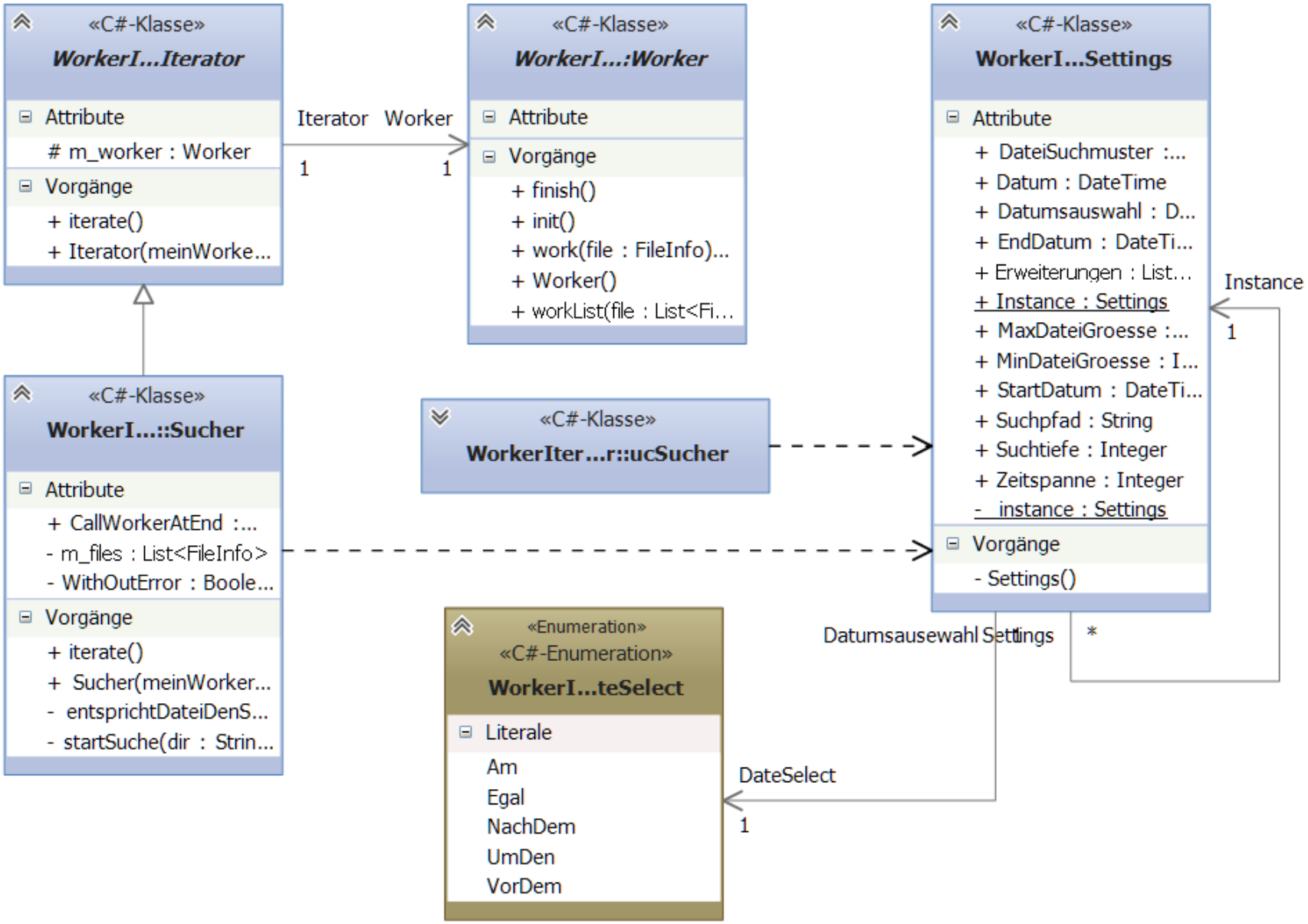
Worker-Iterator-Beispiel

Klassendiagramm der Bibliothek "WorkerIterator"



Die Basisklassen

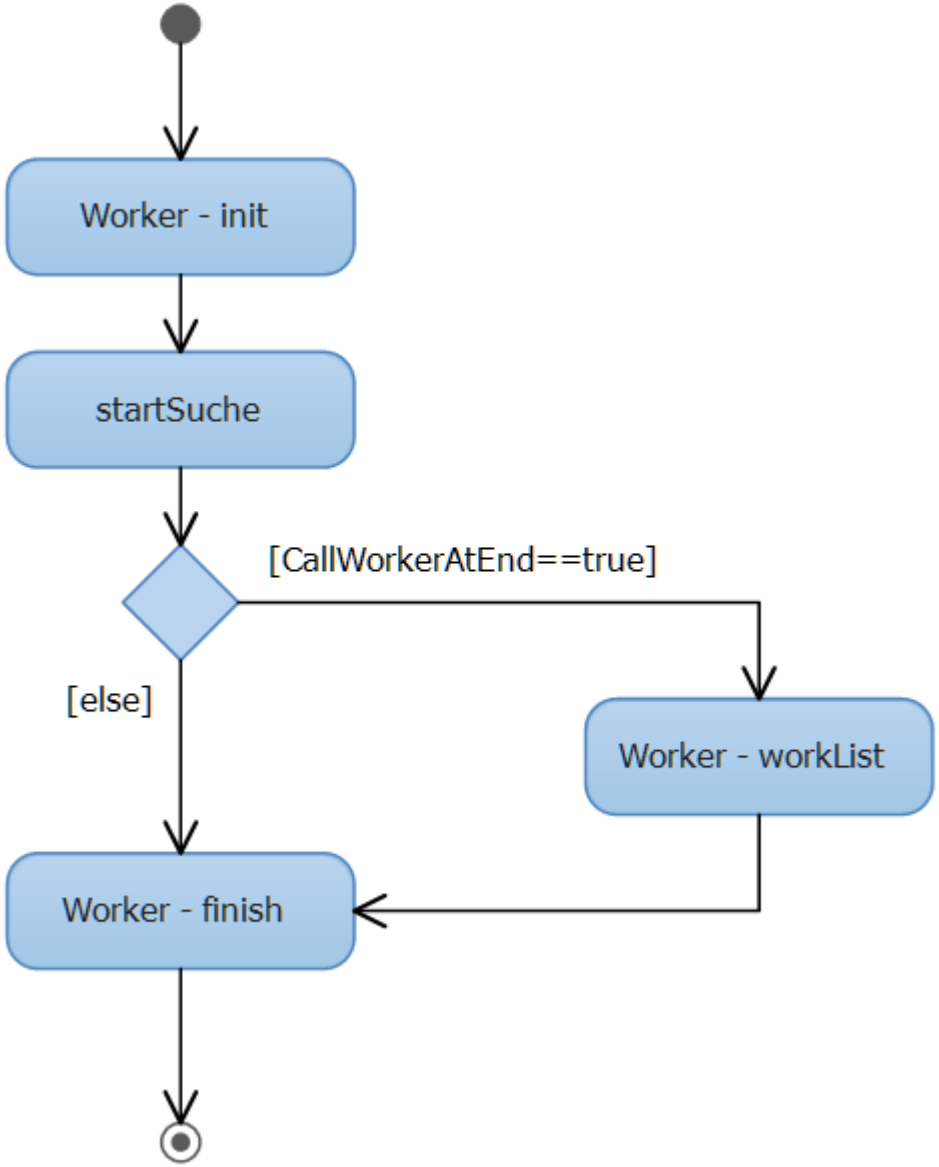
```
public abstract class Iterator
{
    protected Worker m_worker;

    public Iterator(Worker meinWorker)
    {
        m_worker = meinWorker;
    }

    public virtual void iterate()
    {
    }
}
```

```
public abstract class Worker
{
    public virtual void init()
    {
    }

    public virtual void finish()
    {
    }

    public virtual bool work(FileInfo file)
    {
        return false;
    }

    public virtual bool workList(List<FileInfo> file)
    {
        return false;
    }
}
```

Die Grundidee

- Ablauf
  - Der Iterator "durchsucht" eine Objektmenge nach bestimmten Kriterien
  - Wenn der Iterator ein Objekt gefunden hat, wird dieses dem Worker zur Bearbeitung übergeben
  - Wenn der Worker mit dem Objekt fertig ist, sucht der Iterator weiter
- Struktur
  - Der Iterator muß das Worker-Objekt kennen, damit er diesem ein gefundenes Objekt übergeben kann. Dies wird hier dadurch gelöst, daß der Iterator nur erstellt werden kann, wenn dem Konstruktor das Worker-Objekt übergeben wird
  - Die Worker-Klasse bietet eine definierte Methode an, die der Iterator nutzt, um die Bearbeitung anzustoßen (work)
  - Der Iterator selber wird mit iterate zur Suche nach Objekten angestoßen
  - Für bestimmte Worker kann es erforderlich oder sinnvoll sein, vor der Bearbeitung eine Initialisierung bzw. Abschluß der Bearbeitung vorzunehmen. Dazu können die Methoden init und finish vom Iterator verwendet werden.
  - Der Worker bietet darüber hinaus von seiner Schnittstelle die Möglichkeit, die Bearbeitung der gefundenen Objekte am Ende durchzuführen, dazu muß der Iterator die Objekte sammeln und dann am Ende dem Worker übergeben (workList)
  - Worker und Iterator sind abstrakte Basisklassen, von denen jeweils konkrete Klassen abzuleiten sind, in denen die Funktionalität implementiert werden muß

Die Iterator-Klasse Sucher

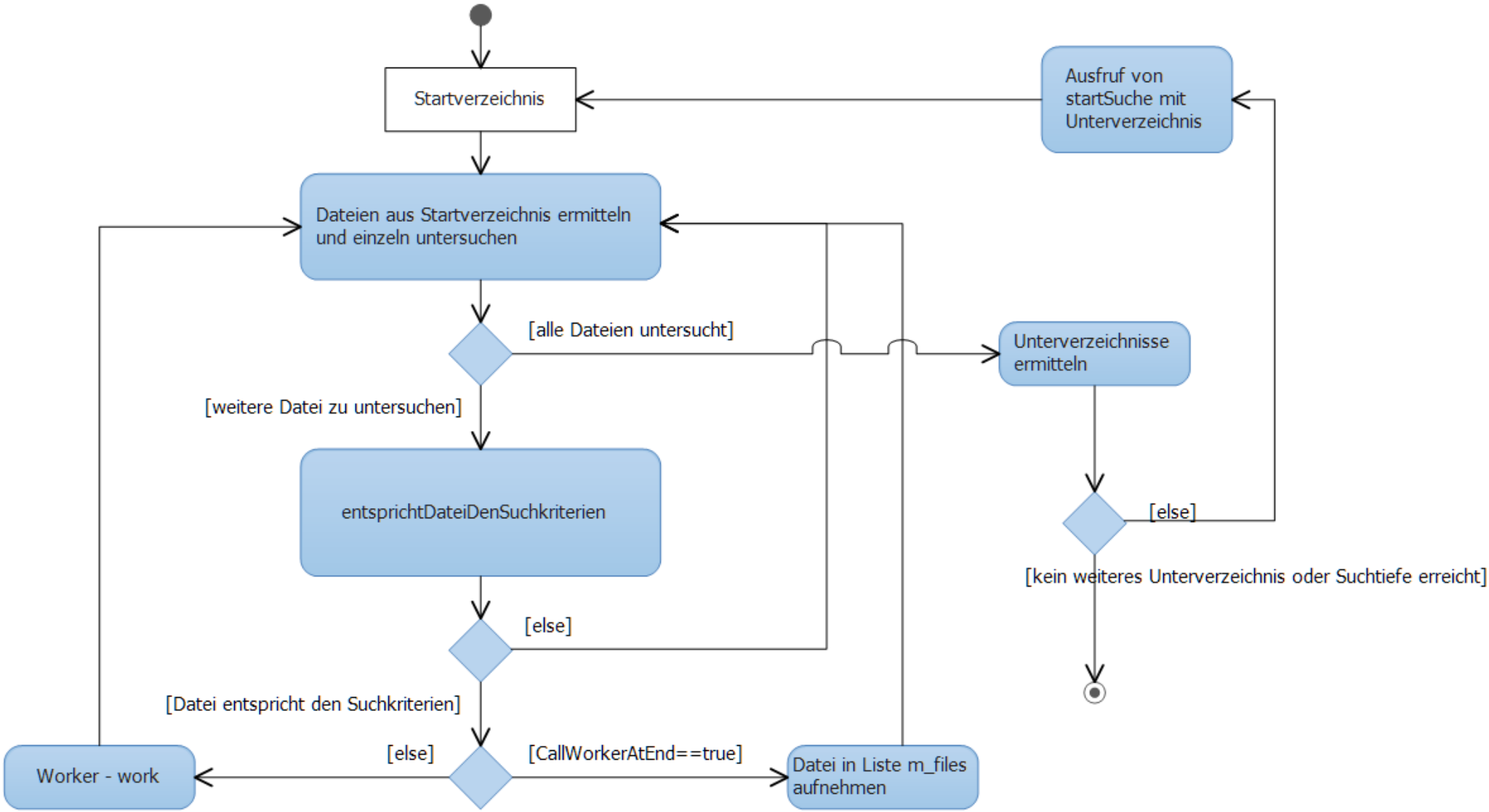


Sucher
Klasse
→ Iterator

Felder
- CallWorkerAtEnd
- m_files
- WithOutError

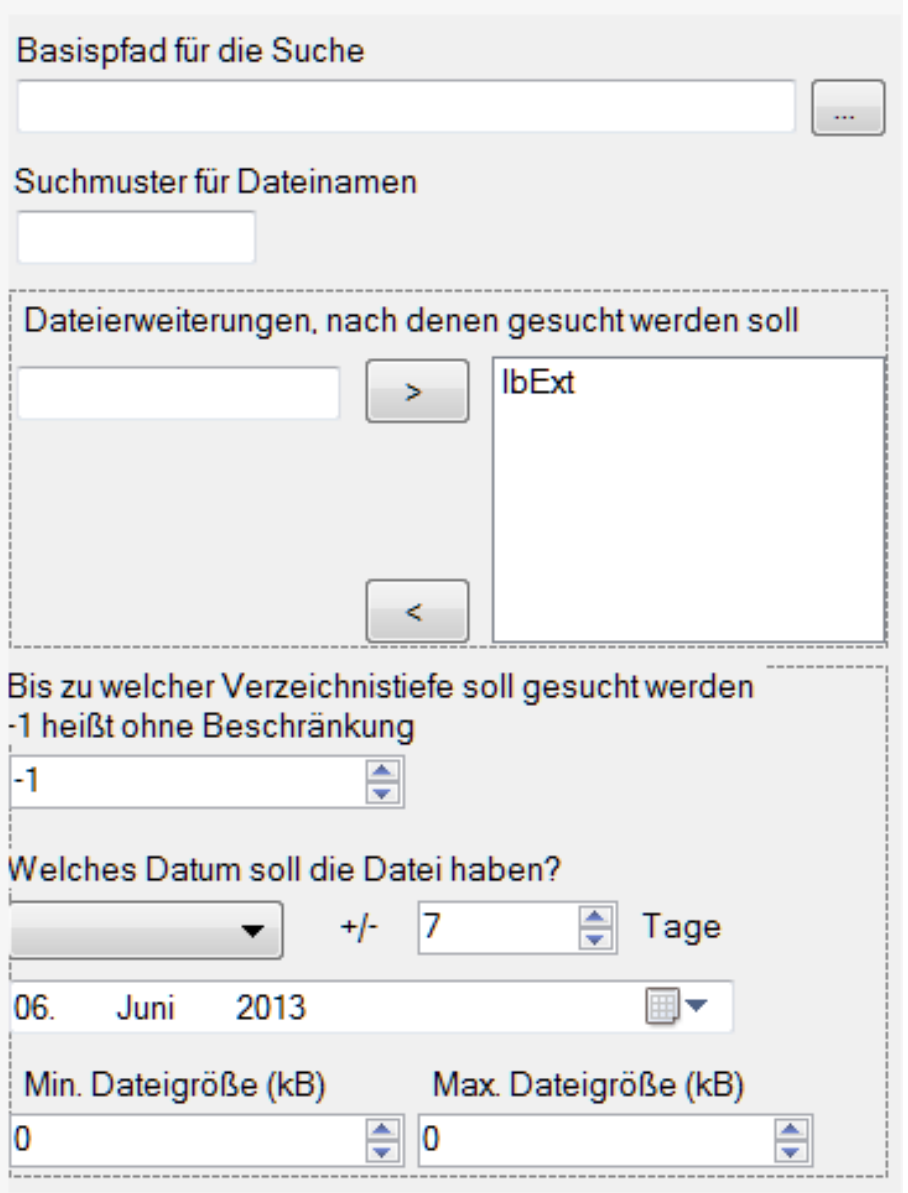
Methoden
- entsprichtDateiDen...
- iterate
- startSuche
- Sucher

startSuche – Aktivitätendiagramm



Das UserControl ucSucher

- Das UserControl soll die grafische Oberfläche zur Bearbeitung des Settings-Objektes darstellen
- Da das UserControl ohne "Start"-Button arbeiten soll, müssen Änderungen an den Steuerelementen unmittelbar in das Settings-Objekt geschrieben werden (anstatt dies beim Start-Aufruf einfach zu aktualisieren)
- Alternativ wäre das "public"-setzen der Steuerelemente eine Alternative, aus Gründen der Kapselung wird der obige Ansatz verfolgt
- Wenn das Control geöffnet wird, soll es die Daten aus dem Settings-Objekt zur Initialisierung nutzen. Dazu muß das Settings-Objekt auch die Angaben zur Datumsauswahl beinhalten und somit erweitert werden



Die Reaktion auf Änderungen

- Alle Steuerelemente lenken bei Changed-Ereignissen auf unten stehende Methode, die das Settings-Objekt auf die aktuellen GUI-Einstellungen aktualisiert
- Über RegisterChanged kann dieser Mechanismus ausgeschaltet werden

```
private void anyThingChanged(object sender, EventArgs e)
{
    if (RegisterChanged)
    {
        //Das Settings-Objekt mit allen Einstellungen der GUI füllen
        Settings.Instance.Suchpfad = tbPath.Text;
        Settings.Instance.Suchtiefe = (int)nudSuchtiefe.Value;
        Settings.Instance.DateiSuchmuster = tbDateiSuchmuster.Text.ToString();
        setExts();
        setDate();
        Settings.Instance.MaxDateiGroesse = (int)nudMaxSize.Value;
        Settings.Instance.MinDateiGroesse = (int)nudMinSize.Value;
    }
}
```

Das Initialisieren des UserControl aus dem Settings-Objekt

- Beim Erstellen des Controls wird unten stehende Methode gerufen, die alle GUI-Elemente aus dem Settings-Objekt aktualisiert
- Hier wird die Variable RegisterChanged auf false gesetzt, damit das Control bei Änderung der Steuerelemente nicht rückgreifend das Settings-Objekt aktualisiert, da sonst eine endlose Kette angestoßen werden könnte,

```
public void initFromSettings()
{
    //Alten Zustand von RegisterChanged merken und dann ausschalten
    bool oldRegisterChanged = RegisterChanged;
    RegisterChanged = false;
    //Steuerelemente setzen
    tbPath.Text = Settings.Instance.Suchpfad;
    nudSuchtiefe.Value = Settings.Instance.Suchtiefe;
    tbDateiSuchmuster.Text =Settings.Instance.DateiSuchmuster;
    nudMaxSize.Value = Settings.Instance.MaxDateiGroesse;
    nudMinSize.Value = Settings.Instance.MinDateiGroesse;
    foreach (string itm in Settings.Instance.Erweiterungen)
    {
        lbExt.Items.Add(itm);
    }
    cbDatumSuche.SelectedIndex = (int)Settings.Instance.Datumsauswahl;
    dtpDatum.Value = Settings.Instance.Datum;
    nodTage.Value = Settings.Instance.Zeitspanne;
    //Änderungs-Management auf alten Wert zurücksetzen
    RegisterChanged = oldRegisterChanged;
}
```

Erweiterung des UserControls

- Das UserControl soll im Folgenden ausgebaut werden, so dass bspw. eingestellt werden kann, ob Dateierweiterungen per GUI auswählbar sind oder nur im Settings-Objekt gehalten werden
  - Hintergrund ist die Suche nach Dateien bestimmter Typen, die nicht auswählbar sein sollen
  - Das UserControl soll dann nur über die anderen Möglichkeiten zur Einschränkung der Dateiauswahl verfügen und die betroffenen Controls ausblenden
- Zur Vereinfachung werden die auszublendenden Controls auf ein Panel gelegt. So wird nur das Panel ausgeblendet anstatt eine Liste von Steuerelementen
- Für die Steuerung der Sichtbarkeit erhält das UserControl eine öffentliche Eigenschaft, mit der die Einstellung erfolgen soll
  - Generell steuert die boole'sche Eigenschaft die Sichtbarkeit des Panels panErweiterungenAngeben
  - Da beim Ausblenden des Panels eine Lücke entstehen würde, werden die darunter stehenden Elemente ebenfalls in ein Panel (panFileProperties) eingefügt. Dieses Panel wird beim Ausblenden hinsichtlich der Top-Position an die Stelle des panErweiterungenAngeben-Panels (ErweiterungPos) geschoben, sonst wieder an die ursprüngliche Position (PropertiesPos)
  - Gleichzeitig wird die Größe des UserControls selber um die entsprechenden Werte verkleinert
  - Beim Erstellen des Controls werden die Werte ErweiterungPos, PropertiesPos, und die ursprüngliche Höhe des UserControls (ucHeight) gemerkt

Die Erzeugung des UserControls

```
public ucSucher()
{
    //Änderungs-Management ausschalten
    bool oldRegisterChanged = RegisterChanged;
    RegisterChanged = false;
    InitializeComponent();
    this.cbDatumSuche.SelectedIndex = 0;
    ErweiterungPos = panErweiterungenAngeben.Top;
    PropertiesPos = panFileProperties.Top;
    ucHeight = this.Height;
    initFromSettings();
    //Änderungs-Management auf alten Wert zurücksetzen
    RegisterChanged = oldRegisterChanged;
}
```

Die Dateierweiterungs-Eigenschaft des UserControls

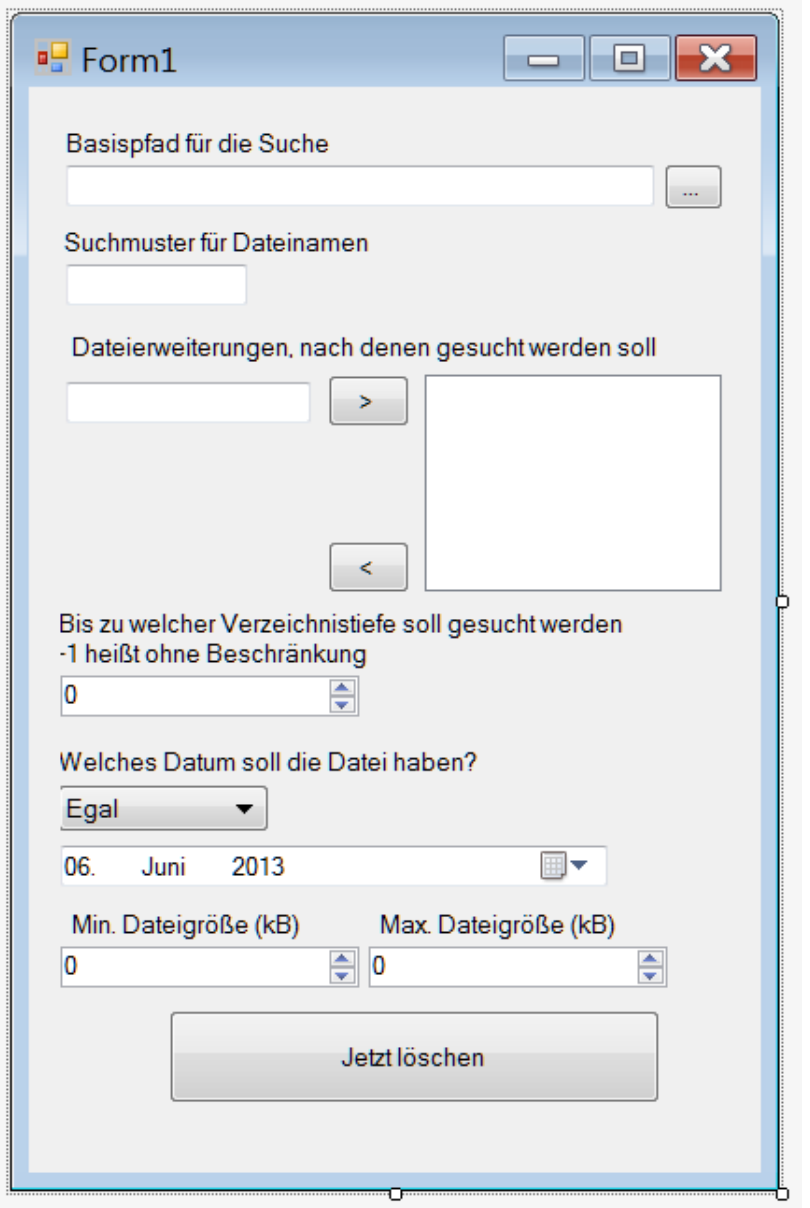
```
public bool ErweiterungenWaehlenAnzeigen
{
    get
    {
        return panErweiterungenAngeben.Visible;
    }
    set
    {
        panErweiterungenAngeben.Visible = value;
        if (panErweiterungenAngeben.Visible)
        {
            panFileProperties.Top = PropertiesPos;
            this.Height = ucHeight;
        }
        else
        {
            panFileProperties.Top = ErweiterungPos;
            this.Height = ucHeight - panErweiterungenAngeben.Height;
        }
    }
}
```

Bibliothek WorkerIterator - Zusammenfassung

- Die Bibliothek stellt die Basisklasse Sucher für eine Dateisuche nach Kriterien, die über die Singleton-Klasse Settings definiert werden, bereit
- Die den Suchkriterien entsprechenden Dateien werden, abhängig von der Eigenschaft CallWorkerAtEnd des Sucher-Objektes, entweder direkt nach Auffinden oder am Ende dem Worker übergeben, der vom Sucher das zugehörige FileInfo-Objekt übergeben bekommt
- Die Klasse Worker der Bibliothek ist abstrakt und kann nur verwendet werden, wenn eine davon abgeleitete Klasse erstellt wird. In dieser Klasse kann in der work- bzw. workList-Methode implementiert werden, wie mit den FileInfo-Objekten verfahren werden soll
- Für die Definiton der Suchkriterien stellt die Bibliothek das UserControl ucSucher bereits, das in ein Formular eingebunden werden kann und das Settings-Singleton-Objekt über eine GUI bearbeitbar macht
- Für die Nutzung des in der Bibliothek angebotenen Worker-Iterator-Musters in einer eigenen Anwendung müssen folgende Arbeiten erledigt werden:
  - Einbinden des Verweises auf die Bibliothek
  - Verwenden des UserControls oder einer eigenen Implementierung zur Definition der Sucheinstellungen im Settings-Singleton-Objekt
  - Erstellen einer Worker-Klasse und die Implementierung der notwendigen Methoden
  - Erzeugen eines Sucher-Objektes mit Übergabe des Worker-Objektes der eigenen Worker-Klasse
  - Start des Suchers über Aufruf der Iterate-Methode

Nutzung der Bibliothek zum Löschen von Dateien

- Nach Erstellen eines Verweises auf die Bibliothek WorkerIterator wird ein Formular erstellt, in dem das UserControl ucSucher und eine Button platziert werden
- Für den Button wird eine Aktion auf das Click-Ereignis angelegt, bei der das Programm die Dateien, die obigem Suchschema entsprechen, löschen soll



Nutzung der Bibliothek zum Löschen von Dateien

- Es wird eine Klasse Loescher erstellt, die vom Worker erbt und nur die work-Methode überschreibt
- workList funktioniert dann natürlich nicht

```
class Loescher:WorkerIterator.Worker
{
    public override bool work(System.IO.FileInfo file)
    {
        file.Delete();
        return true;
    }
}
```

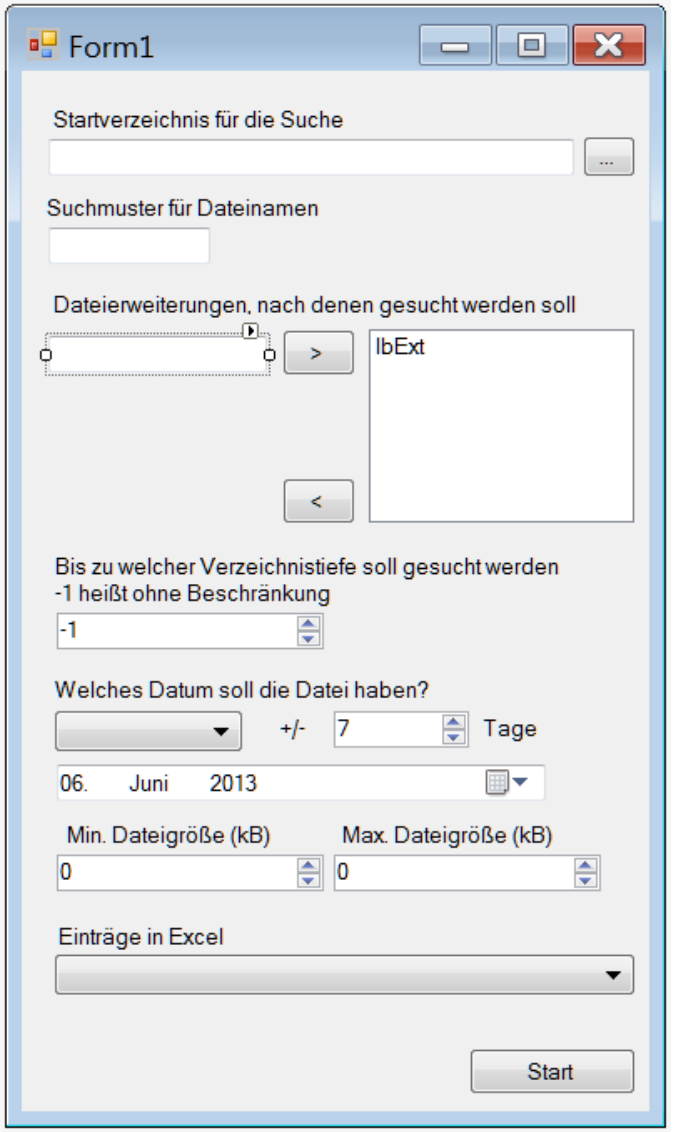
- Beim Klick auf den Button wird nun nur ein Sucher-Objekt erstellt, das ein Loescher-Objekt (seinen Worker) übergeben bekommt
- Da workList nicht implementiert wurde, muß CallWorkerAtEnd auf false gesetzt werden
- Dann wird der Iterator mit iterate gestartet

```
private void button1_Click(object sender, EventArgs e)
{
    Sucher ns = new Sucher(new Loescher());
    ns.CallWorkerAtEnd = false;
    ns.iterate();
}
```

Nutzung der Bibliothek, um Dateien in Excel zu schreiben

- Auch hier wird eine Klasse als Ableitung des Workers erstellt (ExcelSchreiber)
- Auch hier wird das UserControl in eine Oberfläche integriert. Zusätzliche Steuerelemente legen die Art der Einfügung in Excel (ExcelSettings) fest
- Aufruf erfolgt analog wie beim Loescher-Projekt

```
public partial class frmMainMitUserControl : Form
{
    public frmMainMitUserControl()
    {
        InitializeComponent();
        cbExcelSort.SelectedIndex = 0;
    }

    private void btnStart_Click(object sender, EventArgs e)
    {
        excelSettings.Instance.InExcelWieVerteilen =
            excelSettings.CastToEnum<eInExcelWieVerteilen>(cbExcel

        //Suche starten
        Sucher m_sucher = new Sucher(new ExcelSchreiber());
        m_sucher.CallWorkerAtEnd = false;
        m_sucher.iterate();

    }
}
```



Aufbau des ExcelSchreiber-Workers

- Der ExcelSchreiber überschreibt alle vier Methoden der Worker-Basisklasse. Die Aufgaben der einzelnen Methoden sind:
- init
  - Startet Excel und erstellt eine neue Arbeitsmappe. Stellt sicher, daß die Arbeitsmappe nur ein Tabellenblatt enthält. Beim erfolgreichen Abschluss (ohne Fehler) wird die Variable InitSuccessful auf true gesetzt
- work
  - Verarbeitet ein FileInfo-Objekt. Ermittelt anhand der Einstellungen in ExcelSettings den Namen des Tabellenblattes durch die Funktion getSheetName, stellt sicher, daß es dieses Tabellenblatt in richtiger Reihenfolge gibt (getOrCreateWorkSheet mit Unterstützung durch das Dictionary m_sheetDict) und trägt die Daten der Dateien im Tabellenblatt ein. Im Dictionary m_curRow merkt sich der Excelschreiber, in welche Zeile des Tabellenblattes geschrieben werden muss
- workList
  - ruft für jedes Element der Liste work auf
- finish
  - Öffnet jedes Tabellenblatt und passt die Breite verschiedener Spalten an die Inhalte an

Die Funktion getSheetName

```
private string getSheetName(workerObject file)
{
    string SheetName = "Dateien";
    switch (excelSettings.Instance.InExcelWieVerteilen)
    {
        case eInExcelWieVerteilen.ProErweiterungEinTabellenblatt:
            SheetName = file.Extension.ToLower().Replace(".", "");
            break;
        case eInExcelWieVerteilen.ProJahrEinTabellenblatt:
            SheetName = file.CreationTime.Year.ToString();
            break;
        case eInExcelWieVerteilen.ProMonatEinTabellenblatt:
            SheetName = file.CreationTime.Year.ToString() + "-" +
                file.CreationTime.Month.ToString();
            break;
    }
    return SheetName;
}
```

Die Funktion getOrCreateWorkSheet

```
public Excel.Worksheet getOrCreateWorkSheet(string name)
{
    if (!m_sheetDict.ContainsKey(name))
    {
        m_sheetDict[name] = null;
        int i = 0;
        foreach (KeyValuePair<string, Excel.Worksheet> kvp in m_sheetDict)
        {
            if (kvp.Key == name)
            {
                break;
            }
            i++;
        }
        if (!firstWasRenamed)
        {
            m_sheetDict[name] = (Excel.Worksheet)m_wbk.Worksheets[1];
            firstWasRenamed = true;
        }
        else
        {
            if (i == 0)
            {
```

Die work-Methode

```
public override bool work(workerObject file)
{
    if (!InitSuccessful) return false;
    try
    {
        string SheetName = getSheetName(file);
        Excel.Worksheet mm_ws = getOrCreateWorkSheet(SheetName);

        int mm_row = m_curRow[SheetName];
        if (mm_row == 1)
        {
            mm_ws.Cells[mm_row, 1] = "Verzeichnis";
            mm_ws.Cells[mm_row, 2] = "Datei";
            mm_ws.Cells[mm_row, 3] = "Erweiterung";
            mm_ws.Cells[mm_row, 4] = "Datum";
            mm_ws.Cells[mm_row, 5] = "Größe";
        }
        mm_row++;
        mm_ws.Cells[mm_row, 1] = file.DirectoryName;
        mm_ws.Cells[mm_row, 2] = file.Name;
        mm_ws.Cells[mm_row, 3] = file.Extension.Substring(1).ToLower();
        mm_ws.Cells[mm_row, 4] = file.CreationTime;
        mm_ws.Cells[mm_row, 5] = Math.Round(file.Length / 1024.0, 2);
        m_curRow[SheetName] = mm_row;
```