Ein erstes SCORM-Lernpaket

Sebastian Roith, https://www.roith-unterrichtet.de

Zielsetzung

Als Einführung in das Erstellen von SCORM-Lernpaketen werden wir ein Lernpaket programmieren, welches seine Bearbeiterin oder seinen Bearbeiter namentlich begrüßt und mitzählt, wie oft es bereits von derselben Person aufgerufen wurde. Nach dem dritten Aufruf soll es als abgeschlossen gelten.

Hinweise

- Es gibt mehrere Versionen von SCORM. Praktisch sind aber nur zwei relevant: *SCORM 1.2* und *SCORM 2004 4th Edition*. SCORM 1.2 ist einfacher, genießt breitere Unterstützung und bietet alles, was wir für dieses Projekt brauchen. Also beschränken wir uns hierauf.
- Einiges in diesem Tutorial ist flapsig formuliert oder anderweitig vereinfachend. Es geht mir um einen möglichst einfachen Einstieg in Grundlegendes, nicht um größtmögliche Exaktheit. Und auch möglich, dass ich etwas nicht ganz richtig verstanden habe.
- Ich habe auf sinnvolle, fortgeschrittene Sprachelemente von Javascript verzichtet, um nicht dadurch zusätzliche Verständnishürden einzubauen. Mit diesem Programmierstil stößt man bei größeren Projekten jedoch schnell an Grenzen.

Die Schnittstelle zur Lernplattform

Funktion der Schnittstelle

Eine Lernplattform, die SCORM-Lernpakete einbinden kann, hinterlegt für das eingebundene Lernpaket eine Schnittstelle namens `API` in einem dem Lernpaket übergeordneten Fenster (jeder Tab eines Webbrowsers, jeder Iframe usw. zählt in diesem Zusammenhang als Fenster). Neben drei Methoden, die weitere Informationen über auftretende Fehler liefern, besteht diese Schnittstelle aus folgenden Methoden (je mit beispielhafter Verwendung):

`API.LMSInitialize("")` Muss einmalig vor allen anderen Methoden aufgerufen werden, wenn das Lernpaket die Schnittstelle verwenden möchte

`API.LMSGetValue("cmi.core.student_name")` Der Rückgabewert ist der von der Lernplattform gespeicherte Wert für das angegebene Element (im Beispiel `cmi.core.student_name`, was für den Namen der Person steht, die das Lernpaket gerade nutzt).

`API.LMSSetValue("cmi.core.lesson_status", "incomplete")` Speichert den als zweiten Parameter angegebenen Wert als neuen Wert des zuerst angegebenen Elements. Im Beispiel wird also der Wert des Elements `cmi.core.lesson_status` (das ist der Status des Lernpakets, sofern dieses nur aus einem „Inhalt“ besteht) auf `"incomplete"`, also begonnen, aber noch nicht abgeschlossen gesetzt.

`API.LMSCommit("")` Sollte nach zusammengehörenden `LMSSetValue`-Aufrufen aufgerufen werden. Dann weiß die Lernplattform, dass all diese Werte wieder zueinander passen und gespeichert werden können.

`API.LMSFinish("")` Sollte einmalig zuletzt aufgerufen werden, um der Lernplattform mitzuteilen, dass es keine weitere Kommunikation über diese Schnittstelle mehr geben wird (bevor das Lernpaket erneut geöffnet wird).

Von `LMSGetValue` abgesehen ist der Rückgabewert stets `"true"` bzw. im Fehlerfall `"false"`. Wo eine leere Zeichenkette als Parameter angegeben wurde, muss immer als einziger Parameter eine leere Zeichenkette übergeben werden.

Auffinden der Schnittstelle

`window.parent` speichert das dem aktuellen Fenster übergeordnete Fenster. Achtung: Wenn es kein anderes übergeordnetes Fenster gibt, zählt das aktuelle Fenster wieder als übergeordnetes Fenster. Sofern die Schnittstelle vorhanden ist, hat ein direkt oder indirekt übergeordnetes Fenster als Attribut `API` die beschriebene Schnittstelle. Dann könnte sie so gefunden werden:

```
function findeSchnittstelle() {
    // Wir starten mit dem aktuellen Fenster
    var fenster = window;
    // Solange wir die Schnittstelle nicht gefunden haben, weitersuchen
    while (true) {
        // Prüfen, ob das akutell betrachtete Fenster die Schnittstelle hat
        if (fenster.API) {
            // Wenn ja, die Suche abbrechen und die gefundene Schnittstelle zurückgeben
            return fenster.API;
        }
        // Andernfalls weiter mit dem übergeordneten Fenster
        fenster = fenster.parent;
    }
}
```

Wenn allerdings keine Schnittstelle vorhanden ist oder der Zugriff auf ein übergeordnetes Fenster aus Sicherheitsgründen vom Browser unterbunden wird, führt die oben aufgeführte Funktion zu einer Endlosschleife bzw. einem Fehler. Besser ist daher, die Anzahl an zu durchsuchende übergeordnete Fenster zu beschränken und zu prüfen, ob es wirklich ein (anderes) übergeordnetes Fenster gibt. Eine solche verbesserte Version von `findeSchnittstelle` sieht beispielsweise so aus:

```
function findeSchnittstelle() {
    // Maximale Anzahl an verbleibenden, zu durchsuchenden übergeordneten Fenstern
    var nochEbenen = 10;
    // Wir starten mit dem aktuellen Fenster
    var fenster = window;
```

```
    // Solange `fenster` einen gültigen Wert hat
    // und wir die Maximalanzahl nicht erreicht haben, weitersuchen
    while (fenster && nochEbenen > 0) {
        // Prüfen, ob das akutell betrachtete Fenster die Schnittstelle hat
        if (fenster.API) {
            // Wenn ja, die Suche abbrechen und die gefundene Schnittstelle zurückgeben
            return fenster.API;
        }
        // Abbrechen, wenn aktuelles Fenster sich selbst als übergeordnetes Fenster hat
        if (fenster.parent == fenster) {
            return null;
        }
        // Andernfalls weiter mit dem übergeordneten Fenster
        fenster = fenster.parent;
        // Vermerken, das nun ein übergeordnetes Fenster weniger noch zu berachten ist
        nochEbenen = nochEbenen - 1;
    }
    // Wenn keine Schnittstelle gefunden wurde: `null` als Rückgabewert festlegen
    return null;
}
```

Version 1: Dateien im Lernpaket

In diesem Kapitel schnüren wir ein Lernpaket zusammen, indem wir die benötigten Dateien in einem Archiv speichern. Am Kapitelende haben wir ein Lernpaket, dass zwar noch nicht wirklich mit der Lernplattform kommuniziert, allerdings schon eingebunden werden kann.

HTML-Datei

Wenn wir alles richtig machen, öffnet die Lernplattform eine HTML-Datei unseres Lernpakets in einem untergeordneten Fenster (also einem neuen Fenster oder einem Iframe). Diese Datei müssen wir erstellen und mitliefern. Folgende Vorlage eignet sich als Ausgangspunkt für den Dateiinhalt. Den angezeigten Inhalt werden wir später über ein Skript verändern:

```
<!DOCTYPE html>
<html lang="de">
    <head>
        <title>Ein Lernpaket</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <meta name="viewport" content="width=device-width" />
        <script src="lernpaketfunktionen.js"></script>
    </head>
    <body>
        <h1>Wird geladen...</h1>
        <p>Wenn dieser Text nicht verschwindet, ist ein Fehler aufgetreten.</p>
    </body>
</html>
```

Diese Datei speichern wir in einem zunächst leerem Ordner als `index.html` ab. Ein anderer Name wäre auch möglich. Dann muss dieser aber auch im übernächsten Abschnitt statt `index.html` angegeben werden.

Javascript-Datei

Alles zu programmierende, damit das Lernpaket so funktioniert wie es soll, werden wir später in die Datei `lernpaketfunktionen.js` schreiben. Zunächst kopieren wir nur die schon fertige Funktion `findeSchnittstelle` hinein und speichern sie im selben Ordner wie `index.html`. Unterordner usw. wären auch okay, aber dann müssten wir den Pfad in `index.html` und gleich in `imsmanifest.xml` anpassen.

Manifest

Jedes Lernpaket braucht eine Datei namens `imsmanifest.xml`. Diese liefert der Lernplattform Informationen über den Aufbau des Lernpakets. In unserem Fall, dass es die Dateien `index.html` und `lernpaketfunktionen.js` gibt, dass mit `index.html` gestartet werden soll und dass alles zusammen den einzigen „Inhalt“ des Lernpakets bildet. Dazu speichern wir folgendes als `imsmanifest.xml` ab:

```
<?xml version="1.0" standalone="no" ?>
<manifest
    version="1"
    identifier="erstesLernpaket"
    xmlns="http://www.imsproject.org/xsd/imscp_rootv1p1p2"
    xmlns:adlcp="http://www.adlnet.org/xsd/adlcp_rootv1p2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.imsproject.org/xsd/imscp_rootv1p1p2 imscp_rootv1p1p2.xsd
        http://www.imsglobal.org/xsd/imsmd_rootv1p2p1 imsmd_rootv1p2p1.xsd
        http://www.adlnet.org/xsd/adlcp_rootv1p2 adlcp_rootv1p2.xsd"
>
    <metadata>
        <schema>ADL SCORM</schema>
        <schemaversion>1.2</schemaversion>
    </metadata>

    <organizations default="default_org">
        <organization identifier="default_org">
            <title>Ein erstes Lernpaket</title>
            <item identifier="operationcalls" identifierref="content">
                <title>Mehrfache Begrüßung</title>
            </item>
        </organization>
    </organizations>

    <resources>
        <resource
            identifier="content"
```

```
            type="webcontent"
            adlcp:scormtype="sco"
            href="index.html"
        >
            <file href="lernpaketfunktionen.js" />
            <file href="index.html" />
        </resource>
    </resources>
</manifest>
```

Was der Attributwert von `identifier` (im `resource`-Tag) und `identifierref` (im `item`-Tag) ist, ist egal, solange es beide male derselbe Attributwert ist. Ebenso bei `default` im `organizations`-Tag und `identifier` im `organization`-Tag. Der Textinhalt der `title`-Tags ist frei wählbar. Die Attribute `version` (Versionsnummer) und `identifier` (ID des Lernpakets) können auch frei gewählt werden. Lernplattformen können diese Werte nutzen, um abzuschätzen, ob ein Lernpaket durch ein völlig anderes ersetzt wird oder durch eine neue Version desselben Lernpakets. Und entsprechend dieser Abschätzung die bisherigen Lernfortschritte löschen oder beibehalten.

Im `resource`-Tag werden alle Dateien aufgelistet. Wichtig ist das Attribut `href`, das angibt, welche Datei beim Start des Lernpakets geöffnet werden soll.

XSD-Dateien

Aus Kompatibilitätsgründen sollten die vier Dateien `ims_xml.xsd`, `imscp_rootv1p1p2.xsd`, `imsmd_rootv1p2p1.xsd` und `adlcp_rootv1p2.xsd` von einem anderen Lernpaket in das Hauptverzeichnis unseres Lernpakets kopiert werden. (Allerdings wird kaum eine Lernplattform überhaupt nur merken, wenn diese Dateien fehlen; sie beschreiben lediglich maschinenlesbar das Dateiformat.)

Zusammenpacken

Diese sechs Dateien werden zu einem ZIP-Archiv zusammengefügt. Wichtig dabei ist, dass diese Dateien direkt im ZIP-Archiv enthalten sind (und nicht etwa in einem Ordner, der im ZIP-Archiv enthalten ist). Dieses ZIP-Archiv kann in einer Lernplattform als Lernpaket eingebunden werden. Wurde bisher alles richtig gemacht, wird die HTML-Datei `index.html` angezeigt. Es findet allerdings noch keine Kommunikation mit der Lernplattform statt und wir sehen nur die Fehlermeldung.

Version 2: Grundlegende Funktionalität unseres Lernpakets

Wir ergänzen nun in `lernpaketfunktionen.js` fortlaufend Methoden und einiges mehr. So erhalten wir am Kapitelende ein Lernpaket, welches bereits mit der Lernplattform kommuniziert.

Starten der Kommunikation

Der Einfachheit halber speichern wir zentral die Schnittstelle, die wir hoffentlich entdecken. Dazu unter der bereits vorhandenen Funktion `findeSchnittstelle` ergänzen:

```
var API = null;
```

Anfangs müssen wir die Schnittstelle finden (also die Funktion `findeSchnittstelle` aufrufen) und dann `LMSInitialize` der Schnittstelle. Dies soll die Funktion `initKommunikation` erledigen:

```
function initKommunikation() {
    API = findeSchnittstelle();
    if (API == null) {
        zeigeKeineKommunikation();
    } else {
        API.LMSInitialize("");
        zeigeInhalt();
    }
}
```

Die Methoden `zeigeKeineKommunikation` und `zeigeInhalt` sollen den angezeigten Inhalt an die jeweilige Situation anpassen. Diese werden wir später programmieren.

Damit diese Methode aufgerufen wird, sobald die Seite ausreichend geladen ist, ergänzen wir unter ihr:

```
window.addEventListener("DOMContentLoaded", initKommunikation);
```

Beenden der Kommunikation

Das Beenden der Kommunikation soll die Methode `finishKommunikation` übernehmen:

```
function finishKommunikation() {
    if (API != null) {
        API.LMSFinish("");
    }
}

window.addEventListener("beforeunload", finishKommunikation);
```

Die letzte Zeile bewirkt, dass die Methode `finishKommunikation` automatisch aufgerufen wird, bevor die Seite warum auch immer verlassen wird.

Anpassen des Seiteninhalts

`document.body` repräsentiert den `body`-Tag des HTML-Dokuments. Indem wir dessen Attribut `innerHTML` einen HTML-Quelltext zuweisen, können wir ändern, was angezeigt wird:

```
function zeigeKeineKommunikation() {
    var body = document.body;
```

```
    body.innerHTML = "<h1>Keine Verbindung zur Lernplattform</h1>"
        + "<p>Die Verbindungsaufnahme zur Lernplattform ist fehlgeschlagen. "
        + "Die Schnittstelle konnte nicht gefunden werden.</p>";
}
```

Vorname auslesen

Das Element `cmi.core.student_name` speichert den Namen der Person, die das Lernpaket gerade bearbeitet. Meistens zuerst den Nachnamen, dann ein Komma, dann der Vorname. Also beispielsweise `"Mustermann, Max"`. Folgende Funktion sollte also den Vornamen ermitteln:

```
function ermittleVorname() {
    var rohwert = API.LMSGetValue("cmi.core.student_name");
    // Zunächst Leerzeichen zu Beginn und am Ende wegschneiden
    var ohneLeerzeichen = rohwert.trim();
    // Falls kein Name vorhanden ist „unbekannte Person" als Ersatzvorname verwenden
    if (ohneLeerzeichen.length == 0) {
        return "unbekannte Person";
    }
    // Nach dem ersten Komma suchen
    var indexKomma = ohneLeerzeichen.indexOf(",");
    // Kein Komma? Dann ganzen Namen als Vorname nutzen
    if (indexKomma == -1) {
        return ohneLeerzeichen;
    }
    // Nur alles nach dem Komma verwenden
    var nachKomma = ohneLeerzeichen.substring(indexKomma + 1);
    // Leerzeichen am Beginn weg
    var nachKommaOhneLeerzeichen = nachKomma.trim();
    // Falls nichts nach dem Komma steht: Ganzen Namen als Vorname nutzen
    if (nachKommaOhneLeerzeichen.length == 0) {
        return ohneLeerzeichen;
    }
    // Alles nach dem Komma sollte der Vorname sein
    return nachKommaOhneLeerzeichen;
}
```

Begrüßung anzeigen

Den Rückgabewert von `ermittleVorname` brauchen wir für die Methode `zeigeInhalt`:

```
function zeigeInhalt() {
    var vorname = ermittleVorname();
    var gruss = "Hallo " + vorname + "!";
    var ueberschrift = "Herzlich willkommen!";
    var info = "Schön, dass du dieses Lernpaket geöffnet hast.";
    var html = "<h1>" + ueberschrift + "</h1><p>" + gruss + " " + info + "</p>";

    var body = document.body;
```

```
    body.innerHTML = html;
}
```

Zwischenstand

Das bisher Geschaffte können wir in einer Lernplattform testen. Ist alles richtig, sollten wir namentlich begrüßt werden. Ideal wäre es übrigens, auch die Versionsnummer in `imsmanifest.xml` abzuändern.

Abhängig von der Lernplattform kann es vorkommen, dass das Lernpaket so schon beim erstmaligen Öffnen als erfolgreich abgeschlossen angesehen wird. Dieses Verhalten wäre korrekt. Denn ein Lernpaket soll als `"completed"` angenommen werden, wenn über die Schnittstelle kommuniziert wird, aber kein Status festgesetzt wird. Den Status festzusetzen ist eine der Aufgaben, die noch vor uns liegen. Und auch wenn einige Lernplattformen unter diesen Umständen keinen Abschluss annehmen (sondern beispielsweise erst, wenn auch irgendein Wert verändert wird): Auf der sicheren Seite sind wir erst, wenn wir die Festlegung des Status selbst in die Hand nehmen.

Version 3: Bearbeitungsfortschritt speichern und wiederherstellen

In diesem Kapitel bauen wir das bisherige Lernpaket so aus, dass alle anfangs aufgelisteten Zielsetzungen erfüllt sind. Hierzu muss wieder der Inhalt der Datei `lernpaketfunktionen.js` erweitert werden.

Auslesen, der wie vielte Lernpaketaufruf gerade stattfindet

Wir müssen insbesondere speichern, wie oft das Lernpaket bereits gestartet wurde. Weil es sich bei dieser Information um eine Angabe handelt, die sich als Standort der Bearbeitung auffassen lässt, verwenden wir das Element `cmi.core.lesson_location`. Sein Wert ist Anfangs `""`, dieser Wert kann fast beliebig von uns festgelegt werden (genau genommen maximal 255 ASCII-Zeichen, also völlig ausreichend zum Speichern einer in den meisten Fällen nur einstelligen Zahl).

Beginnen wir damit, eine Funktion namens `ermittleVorherigeAufrufe` zu programmieren, die den zuvor gespeicherten Wert ausliest und in eine Zahl umwandelt:

```
function ermittleVorherigeAufrufe() {
    // Gespeicherte Zeichenkette von Lernplattform erfragen
    var gespeichert = API.LMSGetValue("cmi.core.lesson_location");
    // In Zahl umwandeln
    var anzahl = +gespeichert;
    // Falls das Gespeicherte keine Zahl ist, gehen wir von 0 aus
    if (!Number.isFinite(anzahl)) {
        return 0;
    }
    // Ermittelte Zahl zurückgeben
```

```
    return anzahl;
}
```

Mithilfe dieser Funktion können wir bestimmen, der wievielte Aufruf des Lernpakets gerade stattfindet. Diese Information ist für uns so entscheidend, dass wir sie zentral abspeichern:

```
var wievielterAufruf = 0;
```

Eine Funktion `ladeStatus` soll unter anderem dieser Variable den richtigen Wert zuweisen (und noch mehr, was wir später ergänzen werden):

```
function ladeStatus() {
    wievielterAufruf = ermittleVorherigeAufrufe() + 1;
}
```

Im nächsten Abschnitt werden wir dafür sorgen, dass die Funktion `ladeStatus` tatsächlich aufgerufen wird.

Speichern des neuen Status

Umgekehrt soll eine Funktion `speichereStatus` den neuen Status speichern, damit er beim nächsten Start wieder ausgelesen werden kann:

```
function speichereStatus() {
    // Anzahl der bisherigen Aufrufe in Zeichenkette umwandeln
    var zuSpeichern = "" + wievielterAufruf;
    // Tatsächlich speichern
    API.LMSSetValue("cmi.core.lesson_location", zuSpeichern);
    // Lernplattform mitteilen, dass Änderungen vorerst abgeschlossen sind
    API.LMSCommit("");
}
```

Die Funktionen `ladeStatus` und `speichereStatus` sollten in dieser Reihenfolge sofort ausgeführt werden, sobald die Kommunikation mit der Lernplattform gestartet ist. Um dies zu erreichen, ändern wir einfach die Funktion `initKommunikation` entsprechend ab:

```
function initKommunikation() {
    API = findeSchnittstelle();
    if (API == null) {
        zeigeKeineKommunikation();
    } else {
        API.LMSInitialize("");
        ladeStatus();
        speichereStatus();
        zeigeInhalt();
    }
}
```

Status festlegen

Welchen Status das Lernpaket hat (bzw. eigentlich nur der eine „Inhalt“), ist im Element `cmi.core.lesson_status` gespeichert. Nur folgende Werte sind

erlaubt:

not attempted Es wurde noch nicht mit dem bearbeiten des Inhalts begonnen.
browsed Der Inhalt wurde bisher nur vorab betrachtet.
incomplete Mit der Bearbeitung des Inhalts wurde begonnen, die Bearbeitung ist jedoch noch nicht abgeschlossen.
failed Der Inhalt wurde erfolglos bearbeitet.
passed Die Bearbeitung des Inhalts wurde erfolgreich abgeschlossen.
completed Die Bearbeitung des Inhalts ist abgeschlossen. Über den Erfolg der Bearbeitung kann oder braucht keine Aussage getroffen werden.

Welcher Status vorliegen soll, kann anhand des Wertes von `wievielterAufruf` gefolgert werden. Und indem wir den Wert von `cmi.core.lesson_status` explizit festlegen, verhindern wir auch eine automatische und womöglich für uns unpassende Festlegung durch die Lernplattform. Wir ergänzen darum `speichereStatus` so:

```
function speichereStatus() {
    // Anzahl der bisherigen Aufrufe in Zeichenkette umwandeln
    var zuSpeichern = "" + wievielterAufruf;
    // Tatsächlich speichern
    API.LMSSetValue("cmi.core.lesson_location", zuSpeichern);
    // Status speichern
    if (wievielterAufruf < 3) {
        API.LMSSetValue("cmi.core.lesson_status", "incomplete");
    } else {
        API.LMSSetValue("cmi.core.lesson_status", "completed");
    }
    // Lernplattform mitteilen, dass Änderungen vorerst abgeschlossen sind
    API.LMSCommit("");
}
```

Bearbeitungsfortschritt anzeigen

Damit auch die das Lernpaket benutzende Person den „Bearbeitungsfortschritt“ einsehen und nachvollziehen kann, ändern wir `zeigeInhalt` so ab:

```
function zeigeInhalt() {
    var vorname = ermittleVorname();
    var gruss = "Hallo " + vorname + "!";
    var ueberschrift = "Willkommen zurück!";
    var info = "Schön, dass du dieses Lernpaket erneut geöffnet hast. "
        + "Erledigt hast du es ja längst.";

    if (wievielterAufruf == 1) {
        ueberschrift = "Herzlich willkommen!";
        info = "Öffne dieses Lernpaket noch zweimal, um es abzuschließen.";
    }
    if (wievielterAufruf == 2) {
        info = "Öffne dieses Lernpaket erneut, um es abzuschließen.";
    }
    if (wievielterAufruf == 3) {
```

```
        info = "Nun hast du dieses Lernpaket erfolgreich abgeschlossen.";
    }

    var html = "<h1>" + ueberschrift + "</h1><p>" + gruss + " " + info + "</p>";
    var body = document.body;
    body.innerHTML = html;
}
```

Möglichst keine neue Bearbeitung starten

Einige Lernplattformen bieten das Starten einer komplett neuen Bearbeitung an, wenn der aktuelle Versuch abgeschlossen ist. Um das möglicherweise zu unterbinden, können wir noch dem Element `cmi.core.exit` den Wert `"suspend"` zuweisen. Damit deuten wir an, dass beabsichtigt ist, dass die Bearbeitung an der aktuellen Stelle weitergeht. Es gibt keinen Grund, warum diese Festlegung erst am Ende erfolgen sollte. Also ändern wir `speichereStatus` erneut ab:

```
function speichereStatus() {
    // Anzahl der bisherigen Aufrufe in Zeichenkette umwandeln
    var zuSpeichern = "" + wievielterAufruf;
    // Möglichst Weiterarbeiten statt neuer Versuch
    API.LMSSetValue("cmi.core.exit", "suspend");
    // Tatsächlich speichern
    API.LMSSetValue("cmi.core.lesson_location", zuSpeichern);
    // Status speichern
    if (wievielterAufruf < 3) {
        API.LMSSetValue("cmi.core.lesson_status", "incomplete");
    } else {
        API.LMSSetValue("cmi.core.lesson_status", "completed");
    }
    // Lernplattform mitteilen, dass Änderungen vorerst abgeschlossen sind
    API.LMSCommit("");
}
```

Zusammenfassung

Nun haben wir ein Lernpaket, welches alle gesteckten Ziele erfüllt.