatcodes für die Funktion Format 67

Abbildungsverzeichnis

Abbildung 1: Organisation der Produktionslinie 1
Abbildung 2: Maximale Anzahl der Werker pro Karosserie 2
Abbildung 3: Bearbeitungshöhen der Gruppen 2
Abbildung 4: Vorbehandlungswerkzeug Spritzpistole 4
Abbildung 5: Nachbehandlungswerkzeug Pinsel 4
Abbildung 6: Abhängigkeiten für die Zeitberechnung 6
Abbildung 7: Berechnung der Bearbeitungszeit 7
Abbildung 8: Aufteilung der Karosserie in Bearbeitungsbereiche 8
Abbildung 9: Regeln für die Berechsaufteilung 9
Abbildung 10: Zusatzfunktion zur Überprüfung der Ortsübereinstimmung 10
Abbildung 11: Überprüfung ob Werkzeug vorhanden 12
Abbildung 12: Anzahl der Berechnungen 13
Abbildung 13: Selektion durch Bearbeitungshöhe 14
Abbildung 14: Algorithmus 15
Abbildung 15: Bevorzugte Gruppe 16
Abbildung 16: Überprüfung der Bearbeitungshöhe 16
Abbildung 17: Überprüfe Werker 17
Abbildung 18: Listenerstellung für Nachbehandlung 19
Abbildung 19: Abbruchbedingung 20
Abbildung 20: Speichern der Vorschläge 21
Abbildung 21: Speicherreservierung durch AddNew 32
Abbildung 22: Füllen des neuen Datensatzes mit Informationen 32
Abbildung 23: Taktzeit Ändern 35
Abbildung 24: Das Hauptfenster 35
Abbildung 25: Fenster für einfache Tätigkeit 37
Abbildung 26: Fenster für Eintaktung 38
Abbildung 27: Fenster für administrative Tätigkeiten 38
Abbildung 28: Betrachtung eines Arbeitsplatzes 39
Abbildung 29: Diagramm zur Darstellung der Auslastungen 39
Abbildung 30: Passworteingabe 40
Abbildung 31: Intelligente Verschiebung 41
Abbildung 32: Ereignisbehandlung 42
Abbildung 33: Vereinigung und Schnitt zweier Rechtecke 58
Abbildung 34: Netzwerk 69

1 EINFÜHRUNG

Die Beförderungsgeschwindigkeit der Karosserien durch die Produktion ist konstant, was die Basis zur Berechnung der Taktzeit darstellt. Die Taktzeit gibt an wie viel Zeit man hat, um eine Karosserie innerhalb von einer Strecke zu bearbeiten. Jeder Werker muss innerhalb der Taktzeit alle ihm zugeordneten Arbeitsschritte durchführen.

Wenn eine neue Tätigkeit in die Produktion aufgenommen wird, ist es notwendig zu überprüfen, welcher Werker noch in der Lage ist diese neue Tätigkeit durchzuführen. Hierbei ist in den meisten Fällen der zeitliche Faktor nicht das einzige was berücksichtigt werden muss. Das notwendige Werkzeug für die Durchführung der speziellen Tätigkeit muss dem Werker auch zur Verfügung stehen. Das Hinzufügen einer neuen Tätigkeit in die Produktion wird „Eintakten“ genannt.

Derzeit wird eine Eintaktung in der Produktionslinie „PVC-Abdichten 2“ von Hand berechnet. Die Berechnung der Eintaktung von Hand ist sehr langwierig und kostspielig, daher sollte im Rahmen dieser Diplomarbeit ein Programm entwickelt werden, welches die Eintaktung automatisch berechnet. Das Programm soll zusätzlich eine Hilfe bei Umstrukturierungen an der Produktionslinie sein. Weiterhin ist das Programm so zu gestalten, dass es für die ähnlich aufgebaute Produktionslinie „PVC-Abdichten 1“ genutzt werden kann.

1.1 AUFBAU DER PRODUKTIONSLINIE „PVC-ABDICHTEN 2“

Die „Abdichtlinie 2“ ist in verschiedene Gruppen unterteilt. Jede Gruppe hat ein spezielles Bearbeitungsziel. Weiterhin ist jede Gruppe je nach Bearbeitungszweck mit Werkzeugen und Bearbeitungsmitteln ausgestattet.

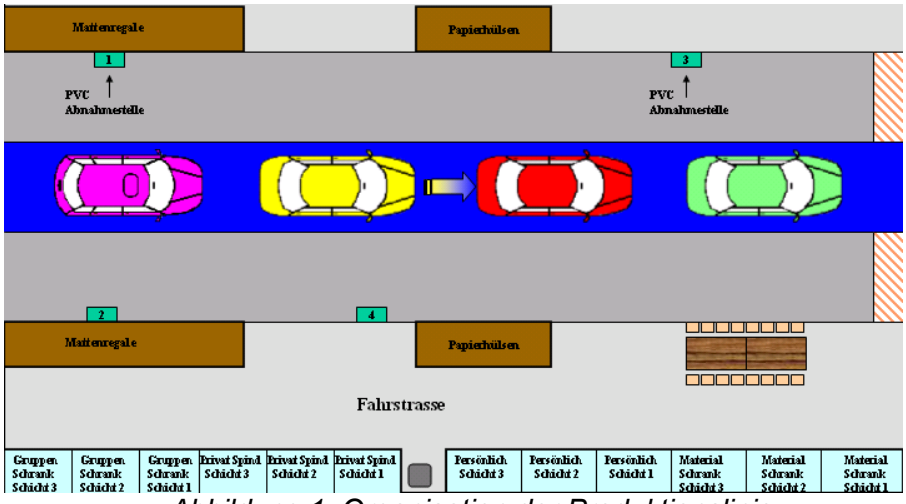


Abbildung 1: Organisation der Produktionslinie

Die Anzahl der Werker in einer Gruppe ist abhängig von der Ausdehnung der Gruppe. Diese Abgrenzung der Werkeranzahl ist notwendig, da bei einer Überbesetzung der Gruppen die Werker sich gegenseitig an der Arbeit behindern würden. An einer Karosserie sind Sinnvollerweise maximal 6 Werker gleichzeitig beschäftigt. Ist eine weitere Bearbeitung der Karosserie innerhalb der Gruppe notwendig, so muss die Gruppe dementsprechend gestreckt gestaltet sein. Die Anordnung der Werker an einer Karosserie ist in Abbildung 2 dargestellt.

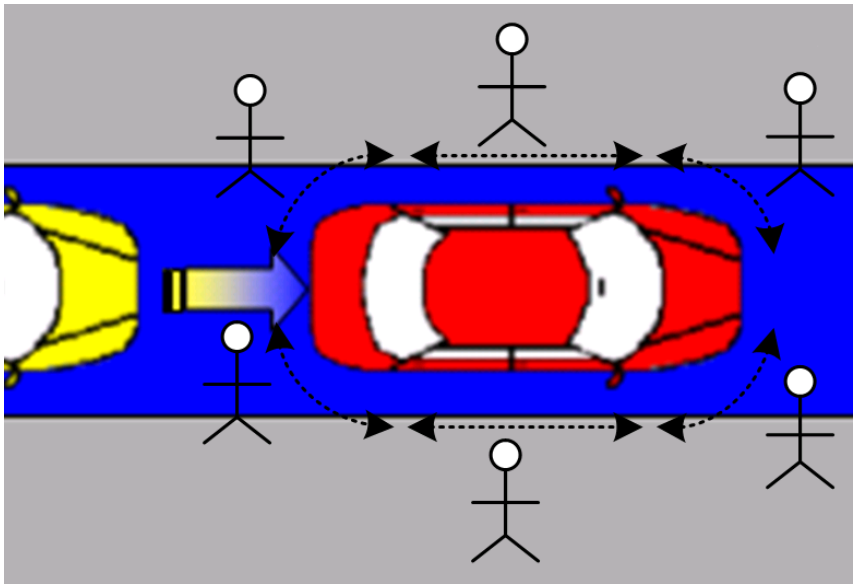


Abbildung 2: Maximale Anzahl der Werker pro Karosserie

Da jede Gruppe ein spezielles Bearbeitungsziel hat, ist es möglich für jede Gruppe eine gültige Bearbeitungshöhe anzugeben. Diese Eigenschaft ist für die automatische Eintaktung eine große Hilfe, wie später noch genauer beschrieben wird.

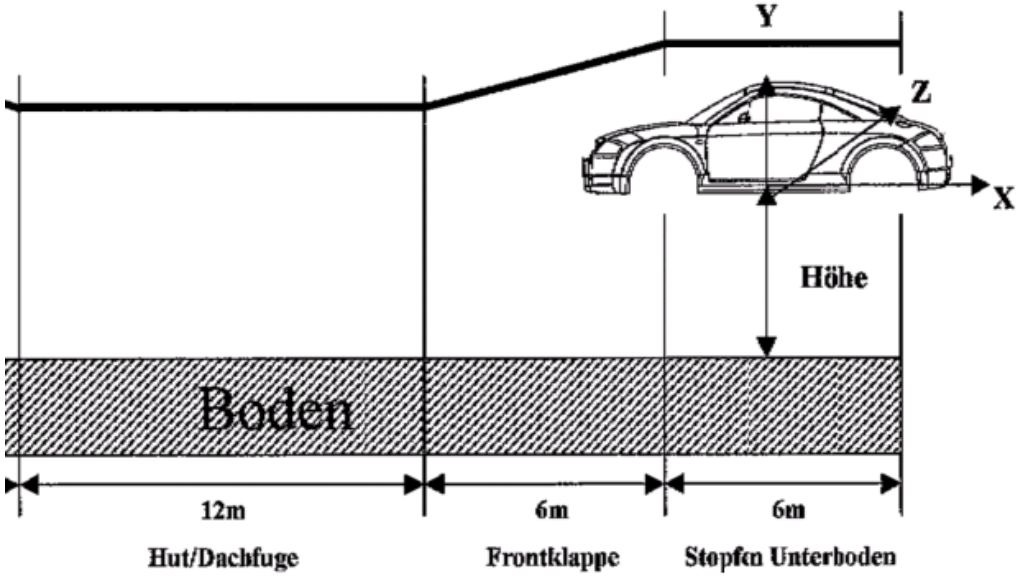


Abbildung 3: Bearbeitungshöhen der Gruppen

1.2 TYPISCHE TAKTARBEITSFOLGEN (TAFS)

Jeder Arbeitsschritt eines Werkers wird im Audi-Internen System „TAKSI“ festgehalten. Um dabei jeden Arbeitsschritt voneinander unterscheiden zu können, gibt es einen Index, welcher für jeden Arbeitsschritt einmalig ist und genau einen Arbeitsschritt kennzeichnet.

Dieser Index wird Taktarbeitsfolge, kurz TAF genannt. Anhand eines TAF´s kann man herausfinden in welchem **P**rodukt **D**etail **M**ontageanweisungsblatt (PDM-Blatt) die Bearbeitungsstelle beschrieben wird. In einem PDM-Blatt ist bildlich die Bearbeitungsstelle aufgeführt. Weiterhin ist es möglich über die TAF die Bearbeitungszeit eines Arbeitsschritts zu erhalten.

Wenn eine neue Tätigkeit in die Produktion aufgenommen wird, muss für diese Tätigkeit eine neue TAF erstellt werden. Um Mehrdeutigkeiten zu vermeiden, werden die TAF´s zentral vergeben.

TAF	Beschreibung	PDM-Blatt	Position	Zeit
5210 A A	**Weg zur Karosse**			0,0777
5210 A B	Frontklappenhalter demontieren			0,1609
5210 A C	Frontklappenhalter abwerfen/holen			0,0555
5210 A D	Einstellpuffer montieren			0,0804
5210 A M	Klebepunkt(Klarsicht-)	256	20a	0,0472

Tabelle 1: Aufbau einer TAF

Ein PDM-Blatt enthält in der Regel die Bearbeitungsorte von mehreren Tätigkeiten, um dabei dennoch den Überblick zu behalten, gibt es zusätzlich die Angabe der Position.

Wenn man die manuellen Tätigkeiten, welche in der „Abdichtlinie 2“ vorhanden sind genauer Betrachtet, kann man Sie in zwei Gruppen unterteilen.

1.2.1 EINFACHE TÄTIGKEITEN

Die Durchführung dieser Tätigkeiten geschieht in der Regel ohne Werkzeug.
Für die Berechnung der Bearbeitungsdauer dieser Tätigkeiten ist es nicht erforderlich spezielle Werkzeug- oder Nahtinformationen zu kennen. Tätigkeiten wie z. B. „Stopfen eindrücken“, nehmen durchgehend eine konstante Bearbeitungszeit in Anspruch. Es sei den der Bearbeitungsort ist schwer zugänglich.

1.2.2 NAHTBEZOGENE TÄTIGKEITEN

Nahtbezogene Tätigkeiten müssen mit mindestens einem Werkzeug behandelt werden. Die erste Behandlung einer Naht wird immer mit einer Pistole durchgeführt, womit man über die Naht eine schützende PVC Schicht legt. Je nachdem wie lang die Naht ist, und welches Werkzeug benutzt wird, variieren hier die Bearbeitungszeiten.

Abbildung 4: Vorbehandlungswerkzeug Spritzpistole

Außer bei dem Einsatz einer „Sprühpistole“, muss jede Naht die gelegt wird, mit einem Verstreichwerkzeug nachbehandelt werden. Wobei es verschiedene Werkzeuge zum Verstreichen gibt. Auch hier variieren die Bearbeitungszeiten in Abhängigkeit von Werkzeug und Nahtlänge.

Abbildung 5: Nachbehandlungswerkzeug Pinsel

Bei der Eintaktung einer neuen Naht, entstehen meistens zwei neue TAF´s. Eine für die Vorbehandlung und falls nötig eine für die Nachbehandlung. Wodurch auch zwei neue Zeiten entstehen die berücksichtigt werden müssen.

Diese Abhängigkeit zwischen Vor- und Nachbehandlung muss in der automatischen Eintaktung berücksichtigt werden. Wie diese Abhängigkeit gehandhabt wird, wird später genauer beschrieben. Weiterhin ist zu beachten dass die Spritzpistolen fest an den Orten gebunden sind, an der Sie eingesetzt werden.

2 DIE AUTOMATISCHE EINTAKTUNG

Nachdem durch die Einführung die Umwelt und die Problematik genauer erklärt ist, muss noch dargestellt werden welche Vorarbeit notwendig ist, um die Eintaktung automatisieren zu können. Hierzu sind folgende Punkte von Bedeutung.

- Berechnung der Bearbeitungsdauer
- Unterteilung der Karosserie in Bearbeitungsbereiche
- Zuweisung von Bearbeitungsbereichen an die TAFs

2.1 BERECHNUNG DER BEARBEITUNGSDAUER

Wie die Zeitwirtschaft der Audi die Berechnung der Bearbeitungszeit handhabt wird hier aus Geheimhaltungsgründen nicht genau beschrieben. Aber das Prinzip wird kurz erläutert.

Es gibt Tabellen indem für alle typischen Handbewegungen, die in der Montage vorkommen, spezielle Konstanten hinterlegt sind. Um die Bearbeitungszeit eines neuen Arbeitsschritts zu erhalten, muss man alle Handbewegungen die in dem Arbeitsschritt vorkommen, berücksichtigen.

Hat man jede Handbewegung die in dem neuen Arbeitsschritt vorkommen aus den Tabellen herausgefiltert, kann man die Bearbeitungszeit errechnen, indem man alle Konstanten der Handbewegungen addiert. Die Summe ist die Konstante, welche die Bearbeitungszeit des gesamten Arbeitsschritts repräsentiert. Nachdem man diese Konstante mit produktionsspezifischen Faktoren multipliziert, erhält man die Bearbeitungszeit in Minuten.

2.2 NOTWENDIGE VORAUSSETZUNGEN FÜR DIE AUTOMATISCHE EINTAKTUNG

Bei einfachen Tätigkeiten reicht die Erfüllung der Grundvoraussetzung aus, um eine Aussage darüber machen zu können, ob eine Eintaktung möglich ist. Die Grundvoraussetzung ist in Worten wie folgt definiert.

$$\text{Taktzeit} \geq \text{Gesamtbearbeitungszeit des Werkers} + \text{neue Zeit}$$

Formel 1: Grundvoraussetzung für gültige Eintaktung

Da die automatische Eintaktung der einfachen Tätigkeiten kein Problem darstellt, werden Sie im Folgenden nur kurz angesprochen. Das Hauptproblem stellt die automatische Eintaktung der „nahtbezogenen Tätigkeiten“ dar.

Die Voraussetzungen für eine gültige Eintaktung der „nahtbezogenen Tätigkeiten“ sind komplexer. Hier müssen mehrere Faktoren gleichzeitig betrachtet werden. Zu den Grundvoraussetzungen müssen hier zusätzlich folgende Faktoren berücksichtigt werden.

- Bearbeitungszeit wird in Bezug zu einem Werkzeug berechnet
  - Eventuell zusätzliche Zeit für Nachbehandlung (weiteres Werkzeug)
- Die Lage der Naht muss mit dem Bearbeitungsbereich des Werkers übereinstimmen
- Werker muss das notwendige Werkzeug zur Verfügung haben

2.2.1 Zeitberechnung in Abhängigkeit von Werkzeug

Da man für die Durchführung der „einfachen Tätigkeiten“ keine Werkzeuge braucht, sind diese leicht zu berechnen. Hierzu hinterlegt man für eine typische Tätigkeit wie z.B. „Stopfen eindrücken“ eine speziell zusammengesetzte Konstante. Anhand dieser Konstante berechnet man wie oben erwähnt die Bearbeitungszeit.

Die Bearbeitungszeit einer „nahtbezogenen Tätigkeit“ stellt keinen konstanten Bewegungsablauf dar, und ist abhängig von der Nahtlänge und der Bearbeitungsgeschwindigkeit des Werkzeugs welches genutzt wird.

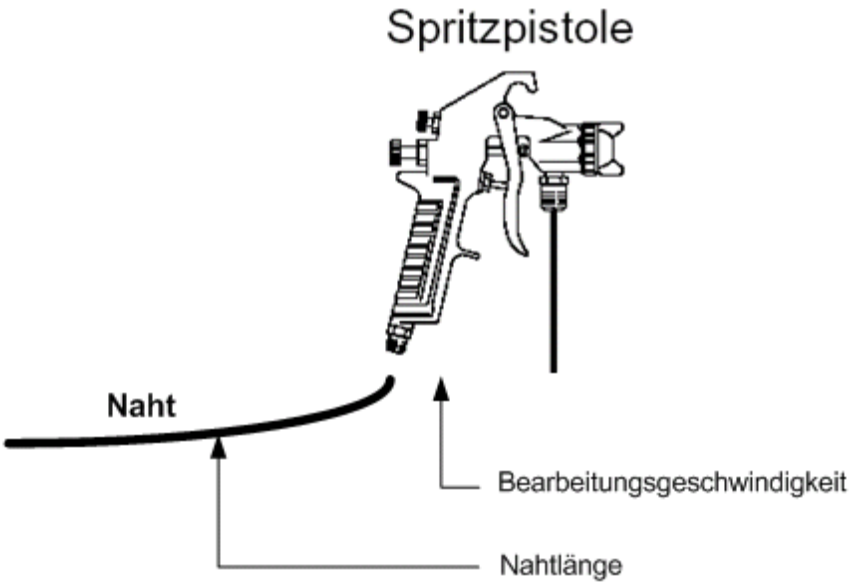


Abbildung 6: Abhängigkeiten für die Zeitberechnung

Hinzu kommt dass die Werkzeuge für die Vorbehandlung von Nähten, über einen Hochdruckschlauch an so genannte Druckregler verbunden sind, und somit an bestimmte Bereiche Ortsgebunden sind.

Um die Abhängigkeiten zu dem Werkzeug und zu der Nahtlänge im Programm berücksichtigen zu können, wird folgende Tabelle erstellt. Diese Tabelle soll das Programm als Entscheidungstabelle nutzen können.

	Werkzeuge		
Bezeichnung	**< 200 mm**	**> 200 mm < 500 mm**	**> 500 mm < 800 mm**
Flachpinsel	45	55	70
Grobnahtrohr	5	5	7
Heizkörperpinsel	7	17	3
Kreuzdüse	5	6	7
Schaber	14	25	37
Spitzdüse	5	6	7
Schlitzdüse	5	7	5
Sprühpistole	7	10	15
Winkeldüse	5	5	6
Wischtuch	14	55	30

Tabelle 2: Aufbau der Entscheidungstabelle

Die Umrechnungskonstanten sind für jedes Werkzeug in Abhängigkeit von der Nahtlänge hinterlegt. Hierbei ist die Nahtlänge in drei sinnvolle Bereiche unterteilt. Diese Entscheidungstabelle ermöglicht eine Zeitberechnung für alle „nahtbezogenen Tätigkeiten“, da auch die Nachbehandlungswerkzeuge in dieser Entscheidungstabelle enthalten sind. Diese Entscheidungstabelle wird seitens des Programms wie folgt genutzt:

- Selektion des notwendigen Werkzeugs (Zeile)
- Selektion des Bereichs der Nahtlänge (Spalte)
- Berechnung der Bearbeitungszeit anhand der selektierten Konstante

Durch die Entscheidungstabelle ist es nun möglich die Berechnung der Bearbeitungszeit automatisch durchzuführen. Als Eingabeinformation braucht man lediglich die Nahtlänge und das Bearbeitungswerkzeug.

Abbildung 7: Berechnung der Bearbeitungszeit

Nachdem die Bearbeitungszeit berechnet ist, ist es möglich zu überprüfen welche Werker in der Lage sind, diese neue Zeit aufzunehmen. Es ist nun möglich die Grundvoraussetzung für eine gültige Eintaktung zu überprüfen.

Wie schon erwähnt muss für eine nahtbezogene Tätigkeit noch mehr berücksichtigt werden, um eine Aussage bezüglich der Eintaktungsmöglichkeit erhalten zu können.

2.2.2 Gliederung der Karosserie in Arbeitsbereiche

Neben der Berechnung der Bearbeitungszeit ist es auch notwendig zu überprüfen welcher Werker diese Tätigkeit ausüben kann. Um dies überprüfen zu können muss man zu jedem Arbeitsschritt der Werker wissen, welchen Bereich an der Karosserie der Werker von seiner aktuellen Position aus bearbeiten kann.

Liegt der Bearbeitungsort der neuen Tätigkeit in dem Bearbeitungsradius eines Werkers, so kommt dieser Werker für eine Bearbeitung in Frage. Um diese Überprüfung zu ermöglichen, muss der aktuelle Bearbeitungsbereich zu jedem Arbeitsschritt als Zusatzinformation hinterlegt werden.

Hierzu braucht man eine sinnvolle Gliederung der Karosserie in typische Bearbeitungsbereiche. Wie Abbildung 8 zeigt, wurde die Karosserie in der X-, Y- und Z-Achse in spezielle Bearbeitungsbereiche aufgeteilt.

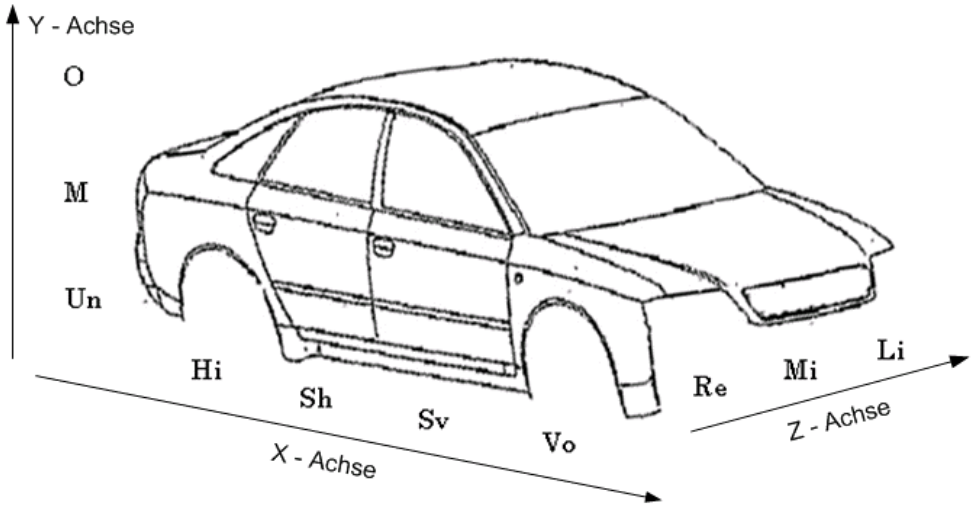


Abbildung 8: Aufteilung der Karosserie in Bearbeitungsbereiche

Die in der Abbildung 8 erwähnten Abkürzungen sind wie folgt zu verstehen:

- X-Achse
  - Hi = Hinten
  - Sh = Seite hinten
  - Sv = Seite vorne
  - Vo = Vorne

- Y-Achse
  - Un = Unten
  - M = Mitte
  - O = Oben

- Z-Achse
  - Re = Rechts
  - Mi = Mitte
  - Li = Links

Diese Kürzel werden in dem Programm genutzt, wobei eine Kombination zwischen den verschiedenen Bereichen möglich ist. Dies ist notwendig, wenn z.B. eine Naht im vorderen Bereich auf der mittleren Höhe auf der rechten Seite beginnt und bis zu der Mitte reicht.

Damit das Programm solche Bereichsinformationen richtig verarbeiten kann müssen Sie einheitlich nach bestimmten Regeln abgespeichert werden. Diese einheitliche Bereichsangabe wird zu jedem Arbeitsschritt hinzugefügt, wodurch der aktuelle Bearbeitungsradius zu jedem Arbeitsschritt angegeben wird. Weiterhin wird die Lage einer neuen Tätigkeit nach denselben Regeln angegeben. Diese Regeln der Bereichsangabe sind wie folgt realisiert.

- Ortsinformationen der selben Achse werden mit Bindestrich verbunden

- Unterschiedliche Achsen werden durch Komma getrennt

- Die Reihenfolge der Achsen ist wie folgt organisiert
  (1) Die X-Achse ist von vorne nach hinten organisiert
  (2) Die Y-Achse ist von oben nach unten organisiert
  (3) Die Z-Achse ist von rechts nach links organisiert

Würde eine Naht im Vorderen Bereich in der mittleren Höhe liegen und dort von der rechten Seit bis zur Mitte reichen, so wird diese Information wie folgt abgespeichert.

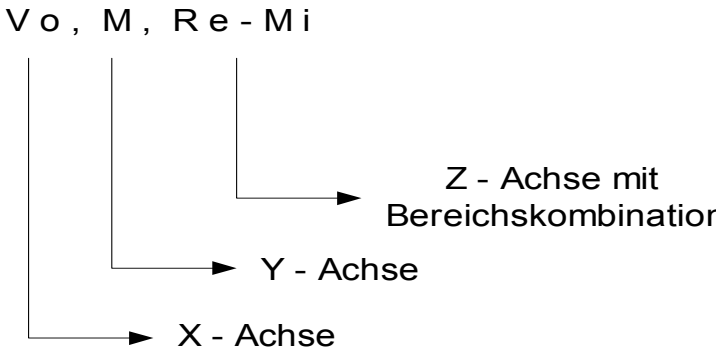


Abbildung 9: Regeln für die Berechsaufteilung

2.2.3 Hinzufügen von Werkzeuginformationen zu den Taktarbeitsfolgen

Es ist nun möglich automatisch zu überprüfen ob die Lage einer Naht in den Bearbeitungsradius eines Werkers fällt und ob die Zeit, welche für die Bearbeitung der neuen Naht notwendig ist, von einem Werker getragen werden kann.

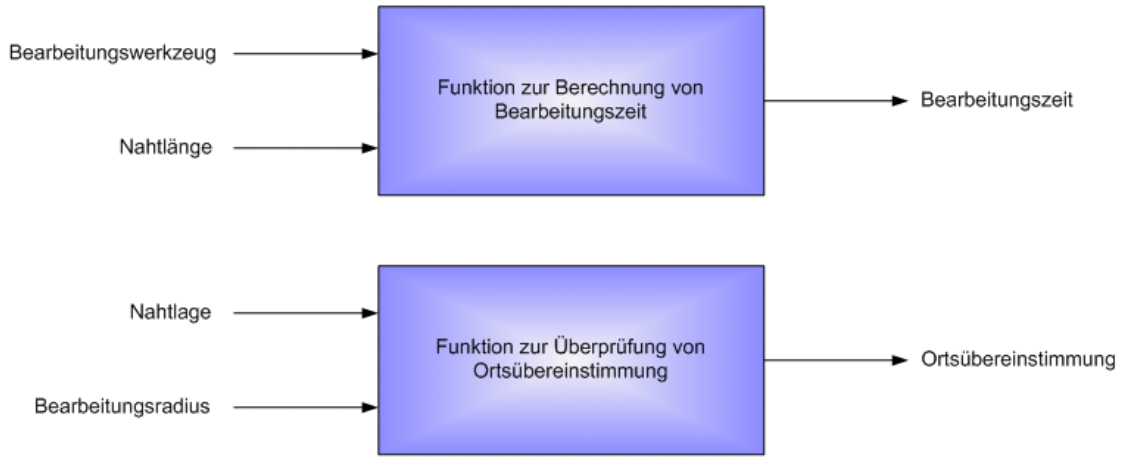


Abbildung 10:Zusatzfunktion zur Überprüfung der Ortsübereinstimmung

Anhand des vorgegebenen Werkzeugs und der Nahtlänge kann man die notwendige Bearbeitungszeit errechnen. Weiterhin kann überprüft werden ob die Lage der Naht in dem Bearbeitungsbereich eines Werkers liegt. Hierzu wird die Tabelle der TAF wie folgt erweitert.

TAF			Beschreibung	Bearbeitungsradius	PDM-Blatt	Position	Zeit
5210	**A**	**A**	**Weg zur Karosse**				0,0777
5210	**A**	**B**	Frontklappenhalter demontieren	Vo,M,Li			0,1609
5210	**A**	**C**	Frontklappenhalter abwerfen/holen	Vo,M,Re-Mi			0,0555
5210	**A**	**D**	Einstellpuffer montieren	Vo,M,Mi			0,0804
5210	**A**	**M**	Klebepunkt(Klarsicht-)	Vo,M,Li	256	20a	0,0472

Tabelle 3:Zusatzinformation Bearbeitungsort

Um eine automatische Aussage darüber zu erhalten, ob eine neue Naht durch einen speziellen Werker bearbeitet werden kann, fehlt noch die Angabe über das Werkzeug welches ein Werker zur Verfügung hat. Hierzu muss noch zu jedem Arbeitsschritt hinterlegt werden, welches Werkzeug der Werker zur Hand hat. Handelt es sich um das vorgegebene Werkzeug, so kommt der Werker für eine Eintaktung in Frage.

Damit die Werkzeuginformationen einheitlich abgespeichert werden, ist es notwendig eine Tabelle für die Werkzeuge zu erstellen, worin zu jedem Werkzeug ein Kürzel hinterlegt ist. Dieser Kürzel wird als weitere Zusatzinformation zu jedem Arbeitsschritt angegeben, und kennzeichnet im gesamten System immer dasselbe Werkzeug.

Werkzeuge	
Werkzeugkürzel	**Bezeichnung**
FP	Flachpinsel
GNR	Grobnahtrohr
HKP	Heizkörperpinsel
KD	Kreuzdüse
SCH	Schaber
SPD	Spitzdüse
SD10	Schlitzdüse 10mm
SD15	Schlitzdüse 15mm
SD5	Schlitzdüse 5mm
SD8	Schlitzdüse 8mm
SPP	Sprühpistole
WD6	Winkeldüse 6mm
WT	Wischtuch

Tabelle 4: Werkzeugkürzel

Nun sind alle notwendigen Voraussetzungen für eine automatische Eintaktung gegeben. Eine TAF ist jetzt wie folgende Tabelle zeigt aufgebaut.

TAF			Beschreibung	Bearbeitungsradius	Werkzeug	PDM-Blatt	Pos.	Zeit
5210	**A**	**A**	**Weg zur Karosse**					0,0777
5210	**A**	**B**	Frontklappenhalter demontieren	Vo,M,Li	KD-FP			0,1609
5210	**A**	**C**	Frontklappenhalter abwerfen/holen	Vo,M,Re-Mi	KD-FP			0,0555
5210	**A**	**D**	Einstellpuffer montieren	Vo,M,Mi	KD-FP			0,0804
5210	**A**	**M**	Klebepunkt(Klarsicht-)	Vo,M,Li	KD-FP	256	20a	0,0472

Tabelle 5: Zusatzinformation Werkzeug(e)

Falls ein Werker mehrere Werkzeuge gleichzeitig zur Hand hat, kann man diese auch kombinieren. Um keine Fehlfunktion herbeizurufen muss bei der Kürzelvergabe folgende Regel beachtet werden.

- Es darf kein Kürzel ein Teil eines anderen Kürzels darstellen.
  - Falsch: **SD** für Spitzdüse UND **SD**5 für Spitzdüse 5mm
  - Richtig: SPD für Spitzdüse UND SD5 für Spitzdüse 5mm

Diese Regel ist notwendig, da das Programm bei der Kontrolle ob das Werkzeug an einem Arbeitsschritt vorhanden ist, wie folgt vorgeht:

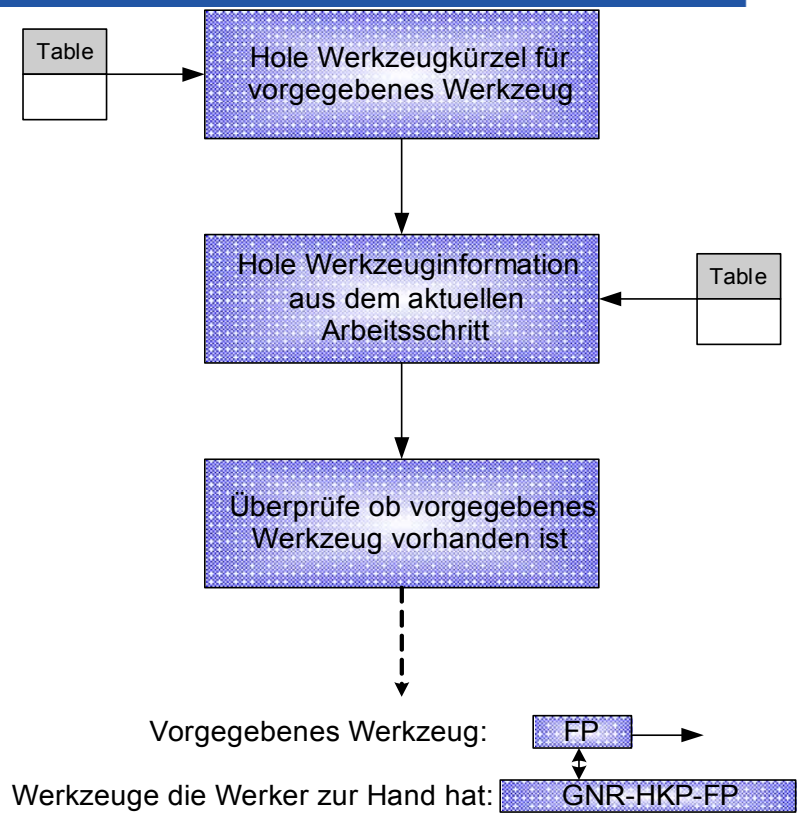


Abbildung 11:Überprüfung ob Werkzeug vorhanden

Es wird überprüft ob die Zeichenfolge für das vorgegebene Werkzeug in der Zeichenfolge der Werkzeuge, die der Werker zur Hand hat vorkommt. Es spielt keine Rolle welches Zeichen als Trennungszeichen genutzt wird.

Durch das Hinzufügen der genannten Zusatzinformationen an die Taktarbeitsfolgen ist es möglich eine automatische Eintaktung für nahtbezogene Tätigkeiten durchzuführen. Eine gültige Eintaktung dieser Tätigkeiten ist gegeben wenn folgende Bedingungen gleichzeitig erfüllt sind.

- Bearbeitungszeit kann von einem Werker getragen werden

&

- Dieser Werker bearbeitet den notwendigen Bearbeitungsbereich

&

- Dieser Werker hat das notwendige Werkzeug zur Hand

Sind diese Bedingungen gleichzeitig erfüllt, so kommt dieser Werker für eine Eintaktung in Frage.

2.3 Die Bearbeitungshöhe der Gruppen

Dass es möglich ist für jede Gruppe eine spezielle Bearbeitungshöhe zu hinterlegen, wurde bereits angesprochen. Weiterhin wurde bereits angesprochen welche Vorarbeit notwendig ist um eine Eintaktung automatisieren zu können. Die automatische Eintaktung wird grob in folgenden Schritten durchgeführt.

- Berechnung der Bearbeitungszeit für die neue Tätigkeit
- Überprüfen ob ein Werker mit dieser neuen Zeit belastet werden kann
- Überprüfen ob dieser Werker diese Tätigkeit durchführen kann
  - Liegt der Bearbeitungsort im Bearbeitungsradius des Werkers
  - Ist das notwendige Werkzeug vorhanden

Die Überprüfung ob ein Werker mit der neuen Zeit belastet werden kann, wird für jeden Werker einmal durchgeführt. Wobei die Überprüfung ob die Lage der neuen Tätigkeit in dem Bearbeitungsradius liegt und ob dabei das notwendige Werkzeug vorhanden ist, für jeden Arbeitsschritt durchgeführt wird.

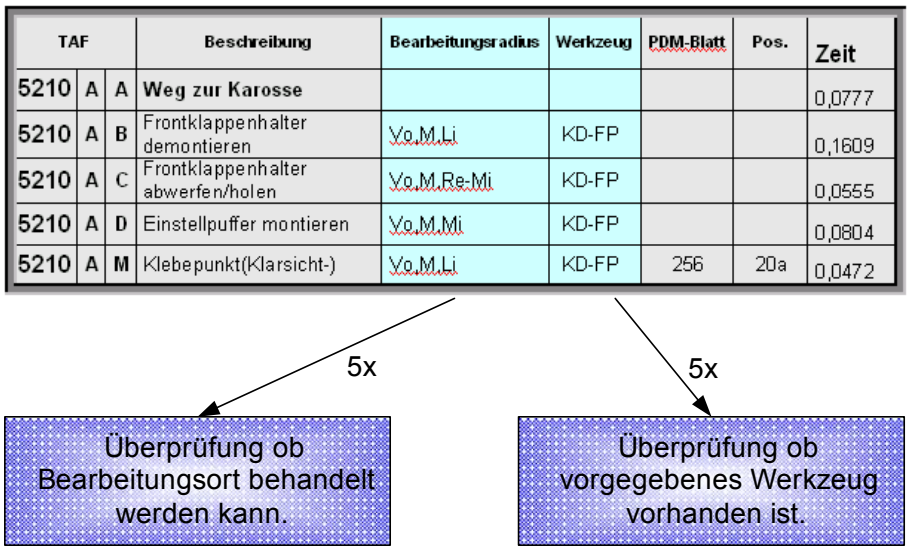


Abbildung 12: Anzahl der Berechnungen

Bei dem Umfang der Produktionslinie kann diese Vorgehensweise zu langen Berechnungszeiten führen. Hier kann man die Eigenschaft der Bearbeitungshöhe der Gruppen nutzen um den Algorithmus zu optimieren.

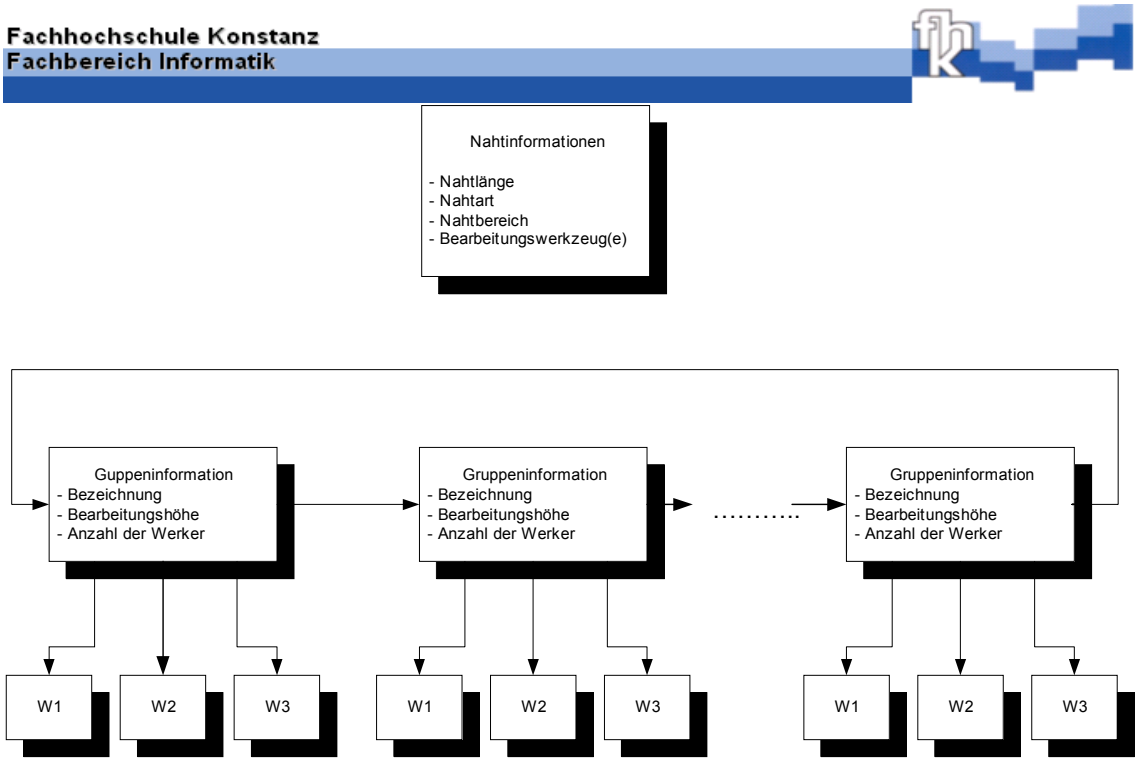


Abbildung 13: Selektion durch Bearbeitungshöhe

Ermöglicht man eine Überprüfung, ob die Bearbeitungshöhe einer Gruppe mit der Höheninformation (Y-Achse) der neuen Tätigkeit übereinstimmt, so kann man alle Werker einer Gruppe ausschließen, bei denen keine Übereinstimmung gegeben ist. Es werden nur die Werker berücksichtigt, dessen Gruppe durch die Höheninformation in Frage kommt. Um diese Überprüfung zu ermöglichen ist es lediglich notwendig für jede Gruppe die Bearbeitungshöhe zu verwalten.

Durch die Nutzung dieser Tatsache, kann man nun die automatische Eintaktung wie folgt realisieren:

- Berechnung der Bearbeitungszeit für die neue Tätigkeit
- Überprüfen ob die Bearbeitungshöhe der Gruppe passt
- Überprüfen ob ein Werker mit dieser neuen Zeit belastet werden kann
- Überprüfen ob dieser Werker diese Tätigkeit durchführen kann
  - Liegt der Bearbeitungsort im Bearbeitungsradius des Werkers
  - Ist das notwendige Werkzeug vorhanden

Wobei alle Arbeitsschritte der Werker, die in Frage kommen zweimal durchlaufen werden. Der erste Durchlauf ist für die Berechnung der aktuellen Auslastung des Werkers notwendig. Nach dem ersten Durchlauf kann erst überprüft werden ob die Bearbeitungszeit getragen werden kann. Ist dies gegeben, so werden alle Arbeitsschritte nochmals durchlaufen mit den in Abbildung 12 gezeigten Überprüfungen.

2.4 DER ALGORITHMUS ZUR AUTOMATISCHEN EINTAKTUNG

Da jede Gruppe vorzugsweise eine bestimmte Art von Bearbeitung durchführt, soll man die Möglichkeit haben dem Programm mitzuteilen, dass die neue Tätigkeit ab einer bestimmten Gruppe Eingetaktet werden soll.

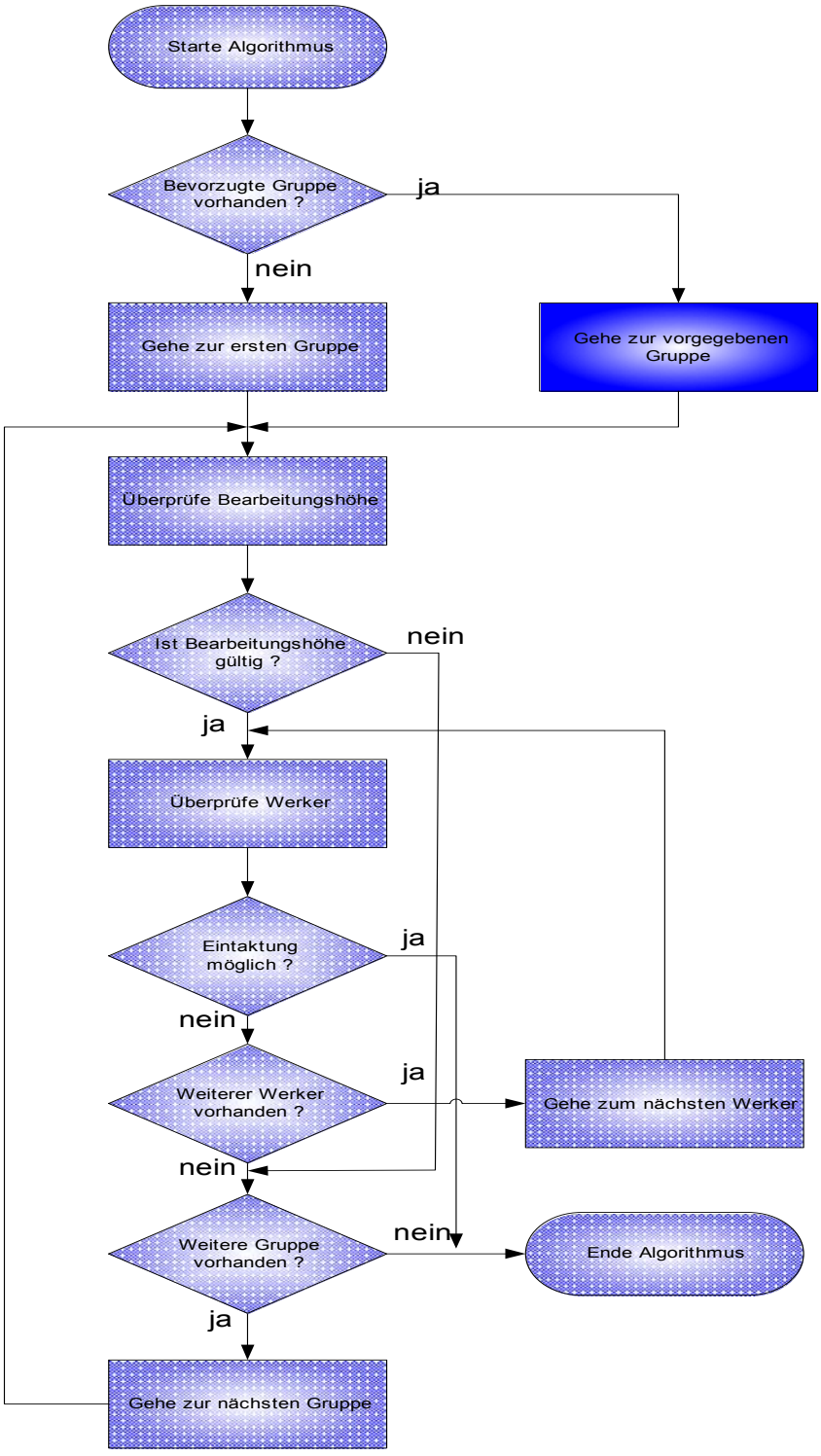


Abbildung 14: Algorithmus

Das Programm soll versuchen die neue Tätigkeit ab der vorgegebenen Gruppe einzutakten. Ist keine bevorzugte Gruppe angegeben, so wird von der ersten Gruppe an versucht Einzutakten.

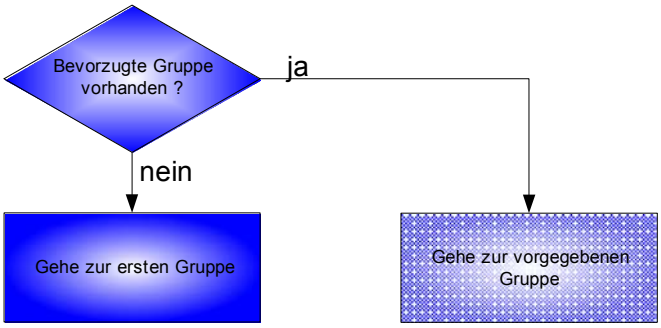


Abbildung 15: Bevorzugte Gruppe

Anschließend wird die Bearbeitungshöhe der aktuellen Gruppe überprüft. Deckt sich diese Bearbeitungshöhe mit der Höhe des Bearbeitungsortes der neuen Tätigkeit, so werden die Arbeitsplätze dieser Gruppe berücksichtigt. Andernfalls wird die Bearbeitungshöhe der nächsten Gruppe überprüft.

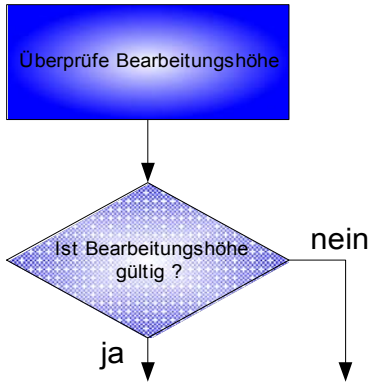


Abbildung 16: Überprüfung der Bearbeitungshöhe

Falls die Bearbeitungshöhe gültig ist, wird der erste Arbeitsplatz dieser Gruppe nach einer möglichen Eintaktung überprüft. Welche Vorarbeit notwendig ist um diese Überprüfung realisieren zu können, wurde bereits beschrieben. Weiterhin wurde grob beschrieben, wie diese Überprüfung durchgeführt wird.

Wie die genaue Überprüfung, ob eine Eintaktung möglich ist, realisiert ist und wie eine gültige Eintaktung gehandhabt wird, ist im Folgenden genau beschrieben. Bisher wurde nur der Fall für die Eintaktung einer einzigen Zeit betrachtet. Ist aber eine Nachbehandlung erwünscht, so muss man zwei Bearbeitungszeiten berechnen und auch beide Zeiten bei der Eintaktung berücksichtigen. Falls dabei nur die Zeit der Vorbehandlung getragen werden kann muss die zugehörige Nachbehandlung speziell errechnet werden.

Ob ein Werker als Eintaktungsort in Frage kommt, und wie die Nachbehandlung berücksichtigt wird, ist in der folgenden Abbildung beschrieben.

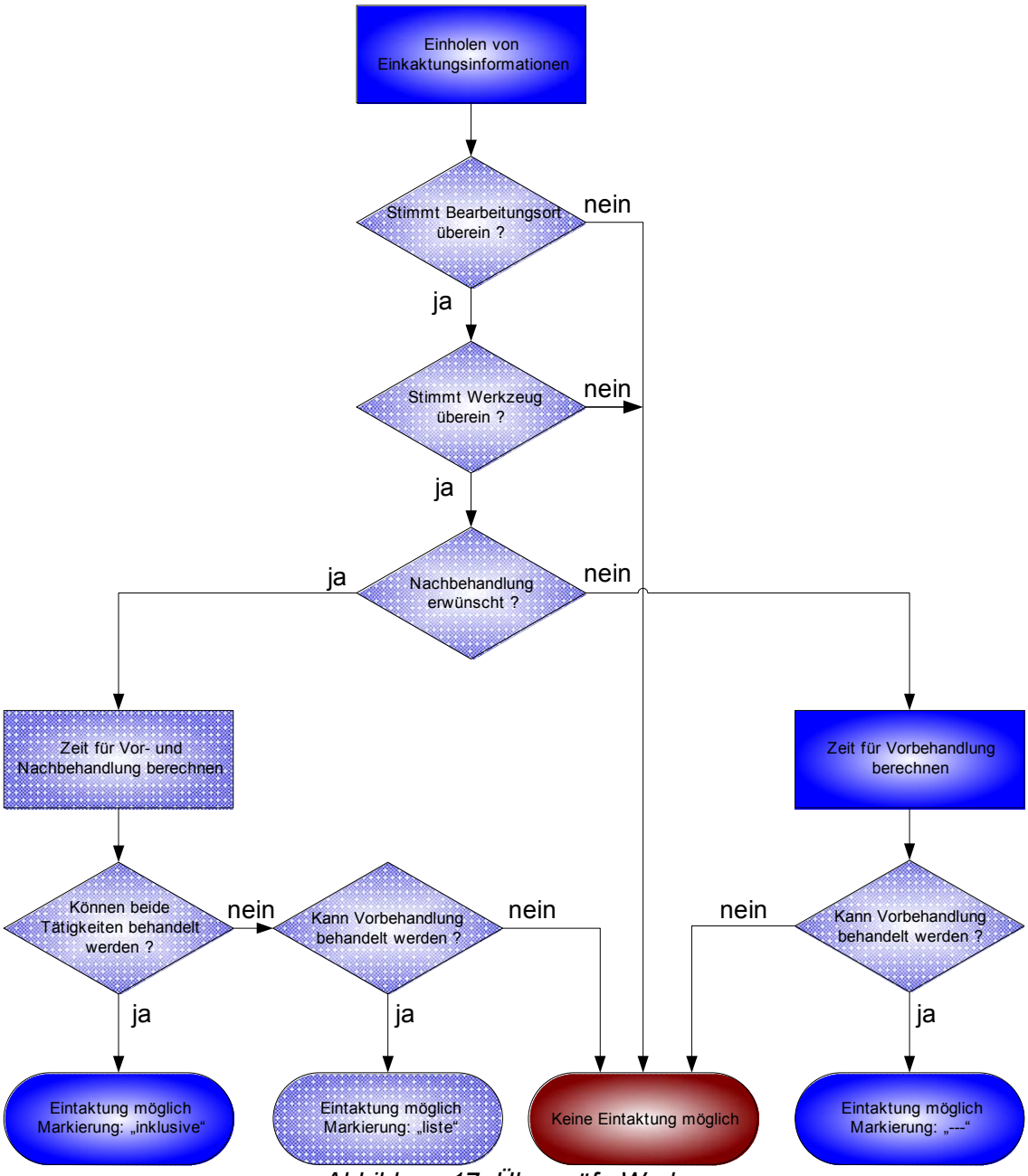


Abbildung 17: Überprüfe Werker

Bei genauer Betrachtung dieser Abbildung fällt auf, dass zuerst die Parameter der Lage und des Werkzeugs überprüft werden. Erst wenn diese Parameter übereinstimmen, wird überprüft ob die jeweilige Tätigkeit bezüglich der Zeit bearbeitet werden kann.

In Kapitel 2.3 wurde erwähnt, dass alle Werker die durch die Gruppenhöhe in frage kommen zweimal durchlaufen werden müssen, um überprüfen zu können ob eine Eintaktung möglich ist. Der erste Durchlauf ist notwendig um die aktuelle Auslastung des Werkers zu berechnen. Anschließend muss der Werker

ein weiteres Mal durchlaufen werden, wobei die Parameter der Lage und des Werkzeugs überprüft werden. Hierbei ist ein zweiter Durchlauf nur notwendig, wenn ein Werker mit der neuen Bearbeitungszeit belastet werden kann.

Dadurch dass nun die Parameter der Lage und des Werkzeugs zuerst überprüft werden, muss der Werker nur ein zweites Mal durchlaufen werden, wenn die neue Naht bezüglich der Lage und des Werkzeugs bearbeitet werden kann. Kommt ein Werker bezüglich des Bearbeitungsbereichs und der Werkzeuge die vorhanden sind nicht für die Behandlung einer neuen Naht in Frage, so muss die aktuelle Auslastung des Werkers nicht berechnet werden.

Zusammenfassend kann man aussagen:

- Überprüft man die Auslastung zuerst, so kann man einen zweiten Durchlauf sparen, wenn die neue Zeit nicht behandelt werden kann.

- Überprüft man die Parameter der Lage und des Werkzeugs zuerst, so kann man einen zweiten Durchlauf sparen, wenn beide Parameter nicht gleichzeitig gegeben sind.

Da die Wahrscheinlichkeit für letztere Aussage höher ist, wird diese Reihenfolge der Überprüfung gewählt.

Ferner kann man in der Abbildung 17 erkennen, dass es für eine erfolgreiche Eintaktung verschiedene Markierungen gibt:

- Markierung „inklusive“

- Markierung „liste“

- Markierung „---“

Ist für die Behandlung einer Naht keine Nachbehandlung notwendig, so wird ein gefundener Eintaktungsvorschlag mit der Markierung „---“ gekennzeichnet. Diese Markierung muss nicht weiter berücksichtigt werden.

Die Markierungen „inklusive“ und „liste“ beziehen sich auf die Behandlung von Nähten bei denen eine Vor- und Nachbehandlung notwendig ist. Hierbei gibt die Markierung „inklusive“ an, dass Vor- und Nachbehandlung von demselben Werker durchgeführt werden können.

Falls ein Werker nur die Vorbehandlung durchführen kann, so wird der Eintaktungsvorschlag mit der Markierung „liste“ gekennzeichnet. In diesem Fall kann man nur die Aussage machen, dass eine Eintaktung prinzipiell möglich ist. Um in diesem Fall einen kompletten Eintaktungsvorschlag zu erhalten, werden Vorschläge die mit der Markierung „liste“ gekennzeichnet sind wie folgt gehandhabt.

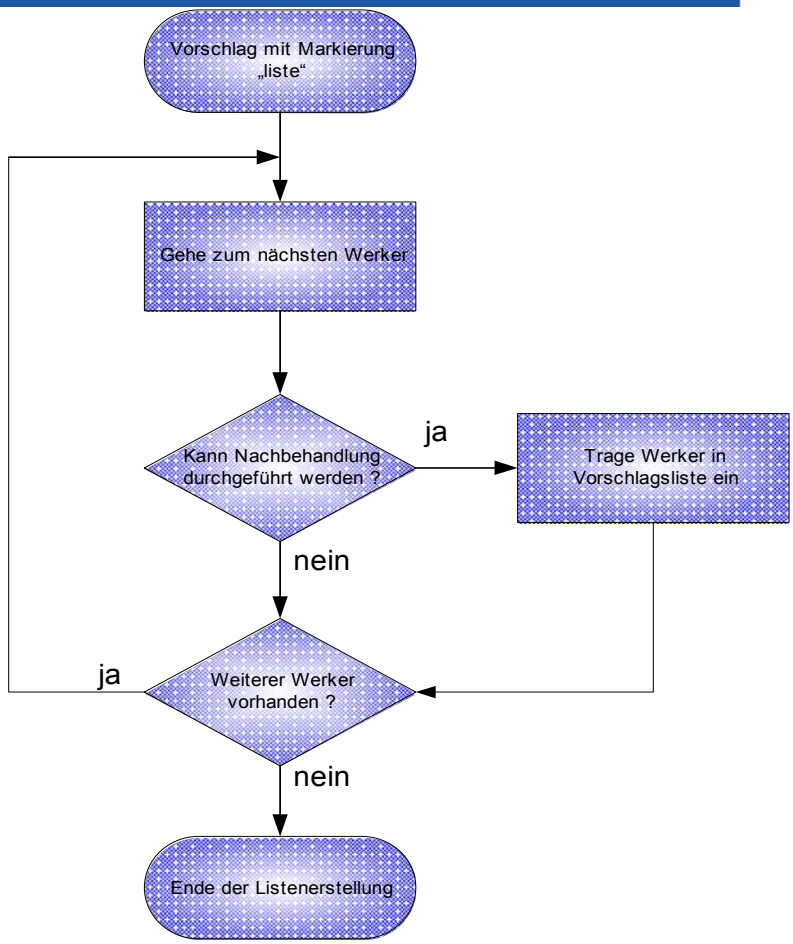


Abbildung 18: Listenerstellung für Nachbehandlung

Der Eintaktungsvorschlag mit der Markierung „liste“ ist gültig, wenn nach dieser Listenerstellung mindestens ein Werker für die Nachbehandlung in der Liste vorhanden ist.

Bei der Überprüfung „*Kann Nachbehandlung durchgeführt werden?*“ werden nur folgende Faktoren berücksichtigt:

- Kann der Werker mit der nötigen Bearbeitungszeit belastet werden?
- Liegt der Bearbeitungsort im Bearbeitungsradius des Werkers?

Die Überprüfung ob das notwendige Werkzeug vorhanden ist, wird vernachlässigt. Dies ist möglich, da es sich bei den Werkzeugen für die Nachbehandlung immer um Verstreichwerkzeuge handelt. Diese Werkzeuge sind im Gegensatz zu den Vorbehandlungswerkzeugen nicht ortsgebunden und können je nach bedarf frei verteilt werden.

Bei weiterer Betrachtung der Abbildung 14 erkennt man, dass die Suche nach einer möglichen Eintaktung abgebrochen wird, sobald eine mögliche Eintaktung gefunden wird, oder alle Arbeitsplätze ohne einen Eintaktungserfolg

durchlaufen sind. Dies wird in folgender Abbildung nochmals hervorgehoben dargestellt.

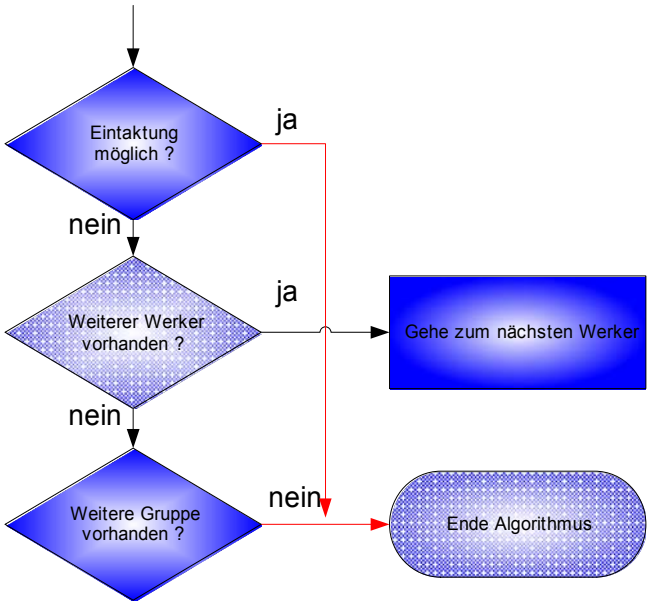


Abbildung 19:Abbruchbedingung

Für die Praxis ist es jedoch nützlicher, wenn das Programm nicht abbricht sobald eine Eintaktungsmöglichkeit gefunden wird. Besser ist es wenn das Programm alle gefundenen Eintaktungsmöglichkeiten in einer Liste speichert und diese anschließend auflistet. Durch diese Vorgehensweise ermöglicht man den Benutzer, Fachwissen und Erfahrung mit in die Eintaktung einzubringen.

2.4.1 Erweiterung der automatischen Eintaktung

Nachdem alle Arbeitsplätze überprüft sind, kann man die gefundenen Eintaktungsmöglichkeiten die in der Liste enthalten sind, auflisten. Bei der Speicherung einer Eintaktungsmöglichkeit muss darauf geachtet werden, dass keine notwendige Information verloren geht. Um dies zu vermeiden, wird folgende Struktur erstellt:

```
struct Vorschlag
{
      CString     Gruppe;
      CString     Arbeitsplatz;
      CString     PrevTaf;
      double      Lastvorher;
      double      Lastnachher;
      double      Schrittdauer;
      CString     Nachbehandlung;
};
```

Diese Struktur speichert den Ort, welches für die Eintaktung in Frage kommt. Wird ein Eintaktungsvorschlag ausgewählt, so wird nach dem angezeigten Arbeitsschritt eingetaktet. Die Variablen „Lastvorher“ und „Lastnachher“ geben an wie die Auslastung im Falle einer Eintaktung sich verändert.

Die Variable „PrevTaf“ gibt den TAF an, welches den Eintaktungsort kennzeichnet. Die Variable „Schrittdauer“ gibt die Bearbeitungszeit der neuen Tätigkeit an. In der Variable „Nachbehandlung“ wird die schon beschriebene Information der Markierung hinterlegt.

Auf der Basis dieser Struktur wird folgende Liste erstellt, welche bis zu einhundert Eintaktungsvorschläge verwalten kann:

```
struct ETListe
{
      Vorschlag Eintaktung[100];
      int size;
};
```

Hier gibt die Variable „size“ an wie viele Eintaktungsvorschläge sich gerade in der Liste befinden. Übernimmt man diese Änderungen in den Algorithmus, wirkt sich das wie folgende Abbildung zeigt aus.

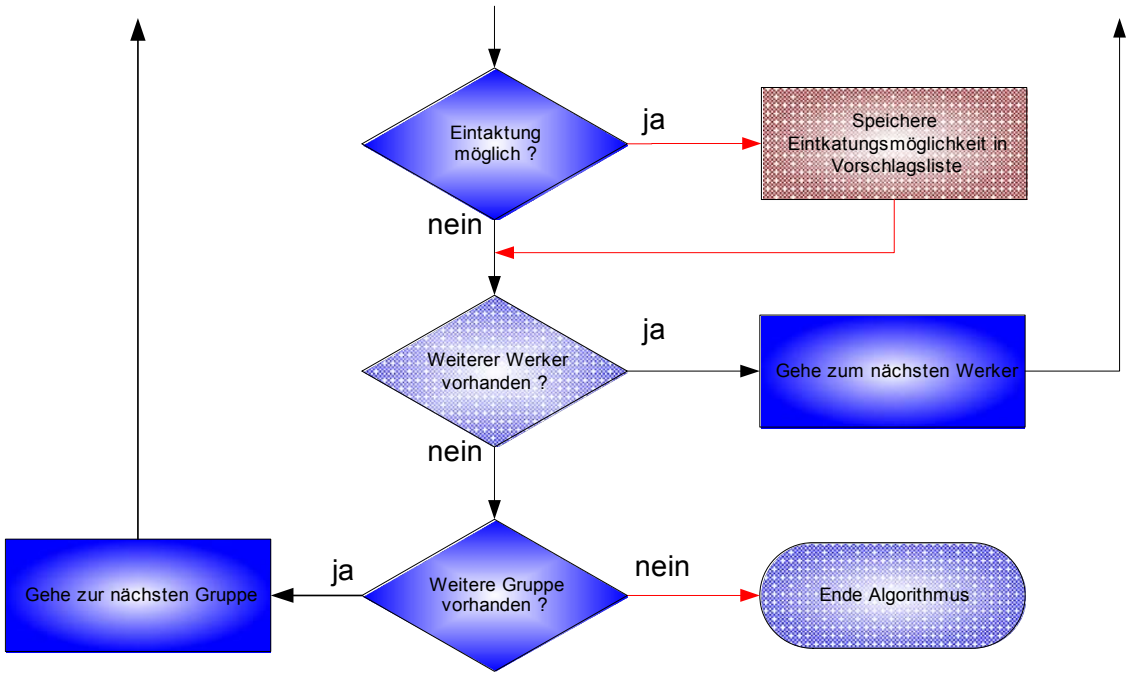


Abbildung 20: Speichern der Vorschläge

Durch den statischen Aufbau der Liste ist die Anzahl der gefundenen Eintaktungsvorschläge, die gespeichert werden können, auf einhundert begrenzt. Der Einsatz einer dynamischen Liste würde keine harte Begrenzung mit sich ziehen, dies ist aber nicht notwendig, da mehr als einhundert Eintaktungsvorschläge sehr selten gefunden werden.

3 DIE WISSENSBASIS ALS RELATIONALE DATENBANK

Die Information der Gruppen, Werker und Werkzeuge müssen so abgespeichert werden, sodass sie die Produktionslinie so Wahrheitsgetreu wie möglich widerspiegeln. Weiterhin soll es möglich sein, auf Änderungen in der Produktionslinie wie etwa der Einsatz neuer Werkzeuge oder die Einbindung neuer Gruppen, ohne großen Aufwand zu reagieren.

Um diese Informationen übersichtlich zu Verwalten bietet sich der Einsatz einer relationale Datenbank an.

3.1 RELATIONALE DATENBANK GRUNDLAGEN

Das besondere Kennzeichen relationaler Datenbanken ist, dass sie es ermöglichen mehrere Tabellen zueinander in Beziehung zu setzen. Die Verbindung zwischen den Tabellen wird dabei durch Schlüsselfelder hergestellt. Damit die Verbindung zwischen verschiedenen Tabellen reibungslos hergestellt werden kann, ist es notwendig dass die Schlüsselfelder in allen Tabellen die gleiche Information repräsentieren.

Der Schlüssel einer Tabelle muss innerhalb der Tabelle einmalig sein. Durch diese Bedingung verhindert man die Entstehung von Zweideutigkeiten. Der Schlüssel einer Tabelle kann aber in einer anderen Tabelle mehrfach vorkommen, in diesem Fall wird dieser Schlüssel dort Fremdschlüssel genannt.

TAF	Arbeitssch	Gruppe	ArbeitsplatzNr	Beschreibung	Bearbeitungsort	Bearbeitungswerkzeug
5310AG	520	G11	4	PVC auftragen mit Pistole	sv,m,re-mi	GNR
5310AH	530	G11	4	PVC auftragen mit Pistole	sv,m,re-mi	GNR
5310AI	540	G11	4	PVC auftragen mit Pistole	sv,m,re-mi	GNR
5310AK	550	G11	4	PVC auftragen mit Pistole	sv,m,re-mi	GNR
	560	G11	4	PVC auftragen mit Pistole	sv,un,re-mi	

Schlüsselfeld

Fremdschlüssel

Tabelle 6: Beispiel Fremdschlüssel

Der Fremdschlüssel stellt das Schlüsselfeld einer anderen Tabelle dar, und ist somit dort einmalig. Dies bedeutet, dass über den Fremdschlüssel die zweite Tabelle mit dieser Tabelle verbunden ist. Über den Fremdschlüssel kann man in der zweiten Tabelle weitere Informationen erhalten, welche somit in einer Beziehung mit dem Datensatz der gezeigten Tabelle stehen. In diesem Beispiel kann man über diesen Fremdschlüssel weitere Informationen über ein Werkzeug erhalten.

Anhand dieses Beispiels wird ein Vorteil einer relationalen Datenbanken leicht ersichtlich, würde man anhand des Kürzels „GNR“ in jeder Spalte die Werkzeugsbezeichnung „Grobnahtrohr“ abspeichern, so würde sich eine speicherintensivere Tabelle bilden. Wenn es nicht möglich wäre durch den

Fremdschlüssel Artverschiedene Informationen in einer anderen Tabelle abzuspeichern und dabei trotzdem die Beziehung zu diesen Informationen beizubehalten, so würde der Aufbau der Tabelle schnell unübersichtlich werden.

3.2 AUFBAU DER TABELLEN UND DEREN BEZIEHUNGEN

Alle Arbeitsschritte in der Produktionslinie, welche für ein spezielles Modell nötig sind, werden in einer Tabelle abgespeichert. In dieser Tabelle sind alle notwendigen Informationen für eine automatische Eintaktung vorhanden. Damit man alle Modelle, die durch diese Produktionslinie bearbeitet werden, übersichtlich verwalten kann, wird folgende Tabelle erstellt.

Modelle

Kürzel	Bezeichnung	Taktzeit
A4A	A4-Avant	1,21
A4L	A4-Limusine	1,21

Tabelle 7: Modelle

In dieser Tabelle werden alle Modelle verwaltet, die bearbeitet werden. Das Programm betrachtet diese Tabelle um herauszufinden, für welche Modelle die Arbeitsschritte verwaltet sind. Hierbei muss die Bezeichnung genau denselben Namen haben wie die Tabelle des jeweiligen Modells, worin alle Arbeitsschritte verwaltet sind.

Theoretisch ist es möglich für jedes Modell eine eigene Taktzeit vorzugeben. Diese Funktionalität ist derzeit in der Praxis nicht möglich. Die Bezeichnungen „A4-Avant“ und „A4-Limusine“ geben dem Programm an, dass es gleichnamige Tabellen gibt. Durch diese Organisation ist es möglich das Programm ohne großen Aufwand an neue Modelle anzupassen. Hierzu muss lediglich die neue Tabelle erstellt werden, worin alle Arbeitsschritte für das neue Modell beschrieben sind, und der Name der neuen Tabelle muss hier aufgeführt werden.

Bevor man die Information aller Arbeitsschritte eines neuen Modells in einer Tabelle abspeichern kann, muss bekannt sein wie diese Tabelle aufgebaut ist. Hier handelt es sich um die Tabelle, welche im Vorfeld schon beschrieben wurde. Damit eine automatische Eintaktung möglich ist, wird in dieser Tabelle für jeden Arbeitsschritt die Information des aktuellen Bearbeitungsbereichs und die Information der Werkzeuge, die der Werker zur Verfügung hat, verwaltet. Weiterhin ist für jeden Arbeitschritt die Information der Bearbeitungsdauer hinterlegt. Die Arbeitsschritte werden in dieser Tabelle sequentiell, beginnend mit dem ersten Arbeitsplatz gespeichert.

Die Reihenfolge der Arbeitsschritte ist durch den Wert, welches in der Spalte ArbeitsschrittNr steht, geregelt.

	Feldname	Felddatentyp
🔑	TAF	Text
	ArbeitsschrittNr	Zahl
	Gruppe	Text
	ArbeitsplatzNr	Text
	Beschreibung	Text
	Bearbeitungsort	Text
	Bearbeitungswerkzeug	Text
	TeileNr	Text
	PDM-Blatt	Text
	Pos	Text
	Zeit	Zahl

Tabelle 8: Aufbau einer Tabelle für ein Modell

Folgende Tabelle stellt einen Ausschnitt einer Tabelle dar, welches alle Arbeitsschritte für die Bearbeitung eines Modells verwaltet. Im Falle einer Eintaktung stellen die Spalten „TAF“ und „ArbeitsschrittNr“ ein Problem dar. Da die „ArbeitsschrittNr“ für die Reihenfolge ausschlaggebend ist, muss sie im Vergleich zum vorangehenden Arbeitsschritt um eins erhöht werden. Diese Vorgehensweise kann im folgenden Fall zu einem Konflikt führen.

5310AL	56	sv,un,re-mi	GNR	0,0555
5310AN	56	sv,un,re-mi	GNR	0,0277

Konflikt wenn zwischen diesen Arbeitsschritten
Eingetaktet werden soll

Tabelle 9: Konflikt ArbeitsschrittNr

Dieses Problem wird gelöst, indem man vor einer Eintaktung überprüft ob diese Konstellation gegeben ist. Falls ja, werden die nachfolgenden Arbeitsschrittnummern solange erhöht, bis sie keine fortlaufende Zahl mehr bilden. Somit wird die notwendige Lücke für die Eintaktung geschaffen. Ein weiteres Problem ist die Vergabe von den TAFs, welche bei der AUDI zentral vergeben werden. In den meisten Fällen wird zum Zeitpunkt einer Eintaktung noch keine TAF vorhanden sein. Um dennoch eine Eintaktung durchführen zu können, werden Platzhalter eingesetzt.

Da die Spalte der TAFs das Schlüsselfeld dieser Tabelle ist, muss dabei gewährleistet sein, dass auch die Platzhalter nicht zweimal in der Tabelle vorkommen. Um dies zu erreichen betracht man bei einer Eintaktung den Arbeitsschritt vor der Eintaktungsstelle. Da die TAFs alle zentral vergeben werden, sind sie für alle Arbeitsschritte einmalig. Diese Eigenschaft wird genutzt um bei der Erstellung der Platzhalter weiterhin einmalige TAFs zu erhalten. Hierzu braucht man lediglich immer einen speziellen Zusatz an die TAF des vorangehenden Arbeitsschritts beizusteuern. Ist die Art der Zusatzinformation

immer gleich, so können weiterhin keine TAFs entstehen die schon vorhanden sind. Das Programm erweitert bei einer Eintaktung die Information des

vorangehenden TAFs lediglich um ein „P“, wie durch folgende Tabelle deutlich wird.

	TAF	Arbeitssch	Gruppe	ArbeitsplatzNr
	6120BD	3870	G14	3
	6120BG	3880	G14	3
	6120BK	3890	G14	3
			14	3
	6120BM	3900	G14	3

Einfügen von einem Platzhaler

Erhöhung der ArbeitsschrittNr

Tabelle 10: Platzhalter

Um die Bearbeitungszeit einer Naht mit einem speziellen Werkzeug berechnen zu können, müssen die Umrechnungskonstanten wie schon beschrieben in einer Tabelle abgespeichert werden. Weiterhin gilt es die Kürzel für die verschiedenen Werkzeuge zu verwalten. Diese zwei Funktionen werden mit folgender Tabelle gelöst.

Werkzeuge

Werkzeugkürzel	Bezeichnung	< 200 mm	> 200 mm < 500 mm	> 500 mm < 800 mm
FP	Flachpinsel	145	255	370
GNR	Grobnahtrohr	55	65	70
HKP	Heizkörperpinsel	75	170	315
KD	Kreuzdüse	55	65	70
SCH	Schaber	145	255	370
SD10	Schlitzdüse 10mm	50	60	65
SD15	Schlitzdüse 15mm	50	60	65
SD5	Schlitzdüse 5mm	50	60	65
SD8	Schlitzdüse 8mm	50	60	65
SPD	Spitzdüse	55	65	70
SPP	Sprühpistole	70	110	145
WD6	Winkeldüse 6mm	50	60	65
WT	Wischtuch	145	255	370

Tabelle 11: Werkzeuge

Das Schlüsselfeld dieser Tabelle bildet die Spalte der Werkzeugkürzel. Diese Kürzel sind in den Tabellen der verschiedenen Modelle als Fremdschlüssel enthalten. Die beiden Tabellen sind durch das Feld der Werkzeugkürzel miteinander verbunden.

Zur Umrechnung der Konstanten in einen Zeitfaktor, müssen diese Werte mit produktionsspezifischen Faktoren multipliziert werden. Um eine Unterscheidung zwischen den Werkzeugen für die Vor- bzw. Nachbehandlung zu erhalten, enthält diese Tabelle eine weitere Spalte, in der die Werkzeuge für die Nachbehandlung speziell markiert sind.

Damit es möglich ist auf Änderungen der produktionsspezifischen Faktoren reagieren zu können, sind alle Faktoren mit denen diese Konstanten multipliziert werden müssen, um die Bearbeitungsdauer in Minuten zu erhalten, in der Tabelle „Konstanten“ gespeichert. Aus Geheimhaltungsgründen wird diese Tabelle nicht genau erläutert.

Analog zu den Werkzeugen ist die Tabelle der Tätigkeiten zu verstehen. Diese Tabelle wird zur Berechnung der Bearbeitungszeit genutzt wenn es sich bei der Eintaktung um eine „einfache Tätigkeit“ handelt. Wie bei der Tabelle der Werkzeuge bildet auch hier das Feld der Kürzel den Schlüssel. Dieser Schlüssel wird in den Tabellen der verschiedenen Modelle als Fremdschlüssel genutzt, um so eine Beziehung zwischen diesen Tabellen herzustellen.

Taetigkeit

Taetigkeit	Kuerzel	TMU
Leinen30x30	LE30	30
Leinen40x40	LE40	90
Stopfen30	ST30	60
Stopfen40	ST40	32

Tabelle 12:Tätigkeit

Die Tabelle der Gruppen ist wichtig für die automatische Eintaktung. Hier ist zu jeder Gruppe die Bearbeitungshöhe angegeben, welches durch den Eintaktungsalgorithmus abgefragt wird. Diese Angabe der Bearbeitungshöhe(n) muss für jedes Modell angegeben werden.

Gruppen

Gruppe	Modell	Beschreibung	Bearbeitungshöhe	Anzahl Werker	Max Anzahl Werker
G11	A4-Avant	m-un	9	9	
G11	A4-Limusine	m-un	9	9	
G12	A4-Avant	m-un	9	9	
G12	A4-Limusine	m-un	9	9	
G13	A4-Avant	o-m	9	9	
G13	A4-Limusine	o-m	9	9	

Tabelle 13: Gruppen

Weiterhin ist es möglich zu überprüfen ob die die maximal erlaubte Anzahl der Werker schon erreicht ist. Diese Überprüfung ist notwendig, wenn das Programm entscheiden muss ob ein weiterer Werker notwendig ist.
Weiterhin sind Tabellen für das Handling der administrativen Tätigkeiten vorhanden, welche später genauer beschrieben werden. Diese Tabellen werden alle durch eine Access Datenbank verwaltet.

Der Zugriff auf diese Tabellen wird über die Open Database Connectivity Schnittstelle realisiert. Durch den Einsatz dieser Schnittstelle ist es theoretisch möglich all diese Tabellen durch ein anderes Datenbankprogramm zu verwalten. Um dabei weiterhin eine einwandfreie Funktionalität des Programms zu erhalten, muss lediglich der ODBC-Treiber auf das neue Programm

umgestellt werden. Diese Umstellung geschieht im Betriebssystem und ist somit von dem Programm komplett entkoppelt.

3.3 Open Database Connectivity (ODBC)

Microsoft hat die Inkompatibilität der Datenbankenschnittstellen als Problem erkannt. Jede Datenbank hatte ihre eigene Anwendungsentwicklungssprache, die zwar gut in die Datenbank integriert war, aber nicht mit irgendeiner anderen Datenbank zusammenarbeitete. Daraus ergaben sich für jeden Entwickler Probleme, der eine Datenbank für die eine Anwendung und dann eine andere Datenbank für die nächste Anwendung benötigte. Der Entwickler musste sich mit der speziellen Datenbanksprache der jeweiligen Datenbank beschäftigen und konnte nicht auf eine bekannte Sprache zurückgreifen.

Damit der Entwickler mit einer beliebigen Datenbank in der bevorzugten Programmiersprache arbeiten konnte, war eine standardisierte Schnittstelle erforderlich, die mit jeder Datenbank funktioniert. Die Schnittstelle Open Database Connectivity (ODBC) ist als standardisierte, SQL-Basierte Schnittstelle in das Betriebssystem Windows integriert. Hinter dieser Schnittstelle verbergen sich für verschiedene Datenbanken Plugins, die die ODBC- Funktionsaufrufe entgegennehmen und in Aufrufe für die spezielle Schnittstelle der betreffenden Datenbank umwandeln.

Damit braucht ein Entwickler nur noch eine einzige Datenbankschnittstelle für alle Datenbanken zu erlernen und einzusetzen. Darüber hinaus ist es den Anbietern von Programmiersprachen möglich, ODBC-Unterstützung in ihre Sprachen und Entwicklungswerkzeuge zu integrieren, um den Datenbankzugriff nahezu transparent zu gestalten.

3.3.1 Die Klasse CRecordset

In der Visual C++ Entwicklungsumgebung ist der größte Teil der ODBC Funktionalität in zwei Klassen verkapselt:

- CRecordset
- CDatabase.

Die Klasse CDatabase enthält die Verbindungsinformationen zur Datenbank und kann in der gesamten Anwendung gemeinsam genutzt werden. Die Klasse CRecordset verkapselt einen Satz von Datensätzen aus der Datenbank. Über diese Klasse kann man eine auszuführende SQL-Abfrage spezifizieren. Die Klasse führt die Abfrage aus und verwaltet die Ergebnismenge, die die Datenbank zurückgibt.

Die Datensätze im Recordset kann man modifizieren und aktualisieren, die Änderungen werden zurück an die Datenbank übergeben. Im Recordset lassen

sich Datensätze hinzufügen und löschen. Auch diese Änderungen kann man zurück in die Datenbank schreiben.

3.3.1.1 VERBINDUNG ZUR DATENBANK HERSTELLEN

Bevor die Funktionalität der Klasse CRecordset genutzt werden kann, muss sie mit einer Datenbank verbunden werden. Dies realisiert man mit Hilfe der Klasse CDatabase. Es ist keine Instanz der Klasse CDatabase explizit zu erzeugen oder einzurichten, die erste Instanz der Klasse CRecordset übernimmt das automatisch.

Wenn man in der Programmierumgebung Visual C++ eine Anwendung mit dem Anwendungs-Assistenten erstellt und die Einbindung der ODBC Funktionalität wählt, bindet der Abwendungs-Assistent automatisch die Informationen welche für die Verbindung zu der Datenbank nötig sind in die erste von ihm erzeugte und von CRecordset abgeleitete Klasse ein. Wenn man diese CRecordset Klasse erzeugt, ohne ein CDatabase Objekt zu übergeben, verwendet sie die vorgegebenen Verbindungsinformationen, die durch den Assistenten hinzugefügt werden, um eine Verbindung zu der Datenbank herzustellen.

3.3.1.2 DAS ÖFFNEN UND SCHLIEßEN EINES RECORDSETS

Nachdem ein `CRecordset` Objekt erstellt und mit der Datenbank verbunden ist, muss der Recordset geöffnet werden, um eine Gruppe von Datensätze aus der Datenbank abzurufen. Dazu wird die Member Funktion `Open` des `CRecordset` Objekts genutzt. Diese Funktion kann man ohne irgendwelche Argumente aufrufen, wenn man die Standardwerte übernehmen möchte, welche bei der Erstellung mit dem Anwendungsassistenten eingegeben wurden.

Das erste Argument der Funktion Open gibt den Typen des Recordsets an. Der Typ de Recordsets gibt an welche Funktionalität bezüglich der Aktualisierung der Datensätze gegeben ist. Der Standardwert für dieses Argument lautet AFX_DB_USE_DEFAULT_TYPE. Bei der Nutzung des Standartwerts, wird der Recordset als Snapshot geöffnet.

Folgende Tabelle listet die vier Typen von Recordsets auf, welche bei Datenbanken üblich sind.

Typ	Beschreibung
CRecordset::dynaset	Eine Gruppe von Datensätzen, die sich durch Aufruf der Funktion `Fetch` aktualisieren lassen, so dass Änderungen, die durch andere Benutzer am Recordset vorgenommen wurden, sichtbar sind.
CRecordset::snapshot	Eine Gruppe von Datensätzen, die sich nur aktualisieren lässt, wenn

	man den Recordset schließt und erneut öffnet.
CRecordset::dynamic	Nahezu ähnlich zum Typ `CRecordset::dynaset`, aber in vielen ODBC-Treibern nicht verfügbar.
CRecordset::forwardOnly	Eine schreibgeschützte Gruppe von Datensätzen, die sich nur vom ersten bis zum letzten Datensatz durchblättern lassen.

Tabelle 14:Typen von Recordsets

Das zweite Argument der Funktion Open ist eine SQL-Anweisung, die auszuführen ist, um den Recordset zu füllen. Übergibt man für dieses Argument eine NULL, so wird die vom Anwendungs-Assistenten erzeugte und vorgegebene SQL-Anweisung ausgeführt. In diesem Fall wird eine spezielle Abfrage oder eine Tabelle geöffnet, welche bei der Erstellung mit Hilfe des Assistenten angegeben wurde. Als drittes Argument wird eine Gruppe von Flags benötigt, womit angegeben werden kann, wie die Gruppe der Datensätze in den Recordset abzurufen ist. Da die meisten Flags ein tiefergehendes Verständnis der ODBC-Schnittstelle erfordern, zeigt folgende Tabelle lediglich ein paar dieser Flags.

Flag	**Beschreibung**
CRecordset::none	Der Standardwert für dieses Argument. Legt fest, dass keine Optionen die Art und Weise beeinflussen, wie der Recordset geöffnet und verwendet wird.
CRecordset::appendOnly	Verhindert, dass der Benutzer die existierenden Datensätze im Recordset bearbeiten oder löschen kann. Der Benutzer kann ausschließlich neue Datensätze in den Recordset einfügen. Diese Option kann nicht in Verbindung mit dem Flag `CRecordset::readOnly` verwendet werden.
CRecordset::readOnly	Legt fest, dass sich der Recordset nur lesen lässt und der Benutzer keinerlei Änderungen daran vornehmen kann. Diese Option kann nicht in Verbindung mit dem Flag `CRecordset::appendOnly` verwendet werden.

Tabelle 15:Flags für das öffnen eines Recordsets

Nachdem die Arbeit mit dem Recordset abgeschlossen ist, kann man die Funktion Close aufrufen, um den Recordset zu schließen und alle von dem Recordset belegten Ressourcen freizugeben. Die Funktion Close wird ohne Argumente aufgerufen.

3.3.1.3 NAVIGATION DURCH DEN RECORDSET

Nachdem eine Gruppe von Datensätzen aus der Datenbank abgerufen wurde, muss man in der Lage sein, durch den Recordset zu navigieren. Hierzu bietet die Klasse CRecordset mehrere Funktionen, die es ermöglichen durch den Datensatz zu navigieren. Folgende Tabelle zeigt welche Funktionen die Navigation durch den Recordset unterstützen.

Funktion	Beschreibung
MoveFirst	Geht zum ersten Datensatz im Recordset.
MoveLast	Geht zum letzten Datensatz im Recordset.
MoveNext	Geht zum nächsten Datensatz im Recordset.
MovePrev	Geht zum vorherigen Datensatz im Recordset.
Move	Schaltet um die angegebene Zahl von Datensätzen ausgehend vom aktuellen Datensatz oder vom ersten Datensatz im Recordset weiter.
SetAbsolutePosition	Geht zum angegebenen Datensatz im Recordset.
IsBOF	Liefert `TRUE` zurück, wenn es sich beim aktuellen Datensatz um den ersten Datensatz im Recordset handelt.
IsEOF	Liefert `TRUE`, wenn es sich beim aktuellen Datensatz um den letzten Datensatz im Recordset handelt.
GetRecordCount	Gibt die Anzahl der Datensätze im Recordset zurück.

Tabelle 16:Navigation durch einen Recordset

Zwei dieser Funktionen für die Navigation durch den Recordset brauchen Argumente für deren Steuerung.

- Move
- SetAbsolutePosition

Der Funktion SetAbsolutePosition wird ein einzelnes Argument übergeben, welche die Zeile des Zieldatensatzes spezifiziert. Wobei die ‚0' den ersten Datensatz markiert. Übergibt man dieser Funktion einen negativen Wert, so beginnt die Selektion am Ende des Recordsets, wobei die ‚-1' den letzen Datensatz und die ‚-2' den vorletzten Datensatz markiert.

Der Funktion Move werden zwei Argumente übergeben. Das erste gibt die Anzahl der Datensätze an, welche übersprungen werden sollen. Hierbei kann man positive und negative Werte angeben. Ein negativer Wert kennzeichnet eine Rückwärtsnavigation, wobei bei einem positiven wert vorwärts navigiert wird. Durch das zweite Argument wird die Art der Navigation durch den Recordset spezifiziert.

Die möglichen Werte und deren Bedeutung für das zweite Argument werden in der folgenden Tabelle beschrieben.

Typ	Beschreibung
SQL_FETCH_RELATIVE	Bewegung um die angegebene Anzahl von Zeilen ab der aktuellen Zeile.
SQL_FETCH_NEXT	Verschiebung zur nächsten Zeile, wobei die angegebene Anzahl von Zeilen ignoriert wird. Das gleiche wie der Aufruf der Funktion `MoveNext`.
SQL_FETCH_PRIOR	Verschiebung zur vorherigen Zeile, wobei die angegebene Anzahl von Zeilen ignoriert wird. Das gleiche wie der Aufruf der Funktion `MovePrev`.
SQL_FETCH_FIRST	Geht zur ersten Zeile, wobei die angegebene Anzahl von Zeilen ignoriert wird. Das gleiche wie der Aufruf der Funktion `MoveFirst`.
SQL_FETCH_LAST	Geht zur letzten Zeile, wobei die angegebene Anzahl der Zeilen ignoriert wird. Das gleiche wie der Aufruf der Funktion `MoveLast`.
SQL_FETCH_ABSOLUTE	Verschiebung um die angegebene Anzahl von Zeilen vom Beginn des Recordsets. Das gleiche wie der Aufruf der Funktion `SetAbsolutePosition`.

Tabelle 17: Argumentbeschreibung der Funktion Move

Wenn ein Eintaktungsvorschlag gefunden wird, welches für den Benutzenden sinnvoll erscheint, so muss dieser Vorschlag eingetaktet werden. Aus der Sicht der Datenbank gilt ein Vorschlag als eingetaktet, wenn der zugehörige Datensatz in den Recordset hinzugefügt wird. Hierzu muss die ODBC-Schnittstelle die Funktionalität für das hinzufügen von neuen Datensätzen unterstützen.

3.3.1.4 Das Hinzufügen, Löschen und Aktualisieren von Datensätzen

Zu der Möglichkeit neue Datensätze in den Recordset aufzunehmen, muss es möglich sein vorhandene Datensätze zu bearbeiten sowie auch zu löschen. Für all diese Aktionen bietet die Klasse CRecordset folgende Funktionen.

Funktion	Beschreibung
AddNew	Fügt einen neuen Datensatz in den Recordset ein.
Delete	Löscht den aktuellen Datensatz aus dem Recordset.
Edit	Erlaubt die Bearbeitung des aktuellen Datensatzes.
Update	Speichert die aktuellen Änderungen in der Datenbank.
Requery	Führt die aktuelle SQL-Abfrage erneut aus, um den Recordset zu aktualisieren.

Tabelle 18: Administrative Funktionen

Diesen Funktionen müssen keine Argumente übergeben werden. Allerdings muss bei einigen Funktionen gewährleistet sein dass bestimmte Schritte eingehalten werden, damit sie richtig funktionieren.

Um einen neuen Datensatz in die Datenbank aufzunehmen muss zunächst die Funktion AddNew aufgerufen werden. Diese Funktion reserviert im Vorfeld den Speicherbereich welches für den neuen Datensatz notwendig ist. Wie dies zu verstehen ist, wird in folgender Abbildung deutlich.

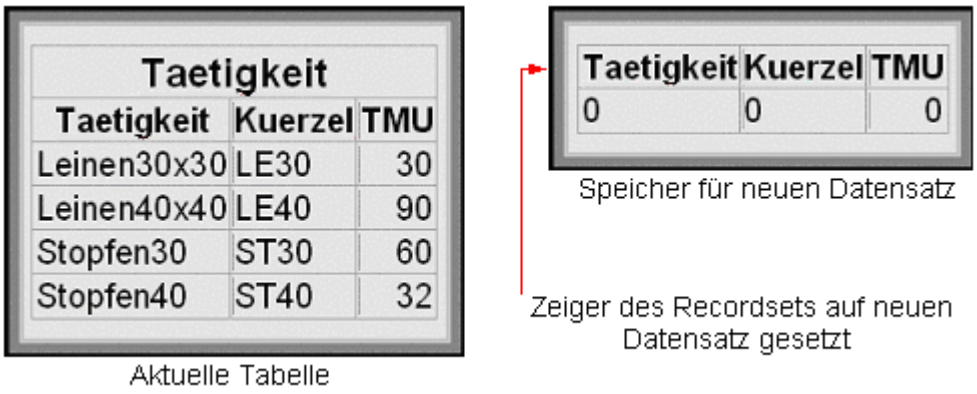


Abbildung 21: Speicherreservierung durch AddNew

Weiterhin wird der aktuelle Zeiger des Recordsets auf den neuen Datensatz gelenkt. Das Zeichen ‚0' in den Spalten des neuen Datensatzes stellt dar, dass die Felder nicht belegt sind. Als nächstes ist es notwendig den neuen Datensatz mit Informationen zu füllen. Hierbei muss mindestens das Schlüsselfeld der Tabelle mit einer gültigen Information besetzt werden.

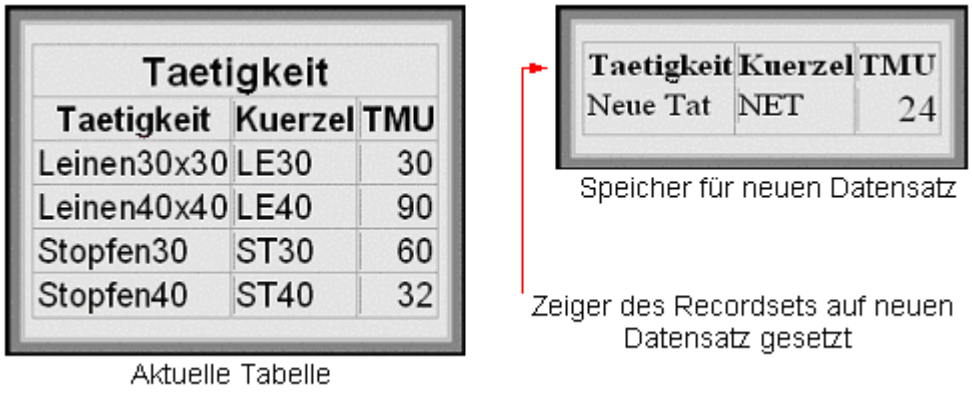


Abbildung 22:Füllen des neuen Datensatzes mit Informationen

Derzeit ist der neue Datensatz zwar mit Informationen gefüllt, aber noch nicht in die Tabelle integriert. Um dies zu erreichen muss nun die Funktion Update aufgerufen werden. Nach dem Aufruf der Funktion Update ist der neue Datensatz in die Tabelle aufgenommen. Um nun einen Recordset auf die neue Tabelle zu erhalten muss die Funktion Requery aufgerufen werden. Hält man diese Reihenfolge ein, so gelingt das Hinzufügen von einem neuen Datensatz in die Tabelle.

In unserem Beispiel würde der Quellcode für das Hinzufügen dieses Beispieldatensatzes wie folgt aussehen:

```
// Einen neuen Datensatz in den Recordset einfügen
m_pSet.AddNew();

// Neuen Datensatz mit Informationen Bereichern
m_pSet.m_Taetigkeit = „Neue Tat“;
m_pSet.m_Kuerzel = „NET“;
m_pSet.m_TMU = 24;

// Den neuen Datensatz in der Datenbank speichern
m_pSet.Update();

// Den Recordset aktualisieren
m_pSet.Requery();

// Zum neuen Datensatz gehen
m_pSet.MoveLast();
```

Der aktuelle Datensatz kann durch den Aufruf der Funktion Delete gelöscht werden. Ist der aktuelle Datensatz gelöscht, so muss zu einem anderen Datensatz navigiert werden, um so nicht mehr den gerade gelöschten Datensatz zu betrachten. Sobald ein Datensatz gelöscht wird, gibt es keinen aktuellen Datensatz mehr, bis zu einem anderen Datensatz navigiert wird. Hierbei muss die Funktion Update nicht aufgerufen werden, da dies durch die Navigationsfunktionen automatisch durchgeführt wird.

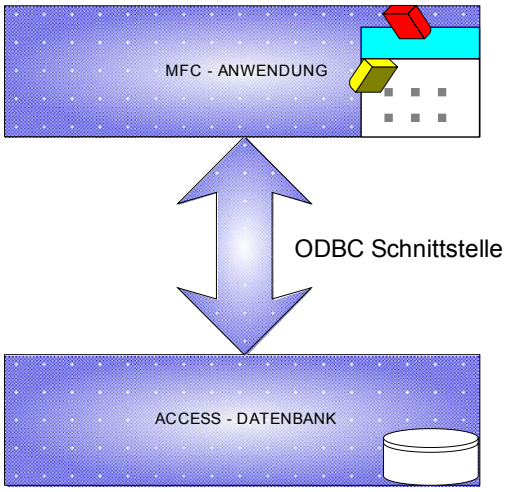
```
// Den aktuellen Datensatz löschen
m_pSet.Delete();

// Zum vorherigen Datensatz gehen
m_pSet.MovePrev();
```

Um eine Aktualisierung eines Datensatzes zu durchzuführen, ruft man zunächst die Funktion Edit auf. Anschließend kann man die Felder bearbeiten, die man aktualisieren will. Hat man alle Änderungen durchgeführt, ruft man die Funktion Update auf, um die Änderungen in die Datenbank zu übernehmen. Hierzu sieht der Quellcode wie folgt aus:

```
// Die Bearbeitung des aktuellen Datensatzes ermöglichen
m_pSet.Edit();
// Bearbeiten der Felder, die man aktualisieren will
.
.
.
.
.
.
// Änderungen des Benutzers im aktuellen Datensatz speichern
m_pSet.Update();
```

4 AUFBAU DES PROGRAMMS

Um eine richtige Funktionalität liefern zu können brauch das Programm eine gut gepflegte Datenbank. Damit diese Datenbank ohne großen Aufwand gut gepflegt werden kann, habe ich eine Access Datenbank ausgewählt. Um keine direkte Bindung zu der Access Datenbank zu haben, wird über die oben beschriebene ODBC-Schnittstelle auf die Datenbank zugegriffen.



Der Einsatz dieser Schnittstelle ermöglicht eine Nutzung des Programms in allen Windows Betriebssystemen, bei denen es möglich ist, eine Verbindung zu der Datenbank über die ODBC-Schnittstelle aufzubauen. Hierbei muss die Schnittstelle bei der Einrichtung im Betriebssystem „ETV1“ genannt werden.

4.1 DAS HAUPTFENSTER

Das Programm soll so gestaltet sein, dass eine einfache Handhabung gegeben ist. Um dies zu erreichen, sind die wichtigsten Funktionen als selbsterklärende Buttons realisiert. Durch das Hauptfenster ist es möglich folgende Nutzungsfälle durchzuführen:

- Eingabe von Informationen einer neuen Naht
- Eingabe von Informationen einer neuen Tätigkeit
- Berechnung von Eintaktungsmöglichkeiten
- Betrachten von Eintaktungsvorschlägen

Weiterhin ist es möglich die Taktzeit zu verändern, um so zu sehen wie sich das auf die Eintaktungsvorschläge auswirkt.

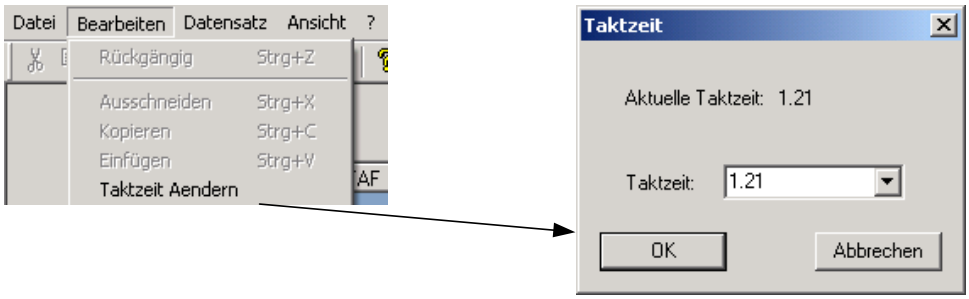


Abbildung 23: Taktzeit Ändern

Da das Ändern der Taktzeit eine Funktion ist, welche in der Praxis sehr selten genutzt wird, wird diese Funktion in das Menü eingebettet. So ist es möglich die sichtbare Programmfläche übersichtlich zu gestalten.

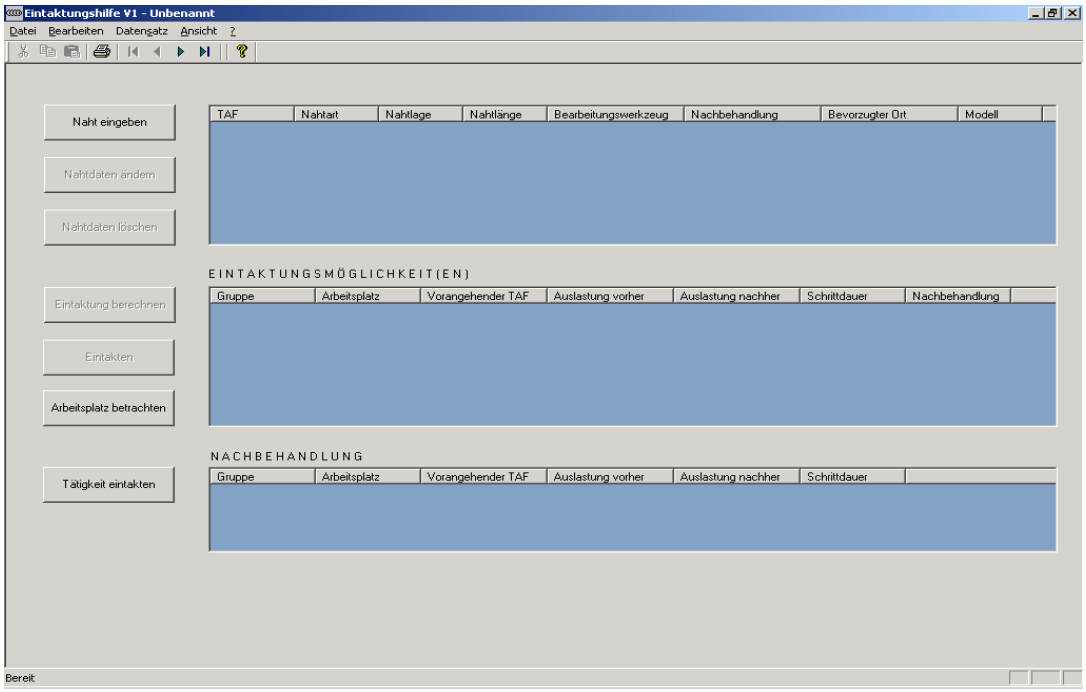
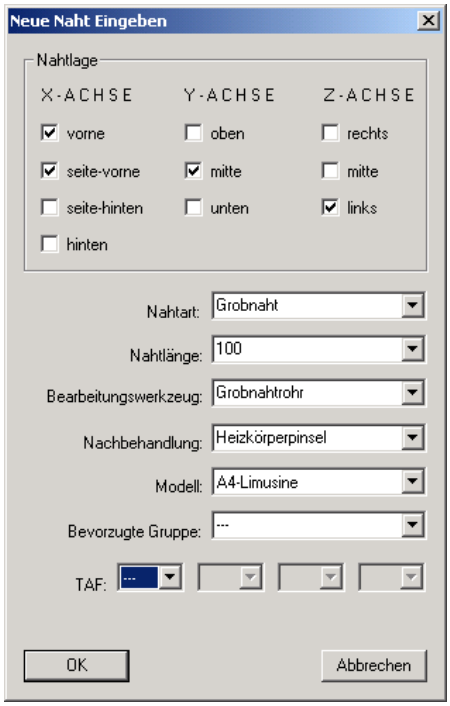


Abbildung 24: Das Hauptfenster

Aus dieser Abbildung wird ersichtlich, dass nicht alle Buttons aktiv sind. Die Buttons werden nur dann aktiviert, wenn es einen Sinn macht die Funktionalität welche durch den speziellen Button ausgelöst wird anzubieten. Zum Beispiel macht es erst einen Sinn Nahtdaten zu ändern, wenn schon welche eingegeben wurden und eine Naht aus der Liste ausgewählt wird. Diese Voraussetzung gilt auch für die Buttons „Nahtdaten löschen“ und „Eintaktung berechnen“. Das Button Eintakten wird erst aktiviert, wenn gültige Eintaktungsvorschläge vorhanden sind und ein Eintaktungsvorschlag aus der Vorschlagsliste ausgewählt ist.

4.2 EINGABEFENSTER FÜR NAHTABHÄNGIGE TÄTIGKEIT



Muss eine neue Naht eingetaktet werden, so kann man alle Daten die notwendig sind durch die dargestellte Eingabemaske eingeben. Wird diese Eingabemaske aufgerufen, so nimmt das Programm kontakt mit der Datenbank auf, um folgende Listen mit Informationen zu füllen.

Bearbeitungswerkzeug → Tabelle Werkzeuge
Nachbehandlung → Tabelle Werkzeuge
Modell → Tabelle Modelle

Die Liste der Bevorzugten Gruppe wird in Abhängig von dem Modell mit Informationen gefüllt, hierbei wird die Tabelle der Gruppen genutzt wird.

Dadurch dass bei jedem Aufruf dieses Fensters kontakt mit der Datenbank aufgenommen wird, wird gewährleistet dass immer die gültigen Modelle bzw. Werkzeuge für die Auswahl zur Verfügung gestellt werden. Weiterhin wird durch die Vorgabe der Auswahlkriterien, in Form von einer Liste, eine mögliche Fehleingabe verhindert.

Nachdem durch dieses Fenster die Nahtinformationen eingegeben sind, wird die Information dieser Naht in der oberen Liste des Hauptfensters aufgelistet. Man hat die Möglichkeit mehrere Nahtinformationen im Voraus einzugeben, um diese anschließend nacheinander einzutakten. Selektiert man eine spezielle Nahtinformation, so kann im Hauptfenster eine Eintaktung durchgeführt werden. Dies würde im besten Fall in folgenden Schritten geschehen:

- Selektierung einer Naht die durch diese Maske eingegeben wurde
  - Aktiviert „Eintaktung berechnen“ Button
- Nutzen des „Eintaktung berechnen“ Buttons
  - Liste der „Eintaktungsmöglichkeiten“ wird gefüllt
- Selektierung eines Eintaktungsvorschlags
  - Aktiviert „Eintakten“ Button
- Durchführen der Eintaktung

In diesem Beispiel wurde eine Eintaktungsmöglichkeit mit der Markierung „inklusive“ beschrieben. Wie die Markierungen gehandhabt werden, wurde in Kapitel 2.4 beschrieben.

4.3 EINGABEFENSTER FÜR EINFACHE TÄTIGKEIT

Da die Überprüfung von Eintaktungsmöglichkeiten bei den Tätigkeiten sehr einfach ist, ist hier die Eingabemaske mit der Eintaktungsberechnung zusammengeführt.

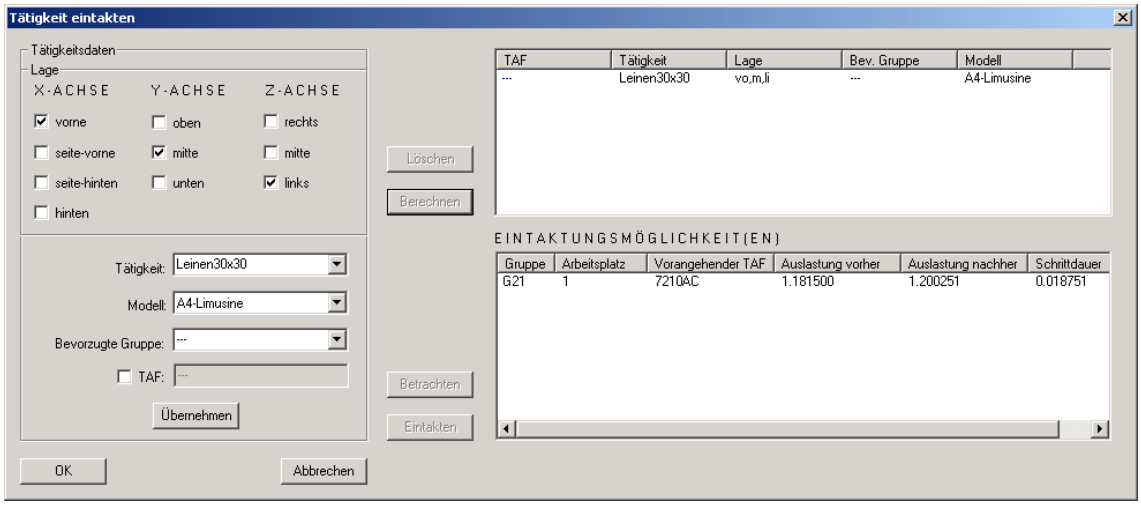


Abbildung 25: Fenster für einfache Tätigkeit

Auch an diesem Fenster sieht man dass nicht alle Buttons aktiviert sind. Alle Buttons in dem Programm werden immer nur dann aktiviert, wenn es einen Sinn macht. Hier kann eine Eintaktung in folgenden Schritten durchgeführt werden.

- Eingabe der notwendigen Informationen
- Übernehmen der Informationen in die Liste der Tätigkeiten, die eingetaktet werden sollen.
- Selektierung einer Tätigkeit in der Liste
  - aktiviert „Berechnen“ Button
- Nutzen des „Berechnen“ Buttons
  - Füllen der unteren Liste mit Eintaktungsvorschlägen
- Selektieren eines Eintaktungsvorschlags
  - aktiviert „Eintakten“ Button
- Durchführen der Eintaktung

Betätigt man den Button für die Eintaktung, so werden die Eintaktungsdaten vorher nochmals kurz angezeigt.

4.4 FENSTER FÜR EINTAKTUNG

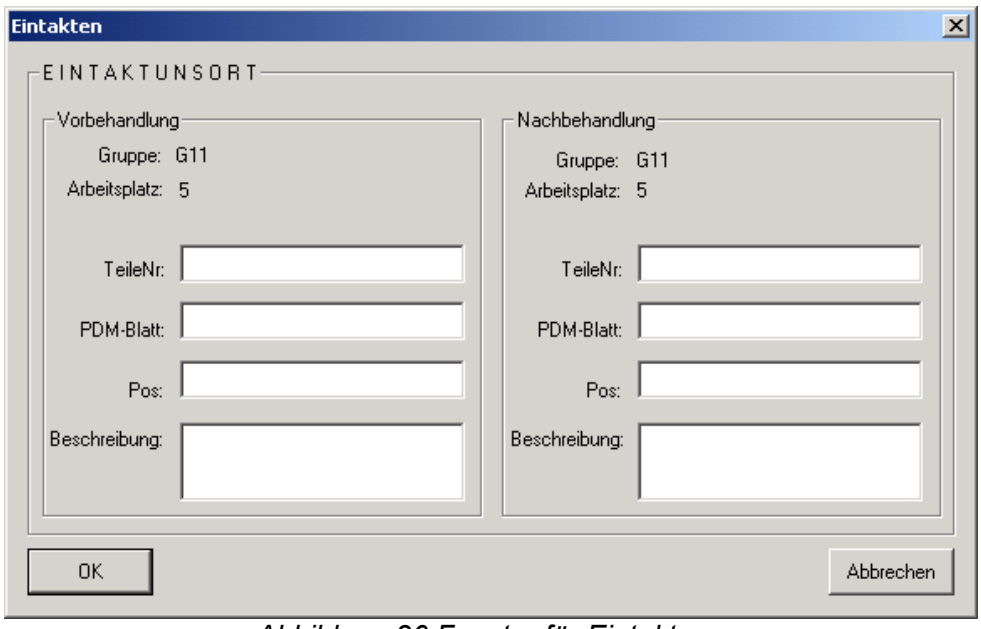


Abbildung 26:Fenster für Eintaktung

Will man eine Tätigkeit eintakten, so wird dieses Fenster angezeigt, bevor eine neue Tätigkeit endgültig eingetaktet wird. Hier hat man die Möglichkeit zu der Tätigkeit, die eingetaktet wird, weitere Informationen anzugeben.

4.5 FENSTER FÜR ADMINISTRATIVE TÄTIGKEITEN

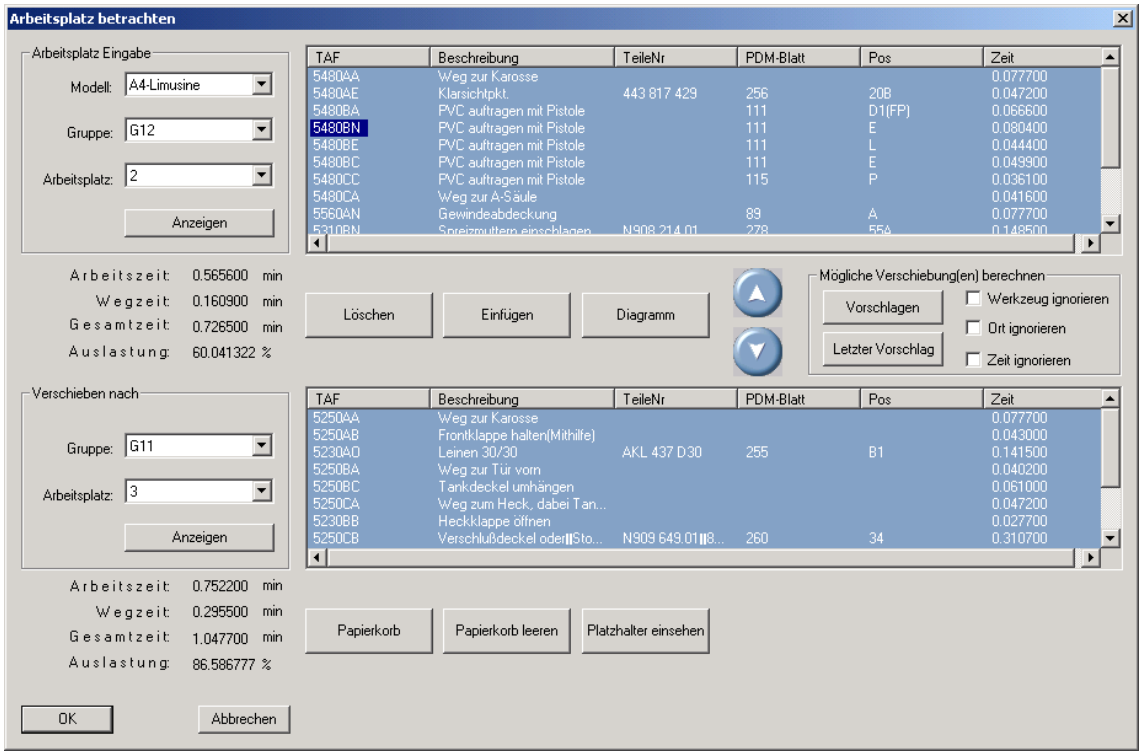


Abbildung 27: Fenster für administrative Tätigkeiten

Dieses Fenster ermöglicht eine Genaue Betrachtung von den Einzelnen Arbeitsplätzen. Bei dieser Betrachtung werden die Effektive Arbeitszeit, die Wegezeit, die Gesamtzeit und die Auslastung in Prozent des ausgewählten Arbeitsplatzes angezeigt.

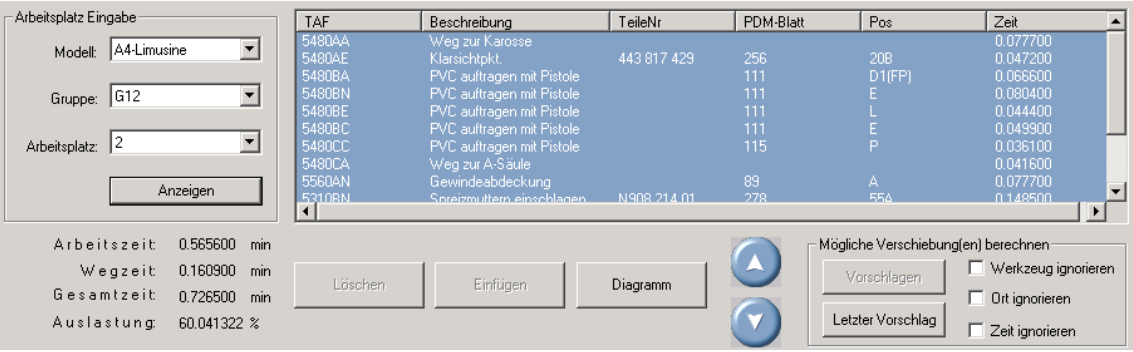


Abbildung 28: Betrachtung eines Arbeitsplatzes

Weiterhin kann man hier jeden Arbeitsschritt eines Werkers einsehen. Will man eine neue Tätigkeit bewusst an einen bestimmten Arbeitsplatz eintakten, so kann man diese hier von Hand eintakten, ohne vorher einen Vorschlag zu generieren.

Ein Überblick in Bezug auf alle Auslastungen der Arbeitsplätze, die in der Gruppe vorkommen, kann durch den Button „Diagramm“ gewonnen werden. Solch ein Diagramm ist wie folgt gestaltet.

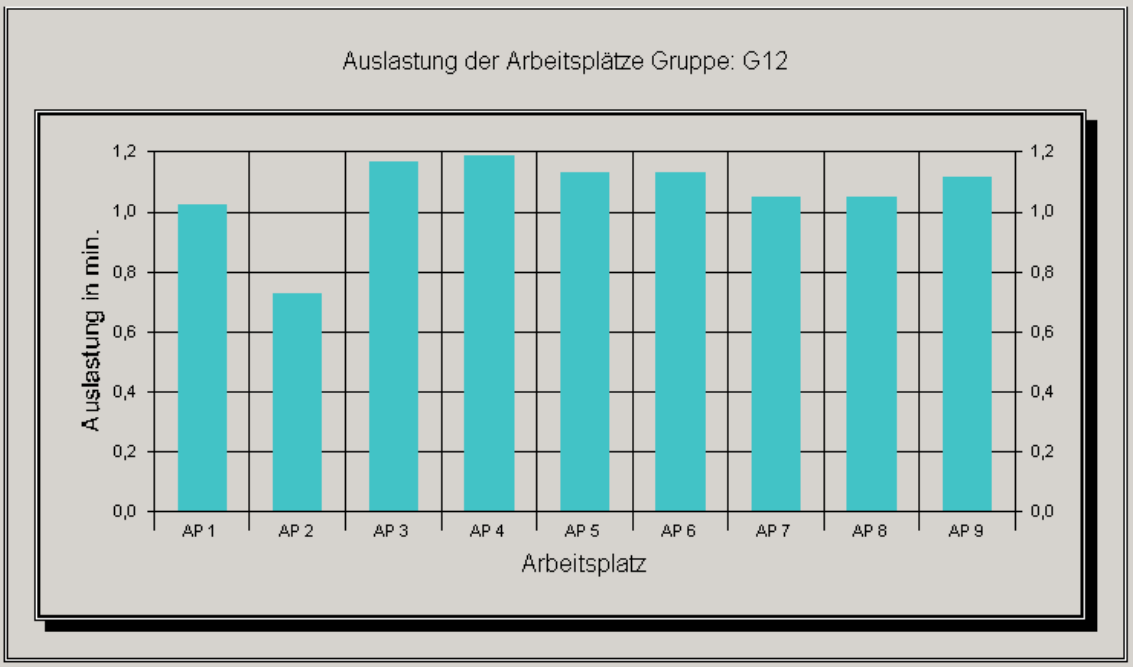


Abbildung 29: Diagramm zur Darstellung der Auslastungen

Wenn bei Umstrukturierungen der Produktionslinie die Arbeitsschritte bewusst verschoben werden müssen, kann man dies auch durch dieses Fenster realisierten. Hierzu wählt man in einer der beiden Listen den Zielarbeitsplatz

aus und in der anderen den Quellarbeitsplatz. Die Verschiebung wird durch das Klicken auf eines der beiden Pfeilbuttons ausgelöst, wobei die Pfeilrichtung die Verschiebungsrichtung angibt.

Für den Fall, dass ein Arbeitsschritt nicht mehr gebraucht wird, kann man diese mit Hilfe des Löschen Buttons entfernen. Da diese Tätigkeiten sich direkt auf die Abbildung der Produktionslinie auswirken, ist es nur befugten Benutzern erlaubt diese administrativen Tätigkeiten durchzuführen. Die Nutzung dieser Funktionen durch Unbefugte wird durch eine Passworteingabe verhindert.

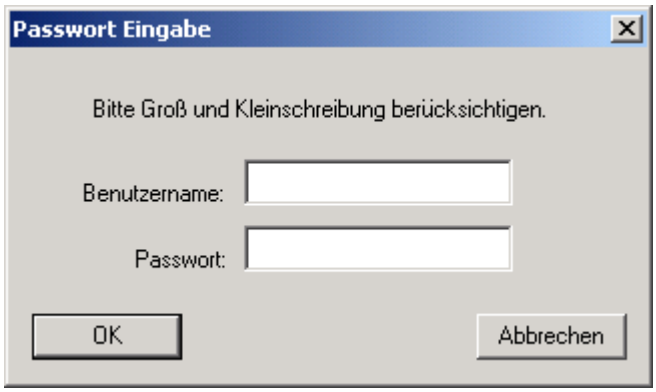


Abbildung 30:Passworteingabe

Die erlaubten Benutzernamen und Passwörter werden in der Datenbank gehalten. Will man einer neuen Person den Zugriff auf diese Funktionalitäten erlauben, so muss man für diese Person einen Benutzernamen und den dazugehörigen Passwort in folgende Tabelle eintragen.

Passwortliste

Name	Passwort
Name1	Passwort1
Name2	Passwort2
Name3	Passwort3

Tabelle 19: Passwortliste

Wird in dem Programm ein Benutzername und ein Passwort eingegeben, so überprüft das Programm ob der eingegebene Name in dieser Tabelle vorhanden ist. Falls der Name gefunden wird, so wird überprüft ob das Passwort übereinstimmt. Ist beides gegeben dürfen Veränderungen vorgenommen werden.

Werden durch einen Benutzer Datensätze gelöscht, so werden diese vorerst in einem Papierkorb gespeichert. Man hat dann die Möglichkeit den Inhalt des Papierkorbs einzusehen und Datensätze wieder in die Produktion einzutakten oder den gesamten Inhalt des Papierkorbs endgültig zu löschen.

4.6 INTELLIGENTES VERSCHIEBEN VON VORHANDENEN ARBEITSSCHRITTEN (TAFS)

Bisher wurde die automatische Eintaktung von neuen Tätigkeiten behandelt. Doch es kommt in der Praxis auch vor dass man schon bestehende Arbeitsschritte verschieben muss. Auch in diesem Fall stellt das Aussuchen von möglichen Verschiebungsorten von Hand, eine langwierige Aufgabe dar. Da zu den manuellen Tätigkeiten alle Informationen die für eine automatische Eintaktung vorhanden sein müssen, hinterlegt sind. Kann man dieselben Algorithmen nutzen um Vorschläge für mögliche Verschiebungsorte zu erhalten. Man kann sagen es wird eine Tätigkeit eingetaktet, dessen Daten nicht eingegeben werden müssen.

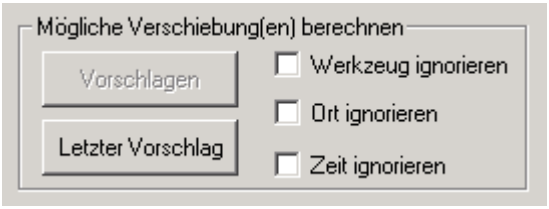


Abbildung 31: Intelligente Verschiebung

Da die Notwendigkeit des Verschiebens von vorhandenen Arbeitsschritten oft mit einer Umstrukturierung an der Produktionslinie zusammenhängt, gibt es die Möglichkeit, bewusst die Überprüfung von Parametern zu unterdrücken.

5 DIE MICROSOFT FOUNDATION CLASSES (MFC)

Die Microsoft Foundation Classes bildet ein Applikationsframework, das als ein Satz von C++ Klassen realisiert ist und Bestandteil des Visual C++ Compilers ist. Es unterstützt gezielt die GUI-Applikationsentwicklung auf der Windows Plattform. Innerhalb der MFC gibt es spezielle Klassen, die dazu dienen, OLE Softwarekomponenten zu entwickeln. Die besten Beispiele für Applikationen welche mit MFC entwickelt wurden, sind die im Microsoft Office Paket enthaltenen Standardanwendungen WinWord und Excel. Zu der MFC Bibliothek sind im Microsoft Visual C++ Compiler Wizards enthalten. Das sind visuelle Codegeneratoren, die zum einen Codegerüste für unterschiedliche Applikationsarten erstellen und zum anderen dabei helfen, das erstellte Codegerüst an die gewünschten Erfordernisse der Applikation anzupassen.

5.1 DIE EREIGNISSBEHANDLUNG

Während eine Applikation läuft, spielt sich der größte Teil des Kontrollflusses innerhalb des Codes des Frameworks ab. Das Framework behandelt die Ereignisverwaltung des GUI-Systems. Es erhält Ereignisse vom System, wenn der Benutzer Kommandos ausführt. Ein Teil der Ereignisse werden durch das Framework selbst behandelt, wie zum Beispiel das Schließen von Fenstern und das Beenden der Applikation. Ein Ereignis wird innerhalb der Applikation jeweils

an eine zu dem Ereignis passende virtuelle Ereignisbehandlungsmethode weitergeleitet, die zu einer Klasse der Applikation gehört. So stellt jede Klasse verschiedene Ereignisbehandlungsmethoden zur Verfügung. Der Entwickler hat während der Implementierung dieser Klassen die Möglichkeit, diese virtuelle Ereignisbehandlungsmethode zu erben und zu überschreiben um auf diese Weise auf ein bestimmtes Ereignis so zu reagieren, wie es in dem speziellen Fall vorgesehen ist.

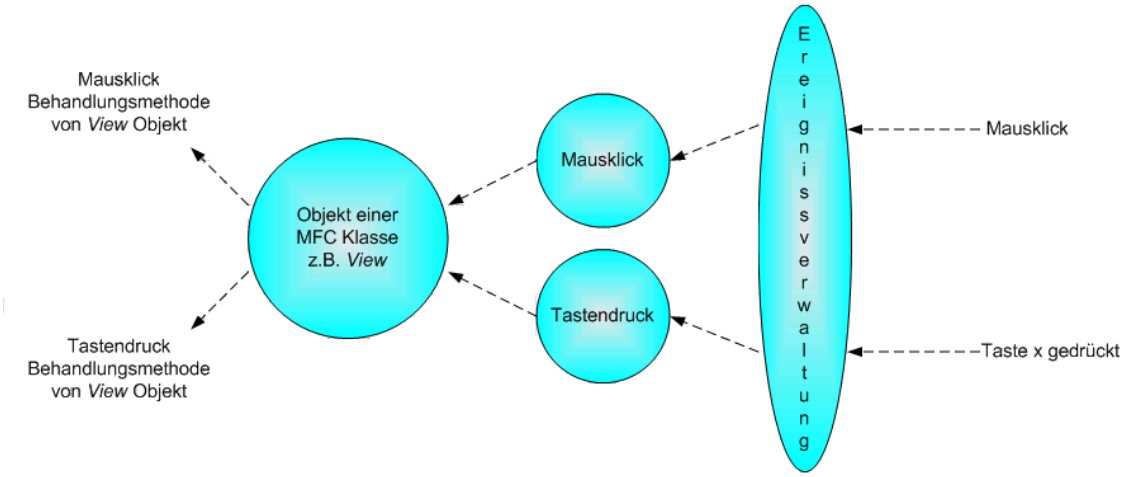


Abbildung 32:Ereignisbehandlung

Insgesamt ist die Ereignisbehandlung sehr an der GUI-Systemschicht orientiert. Wie in Abbildung 32 verdeutlicht, kapselt MFC eintreffende Ereignisse und ruft je nach Art des Ereignisses speziell implementierte Methoden der MFC Klassen mit den dazugehörenden Parametern auf. Die Ereignisbehandlungsmethoden selbst sind umständlich über Makrodefinitionen zu implementieren. Außerdem muss der Programmierer über die Ereignisbehandlung des Systems Bescheid wissen und die Schlüssel der einzelnen Ereignisarten kennen.

Für die Realisierung der automatischen Eintaktung, war es notwendig verschiedene Listen zu erstellen, Zeichenmanipulationen durchzuführen und auf Tabellen zuzugreifen. Um diese Probleme programmiertechnisch zu erleichtern, bietet die MFC eine Reihe von Hilfsklassen an, die im Folgenden genauer beschreiben werden.

5.2 MFC Hilfsklassen

Da die Hilfsklassen die Arbeit eines Entwicklers beschleunigen, werde ich hier die wichtigsten genauer beschreiben. Diese am häufigsten genutzten Hilfsklassen unterstützen die Bearbeitung von:

- Auflistungen
  - Feldklassen
  - Listenklassen
  - Tabellen
  - Benutzerdefinierte Auflistungsklassen

- Koordinaten
  - CPoint
  - CRect
  - CSize

- Zeit
  - COleDateTime
  - COleDateTimeSpan

- Zeichenfolgen
  - CString

5.2.1 Behandlung von Listen

In jedem objektorientierten Programm, ist es notwendig die Objekte gruppiert und in Auflistungen unterschiedlicher Typen und Größen zusammenfassend abzuspeichern.

Hier kommt die MFC zu Hilfe und stellt Gruppen einfach anzuwendender Klassen und Vorlagen bereit. Die Auflistungsklassen fallen in drei große Kategorien:

- Felder

- Listen

- Tabellen

Felder (oder Arrays) bilden die Hauptstütze der Auflistungsklassen und eignen sich für die Implementierung von Objektcontainern. Jedes Objekt in einem Array hat eine bei null beginnende Position, über die man sich auf die Objekte bezieht. Die Listenklasse organisiert Elemente in Form einer doppelt verketteten Liste, bei der die Daten sequentiell miteinander verbunden sind.

Listen bieten sich an, wenn man Elemente am Anfang oder am Ende der Liste schnell hinzufügen oder entfernen muss. Man kann die Liste auch vorwärts oder rückwärts von einem Element zum nächsten durchlaufen.

Bei Tabellen verknüpft man ein Schlüsselobjekt, etwa einen String oder eine Zahl, mit einem Wertobjekt, wo Verknüpfungen von Haus aus nur selten oder zufällig vorkommen. Beispielsweise kann man eine Tabelle verwenden, um Objekte mit Postleitzahlen zu verknüpfen. Tabellen eignen sich hervorragend bei schnellen Abfragen von Objekten, wenn der Verknüpfungsschlüssel gegeben ist, und können für die temporäre Speicherung von Daten bei großen Datenbanken verwendet werden.

5.2.1.1 FELDKLASSEN

MFC stellt mehrere vordefinierte Feldklassen und eine allgemeine Feldvorlage bereit, so dass man Felder erstellen kann, die eigene benutzerdefinierte Objekte aufnehmen. Verschiedene vordefinierte Feldklassen bieten einen schnellen und einfachen Zugriff auf gebräuchliche Typen von Variablen und Objekten.

Feldklasse	**Typen der gespeicherten Variablen**	**Numerischer Bereich des Typs**
CByteArray	`BYTE` - vorzeichenlose 8 Bit-Werte	0 bis 255
CWordArray	`WORD` - vorzeichenlose 16 Bit-Werte	0 bis 65535
CUIntArray	`UINT` - vorzeichenlose Ganzzahlen mit 32 Bit	0 bis 4 294 967 295
CDWordArray	`DWORD` - vorzeichenlose Ganzzahlen mit 32 Bit	0 bis 4 294 967 295
CStringArray	`CString` - Textobjekte	
CObArray	`CObject` - von `CObject` abgeleitete Objekte	
CPtrArray	`void*` - Objektzeiger oder Speicheradressen	

Tabelle 20: Vordefinierte Arraybasierte Klassen

Für jede Feldklasse gibt es mehrere Member-Funktionen, die sich nur durch den Typ der aufgenommenen Variablen unterscheiden. Jede hier vorgestellte Funktion lässt sich mit allen Feldklassen verwenden, um die Variablen des entsprechenden Typs zu behandeln.

Einer der nützlichsten Aspekte dieser Feldklassen ist deren Fähigkeit, dynamisch zu wachsen. Normale C/C++ Arrays sind in ihrer Größe vordefiniert und lassen sich nur durch umfangreiche Neuzuweisungen von Speicher erweitern. Die Auflistungsklassen verbergen diese Neuzuweisungen, so dass man einfach die Member-Funktion Add eines Feldobjekts aufrufen kann, um einen neuen Wert hinzuzufügen. Will man beispielsweise Strings in ein CStringArray aufnehmen, verwendet man etwa folgenden Code:

```
CStringArray Werkzeuge;
Werkzeuge.Add("Flachpinsel");
Werkzeuge.Add("Hammer");
Werkzeuge.Add("Zange");
```

Die Größe eines Arrays lässt sich mit der Funktion GetSize ermitteln. Führt man im Anschluss an die obigen Zeilen die folgende Zeile aus, erhält man in nNumberOfItems drei Elemente zurück:

```
int nNumberOfItems = Werkzeuge.GetSize();
```

Ein Array kann man mit der korrespondierenden Funktion SetSize auch auf eine bestimmte Größe einstellen. Dabei wird das Array entweder abgeschnitten oder erweitert, um der übergebenen Größe zu entsprechen.

Mit der Funktion SetAt, der man einen bei null beginnenden Index und den zu speichernden Wert übergibt, kann man Werte in das Array eintragen. Man muss sicherstellen, dass der Index innerhalb der Feldgrenzen liegt, da SetAt ansonsten einen Assertion-Fehler auslöst.

Mit der Funktion GetAt kann man Werte aus dem Array abrufen. Die Funktion liefert den Wert an der angegebenen Indexposition zurück. Bei einem CWordArray setzt man diese Funktionen zum Beispiel folgendermaßen ein:

```
CWordArray myWordArray;
myWordArray.SetSize(20);
myWordArray.SetAt(0,200);
myWordArray.SetAt(19,500);
TRACE("Der Wert an Indexposition 19 lautet %d\n",
myWordArray.GetAt(19));
```

Diese Zeilen setzen das erste Element eines 20elementigen Arrays auf 200 und das letzte auf 500. Die letzte Zeile zeigt dann das letzte Element, den Wert 500, an. Mit der Funktion Add lässt sich das Array weiter vergrößern. Den größten gültigen Index ermittelt man mit der Funktion GetUpperBound. Diese Funktion liefert einen bei null beginnenden Index zurück, oder -1, wenn keine Elemente vorhanden sind.

Mit dem Indexoperator [ ] kann man Werte an einer bestimmten Indexposition wie bei einem normalen C++-Array setzen und abrufen. Beispielsweise lassen sich die Funktionen GetAt und SetAt in den obigen Zeilen folgendermaßen durch den Indexoperator ersetzen:

```
myWordArray[0] = 200;
myWordArray[19] = 500;
TRACE("Der Wert an Indexposition 19 lautet %d\n",
myWordArray.GetAt[19]);
```

Mit den Funktionen InsertAt und RemoveAt kann man Elemente an einer bestimmten Position einfügen bzw. entfernen. Dabei werden alle Elemente ab dieser Position um ein oder mehrere Elemente nach oben bzw. unten verschoben.

Die Funktion InsertAt hat zwei Formen:

- Die erste benötigt eine Indexposition und das an dieser Stelle einzufügende Element. Optional kann man eine Anzahl übergeben, um mehrere Kopien des angegebenen Elements einzufügen.

- Die zweite Form erlaubt es, ein anderes komplettes Array an der angegebenen Indexposition einzufügen.

Die Funktion RemoveAt braucht nur einen Parameter, den Index des zu entfernenden Elements. Man kann auch hier einen optionalen zweiten Parameter übergeben, um eine entsprechende Anzahl von Elementen zu entfernen. Die verbleibenden Feldelemente werden dann nach unten verschoben, um die Lücke aufzufüllen. Alle Elemente lassen sich aus einem Array mit der Funktion RemoveAll entfernen.

5.2.1.2 LISTENKLASSEN

Wie Tabelle 21 zeigt, gibt es nur drei Kategorien von Listen und eine Vorlage für selbstentwickelte Typen. Listen mit einfachen Ganzzahlen benötigt man nur selten. Stattdessen benötigt man oft verkettete Listen, in denen man die von CObject abgeleiteten Klassen, Zeiger auf C++- Klassen oder Strukturen verwaltet.

Klassenname	**Typ der gespeicherten Variablen**
CObList	`CObject` - Zeiger auf von `CObject` abgeleitete Objekte.
CPtrList	`void*` - Zeiger auf Speicheradressen, die beliebige Daten aufnehmen.
CStringList	`CString` - Zeichenfolgen.

Tabelle 21: Auflistungsklassen für Listen

Bei verketteten Listen sind mehrere Objekte miteinander in sequentieller Art verkoppelt. Es gibt eine eindeutige Anfangs- und Endposition, aber jedes andere Element kennt nur seinen unmittelbaren Nachbarn. Eine „Positionsvariable“ protokolliert die aktuelle Position in einer Liste. Man kann mehrere „Positionvariablen“ deklarieren, um verschiedene Stellen in derselben Liste zu verfolgen. Die Member-Funktionen der Listen verwenden dann eine Positionsvariable, um den Anfang, das Ende oder das nächste bzw. vorherige Element in der Liste zu ermitteln.

Mit den Funktionen AddHead und AddTail kann man Elemente in eine Liste am Anfang bzw. am Ende hinzufügen oder mit den Funktionen InsertBefore und InsertAfter vor bzw. nach einer bestimmten Position einfügen. Alle Funktionen liefern dann einen Positionswert zurück, der die Position des neu hinzugefügten Elements angibt. Das folgende Beispiel konstruiert eine vierelementige Liste von CString Elementen:

```
CStringList listMyStrings;
POSITION pos;
pos = listMyStrings.AddHead("Hammer");
listMyStrings.AddTail("Pinsel");
listMyStrings.InsertBefore(pos, "Wischtuch");
listMyStrings.AddTail("Zange");
```

Diese Codezeilen liefern eine verknüpfte Liste von CStrings, die von Anfang bis Ende folgendes Aussehen hat:

```
Wischtuch-Hammer-Pinsel-Zange
```

Man kann auch andere gleichartige Listenobjekte an die Funktionen AddHead und AddTail übergeben, um eine weitere Liste am Beginn oder Ende der aktuellen Liste hinzuzufügen.

Nachdem eine Liste aufgebaut ist, kann man mit Hilfe einer Positionsvariablen durch die Elemente der Liste navigieren. Die Anfangs- oder Endposition lässt sich mit den Funktionen GetHeadPosition bzw. GetTailPosition ermitteln. Diese Funktionen geben einen Positionswert zurück, der die aktuelle Position in der Liste kennzeichnet. Dann kann man die Positionsvariable als Referenz an die Funktionen GetNext oder GetPrev übergeben, um das nächste bzw. vorherige Element in der Liste zu erhalten. Diese Funktionen liefern dann das betreffende Objekt zurück und passen die aktuelle Position an. Wenn das Ende der Liste erreicht ist, wird die Variable „POSITION“ auf NULL gesetzt.

Die Zeilen im folgenden Beispiel durchlaufen die oben erzeugte listMyStrings und zeigen die Elemente nacheinander an:

```
POSITION posCurrent = listMyStrings.GetHeadPosition();
while(posCurrent)
 TRACE("%s\n",listMyStrings.GetNext(posCurrent);
```

Bestimmte Listenelemente lassen sich mit der Funktion Find aufsuchen, die einen Positionswert zurückliefert, wenn der übergebene Suchparameter gefunden wurde. Optional können Sie einen Positionswert übergeben, von dem aus die Suche beginnen soll. Um zum Beispiel nach dem String Zange der oben angelegten Liste zu suchen, ruft man die Funktion Find wie folgt auf:

```
POSITION posFinger = Find("Zange");
```

Diese Funktion liefert einen Null-Wert zurück, wenn das gesuchte Element nicht vorhanden ist. Mit der Funktion FindIndex kann man das **n**´te Element vom Beginn der Liste an suchen, wobei man **n** als Parameter übergibt. Die Anzahl der Elemente in der Liste liefert die Member-Funktion GetCount. Die Funktion benötigt keine Parameter und gibt die Anzahl der Elemente zurück. Der Wert von Elementen an einer bestimmten Position lässt sich mit den Funktionen GetAt und SetAt abrufen bzw. zurücksetzen. Die Funktionen setzt man ähnlich ein wie die äquivalenten Feldfunktionen, übergibt aber einen Positionswert statt eines Feldindexes.

Mit der Funktion RemoveAt entfernt man Elemente aus der Liste. Der Funktion übergibt man den Positionswert, um das zu entfernende Element zu kennzeichnen. Mit dem folgenden Code löschen Sie zum Beispiel den Eintrag Finger aus der Liste des obigen Beispiels:

```
RemoveAt(posFinger);
```

Microsoft hat klar erkannt, mit welchen Datenstrukturen oft gearbeitet wird, und hat diese Datentypen mit einer ausreichenden Funktionalität bereitgestellt. Eine weitere Form von Datenaufbereitung sind die Tabellen.

5.2.1.3 TABELLENKLASSEN

Tabellenklassen verbinden einen Typenwert bzw. Element mit einem Schlüsselwert, womit man nach dem Element suchen kann. Die verschiedenen Tabellenklassen sowie deren Schlüsselwerte und zugeordneten Elementtypen sind in Tabelle 22 aufgeführt.

Klassenname	Schlüsseltyp	Elementtyp
CMapWordToOb	`WORD` - vorzeichenloser 16-Bit-Wert	`CObject` - von `CObject` abgeleitete Objekte
CMapWordToPtr	`WORD` - vorzeichenloser 16-Bit-Wert	`void*` - Zeiger auf Speicher
CMapPtrToPtr	`void*` - Zeiger auf Speicher	`void*` - Zeiger auf Speicher
CMapPtrToWord	`void*` - Zeiger auf Speicher	`WORD` - vorzeichenloser 16-Bit-Wert
CMapStringToOb	`CString` - Zeichenfolgen	`CObject` - von `CObject` abgeleitete Objekte
CMapStringToPtr	`CString` - Zeichenfolgen	`void*` - Zeiger auf Speicher
CMapStringToString	`CString` - Zeichenfolgen	`CString` - Textstrings

Tabelle 22: Auflistungsklassen für Tabellen

In eine Tabelle kann man mit der Funktion SetAt Elemente einfügen. Der Funktion übergibt man einen Schlüsselwert als ersten Parameter und den Wert des Elementes, welches hinzugefügt werden soll, als zweiten. Will man zum Beispiel von CObject abgeleitete Objekte, die mit einem Zeichenfolgenwert indiziert sind speichern, kann man die Klasse CMapStringToOb verwenden und Elemente wie folgt hinzufügen:

```
CMapStringToOb mapPlanetDetails;
mapPlanetDetails.SetAt("Merkur", new CPlanetDets (487));
```

In diesem Beispiel ist CPlanetDets eine von CObject abgeleitete Klasse mit einem Konstruktor, der einen Detailparameter für Planeten übernimmt. Den

neuen Objekten sind dann die Planetennamen als Schlüssel zugeordnet. Statt der Funktion SetAt kann man auch den Indexoperator [ ] verwenden, indem man den Schlüsselwert in den eckigen Klammern angibt:

```
mapPlanetDetails["Mars"] = new CPlanetDets (6762);
```

Nachdem die Daten in eine Tabelle eingetragen sind, kann man sie mit der Member-Funktion Lookup wieder abrufen. Der Funktion übergibt man den Schlüsselwert als Referenz auf eine Variable, die das zugehörige Element aufnimmt, falls es existiert. Ist das Element nicht vorhanden, liefert die Funktion Lookup den Wert „FALSE“ zurück. Um zum Beispiel die Angaben über einen Planet aus dem vorherigen Beispiel abzurufen, kann man etwa folgende Zeile verwenden:

```
CPlanetDets* pMyPlanet = NULL;
if (mapPlanetDetails.Lookup("Erde", (CObject*&)pMyPlanet))
TRACE("Umlaufzeit = %d Tage\n", pMyPlanet->m_dSidereal);
```

Die Typumwandlung (CObject*&) dient dazu, den Objektzeiger pMyPlanet in eine allgemeine Zeigerreferenz auf CObject umzuwandeln. Die Funktion GetCount liefert die Anzahl der momentan in der Tabelle vorhandenen Elemente zurück. Die Elemente lassen sich durch Aufruf der Funktion RemoveKey entfernen. Der Funktion übergibt man den Schlüssel des zu entfernenden Elements:

```
mapPlanetDetails.RemoveKey("Jupiter");
```

Man darf nicht vergessen die reservierten Objekte zu löschen. RemoveKey entfernt lediglich den Zeiger auf das Objekt, und nicht das Objekt selbst, und gibt den verwendeten Speicher nicht frei. Durch Aufruf der Funktion RemoveAll kann man alle Elemente auf einmal entfernen.

Mit Hilfe der Funktion GetNextAssoc kann man die Liste der Zuordnungen durchlaufen. Die Funktion benötigt Parameter, die eine Variable mit der aktuellen Position referenzieren, eine Schlüsselvariable und eine Elementvariable. Die Position des ersten Elements ermittelt man mit der Funktion GetFirstPosition. Die Funktion liefert den Positionswert für das erste Element zurück. Um durch die Zuordnungen zu gehen, kann man etwa den folgenden Code verwenden:

```
POSITION pos = mapPlanetDetails.GetStartPosition();
while(pos!=NULL)
{
      CString strPlanet;
      CPlanet* pMyPlanet;
      mapPlanetDetails.GetNextAssoc(pos, strPlanet,
      CObject*&)pMyPlanet);
      TRACE("%s hat einen Durchmesser von %d km\n", strPlanet,
      pMyPlanet->m_dDiameter);
}
```

Nach Rückkehr aus GetNextAssoc enthält „pos“ die Position für die nächste Zuordnung oder NULL, wenn es keine Zuordnungen mehr gibt. Die Schlüssel- und Elementwerte (strPlanet und pMyPlanet im obigen Beispiel) werden nacheinander auf das jeweilige Schlüssel/Element-Paar gesetzt.

Durch die Fähigkeit der Tabelle, schwach besetzte Datenbestände schnell und effizient abzurufen, ist es oftmals von Vorteil, eine Tabelle als Zwischenspeicher bei einer langsamen Datenbanksuche zu verwenden.

Beispielsweise sind in den folgenden Zeilen die mit strPlanetName verbundenen Angaben erforderlich. Beim ersten Aufruf verfügt dieser Code noch nicht über eine abgebildete Version des angeforderten Planeten, so dass man ihn mit GetPlanetFromSlowDB suchen muss. Da der Code dann den abgerufenen Planeten in der Tabelle mapPlanetDetails speichert, lassen sich beim nächsten Aufruf mit demselben strPlanetName die Angaben schnell aus der zwischengespeicherten Version abrufen:

```
CPlanetDets* pMyPlanet = NULL;
if (mapPlanetDetails.Lookup(strPlanetName, (CObject*&)pMyPlanet)
== FALSE)
{
      pMyPlanet = GetPlanetFromSlowDB(strPlanetName);
      mapPlanetDetails.SetAt(strPlanetName,pMyPlanet);
}
return pMyPlanet;
```

Dieses Verfahren ist leicht zu implementieren und kann die Geschwindigkeit der Anwendung erhöhen, wenn man mit langsamen Abfragegeräten wie etwa Datenbanken oder Dateien arbeitet.

5.2.1.4 BENUTZERDEFINIERTE AUFLISTUNGSKLASSEN

Es kommt oft vor, dass man die Auflistungsklassen an eigene Objekte anpassen will, um so die Funktionalität der Auflistungsklassen für das eigene Objekt nutzen zu können.

Die Anpassung bietet mehrere Vorteile, da man ein Feld, eine Liste oder eine Tabelle erstellen kann, die nur den speziellen Typ des Objekts akzeptiert und zurückgibt. Wenn man versehentlich versucht, die falsche Objektart in ein benutzerdefiniertes Feld, eine Liste oder eine Tabelle hinzuzufügen, löst der Compiler eine Fehlermeldung aus. Ein weiterer Vorteil besteht darin, dass man keine Typumwandlungen von allgemeinen Zeigern auf CObject* (das heißt, von einem CObArray) zurück auf das verwendete Objekt vornehmen muss.

Diese Art der Anpassung bezeichnet man als Typsicherheit. In großen Programmen kann es unschätzbar sein, bei versehentlichen Zuweisungen der falschen Klasse zu stoppen. Mit einer Gruppe von Vorlagen, CArray, CList und CMap, kann man in einfacher Weise ein Array, eine Liste oder eine Tabelle

erstellen, um Objekte nur des spezifizierten Typs zu speichern, zu verwenden und zurückzugeben.

Vorlagen (Templates) sind ein kompliziertes Thema, aber Sie brauchen Vorlagen nicht selbst zu schreiben. Die in der Header-Datei afxtempl.h definierten und von MFC bereitgestellten Vorlagen genügen für diese typsicheren Auflistungsklassen. In Bezug auf den vorliegenden Abschnitt stellt man sich Templates einfach als große Makros vor, die beim Kompilieren auf der Basis Ihrer Parameter eine Menge Code generieren.

Die Vorlagen bieten den Zugriff auf alle normalen Funktionen in den Feld-, Listen- oder Tabellenklassen, die in den vorherigen Abschnitten behandelt wurden. Statt allerdings die auf dem allgemeinen Objekt *CObject* basierenden Parameter und Rückgabewerte zu verwenden, können Sie Ihre eigenen Typen als Parameter und Rückgabewerte definieren.

Um die Vorlagen in dem Programm einzusetzen, muss man folgende folgende Header- Zeile in alle Module (.cpp/.h-Dateien) einbinden, die auf die Vorlagendefinitionen zurückgreifen:

```
#include "afxtempl.h"
```

Eine benutzerdefinierte, typsichere Klasse kann gemäß der folgenden Vorlagensyntax für ein Array von benutzerdefinierten Objekten implementiert werden:

```
CArray <CMyCustomClass*, CMyCustomClass*> myArray;
```

Die Symbole „<“ und „>“ in der obigen Definition sind als spitze Klammern zu interpretieren. Die obige Zeile verwendet die Vorlage CArray, um eine Instanz von myArray zu erzeugen. Der erste Parameter CMyCustomClass* spezifiziert Typen von Objektzeigern, die das Array zurückgeben soll, wenn man mit GetAt und anderen Zugriffsfunktionen arbeitet.

Der zweite Parameter CMyCustomClass* legt den Typ fest, der für die Definitionen der Eingabeparameter zu verwenden ist. Somit akzeptieren alle Funktionen, die Objekte speichern, wie etwa SetAt und Add, nur Zeiger auf Objekte der speziellen CMyCustomClass. Zum Beispiel kann man ein Array erzeugen, das nur Zeiger auf die spezielle Klasse CPlanetDets übernimmt und zurückgibt. Die Klasse ist wie folgt definiert:

```
class CPlanetDets : public CObject
{
public:
      CPlanetDets(double dDiameter, double dGravity):
      m_dDiameter(dDiameter),
      m_dGravity(dGravity) {}
      double m_dDiameter,
      double m_dGravity,
};
```

Um ein typsicheres auf CArray basierendes Array namens myPlanetArray zu deklarieren, kann man dann folgenden Code schreiben:

```
CArray<CPlanetDets*, CPlanetDets*> myPArray;
```

Diese Zeile deklariert, dass myPArray nur Zeiger auf ein CPlanetDets-Objekt akzeptiert und Zeiger auf ein CPlanetDets-Objekt zurückgibt.

Das neue Array kann man dann wie folgt einsetzen:

```
myPArray.Add(new CPlanetDets (4878, 0.054  ));
myPArray.Add(new CPlanetDets (12100, 0.815 ));
myPArray.Add(new CPlanetDets (12756, 1.000 ));

for(int i=0; i<myPArray.GetSize(); i++)
      TRACE("Durchm = %f\n", myPArray[i]->m_dDiameter);
```

Diese Zeilen erzeugen drei neue Objekte vom Typ CPlanetDets und fügen sie in das Array ein. Die letzte Zeile zeigt den Durchmesser im Makro TRACE an, ohne dass man den Typ des Rückgabewerts von myPlanetArray[i] umwandeln muss, da es sich bereits um einen Zeiger auf den Typ CPlanetDets* handelt.

Falls man im Zuge der Programmentwicklung die genaue Natur von myPArray vergisst, kann es vorkommen, dass man versucht einen CStatic-Objekt hinzuzufügen:

```
myPlanetArray.Add(new CStatic());
```

In diesem Fall bemerkt der Compiler diese Übertretung und löst folgenden Compiler-Fehler aus:

```
... error C2664: 'Add' : Konvertierung des Parameters 1 von
'class CStatic *' in 'class CPlanetDets *' nicht moeglich
```

Der Fehler bliebe jedoch unbemerkt, wenn man ein CObArray verwendet hätte, um die Planetendaten zu speichern:

```
CObArray myPArray;
```

Das CStatic-Objekt lässt sich hier ohne weiteres zusammen mit den CPlanetDets-Objekten speichern, bewirkt aber Unvorhersehbares, wenn man versucht, das CStatic- Objekt abzurufen und annimmt, dass es sich um ein CPlanetDets-Objekt handelt.

Die Vorlage zur Generierung von typsicheren Listen ist *CList*. Sie weist die gleiche allgemeine Form wie *CArray* auf:

```
CList<CMyCustomClass*, CMyCustomClass *> myCustomList;
```

Auch hier ist der erste Parameter der erforderliche Rückgabetyp des Objekts. Der zweite Parameter spezifiziert die akzeptierten Objekttypen für Funktionen, die Elemente für die Speicherung akzeptieren.

Alle Funktionen, die es für Listen gibt, sind auch für eigene spezielle typsichere Listen verfügbar. Auch diese Funktionen prüfen und geben die spezifizierten Typen zurück.

Demzufolge sieht der äquivalente Code mit einer Listenklasse für das Speichern der Planetendaten etwa folgendermaßen aus:

```
CList<CPlanetDets*,CPlanetDets*> myPList;

myPList.AddTail(new CPlanetDets (4878, 0.054  ));
myPList.AddTail(new CPlanetDets (12100, 0.815 ));
myPList.AddTail(new CPlanetDets (12756, 1.000 ));

POSITION pos = myPList.GetHeadPosition();
while(pos)
      TRACE("Durchmesser = %f\n",
      myPList.GetNext(pos)->m_dDiameter);
```

Die Vorlage für benutzerdefinierte Tabellen unterscheidet sich von Listen und Arrays darin, dass vier Parameter erforderlich sind. Es sind je ein Eingabe- und ein Rückgabewert sowohl für den Schlüssel als auch den Elementwert notwendig. Die allgemeine Form lautet damit:

```
CMap<MyType, MyArgType, CMyCustomClass *, CMyCustomClassArg *>
myCustomMap;
```

Der erste Parameter, MyType, legt den intern gespeicherten Schlüsselwert für jede Tabellenzuordnung fest. Das kann einer der Basistypen wie int, WORD, DWORD, double, float oder CString sein oder ein Zeiger auf einen eigenen speziellen Typ. Der zweite Parameter, MyArgType, spezifiziert den Argumenttyp, der bei der Übergabe der Schlüsselwerte in und aus Tabellenfunktionen zu verwenden ist. Mit dem dritten Parameter, CMyCustomClasss *, legt man fest, wie die internen Elementwerte zu speichern sind. Der vierte Parameter, CMyCustomClassArg *, spezifiziert den Argumenttyp, der für die Übergabe der Elementwerte in und aus den Tabellenfunktionen zu verwenden ist.

Um beispielsweise die Daten der Planeten mit deren Namen zu verbinden, könnte man folgendes kodieren:

```
CMap<CString, LPCSTR, CPlanetDets*, CPlanetDets*> myPMap;

myPMap.SetAt("Merkur", new CPlanetDets(4878, 0.054));
myPMap.SetAt("Venus", new CPlanetDets(12100, 0.815));
myPMap.SetAt("Erde", new CPlanetDets(12756, 1.000));
CPlanetDets* pPlanet = NULL;
```

```
if (myPlanetMap.Lookup("Venus", pPlanet))
      TRACE("Durchmesser = %f\n", pPlanet->m_dDiameter);
```

Die Tabellendeklaration gibt an, dass die Objekte intern als CStrings zu speichern sind, aber LPCSTR (Zeiger auf konstante Zeichenarrays) als Übergabewerte in und aus der Tabelle zu verwenden sind. Die Angaben zu den Planeten selbst werden sowohl intern gespeichert als auch als Zeiger auf CPlanetDets-Objekte zugänglich gemacht.

5.2.2 Behandlung von Koordinaten

Da Windows eine grafisch orientierte Umgebung darstellt, muss man oftmals Punktpositionen, Rechtecke und Größen speichern. Drei MFC-Klassen unterstützen das Speichern und Manipulieren dieser Koordinaten:

- `CPoint`
- `CRect`
- `CSize`

Jede Klasse verfügt über mehrere Elementfunktionen und überladene Operatoren, die den größten Teil der Arbeit beim Addieren, Konstruieren und Finden von Ableitungen dieser Koordinaten realisieren. Darüber hinaus verstehen verschiedene MFC- und GDI-Funktionen diese Typen oder die zugrundeliegenden Typen als Parameterwerte, so dass man keine umständlichen Operationen auszuführen hat, um Werte an diese Funktionen zu übergeben.

5.2.2.1 Die Klasse CPoint

CPoint kapselt eine POINT-Struktur, die lediglich eine x- und eine y-Position speichert, um einen Punkt auf einer zweidimensionalen Oberfläche darzustellen. Auf die Elemente x und y kann man immer direkt zugreifen, um deren aktuelle Werte zu erhalten oder zu setzen:

```
CPoint ptEins;
ptEins.x = 5;
ptEins.y = 20;
TRACE("Koordinate = (%d, %d)\n", ptEins.x, ptEins.y);
```

Diese Werte setzt man, wenn man ein CPoint-Objekt konstruiert, indem man die Werte an einen der verschiedenen Konstruktoren von CPoint übergibt, wie sie Tabelle 23 zeigt.

Konstruktordefinition	Beschreibung

CPoint()	Konstruiert ein nicht initialisiertes Objekt.
CPoint(POINT ptInit)	Kopiert die Einstellungen aus einer `POINT`-Struktur oder einem anderen `CPoint`-Objekt.
CPoint(int x, int y)	Initialisiert das Objekt mit den Parameterwerten von `x` und `y`.
CPoint(DWORD dwInit)	Verwendet die unteren 16 Bit für den `x`-Wert und die höherwertigen 16 Bit für den `y`-Wert.
CPoint(SIZE sizeInit)	Kopiert die Einstellungen aus einer `SIZE`-Struktur oder einem `CSize`-Objekt.

Tabelle 23: Konstruktortypen für die Klasse CPoint

Zum Beispiel kann man die letzten Beispielzeilen durch die folgenden ersetzen und das gleiche Ergebnis erzielen:

```
CPoint ptEins(5,20);
TRACE("Koordinate = (%d, %d)\n", ptEins.x, ptEins.y);
```

Einer der nützlichsten Aspekte der Klasse CPoint sind die vielen überladenen Operatoren. Indem man die Operatoren +, -, += und -= mit anderen CPoint-, CRect- oder CSize-Objekten verwendet, kann man Koordinatenpaare zu/von anderen Koordinatenpaaren, Rechtecken oder Größen addieren/subtrahieren. Der längere herkömmliche Weg, um zwei Punkte voneinander zu subtrahieren, um einen dritten zu erhalten, sieht zum Beispiel folgendermaßen aus:

```
CPoint ptEins(5,20);
CPoint ptZwei(25,40);
CPoint ptDrei;
ptDrei.x = ptZwei.x - ptEins.x;
```

Das Ganze lässt sich mit den überladenen Operatoren folgendermaßen vereinfachen:

```
CPoint ptEins(5,20);
CPoint ptZwei(25,40);
CPoint ptDrei = ptZwei - ptEins;
```

Mit den überladenen logischen Operatoren == und != lassen sich auch Vergleiche ausführen. Um zum Beispiel zu prüfen, ob ptZwei sowohl im x- als auch im y-Wert mit ptEins gleich ist, schreibt man folgenden Code:

```
if(ptEins == ptZwei)
  TRACE("Die Punkte sind identisch.");
```

Die Funktion Offset addiert einen Verschiebungswert, der durch die übergebenen x- und y-Werte, eine CPoint-Klasse, eine POINT-Struktur, eine CSize-Klasse oder eine SIZE-Struktur spezifiziert ist. Demzufolge sind die beiden folgenden Zeilen funktionell identisch:

```
ptEins.Offset(75, -15);
ptEins-=CPoint(-75, 15);
```

Neben der Behandlung von Koordinatenpunkten, ist auch der Umgang mit Rechtecken häufig notwendig.

5.2.2.2 Die Klasse CRect

Die Klasse CRect kapselt eine RECT-Struktur, um zwei Koordinatenpaare aufzunehmen, die ein Rechteck durch die Punkte der linken oberen und unteren rechten Ecke beschreiben.

Ein CRect-Objekt können Sie mit mehreren Konstruktoren erstellen, wie sie Tabelle 24 zeigt.

Konstruktordefinition	Beschreibung
CRect()	Konstruiert ein nicht initialisiertes Objekt.
CRect(const RECT& rcInit)	Kopiert die Einstellungen aus einer anderen `RECT`-Struktur oder einem `CRect`-Objekt.
CRect(LPRECT lprcInit)	Kopiert die Einstellungen über einen `RECT`- oder `CRect`-Zeiger.
CRect(int l, int t, int r, int b)	Initialisiert die Koordinaten des linken, oberen, rechten und unteren Parameters.
CRect(POINT point, SIZE size)	Initialisiert von einer `POINT`-Struktur oder einem `CPoint`-Objekt und einer `SIZE`-Struktur oder einem `CSize`-Objekt.
CRect(POINT ptTL, POINT ptBR)	Initialisiert von einem POINT links oben und einem POINT rechts unten.

Tabelle 24: Konstruktortypen für die Klasse CRect

Nachdem ein CRect-Objekt konstruiert ist, kann man einzeln auf die Elemente top, left, bottom und right zugreifen, indem man die Typumwandlung (LPRECT) verwendet, um die Werte in eine RECT-Struktur wie in den folgenden Zeilen umzuwandeln:

```
CRect rcEins(15,15,25,20);
((LPRECT)rcEins)->bottom += 20;
TRACE("Rechteck: (%d,%d)-(%d,%d)",
((LPRECT)rcEins)->left,((LPRECT)rcEins)->top,
((LPRECT)rcEins)->right,((LPRECT)rcEins)->bottom);
```

Alternativ kann man auf die Elemente entweder über den CPoint für oben links oder den CPoint für unten rechts zugreifen. Die Funktionen TopLeft und BottomRight liefern Referenzen auf diese Elementobjekte zurück. Wenn man entweder auf den Punkt oben links oder unten rechts zugreift, kann man die Punkte dann mit einer der CPoint-Funktionen aus dem vorherigen Abschnitt manipulieren.

Beispielsweise sind die folgenden Zeilen funktionell zu den vorherigen identisch, unterscheiden sich aber darin, dass sie das Rechteck mit CPoint-Objekten konstruieren und darauf zugreifen:

```
CRect rcEins(CPoint(15,15),CPoint(25,20));
rcEins.BottomRight().y += 20;
TRACE("Rechteck: (%d,%d)-(%d,%d)",
rcOne.TopLeft().x,rcOne.TopLeft().y,
rcOne.BottomRight().x,rcOne.BottomRight().y);
```

Mit der Funktion SetRect kann man auch die Koordinaten setzen, wobei man vier Integer-Werte für die x- und y-Koordinaten der oberen linken und der unteren rechten Ecke übergibt. Die Funktion SetRectEmpty setzt alle Koordinaten auf null, um ein NULL-Rechteck zu erzeugen. Die Funktion IsRectNull liefert TRUE, wenn sie auf einem derartigen NULL-Rechteck aufgerufen wird, und IsRectEmpty gibt TRUE zurück, wenn sowohl die Breite als auch die Höhe gleich null sind, selbst wenn einzelne Werte ungleich null sind.

Mehrere Hilfsfunktionen unterstützen die Berechnung verschiedener Aspekte der Rechteckgeometrie. Die Breite und Höhe kann man mit den Funktionen Width bzw. Height ermitteln. Beide Funktionen liefern den diesbezüglichen Integer-Wert zurück. Alternativ kann man ein CSize suchen, das sowohl Breite als auch Höhe repräsentiert, indem man die Funktion Size aufruft. Beispielsweise zeigt die folgende Zeile die Breite und Höhe des Rechtecks rcOne an:

```
TRACE("Breite: %d,Höhe: %d\n", rcOne.Width(), rcOne.Height());
```

Oftmals muß man den Punkt im Zentrum des Rechtecks kennen. Dazu kann man die Funktion CenterPoint aufrufen, die ein CPoint-Objekt zurückgibt, das den Mittelpunkt des Rechtecks darstellt. Das folgende Beispiel bestimmt mit dieser Funktion den Mittelpunkt des Client-Bereichs eines Fensters und zeichnet an dieser Stelle einen Punkt:

```
CRect rcClient;
GetClientRect(&rcClient);
dc.SetPixel(rcClient.CenterPoint(),0);
```

Mit den Funktionen UnionRect und InterSectRect kann man auch die Vereinigungs- bzw. Schnittmenge zweier Rechtecke ermitteln. Beide Funktionen übernehmen zwei Quellrechtecke als Parameter und setzen die Koordinaten des aufrufenden CRect-Objekts auf die Vereinigung oder den Schnitt. Die Vereinigung ist das kleinste Rechteck, das die beiden Quellrechtecke umschließt. Der Schnitt ist das größte Rechteck, das von beiden Quellrechtecken umschlossen wird.

Die Zeichnung in Abbildung 33 zeigt eine Vereinigung und den Schnitt zweier Quellrechtecke mit den Bezeichnern A und B. Die folgenden Zeilen berechnen den Schnitt und die Vereinigung der Quellrechtecke rcEins und rcZwei:

```
CRect rcEins(10,10,100,100);
CRect rcZwei(50,50,150,200);
CRect rcUnion, rcIntersect;
rcUnion.UnionRect(rcEins, rcZwei);
rcIntersect.IntersectRect(rcEins, rcZwei);
```

Hier wird rcUnion auf die Koordinaten (10,10) – (150, 200) und rcIntersect auf die Koordinaten (50, 50) – (100, 100) gesetzt. Mit der Funktion SubtractRect kann man ein Rechteck von einem anderen abziehen. Das Ergebnis ist das kleinste Rechteck, das alle Punkte enthält, die nicht von den beiden Quellrechtecken geschnitten werden. Wenn Sie beispielsweise die folgenden Zeilen in eine OnPaint-Behandlungsroutine einfügen, können Sie sich die Wirkung von SubtractRect ansehen. Der Code zieht rcZwei von rcEins ab und liefert rcDrei. Das Ergebnis der Subtraktion ist der Bereich, der am unteren Rand der Zeichnung in Blau dargestellt ist, wie es Abbildung 33 zeigt.

```
CRect rcEins(10,10,220,220), rcZwei(50,50,150,260), rcDrei;
rcDrei.SubtractRect(rcZwei, rcEins);
dc.FillSolidRect(rcEins, RGB(255,0,0)); // Rot
dc.FillSolidRect(rcZwei, RGB(0,255,0)); // Grün
dc.FillSolidRect(rcDrei, RGB(0,0,255)); // Blau
```

Wenn dieser Code ausgeführt wird, enthält das resultierende Rechteck rcDrei die Koordinaten (50,220)-(150,26).

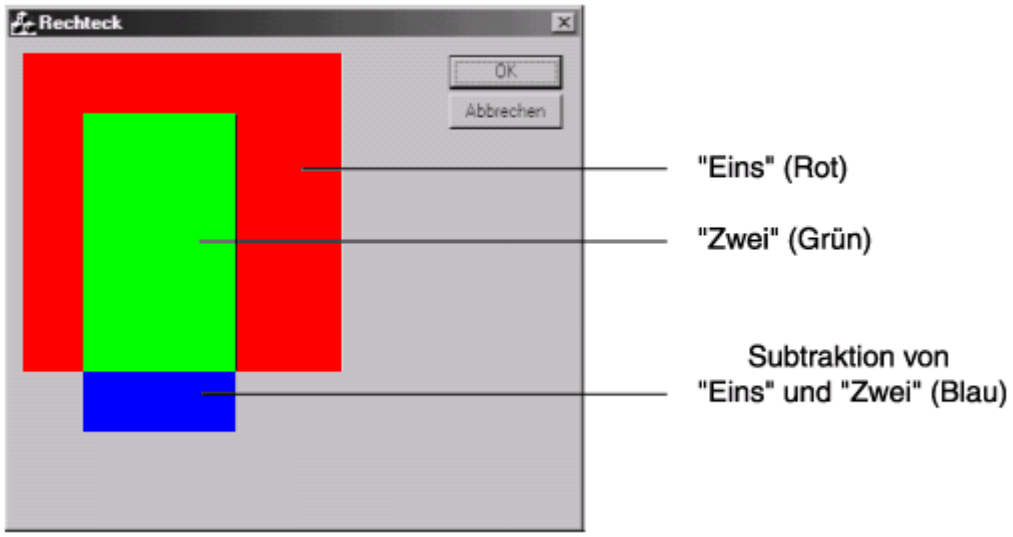


Abbildung 33: Vereinigung und Schnitt zweier Rechtecke

Die Größe eines Rechtecks lässt sich mit den Funktionen InflateRect und DeflateRect vergrößern bzw. verkleinern. Beide Funktionen haben mehrere Formen, die verschiedene Parametertypen gemäß Tabelle 25 akzeptieren.

Parameter	Beschreibung

(int x, int y)	Vergrößert oder verkleinert die linke und rechte Seite nach dem Wert `x` sowie die obere und untere Seite nach dem Wert `y`.
(SIZE size)	Vergrößert oder verkleinert die linke und rechte Seite nach `size.cx` sowie die obere und untere Seite nach dem Wert von `size.cy`.
(LPCRECT lpRect)	Vergrößert jede Seite um die entsprechenden Werte von `left`, `top`, `right` und `bottom` der Struktur `lpRect`.
(int l, int t, int r, int b)	Vergrößert jede Seite um die entsprechenden Werte für links (`l`), oben (`t`), rechts (`r`) und unten (`b`).

Tabelle 25: Parameterformen für InflateRect und DeflateRect

Der folgende Beispielcode vergrößert rcEins und verkleinert rcZwei:

```
CRect rcEins(10,10,100,100);
CRect rcZwei(50,50,150,200);
rcEins.InflateRect(5,5);
rcZwei.DeflateRect(10,20,30,40);
```

Die Ausführung dieser Zeilen setzt rcEins auf die Koordinaten (5,5)-(105,105) und rcZwei auf die Koordinaten (60,70)-(120,160). Eine Trefferprüfung lässt sich ausführen, indem man ermittelt, ob ein bestimmter Punkt innerhalb der Grenzen eines Rechtecks liegt. Dazu ruft man die Funktion PtInRect auf und übergibt den zu testenden Punkt. Wenn der Punkt innerhalb des Rechtecks liegt, gibt die Funktion `TRUE` zurück, andernfalls `FALSE`.

In den folgenden Zeilen wird die Meldung Treffer! - ptTest1 angezeigt, da ptTest1 innerhalb des Testbereichs von rcTestArea liegt, während ptTest2 nicht in diesen Bereich fällt. Demzufolge liefert PtInRect den Wert TRUE für ptTest1 und FALSE für ptTest2:

```
CRect rcTestArea(10,20,440,450);
CPoint ptTest1(200,200), ptTest2(500,500);

if (rcTestArea .PtInRect(ptTest1))
 AfxMessageBox("Treffer! - ptTest1");

if (rcTestArea .PtInRect(ptTest2))
 AfxMessageBox("Treffer! - ptTest2");
```

Wie folgt gibt es für CRect folgende verschiedene überladene Operatoren.

Operator	**Beschreibung**
=	Kopiert wie bei einer normalen numerischen Zuweisung alle Koordinaten vom rechten Operanden (Rechteck) in das linke Rechteck.

+	Der Operator führt entweder eine Verschiebung der Rechteckposition aus, wenn zum Rechteck ein `CPoint`- oder `CSize`-Objekt addiert wird, oder vergrößert die Koordinaten für die entsprechenden Seiten, wenn ein `CRect`-Objekt addiert wird.
-	Analog zu `+`, außer dass die Koordinaten in negativer Richtung verschoben oder bei einem `CRect` verkleinert werden.
+=	Wirkung wie bei `+`, beeinflusst aber nur das aktuelle Rechteck.
-=	Wirkung wie bei -, beeinflusst aber nur das aktuelle Rechteck.
&	Erzeugt ein Rechteck als Schnittmenge zweier Rechteckoperanden.
|	Erzeugt ein Rechteck als Vereinigungsmenge zweier Rechteckoperanden.
&=	Wirkung wie `&`, beeinflusst aber nur das aktuelle Rechteck.
|=	Wirkung wie bei `|`, beeinflusst aber nur das aktuelle Rechteck.
==	Liefert `TRUE`, wenn die Rechtecke identisch sind, ansonsten `FALSE`.
!=	Liefert `FALSE`, wenn die Rechtecke identisch sind, andernfalls `TRUE`.

Tabelle 26: Überladene Operatoren für CRect Objekte

Die folgenden Zeilen zeigen, wie man mit überladenen CRect-Operatoren das Rechteck rcStart manipuliert:

```
CRect rcStart(10,10,100,100);
rcStart = rcStart + CPoint(5,5);
rcStart -= CSize(5,5);
rcStart += CRect(1,2,3,4);
if (rcStart == CRect(9,8,103,104))
 AfxMessageBox("TRUE");
```

Die letzte Bedingung liefert TRUE, weil die Koordinaten nach Ausführung der Anweisungen auf (9,8)-(103,104) gesetzt sind.

5.2.2.3 Die Klasse CSize

Die Klasse CSize kapselt die Struktur SIZE und bietet verschiedene Konstruktoren und überladene Operatoren, mit denen sich die internen Werte cx und cy, die eine Größe definieren, manipulieren lassen. Tabelle 27 zeigt die Konstruktoren, mit denen man eine Instanz eines CSize-Objekts erzeugen kann.

Konstruktordefinition	**Beschreibung**
CSize()	Erzeugt ein nicht initialisiertes `CSize`-Objekt.
CSize(SIZE sizeInit)	Kopiert die Werte `cx` und `cy` aus einem anderen `CSize`-Objekt oder einer `SIZE`-Struktur.

CSize(initCX, initCY)	Initialisiert das Objekt mit `initCX` für die horizontale Größe und `initCY` für die vertikale.
CSize(POINT ptInit)	Initialisiert das Objekt mit den Werten `x` und `y` aus einem `CPoint`-Objekt oder einer `POINT`-Struktur.
CSize(DWORD dwSize)	Setzt den Wert `cx` auf das niederwertige Wort (die unteren 16 Bit) von `dwSize` und `cy` auf das höherwertige Wort (die oberen 16 Bit).

Tabelle 27: Konstruktortypen für die Klasse CSize

Folgendes Beispiel zeigt, dass man die Elemente cx und cy direkt manipulieren kann:

```
CSize tstSize(10,10);
tstSize.cx = tstSize.cy * 2;
```

Die einzigen Funktionen, die die Klasse CSize bietet, sind die überladenen Operatoren gemäß Tabelle 28.

Operator	Beschreibung
+	Addiert zwei `SIZE`-Objekte.
-	Subtrahiert ein `SIZE`-Objekt von einem anderen.
+=	Addiert ein `SIZE`-Objekt.
-=	Subtrahiert ein `SIZE`-Objekt.
==	Bestimmt, ob die beiden Größen gleich sind, und liefert in diesem Fall `TRUE` zurück.
!=	Bestimmt, ob die beiden Größen unterschiedlich sind, und liefert in diesem Fall `TRUE` zurück.

Tabelle 28: Überladene Operatoren für CSize

Diese Operatoren kann man wie normale arithmetische Operatoren einsetzen. Sie beeinflussen sowohl cx als auch cy, wie es die folgenden Zeilen zeigen, die den Inhalt von tstSize manipulieren:

```
CSize tstSize(10,15);
tstSize += tstSize + tstSize - CSize(1,2);
if (tstSize == CSize(29,43)) AfxMessageBox("TRUE");
```

Bei Ausführung des Codes zeigt das Meldungsfeld TRUE, weil tstSize am Ende eine Größe von 29 mal 43 hat.

5.2.3 Behandlung von Zeichenfolgen

Vor mehreren Jahren beneideten die C-Programmierer heimlich ein Werkzeug, das BASIC-Programmierer zur Verfügung hatten. Die ausgeklügelte und einfache Zeichenfolgenbehandlung. Bei C++ lässt sich diese Funktionalität natürlich nachbilden und steht mit der MFC-Klasse CString zur Verfügung.

Die Zeichenfolgenbehandlung ist in Anwendungen häufig gefragt, und Visual C++ Anwendungen tendieren dazu, mit Instanzen von Objekten auf Basis der Klasse `CString` durchsetzt zu sein, um diese Aufgabe zu realisieren.

5.2.3.1 Die Klasse CString

CString-Objekte kann man in einfacher Weise als leeren String konstruieren oder initialisieren, indem man einen der vielen verschiedenen Typen von Textdarstellungssystemen an den Konstruktor übergibt. Der C++ Anfänger tut es sich leichter mit dem Verständnis dieser Klasse, wenn er sich einen Array von dem Variablentyp char vorstellt. Im Prinzip ist eine CString-Klasse ähnlich, mit dem Unterschied, dass die Zeichenmanipulation hier sehr vereinfacht ist.

Die verschiedenen Formen der CString-Konstruktion sind in Tabelle 29 aufgeführt.

Konstruktordefinition	Beschreibung
CString()	Erzeugt einen leeren String der Länge null.
CString(const CString& strSrc)	Kopiert den Inhalt aus einem anderen String.
CString(LPCSTR lpsz)	Kopiert den Inhalt aus einem nullterminierten String.
CString(const unsigned char* psz)	Kopiert den Inhalt aus einem nullterminierten String.
CString(LPCSTR lpch, int nLength)	Kopiert `nLength` Zeichen aus einem Zeichenfeld.
CString(TCHAR ch, int nRepeat = 1)	Füllt den String mit `nRepeat` Kopien des Zeichens `ch`.
CString(LPCWSTR lpsz)	Kopiert einen nullterminierten Unicode-String.

Tabelle 29: Mit CString verwendete Konstruktortypen

Nachdem ein CString-Objekt konstruiert ist, gibt es viele Möglichkeiten, Text hinzuzufügen oder zuzuweisen. Die überladenen Operatoren erlauben einfache Zuweisungen über den Operator = oder die Verkettung von zwei Strings mit den Operatoren + und +=, wie es in den folgenden Zeilen gezeigt wird:

```
CString strTest;
strTest = "Mr Gorsky";
strTest = "Viel Glück " + strTest;
AfxMessageBox(strTest);
```

Dieses Beispiel setzt den String anfänglich auf "Mr Gorsky". Anschließend wird mit dem Operator + der Text "Viel Glück " vorangestellt.

Die Länge eines Strings lässt sich mit der Elementfunktion GetLength ermitteln. Diese Funktion liefert eine Ganzzahl zurück, die die momentane Anzahl der Zeichen im String darstellt. Mit der Funktion IsEmpty kann man auch testen, ob ein String leer ist. Die Funktion liefert TRUE zurück, wenn der String keine Zeichen enthält. Die Funktion Empty dient dazu, den Inhalt eines CString-Objekts zu löschen. Das Objekt weist dann die Länge null auf.

Viele Funktionen erfordern Strings im alten Stil von C und keine CString-Objekte. Die Typumwandlungen (const char *) oder LPCTSTR erlauben diesen Funktionen den Zugriff auf den internen Puffer des CString-Objekts als würde es sich um einen nullterminierten C-String handeln. Der Zugriff ist allerdings nur für den Lesezugriff als nullterminierter String zu behandeln.

Visual C++ wandelt implizit den CString in einen nullterminierten String um, wenn der Prototyp einer bestimmten Funktion das erfordert. Da jedoch einige Funktionen einen void* Prototyp haben, übergibt der Compiler einen Zeiger auf

das CString-Objekt statt den erwarteten nullterminierten String. In diesem Fall muss man die Umwandlung (LPCTSTR) von Hand erledigen.

Mit den Funktionen GetAt und SetAt kann man auf einen String als Array von Zeichen zugreifen. GetAt liefert das Zeichen an der als Parameter übergebenen Position zurück, wobei die das erste Element mit der 0 angesprochen wird. Die Funktion SetAt setzt ein Zeichen an eine Position innerhalb der Länge des Strings. Anstelle der Funktion GetAt kann man auch mit dem Indexoperator [ ] einen Zeichenwert von einer bestimmten Position ermitteln. Beispielsweise tauschen die folgenden Zeilen die Zeichen b und g aus, um den Rechtschreibfehler zu korrigieren:

```
CString strText("Rechtschreigung ");
TCHAR ch1 = strText.GetAt(11);
strText.SetAt(11, strText[14]);
strText.SetAt(14, ch1);
```

Mit den überladenen Operatoren <, <=, ==, !=, >= und > kann man Strings gemäß der lexikographischen Reihenfolge miteinander vergleichen. Bei Zeichenfolgenvergleichen werden die ASCII-Codes miteinander verglichen, so dass Ziffern kleiner als Buchstaben und Großbuchstaben kleiner als Kleinbuchstaben sind. Demzufolge bewirken die folgenden Codezeilen, dass im Meldungsfeld TRUE erscheint:

```
CString str1("123");
CString str2("ABC");
CString str3("abc");
CString str4("bcd");
if (str1 < str2 && str2 < str3 && str3 < str4)
      AfxMessageBox("TRUE");
```

Um den aktuellen String mit einem anderen zu vergleichen, kann man auch die Funktion Compare einsetzen. Die Funktion liefert null zurück, wenn zwei Strings gleich sind, einen negativen Wert, wenn der aktuelle String kleiner als der getestete String ist, oder einen positiven Wert, wenn der aktuelle String größer als der Teststring ist.

Beispielsweise bewirken die folgenden Zeilen, dass ein Meldungsfeld mit dem Text TRUE erscheint:

```
CString strName("Peter");
if (strName.Compare("Piper")<0)
      AfxMessageBox("TRUE");
```

Dieser Vergleich beachtet auch die Groß-/Kleinschreibung der zu vergleichenden Strings. Vergleiche ohne Unterscheidung der Groß-/Kleinschreibung lassen sich mit der äquivalenten Funktion `CompareNoCase` durchführen.

5.2.3.2 ZEICHENFOLGENMANIPULATION

In der Sprache BASIC stehen drei nützliche Funktionen zur Manipulation von Strings zur Verfügung: Mid$, Left$ und Right$. Diese Funktionen sind nun auch als Mid, Left und Right in der Klasse CString verfügbar. Als Ergebnis liefern die Funktionen Kopien von Teilstrings zurück.

Der Funktion Mid übergibt man eine Anfangsposition und optional die Anzahl der zu kopierenden Zeichen. Die Funktion gibt dann einen anderen CString mit dem angegebenen Teilstring zurück. Mit der Funktion Left extrahiert man eine Anzahl Zeichen vom linken Teil eines Strings. Dabei übergibt man der Funktion die Anzahl der gewünschten Zeichen. Die Funktion Right liefert die angegebene Anzahl der Zeichen von dem Ende des Strings an gerechnet. Dazu das folgende Beispiel:

```
CString strText("Ich habe heute drei Segelschiffe gesehen");
TRACE("%s\n", strText.Left(8));
TRACE("%s\n", strText.Mid(15, 17));
TRACE("%s\n", strText.Right(7));
```

Die Ausgabe der drei TRACE-Makros liefert:

```
Ich habe
drei Segelschiffe
gesehen
```

Das Wort heute zwischen habe und drei erscheint nicht in der Trace-Ausgabe, da dieser Teil des Strings überhaupt nicht extrahiert wird.

Mit den Funktionen MakeUpper und MakeLower ändert man alle Zeichen in einem String in Groß- bzw. Kleinbuchstaben, während die Funktion MakeReverse den String in umgekehrter Zeichenfolge liefert.

Leerzeichen, Zeichen für neue Zeile und Tabulatoren lassen sich vom linken Teil eines Strings mit der Funktion TrimLeft entfernen. Auf der rechten Seite realisiert die Funktion TrimRight diese Aufgabe.

5.2.3.3 Zeichenfolgen suchen

Die Suche nach bestimmten Zeichen innerhalb eines Textes ist oft notwendig. In der Diplomarbeit musste z.B. überprüft werden, ob die Lage der Naht in den Bearbeitungsbereich eines Werkers fällt. Mit den Funktionen Find, ReverseFind und FindOneOf kann man nach bestimmten Teilstrings oder Zeichen in einem String suchen. Diese Suchfunktionen nach Teilstrings ermöglichen die Überprüfung ob der Vorgegebene Bereich ein Teil von dem Bereich ist, welches ein Werker bearbeiten kann.

Der Elementfunktion Find übergibt man ein einzelnes Zeichen oder einen String, um nach einem Vorkommen dieses Zeichens oder des Strings im Kontextstring zu suchen. Wenn das Zeichen oder der Teilstring gefunden wurde, liefert die Funktion die Position zurück.

Andernfalls lautet das Ergebnis -1. Es kennzeichnet, dass der Teilstring oder das Zeichen nicht vorhanden ist.

Die folgenden Zeilen suchen nach dem Wort »du« und zeigen den Teilstring ab dieser Position an:

```
CString strTest("Fang an, wenn du denkst");
int nPosn = strTest.Find("du");
if (nPosn!=-1)
      TRACE(strTest.Mid(nPosn) + "?");
```

Die Funktion ReverseFind sucht nach einem bestimmten Zeichen vom Ende eines Strings. Diese Funktion kann nur nach Zeichen suchen, die Suche nach Teilstrings kann nicht mit dieser Funktion realisiert werden. Wird das Zeichen gefunden, liefert die Funktion die Position bezüglich des Anfangs vom String zurück, andernfalls wird eine -1 zurückgeliefert.

Bei der Funktion FindOneOf kann man eine Anzahl von Zeichen übergeben, die in einem String zu suchen sind. Zurückgegeben wird die Position des ersten Suchzeichens der Gruppe. Das Ergebnis -1 gibt an, dass kein Zeichen gefunden wurde. Die folgenden Beispielzeilen suchen nach den Zeichen d, e, k und w. Das w wird zuerst gefunden, so dass der String »wenn du denkst« in der Trace-Ausgabe erscheint:

```
CString strTest("Fang an, wenn du denkst");
int nPosn = strTest.FindOneOf("dekw");
if (nPosn!=-1)
      TRACE(strTest.Mid(nPosn));
```

5.2.3.4 TEXT ZUR ANZEIGE FORMATIEREN

Häufig setzt man CString-Objekte ein, um Text vor der Ausgabe zu formatieren. Dazu verwendet man die Funktion Format, der man im ersten Parameter eine Gruppe von Formatanweisungen als Prozentzeichencodes übergibt. Daran schließt sich die Übergabe von Wertparametern an. Diese Funktion erinnert sehr an die von prinft funktion, der Programmiersprache C.

Die Anzahl der Wertparameter muß der Anzahl der angegebenen Formatcodes entsprechen. Einige dieser Formatflags und die jeweiligen Parametertypen sind in Tabelle 30 aufgelistet. Mehrere Codes kann man zu einem Formatstring zusammenfassen, der dann im aufrufenden CString-Objekt gespeichert wird.

Diesen Formatstring kann man auch als Stringressourcen-ID aus den Ressourcen der Anwendung übergeben.

Flag	Parametertyp	Beschreibung
%c	int	Zeigt ein einzelnes Textzeichen an.
%d	int	Zeigt eine Dezimalzahl mit Vorzeichen an.
%u	int	Zeigt eine Ganzzahl ohne Vorzeichen an.
%o	int	Zeigt eine Oktalzahl ohne Vorzeichen an.
%x	int	Zeigt eine ganze Hexadezimalzahl ohne Vorzeichen in Kleinbuchstaben an.
%X	int	Zeigt eine ganze Hexadezimalzahl ohne Vorzeichen in Großbuchstaben an.
%f	double	Zeigt eine Gleitkommazahl mit Vorzeichen an.
%s	string	Zeigt bei Übergabe eines Stringzeigers wie etwa `char*` einen Textstring an.
%%	keiner	Zeigt das Prozentzeichen `%` an.

Tabelle 30: Formatcodes für die Funktion Format

Das folgende Beispiel kombiniert einige der gebräuchlichen Typen, um ein Meldungsfeld mit dem Text „Die BearbZeit beträgt: 0.34 Minuten“ anzuzeigen:

```
CString strBearbZeit;
char *szM = " Minuten";
double dTime = 0.34;
strBearbZeit.Format("Die BearbZeit beträgt: %f %s" , dTime, szM);
AfxMessageBox(strFormatMe);
```

Mit zusätzlichen Formatangaben kann man für jedes Ausgabefeld die Breite, Genauigkeit und Ausrichtung spezifizieren. Diese Attribute sind nach dem Prozentzeichen aber noch vor dem Typzeichen in folgendem Format zu schreiben:

```
%[Flag][Breite][.Genauigkeit]Typ
```

Für den Flag-Wert kann man die folgenden Zeichen angeben:

- Das Zeichen << - >>
  - Richtet das Feld linksbündig aus.
- Das Zeichen << + >>
  - Zeigt vor einer Zahl ein Plus- oder Minuszeichen an.

- Das Zeichen << 0 >>
  - Füllt die Feldbreite mit Nullen auf.

Mit Breite gibt man dann die Mindestzahl von Zeichen an, die das Feld anzeigen soll. Der Parameter „Genauigkeit“ bestimmt, wie viele Zeichen nach dem Dezimalpunkt stehen sollen.

6 AUSBLICK

Das von mir entwickelte Programm stellt den Grundstein für diese Thematik dar, welche in folgenden Bereichen weiterentwickelt werden kann.

6.1 SCHNITTSTELLE ZUM CAD-PROGRAMM CATIA

Ziel dieser Schnittstelle ist es, Nahtinformationen direkt aus einem CAD Programm zu erhalten. Hier stellt sich die Frage welche Information lässt sich von einer Zeichnung gewinnen. Zur der Informationsgewinnung aus der CAD Zeichnung gibt es zwei Lösungsansätze:

- Lösungsansatz 1: Nutzung der Schnittstelle VDAFS
- Lösungsansatz 2: Zusatzmodul in dem CAD Programm CATIA

Prinzipiell können alle Zeichnungen in das VDAFS Format konvertiert werden. Dieses Dateiformat ist standardisiert, und beschreibt die Oberflächenstruktur von Karosserien. Aus dieser Datei können folgende Informationen gewonnen werden:

- Bearbeitungsort (X-, Y- und Z-Achse)
- Nahtlänge

Die Nutzung dieser Schnittstelle würde den Vorteil mit sich bringen, dass man Plattformunabhängig ist. Falls später die Zeichnungen mit einem anderen CAD Programm erstellt werden, so könnte man diese Zeichnungen weiterhin in den VDAFS Format konvertieren. Ein großer Nachteil stellt hier die Dateigröße einer VDAFS-Datei dar. Dadurch dass zur Abspeicherung der Informationen ASCII Zeichen genutzt werden, bilden sich schnell große Dateien.

Eine weitere Lösungmöglichkeit ist es, ein Modul in dem verwendeten CAD Programm zu entwickeln, welches aus der Zeichnung direkt die notwendigen Informationen filtert und nur diese Informationen in einer Datei abspeichert. Bei dieser Vorgehensweise ist es möglich folgende Informationen zu gewinnen:

- Bearbeitungsort (X-, Y- und Z-Achse)
- Nahtlänge

- Nahtart

Ein klarer Vorteil ist bei diesem Lösungsansatz, dass die Datei welches hinterher die Informationen enthält, frei nach bedarf gestaltet werden kann. Weiterhin wird diese Datei auch sehr klein sein, da nur die notwendigen Informationen für unser Programm abgespeichert werden.

Allerdings müsste man dieses Modul auf der Seite des CAD Programms neu entwickeln, wenn ein neues CAD-Programm eingeführt wird.

6.2 DATENBANKZUGRIFF ÜBER DAS NETZ

Derzeit greift das Programm auf eine Datenbank zu, welche lokal auf der Festplatte abgespeichert ist. Wodurch das Programm momentan nur an einem Rechner installiert werden kann, damit die vorgenommen Änderungen noch überblickt werden können. Das Programm sollte so erweitert werden, dass es möglich wird über das Netzwerk auf diese Datenbank zuzugreifen.

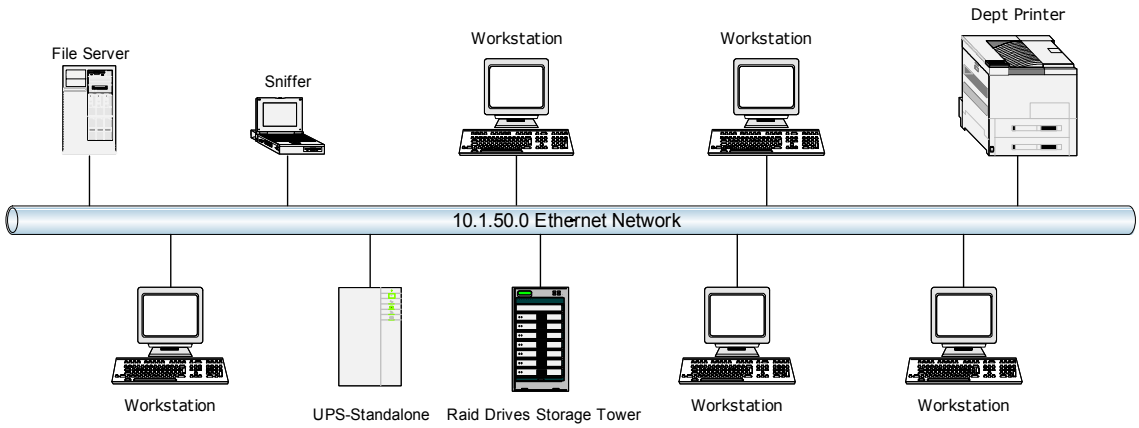


Abbildung 34: Netzwerk

Literaturverzeichnis

Bücher:

Dirk Louis: Jetzt lerne ich Visual C++ 6 [Hrsg.] Markt & Technik, S. 471 – 495

Davis Chapman: Visual C++6 in 21 Tagen [Hrsg.] SAMS, S. 760 – 800 &
S. 350 - 355

Quellcodebeispiele sind größtenteils von Davis Chapman´s Visual C++6 in 21 Tagen übernommen.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, Rahman Dilsiz, geboren am 28.12.1978 in Singen, ehrenwörtlich,

(1) dass ich meine Diplomarbeit mit dem Titel:

„Automatische Eintaktung von manuellen Tätigkeiten in der Lackiererei“

bei der AUDI AG unter Anleitung von Professor Dr. Arndt selbständig und ohne fremde Hilfe angefertigt habe und keine anderen als in der Abhandlung angeführten Hilfen benutzt habe;

(2) dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, den 28.02.2003 Unterschrift:……..