1 Lösungsansatz

Als Basis wird der Source der Übung 5 herangezogen, da es sich bei dieser Übung nur um ein Austauschen des Gedächtnisses und das Hinzufügen einer neuen Methode handelt.
Das Gedächtnis dieses Moduls soll mit einer einfach verketteten Liste ersetzt werden. Der Benutzer soll keine Möglichkeit mehr haben, die Größe des Gedächtnisses zu konfigurieren, da dieses Gedächtnis jetzt dynamisch wachsen kann. Daher soll der Übergabeparameter size der Prozedur Reset entfernt werden. Diese Prozedur soll jetzt die Elemente der Liste löschen (Dispose) und die Referenz der Liste anschließen auf NULL setzen. Im Implementierungsteil soll als Standard die Liste mit NIL initialisiert werden, um zu verhindern das der Pointer der Liste eine ungültige Adresse enthält.
Es sollen die in der Aufgabenstellung geforderten Datentypen implementiert werden, wobei der TimeRegistrationEntryRec Record als Komponente den bereits bestehenden WorkEntry Datentyp verwenden soll, damit sich für den Benutzer sich an der Schnittstelle nichts ändert. Diese Datentypen sollen nur innerhalb des Moduls sichtbar sein.
Es soll eine Funktion implementiert werden, die eine TimeRegistrationEntry Node Instanz erstellt, wobei darauf zu achten ist, das dessen Next Komponente auf NULL gesetzt wird, damit auch dieser Pointer korrekt initialisiert wird.
Anschließend sollen die enthaltenen Prozeduren und Funktionen, die mit dem Gedächtnis arbeiten, an das neue Gedächtnis angepasst werden, wobei sich lediglich die Iteration über die Einträge und das Einfügen in das Gedächtnis ändert.

Als Zusatz soll eine neue Funktion eingeführt werden, die den Namen der Person mit den meisten Work Entry Einträgen zurückgibt. Sollten keine Einträge gefunden werden so soll ein Leerstring zurückgegeben werden. Da gefordert ist hier eine temporäre Liste aufzubauen, die die Einträge der Namen mit ihre Anzahl von Einträgen speichert, sollen auch hier Datentypen für diese Liste erstellt werden.

1. TimeRegistrationCount (Pointer auf eine List Node)
2. TimeRegistrationCountRec (Compound für die zu speichernden Daten)
3. TimeRegistrationCountList (Pointer für die Liste)

Es soll darauf geachtet werden, dass auch eine unsortierte Liste gehandhabt werden kann, wo die Einträge einer Person nicht hintereinander liegen. Um dies einfacher zu gestalten soll eine Funktion eingeführt werden, die einen bereits existierenden TimeRegistrationCount Eintrag aus der Liste für einen Namen einer Person findet und zurückgibt. Ebenso soll auch für diesen Datentyp eine Funktion eingeführt werden, die die TimeRegistrationCount Instanz erstellt und deren NEXT Komponente auf NULL setzt. Auch für diese Liste soll es eine eigene Prozedur geben, die alle Elemente aus der Liste löscht und die Liste anschließend auf NIL setzt.

Für die Tests soll der Source des Programms WorkManagementUnitTest wiederverwendet werden und um die neuen Funktionen und Prozeduren der Schnittstelle erweitert werden, wobei ein besonderer Augenmerk auf die Prozedur Reset gelegt werden soll, da es hier bei einer fehlerhaften Implementierung zu Speicherleichen kommen kann.

2 Source

Folgend ist der Source der WorkManagementUnit und der WorkManagementUnitTest angeführt.

2.1 WorkManagementUnit

Folgend ist der Test der WorkManagementUnit angeführt.

```
{
  Unit for handling work entries which are hold by a single connected  list.
  Call Reset to reset the memory.
  This unit can handle as much entries as the caller wishes, depends on the
  available memory.
}
UNIT WorkManagementUnit;

{ ######################### Interface part ######################### }
INTERFACE

{ Uses the time span unit }
USES TImeSpanUnit;

{
  The types which are visible and usable for the caller.
}
TYPE
  { The error codes which can be handled by the caller }
  ErrorCode = (NONE, TO_SHORT, TO_LONG, INVALID_SPAN);

  { Compound representing a work entry }
  WorkEntry = RECORD
    name: STRING;
    task: STRING;
    spendTime: TimeSpan;
  END;

{
  Resets the memory of this module
}
PROCEDURE Reset;

{
  Creates a work entry for the given data.

  @param
      name: the name of the person
  @param
      task: the name of the task the person worked on
  @param
      spendTime: the TimeSpan instance which represents the time the person has worked
      on the defined task
  @return
      the create workEntry instance
}
FUNCTION CreateWorkEntry(name, task: STRING; spendTime: TimeSpan): WorkEntry;

{
  Adds a WorkEntry to the backed list.

  @param
      entry: the entry to be added
  @param
      error: the error parameter which will be set <> NONE if an error occurs
  @returns
      the occurred error or the given error from the caller which should be NONE
}
PROCEDURE AddWorkEntry(entry: WorkEntry; VAR error: ErrorCode);

{
  Gets the total work time for the given person.

  @param
      name: the name of the person
  @param
```

```
      spendtime: the TimeSpan instance given by the caller which will get set with the total
work time,
      should be initialized by the caller with 0:0:0
  @return
      the TimeSpan representing the total work time, or all values set to 0 when the person
would not be found,
      or no entry exists
}
PROCEDURE GetTotalWorkTimeForPerson(name: STRING; VAR spendTime: TimeSpan);

{
  Gets the average time worked on a task.

  @param
      task: the task to get the average work time for
  @param
      average: the TimeSpan instance where the result will be saved, shall be initialized by
the
      caller with 0:0:0
  @return
      the TimeSpan instance which contains the average time worked on the task, or all field
set to 0,
      if the task could not be found, because there can no task be present in the entry holder
which has
      a time spend less than 1 minute.
}
PROCEDURE GetAverageWorkTimeForTask(task: STRING; VAR average: TimeSpan);

{
  Prints the persons which has worked on the task, along with their time spend.

  @param
      task: the task to be printed.
}
PROCEDURE PrintPersonForTask(task: STRING);

{
  Prints the work summary for the given person.

  @param
      name: the name of the persos to search for the work summary
}
PROCEDURE PrintWorkSummaryForPerson(name: STRING);

{
  Gets the name of the person with the most time spent on the saved tasks.

  @return
      the name of the busiest person
}
FUNCTION BusiestPerson: STRING;

{ ######################## Implementation part ######################## }
IMPLEMENTATION

TYPE
  { The pointer type used for the work entry list nodes }
  TimeRegistrationEntry = ^TimeRegistrationEntryRec;
  { The node type }
  TimeRegistrationEntryRec = RECORD
    entry: WorkEntry;
    next: TimeRegistrationEntry;
  END;
  { Separate pointer type for the work entry list }
  TimeRegistrationEntryList = TimeRegistrationEntry;

  { The pointer type for the time registration count list nodes }
  TimeRegistrationCount = ^TimeRegistrationCountRec;
  { The type for the time registration count node }
  TimeRegistrationCountRec = RECORD
    name: STRING;
    count: INTEGER;
    next: TimeRegistrationCount;
  END;
  { Separate pointer type for the time registration count list }
  TimeRegistrationCountList = TimeRegistrationCount;
```

```
{
  The memory which contains the work entries, represented by a list
}
VAR
  list: TimeRegistrationEntryList;

{ ######################## Private Function/Procedure part ######################## }
{
  Creates a TimeRegistrationEntry node instance.
  ATTENTION: Be aware that the created instance has to be disposed manually.

  @param
      entry: The WorkEntry instance hold the by the node
  @return
      the created TimeRegistrationEntry instance
}
FUNCTION CreateTimeRegistrationEntryNode(entry: WorkEntry): TimeRegistrationEntry;
BEGIN
    CreateTimeRegistrationEntryNode := new(TimeRegistrationEntry);
    CreateTimeRegistrationEntryNode^.entry := entry;
    CreateTimeRegistrationEntryNode^.next := NIL;
END;

{
  Creates a TimeRegistrationCount instance.
  ATTENTION: Be aware that the created instance has to be disposed manually.

  @param
      name: the name for the TimeRegistrationCount instance
  @param
      count: the count for the TimeRegistrationCount instance
  @return
      the created TimeRegistrationCount instance.
}
FUNCTION CreateTimeRegistrationCountNode(name: STRING; count: LONGINT): TimeRegistrationCount;
BEGIN
    CreateTimeRegistrationCountNode := new(TimeRegistrationCount);
    CreateTimeRegistrationCountNode^.name := name;
    CreateTimeRegistrationCountNode^.count := count;
    CreateTimeRegistrationCountNode^.next := NIL;
END;

{
  Finds the TimeRegistrationCount instance contained in the given list, for the given name.
  The name is handled with ignoreCase.
  Used because the backed TimeRegistrationEntryList could not contain sorted entries (sorted
by name).

  @param
      name: the name set on the TimeRegistrationCount of the given list
  @param
      tempList: the given list to search for the set name.
  @return
      the found TimeRegistrationCount instance or NIL if the instance could not be found, or
the given
      tempList is NIL.
}
FUNCTION FindTimeRegistryCountByName(name: STRING; tempList: TimeRegistrationCountList):
TimeRegistrationCount;
VAR
  node: TimeRegistrationCount;
  found: BOOLEAN;
BEGIN
  found := false;
  FindTimeRegistryCountByName := NIL;
  node := NIL;

  IF (tempList <> NIL) THEN BEGIN
    node := tempList;
    WHILE ((found = false) AND (node <> NIL)) DO BEGIN
      IF (name = node^.name) THEN BEGIN
        FindTimeRegistryCountByName := node;
        found := true;
      END
      ELSE BEGIN
```

```
        node := node^.next;
      END;
    END;
  END;
END;

{
  Gets the TimeRegistrationCount instance with the highest count value.
  If multiple instances do have the same count, then the first one is returned
  and the others are ignored.

  @param
      tempList: the TimeRegistrationCountList instance where to search for the highest
contained count.
  @return
      the TimeRegistrationCount with the highest count value, or NIL otherwise
}
FUNCTION GetHighestTimeRegistryCount(tempList: TimeRegistrationCountList):
TimeRegistrationCount;
VAR
  node: TimeRegistrationCount;
  count: LONGINT;
BEGIN
  GetHighestTimeRegistryCount := NIL;
  count := 0;

  IF (tempList <> NIL) THEN BEGIN
    node := tempList;
    WHILE (node^.next <> NIL) DO BEGIN
      (* writeln(node^.name, ' - ', node^.count); *)
      IF (count < node^.count) THEN BEGIN
        count := node^.count;
        GetHighestTimeRegistryCount := node;
      END;
      node := node^.next;
    END;
  END;

END;

{
  Disposes all contained entries of the given TimeRegistrationCountList list.

  @param
      list: the list which elements shall be disposed
  @return
      list: the disposed list, which will be set to NIL.
}
PROCEDURE DisposeTempList(VAR countList: TimeRegistrationCountList);
VAR
  pred, succ: TimeRegistrationCount;
BEGIN
  pred := countList;
  { WriteLn(''); }
  WHILE (pred <> NIL) DO BEGIN
    succ := pred^.next;
    Dispose(pred);
    { WriteLn('Count Entry disposed'); }
    pred := succ;
  END;
  { WriteLn(''); }
  countList:= NIL;
END;

{ ######################### Public Function/Procedure part ######################### }
{ Reset memory }
PROCEDURE Reset;
VAR
  pred, succ: TimeRegistrationEntry;
BEGIN
  pred := list;
  WHILE (pred <> NIL) DO BEGIN
    succ := pred^.next;
    Dispose(pred);
    { WriteLn('Work Entry disposed'); }
    pred := succ;
```

```
  END;
  list := NIL;
END;

{ Creates a WorkEntry instance }
FUNCTION CreateWorkEntry(name, task: STRING; spendTime: TimeSpan): WorkEntry;
BEGIN
  CreateWorkEntry.name := name;
  CreateWorkEntry.task := task;
  CreateWorkEntry.spendTime := spendTime;
END;

{ Adds the work entry to the memory}
PROCEDURE AddWorkEntry(entry: WorkEntry; VAR error: ErrorCode);
VAR
  newNode, node: TimeRegistrationEntry;
BEGIN
  { Converts the time span to seconds, and will contain any error }
  entry.spendTime := TimeSpanToSeconds(entry.spendTime);
  { Invalid time span detected }
  IF (entry.spendTime.error <> '') THEN BEGIN
    error := INVALID_SPAN
  END
  { To less time span }
  ELSE IF (entry.spendTime.timeInSeconds < 60) THEN BEGIN
    error := TO_SHORT
  END
  { To much time }
  ELSE IF (entry.spendTime.timeInSeconds > (8 * 60 * 60)) THEN BEGIN
    error := TO_LONG
  END
  { valid to add entry }
  ELSE BEGIN
    newNode := CreateTimeRegistrationEntryNode(entry);
    { First element }
    IF (list = NIL) THEN BEGIN
      list := newNode;
    END
    ELSE BEGIN
      node := list;
      WHILE (node^.next <> NIL) DO BEGIN
        node := node^.next;
      END;
      node^.next := newNode;
    END;
  END;
END;

{ Gets total work time }
PROCEDURE GetTotalWorkTimeForPerson(name: STRING; VAR spendTime: TimeSpan);
VAR
  node: TimeRegistrationEntry;
  sec: LONGINT;
  entry: WorkEntry;
BEGIN
  sec := 0;
  { Only get total time when entries are present }
  IF (list <> NIL) THEN BEGIN
    node := list;
    WHILE (node <> NIL) DO BEGIN
      entry := node^.entry;
      { Build sum for the given name }
      IF (LowerCase(entry.name) = LowerCase(name)) THEN BEGIN
        sec := sec + TimeSpanUnit.TimeSpanToSeconds(entry.spendTime).timeInSeconds;
      END;
      node := node^.next;
    END;
  END;

  spendTime := TimeSpanUnit.SecondsToTimeSpan(sec);
END;

{ Gets average of spend time }
PROCEDURE GetAverageWorkTimeForTask(task: STRING; VAR average: TimeSpan);
VAR
  node: TimeRegistrationEntry;
```

```
  sec: LONGINT;
  count: INTEGER;
  entry: WorkEntry;
BEGIN
  sec := 0;
  count := 0;
  average.timeInSeconds := 0;

  { Only get average time when entries are present }
  IF (list <> NIL) THEN BEGIN
    node := list;
    WHILE (node <> NIL) DO BEGIN
      entry := node^.entry;
      { Search for this task }
      IF (entry.task = task) THEN BEGIN
        sec := sec + TimeSpanUnit.TimeSpanToSeconds(entry.spendTime).timeInSeconds;
        Inc(count);
      END;
      node := node^.next;
    END;
  END;

  { Calculate the average if the task on working persons have been found }
  IF (sec <> 0) THEN BEGIN
    sec := (sec DIV count);
  END;

  { Create time span out of seconds }
  average := TimeSpanUnit.SecondsToTimeSpan(sec);
END;

{ Print persons for task to table }
PROCEDURE PrintPersonForTask(task: STRING);
VAR
  node: TimeRegistrationEntry;
  found: BOOLEAN;
  count: INTEGER;
  entry: WorkEntry;
  name: STRING;
BEGIN
  found := false;
  count := 0;
  name := '';

  WriteLn('################### Persons working on task ###################');
  WriteLn('Task    : ', task);
  WriteLn('Persons : ');

  { Only print result if there are entries available }
  IF (list <> NIL) THEN BEGIN
    node := list;
    WHILE (node <> NIL) DO BEGIN
      entry := node^.entry;
      { Search for this task }
      IF (entry.task = task) THEN BEGIN
        IF NOT found THEN BEGIN
          found := true;
        END;
        IF (entry.name <> name) THEN BEGIN
          name := entry.name;
          Inc(count);
          WriteLn('          ', count, ': ', name);
        END;
      END;
      node := node^.next;
    END;
  END;
  { Either no entries or task not found }
  IF (NOT found) THEN BEGIN
    WriteLn('          No work entries found for the task !!!');
  END;
  WriteLn('################### Persons working on task ###################');
END;

{ Print work summary for a person }
PROCEDURE PrintWorkSummaryForPerson(name: STRING);
```

```
VAR
  node: TimeRegistrationEntry;
  found: BOOLEAN;
  entry: WorkEntry;
BEGIN
  found := false;

  WriteLn('################### Work summary for person ###################');
  WriteLn('Person : ', name);

  { Only print result if entries are available }
  IF (list <> NIL) THEN BEGIN
    node := list;
    WHILE (node <> NIL) DO BEGIN
      entry := node^.entry;
      { Search for this person }
      IF (entry.name = name) THEN BEGIN
        WriteLn('         Task: ', entry.task:15, ' | Spend time: ',
TimeSpanUnit.TimeSpanToString(entry.spendTime));
        found := true;
      END;
      node := node^.next;
    END;
  END;
  { Either no entries or person not found }
  IF (NOT found) THEN BEGIN
    WriteLn('         No work entries found for the person !!!':10);
  END;
  WriteLn('################### Work summary for person ###################');
END;

{ Get busiest person }
FUNCTION BusiestPerson: STRING;
VAR
  tempList: TimeRegistrationCountList;
  tempNode, itNode, result: TimeRegistrationCount;
  node: TimeRegistrationEntry;
  entry: WorkEntry;
  i: INTEGER;
BEGIN
  BusiestPerson := '';
  tempList := NIL;
  itNode := NIL;
  i := 0;

  IF (list <> NIL) THEN BEGIN
    node := list;
    { iterate over all work entries }
    WHILE (node <> NIL) DO BEGIN
      Inc(i);
      entry := node^.entry;
      { writeln(i, '-node: ', entry.name); }
      { Search if entry already exists }
      tempNode := FindTimeRegistryCountByName(entry.name, tempList);
      { No entries has been found for the current name }
      IF (tempNode = NIL) THEN BEGIN
        tempNode := CreateTimeRegistrationCountNode(entry.name, 1);
        { writeln(i, '-new: ', tempNode^.name, ' - ', tempNode^.count); }
        { Add entry to empty list }
        IF (tempList = NIL) THEN BEGIN
          tempList := tempNode;
          { writeln(i, '-first: ', tempList^.name); }
        END
        { Add entry to not empty list }
        ELSE BEGIN
          itNode := tempList;
          WHILE (itNode^.next <> NIL) DO BEGIN
            { writeln(i, '-iterate: ', itNode^.next^.name); }
            itNode := itNode^.next;
          END;
          itNode^.next := tempNode;
          { writeln(i, '-added: ', itNode^.next^.name); }
        END;
      END
      ELSE BEGIN
        { Increase count by one }
```

```
        tempNode^.count := tempNode^.count + 1;
        { writeln(i, '-found: ', tempNode^.name, ' - ', tempNode^.count); }
      END;
      node := node^.next;
    END;

    { Get the entry with the highest count value }
    result := GetHighestTimeRegistryCount(tempList);
    { Get person name if entry has been found }
    IF (result <> NIL) THEN BEGIN
      BusiestPerson := result^.name;
    END;

    { Dispose list elements and the list }
    DisposeTempList(tempList);
  END;
END;

BEGIN
  { Set list to null as default }
  list := NIL;
END.
```

2.2 WorkManagementUnitTest

Folgend ist der Test der WorkManagementUnitTest angeführt.

```
PROGRAM WorkManagementUnitTest;

USES WorkManagementUnit, TimeSpanUnit;

{
  Adds a entry and prints the added entry to the console
}
PROCEDURE AddEntry(name, task: STRING; span: TimeSpan);
VAR
  entry: WorkEntry;
  error: ErrorCode;
BEGIN
  error := NONE;
  entry := WorkManagementUnit.CreateWorkEntry(name, task, span);
  WorkManagementUnit.AddWorkEntry(entry, error);
  WriteLn('Error: ', error:10, 'Name: ', entry.name, ' | Task: ', entry.task, ' | Time: ',
TimeSpanUnit.TimeSpanToString(entry.spendtime));
END;

{
  Tests GetTotalWorkTimeForPerson
}
PROCEDURE TestGetTotalWorkTimeForPerson;
VAR
  span: TimeSpan;
BEGIN
  WriteLn('---------------------- TestGetTotalWorkTimeForPerson ----------------------');
  WorkManagementUnit.Reset();
  span := TimeSpanUnit.CreateTimeSpan(0, 0, 0);
  WriteLn('No entries are present: ');
  WorkManagementUnit.GetTotalWorkTimeForPerson('Thomas', span);
  WriteLn('Total work time for person: ', TimeSpanUnit.TimeSpanToString(span));
  WriteLn('');

  WorkManagementUnit.Reset();
  AddEntry('Thomas' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Thomas' , 'Spec', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Thomas' , 'Test', TimeSpanUnit.CreateTimeSpan(1, 0, 0));
  AddEntry('Hannes' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Spec', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Test', TimeSpanUnit.CreateTimeSpan(2, 2, 0));
  span := TimeSpanUnit.CreateTimeSpan(0, 0, 0);
  WorkManagementUnit.GetTotalWorkTimeForPerson('Thomas', span);
  WriteLn('');
  WriteLn('Total work time for Thomas: ', TimeSpanUnit.TimeSpanToString(span));
  span := TimeSpanUnit.CreateTimeSpan(0, 0, 0);
  WorkManagementUnit.GetTotalWorkTimeForPerson('Hannes', span);
```

```
  WriteLn('Total work time for Hannes: ', TimeSpanUnit.TimeSpanToString(span));
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestGetTotalWorkTimeForPerson ----------------------');
  WriteLn('');
END;

{
  Tests the AddWorkEntry
}
PROCEDURE TestAddWorkEntry;
BEGIN
  WriteLn('---------------------- TestAddWorkEntry ----------------------');
  WorkManagementUnit.Reset();
  AddEntry('Thomas', 'Impl', TimeSpanUnit.CreateTimeSpan(8, 0, 1));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 0, 1));
  AddEntry('Thomas', 'Impl', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestAddWorkEntry ----------------------');
  WriteLn('');
END;

{
  Tests the procedure Rest
}
PROCEDURE TestReset;
BEGIN
  WriteLn('---------------------- TestReset ----------------------');
  WorkManagementUnit.Reset();
  AddEntry('Thomas', 'Impl', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestReset ----------------------');
  WriteLn('');
END;

{
  Tests the GetAverageWorkTimeForTask
}
PROCEDURE TestGetAverageWorkTimeForTask;
VAR
  span: TImeSpan;
BEGIN
  WriteLn('---------------------- TestGetAverageWorkTimeForTask ----------------------');
  WorkManagementUnit.Reset();
  WriteLn('No entries are present: ');
  WorkManagementUnit.GetAverageWorkTimeForTask('Doku', span);
  WriteLn('AverageTime for Doku: ', TimeSpanUnit.TimeSpanToString(span));
  WriteLn('');

  WorkManagementUnit.Reset();
  AddEntry('Thomas' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Thomas' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Hannes' , 'Spec', TimeSpanUnit.CreateTimeSpan(1, 0, 0));
  AddEntry('Hannes' , 'Spec', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  WriteLn('');
  span := TimeSpanUnit.CreateTimeSpan(0, 0, 0);
  WorkManagementUnit.GetAverageWorkTimeForTask('Impl', span);
  WriteLn('Average time for Impl: ', TimeSpanUnit.TimeSpanToString(span));
  span := TimeSpanUnit.createTimeSpan(0, 0, 0);
  WorkManagementUnit.GetAverageWorkTimeForTask('Spec', span);
  WriteLn('Average time for Spec: ', TimeSpanUnit.TimeSpanToString(span));
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestGetAverageWorkTimeForTask ----------------------');
  WriteLn('');
END;

{
  Tests the procedure PrintPersonForTask
}
PROCEDURE TestPrintPersonForTask;
BEGIN
  WorkManagementUnit.Reset();
  WriteLn('No entries are present: ');
  WorkManagementUnit.Reset();
  WorkManagementUnit.PrintPersonForTask('Implementation');
```

```
  WriteLn('');
  WriteLn('---------------------- TestPrintPersonForTask ----------------------');
  AddEntry('Maria' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(1, 0, 0));
  AddEntry('Thomas' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(1, 1, 1));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  WriteLn('');
  WorkManagementUnit.PrintPersonForTask('Implementation');
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestPrintPersonForTask ----------------------');
  WriteLn('');
END;

{
  Test the procedure PrintWorkSummaryForPerson.
}
PROCEDURE TestPrintWorkSummaryForPerson;
BEGIN
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestPrintWorkSummaryForPerson ----------------------');
  WriteLn('No entries are present');
  WorkManagementUnit.PrintWorkSummaryForPerson('Thomas');
  WriteLn('');
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 2, 2));
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Thomas' , 'Specification', TimeSpanUnit.CreateTimeSpan(1, 1, 1));
  AddEntry('Thomas' , 'Test', TimeSpanUnit.CreateTimeSpan(2, 2, 2));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Test', TimeSpanUnit.CreateTimeSpan(0, 1, 2));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(1, 1, 1));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 1));

  WorkManagementUnit.PrintWorkSummaryForPerson('Thomas');
  WriteLn('');
  WorkManagementUnit.PrintWorkSummaryForPerson('Hannes');
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestPrintWorkSummaryForPerson ----------------------');
  WriteLn('');
END;

{
  Tests the function BusiestPerson
}
PROCEDURE TestBusiestPerson;
VAR
  name: STRING;
BEGIN
  WriteLn('---------------------- TestBusiestPerson ----------------------');
  WorkManagementUnit.Reset();
  WriteLn('No entries:');
  WriteLn('Buisiest person: ', WorkManagementUnit.BusiestPerson);
  WriteLn('');
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(1, 1, 1));
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 2, 2));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Maria' , 'Test', TimeSpanUnit.CreateTimeSpan(2, 2, 2));
  name := WorkManagementUnit.BusiestPerson;
  WriteLn('Buisiest person: ', name);
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestBusiestPerson ----------------------');
  WriteLn('');
END;

BEGIN
  { Test AddWorkEntry }
  TestAddWorkEntry;
  { Test TestReset }
  TestReset;
  { TestGetTotalWorkTimeForPerson }
  TestGetTotalWorkTimeForPerson;
  { Tests GetAverageWorkTimeForTask }
```

```
  TestGetAverageWorkTimeForTask;
  { Test for PrintWorkSummaryForPerson }
  TestPrintWorkSummaryForPerson;
  { Test PrintPersonForTimeTask }
  TestPrintPersonForTask;
  { Test TestBusiestPerson }
  TestBusiestPerson;
END.
```

3 Tests

Folgend sind die Tests angeführt, die die WorkManagementUnit testen.
Bezüglich der Änderungen der Tests hat sich lediglich die Reset Prozedur geändert, da sich hier keine Größenangabe für den Speicher mehr definieren lässt. Ansonsten hat sich die bereits spezifizierte Schnittstelle nicht geändert. Der nennenswerteste Vorteil ist nun eine unbegrenzte Anzahl der speicherbaren Datensätze.

3.1 Reset

Dieser Test testet die Prozedur Rest, welche das Gedächtnis der WorkManagementUnit zurücksetzt indem sie alle Elemente der Liste löscht und den Pointer der Liste auf NULL setzt.

```
---------------------- TestReset ----------------------
Error: NONE       Name: Thomas | Task: Impl | Time: 8:0:0
Error: NONE       Name: Hannes | Task: Specification | Time: 8:0:0

Reset memory for next test
Work Entry disposed
Work Entry disposed
---------------------- TestReset ----------------------
```

Es werden testhalber Konsolenausgaben verwendet, welche nach dem Aufruf von Dispose angeführt sind, um zu zeigen das alle Einträge der Liste erfolgreich gelöscht wurden.

3.2 AddWorkEntry

Dieser Test testet das Verhalten der Prozedur AddWorkEntry, welche einen Eintrag im Gedächtnis anlegt.

Ungültige Zeitangaben:

Dieser Test testet das Verhalten der Prozedur, wenn die WorkEntry Instanzen ungültige Zeitangaben beinhalten.

```
---------------------- TestAddWorkEntry ----------------------
Error: TO_LONG    Name: Thomas | Task: Impl | Time: 8:0:1
Error: TO_SHORT   Name: Hannes | Task: Specification | Time: 0:0:1
---------------------- TestAddWorkEntry ----------------------
```

Wenn Zeitangaben kleiner 1 Minute oder größer 8 Stunden definiert werden, dann werden diese Einträge nicht gespeichert und der entsprechende Error zurückgegeben.

Gültige Zeitangaben:

Dieser Test testet das Verhalten der Prozedur, wenn die WorkEntry Instanzen gültige Zeitangaben beinhalten.

```
Error: NONE       Name: Thomas | Task: Impl | Time: 8:0:0
Error: NONE       Name: Hannes | Task: Specification | Time: 0:1:0
```

Sollten die Zeitangaben innerhalb der erlaubten Parameter liegen, so werden die Einträge erfolgreich gespeichert.

3.3 GetTotalWorkTimeForPerson

Dieser Test testet die Prozedur GetTotalWorkTimeForPerson, welche die Gesamtzeit die eine Person an irgendeinen Task gearbeitet hat liefert.

Keine Einträge vorhanden:

Dieser Test testet das Verhalten der Prozedur, wenn keine Einträge im System vorhanden sind.

```
No entries are present:
Total work time for person: 0:0:0
```

Es wird weiterhin eine TimeSpanUnit Instanz zurückgeben, welche mit 0:0:0 initialisiert wurde. Dieser Wert ist weiterhin ungültig und impliziert, dass es keine Einträge für diese Person gibt.

Einträge vorhanden:

Dieser Test testet das Verhalten der Prozedur, wenn Einträge im System vorhanden sind.

```
Error: NONE      Name: Thomas | Task: Impl | Time: 0:1:0
Error: NONE      Name: Thomas | Task: Spec | Time: 0:2:0
Error: NONE      Name: Thomas | Task: Test | Time: 1:0:0
Error: NONE      Name: Hannes | Task: Impl | Time: 0:2:0
Error: NONE      Name: Hannes | Task: Spec | Time: 0:2:0
Error: NONE      Name: Hannes | Task: Test | Time: 2:2:0

Total work time for Thomas: 1:3:0
Total work time for Hannes: 2:6:0
```

Es werden weiterhin die Gesamtzeiten, der gewählten Personen ausgegeben.

3.4 GetAverageWorkTimeForTask

Dieser Test testet die Prozedur GetAverageWorkTImeForTask, welche die Durchschnittszeiten von einem Task zurückliefert (Gesamtzeit DIV Anzahl der Einträge). Es wird weiterhin eine ganzzahlige Division verwendet, daher besteht immer noch eine gewisse Ungewissheit, bei bestimmten Werten.

Keine Einträge vorhanden:

Dieser Test testet das Verhalten der Prozedur, wenn keine Einträge im System vorhanden sind.

```
No entries are present:
AverageTime for Doku: 0:0:0
```

Auch hier wird weiterhin eine TimeSpan Instanz zurückgegeben, welche mit 0:0:0 initialisiert wurde. Auch in dieser Prozedur ist dieser Wert ein Indikator dafür, dass keine Einträge für den Task gefunden wurden.

Einträge vorhanden:

Dieser Test testet das Verhalten der Prozedur, wenn Einträge im System vorhanden sind.

```
Error: NONE      Name: Thomas | Task: Impl | Time: 0:2:0
Error: NONE      Name: Thomas | Task: Impl | Time: 0:1:0
Error: NONE      Name: Hannes | Task: Spec | Time: 1:0:0
Error: NONE      Name: Hannes | Task: Spec | Time: 0:2:0

Average time for Impl: 0:1:30
Average time for Spec: 0:31:0
```

Es werden die Durchschnittszeiten der gewählten Personen korrekt ausgegeben.

3.5 PrintPersonsForTask

Dieser Test testet die Prozedur PrintPersonForTaks, welche die Personen die an einen Task arbeiten auf der Konsole ausgibt.

Keine Einträge vorhanden:

Dieser Test testet das Verhalten der Prozedur, wenn keine Einträge im System vorhanden sind.

```
#################### Persons working on task ####################
Task    : Implementation
Persons :
          No work entries found for the task !!!
#################### Persons working on task ####################
```

Werden keine Einträge gefunden, so wird dies über die Ausgabe angezeigt.

Einträge vorhanden:

Dieser Test testet das Verhalten der Prozedur, wenn Einträge im System vorhanden sind.

```
---------------------- TestPrintPersonForTask ----------------------
Error: NONE       Name: Maria | Task: Implementation | Time: 0:2:0
Error: NONE       Name: Thomas | Task: Implementation | Time: 0:1:0
Error: NONE       Name: Maria | Task: Specification | Time: 1:0:0
Error: NONE       Name: Thomas | Task: Specification | Time: 0:2:0
Error: NONE       Name: Hannes | Task: Implementation | Time: 0:2:0
Error: NONE       Name: Hannes | Task: Implementation | Time: 0:1:0
Error: NONE       Name: Hannes | Task: Specification | Time: 1:1:1
Error: NONE       Name: Hannes | Task: Specification | Time: 0:1:1

#################### Persons working on task ####################
Task    : Implementation
Persons :
          1: Maria
          2: Thomas
          3: Hannes
#################### Persons working on task ####################
---------------------- TestPrintPersonForTask ----------------------
```

Es werden alle Personen eines Tasks korrekt ausgegeben.

3.6 PrintWorkSummaryForPerson

Dieser Test testet das Verhalten der Prozedur PrintWorkWummaryForPerson, welche alle Tasks an denen eine Person arbeitet ausgibt.

Keine Einträge vorhanden:

Dieser Test testet das Verhalten der Prozedur, wenn keine Einträge im System vorhanden sind.

```
#################### Work summary for person ####################
Person : Thomas
          No work entries found for the person !!!
#################### Work summary for person ####################
```

Wenn keine Einträge vorhanden sind wir dies in der Ausgabe angezeigt.

Einträge vorhanden:

Dieser Test testet das Verhalten der Prozedur, wenn Einträge im System vorhanden sind.

```
Error: NONE      Name: Thomas ¦ Task: Implementation ¦ Time: 0:2:2
Error: NONE      Name: Thomas ¦ Task: Implementation ¦ Time: 0:1:0
Error: NONE      Name: Thomas ¦ Task: Specification ¦ Time: 1:1:1
Error: NONE      Name: Thomas ¦ Task: Test ¦ Time: 2:2:2
Error: NONE      Name: Hannes ¦ Task: Implementation ¦ Time: 0:2:0
Error: NONE      Name: Hannes ¦ Task: Test ¦ Time: 0:1:2
Error: NONE      Name: Hannes ¦ Task: Specification ¦ Time: 1:1:1
Error: NONE      Name: Hannes ¦ Task: Specification ¦ Time: 0:1:1
#################### Work summary for person ####################
Person : Thomas
         Task:  Implementation ¦ Spend time: 0:2:2
         Task:  Implementation ¦ Spend time: 0:1:0
         Task:   Specification ¦ Spend time: 1:1:1
         Task:            Test ¦ Spend time: 2:2:2
#################### Work summary for person ####################
```

Es werden alle Task an die eine Person arbeitet korrekt ausgegeben.

3.7 BuisiestPerson

Dieser Test testet das Verhalten der Funktion BuisiestPerson, welche den Namen der Person liefert, die die meisten Einträge im System hat.

Keine Einträge vorhanden:

Dieser Test testet das Verhalten der Prozedur, wenn keine Einträge im System vorhanden sind.

```
No entries:
Buisiest person:
```

Wenn keine Einträge vorhanden sind, so wird ein Leerstring zurückgegeben.

Einträge vorhanden:

Dieser Test testet das Verhalten der Prozedur, wenn Einträge im System vorhanden sind.

```
Error: NONE      Name: Thomas ¦ Task: Implementation ¦ Time: 0:1:0
Error: NONE      Name: Maria ¦ Task: Specification ¦ Time: 1:1:1
Error: NONE      Name: Thomas ¦ Task: Implementation ¦ Time: 0:2:2
Error: NONE      Name: Hannes ¦ Task: Implementation ¦ Time: 0:2:0
Error: NONE      Name: Maria ¦ Task: Test ¦ Time: 2:2:2

Count Entry disposed
Count Entry disposed
Count Entry disposed

Buisiest person: Thomas
---------------------- TestBusiestPerson ----------------------
```

Wenn Einträge vorhanden sind, und es mehrere Personen mit derselben Anzahl von Einträgen gibt, so wird die erste Person für die Einträge in der temporären Liste gefunden wurden, angezeigt und die weiteren werden ignoriert. Des Weiteren wurde eine Testausgabe eingeführt um zu zeigen, dass die Elemente der Liste vor dem Verlassen der Funktion gelöscht werden.

4 Diskussion

Das Gedächtnis mit einer dynamischen Datenstruktur (Liste) austauschen war nicht besonders aufwendig. Nun ist es möglich unbegrenzt Daten im System zu speichern, daher ist diese Lösung besser als die Vorherige der Übung 5, die mit einer statischen Datenstruktur (Array) gearbeitet hat. Bezüglich der Performance sind mir keine nennenswerte Unterschiede aufgefallen.
Die Funktion, die die am meisten beschäftigte Person liefert, sollte meiner Meinung nach die Person liefern, die die meiste Zeit daran gearbeitet hat und nicht die Person, die die meisten Einträge hat.
Als Beispiel sei angeführt:

1. Thomas 0:0:2
2. Thomas 0:0:2
3. Hannes 2:0:0

Hierbei ist zu sehen, dass zwar Thomas die meisten Einträge besitzt, aber Hannes die meiste Zeit gearbeitet hat. Andererseits sollte es vielleicht auch möglich sein, die am meisten beschäftigte Person nach Task zu erhalten, um auch in der Lage zu sein, die meist beschäftigte Person per Task zu erhalten.
Die Aufgabenstellung der Aufgabe 5 verlangte eine Implementierung über Prozeduren, vielleicht könnte als Rückgabewert ein Datentyp verwendet werden, der das Resultat und einen möglichen Fehler als Attribute beinhaltet, besser wäre noch nur das Resultat zurückzugeben und eine Ausnahme (Exception) zu werfen, die vom Aufrufer behandelt werden muss.
Des Weiteren wäre eine Datenstruktur besser, die die Tasks/Personen repräsentiert und ihrerseits eine Liste aller Tasks/Personen beinhaltet. Damit könnten die Einträge optimaler erfasst und gehandhabt werden (zumindest für den Root. Bsp.: Tasks.persons, Person.tasks) und müssten nicht mühsam in einer unsortierten Liste gesucht werden.
Gegenüber der Verwendung von statischen Datenstrukturen muss hier besonders darauf geachtet werden, das die mit New erstellten Objekt auch immer über Dispose gelöscht werden, also das der reservierte Speicherplatz auch wieder freigegeben wird, was aber als akzeptabler Aufwand anzusehen ist, da es jetzt möglich ist den Speicher dynamisch, wenn er benötigt wird, zu verwenden.
Vielleicht wäre es auch gut eine Prozedur zu haben, die es ermöglicht Einträge zu löschen, damit Einträge einer Person oder eines Tasks gelöscht werden können.