

Das OSG Lecture Bundle

Benutzerhandbuch

Matthias Werner

23. Oktober 2026

Inhaltsverzeichnis

1	Einführung	3
1.1	Idee	3
1.2	Was genau <i>ist</i> denn nun OSG Lecture?	4
1.3	Polymorphie in OSG Lecture	6
2	Kurztutorial	8
3	Konzepte	19
3.1	Projekt	19
3.2	Lehreinheit	19
3.3	Verzeichnisstruktur	20
3.4	Modi	22
3.5	Dokumententypen und Profile	23
4	Inhalte gestalten	26
4.1	Titelei und Metadaten	26
4.2	Präsentationen	27
4.3	Anmerkungen und Quellenangaben	27
4.4	Mehrsprachige Dokumente	28
4.5	Inhalte mit OSG Lecture-Modi auswählen	29
4.6	Aufzählungen als Fließtext	30
4.7	Zweispaltige Inhalte	31
4.8	Querverweise	32
4.9	Literatur	33
4.10	Befehlsreferenz	34
5	Serien	37
5.1	Zähler fortsetzen	37
5.2	Integration	38

Inhaltsverzeichnis

5.3	Rahmendokumentunit	38
5.4	Tags, Links und Fallback	39
5.5	Referenzübersicht	40
6	Standalone	41
7	Projekteinstellungen	43
7.1	Das Projektmanifest	43
7.2	Bundle-Preset	44
7.3	Standardeinstellungen für Nutzer	45
7.4	L ^A T _E X-Konfiguration	46
7.5	Dokumentenprofile	47
7.6	Den Modusgraphen erweitern	48
7.7	Konfigurationsreferenz	49
8	OLLM	52
8.1	Bauen: von einer Instanz bis zur Serie	52
8.2	Prüfen und Diagnostizieren	53
8.3	Aufräumen und Zustände pflegen	53
8.4	Bereitstellen	54
8.5	Projekt anlegen und allgemeine Optionen	55
8.6	Vollständige Kommandoübersicht	55
	Index	58

1 Einführung

1.1 Idee

Lehrmaterialien treten in vielerlei Formen und Versionen auf, beispielsweise als Vorlesungsskript, Folien oder Handouts, und das möglicherweise jeweils in verschiedenen Sprachen. Dabei sind die Inhalte oft gleich oder ähnlich, und die Strukturen – beispielsweise die Reihenfolge der Lehr- und Lerneinheiten sollten konsistent sein. Mit anderen Worten: Die gleiche Sache (hier: semantische Information) soll in unterschiedlicher Gestalt auftreten. In der Informatik oder in der Chemie nennt man dieses Konzept auch *Polymorphie*.¹

Gleichzeitig werden Lehrmaterialien in der Regel permanent überarbeitet. Einzelne Lehreinheiten werden modernisiert, reorganisiert oder gleich ganz entfernt, neue werden ergänzt. Dabei stets die gewünschte Konsistenz aufrechtzuerhalten, ist schwierig. Das *OSG Lecture* Bundle versucht, Lehrende dabei zu unterstützen. Es folgt dabei einer zentralen Idee:²

Leitsatz

Es sollten so **wenig** wie möglich **Redundanzen** auftreten.

Wo Redundanzen notwendig sind (beispielsweise bei verschiedenen Textversionen), ist die Aufrechterhaltung von Konsistenz einfacher, wenn aufeinander bezogenes Quellmaterial **möglichst nahe** beieinander ist.

Daher soll *OSG Lecture* u. a. ermöglichen :

- Generierung verschiedener Dokumentarten aus der gleichen Quelle;
- einfache Reorganisation von Lehreinheiten;
- verschiedene Sprachvarianten aus der gleichen Quelle, wobei gemeinsame Daten nicht redundant dargestellt werden brauchen;
- Unterstützung bei individuellen Erstellen und Überarbeiten von Lehreinheiten mit einfacher Integration (z. B. Skriptkapitel zu einem Gesamtskript);
- schnelle Bereitstellung der verschiedenen Dokumentarten.

1. Vielleicht wären *Polyrepräsentation* oder sogar *Multimodalität* noch passendere Begriffe, aber ersteres ist nicht sehr etabliert und beim zweiten müsste ich vermutlich mit Kulturwissenschaftlern und Semiotikern darüber diskutieren, ob unterschiedliche Sprachen als *Modi* aufzufassen sind. Außerdem bin ich Informatiker, da liegt die Polymorphie nahe.

2. Es ist mir durchaus bewusst, dass dieser Leitgedanke nicht allgemein geteilt wird: ein T_EX-Riese wie Joseph WRIGHT steht dieser Idee z. B. skeptisch gegenüber (<https://github.com/josephwright/ltx-talk/discussions/68#discussioncomment-13755453>), und er kann gute Gründe anbringen. Falls Sie also glauben, dass solche Gründe überwiegen, nutzen Sie *OSG Lecture* nicht.

Dabei wurde von Modalitäten und Arbeitsabläufen ausgegangen, die an der Professur für Betriebssysteme (*Operating Systems Group, OSG*) der TU Chemnitz derzeit üblich sind. Wenn Sie nicht dort tätig sind und völlig andere Abläufe haben, könnte **OSG Lecture** trotzdem für Sie nützlich sein: einzelne Pakete des Bundles lassen sich auch unabhängig von den anderen nutzen.

OSG Lecture ist das Nachfolgeprojekt von **OSG Beamer**,³ das sich aus einem Thema und einer handvoll Ergänzungsmacros für die überaus verbreitete Beamer-Klasse⁴ entwickelt hat. Gegenüber seinem Vorgänger zeichnet sich **OSG Lecture** durch einen größeren Funktionsumfang, ein überarbeitetes und damit hoffentlich besseres Interface, eine umfangreichere Dokumentation und nicht zuletzt dadurch aus, dass es ermöglicht, barrierefreie getaggte Lehrdokumente zu erstellen, indem es Klassen wie **ltx-talk** unterstützt.

1.2 Was genau *ist* denn nun **OSG Lecture**?

OSG Lecture ist ein L^AT_EX-Bundle (genauer gesagt: ein LuaL^AT_EX-Bundle), das verschiedene Ideen und Ansätze zusammenbringt um Lehrmaterialien zu erstellen. Keine dieser Ideen oder Ansätze ist besonders neu.

- **osglecture** ist eine Metaklasse, die die meiste Arbeit an andere Klassen delegiert, so wie es z. B. auch **beamer-switch**⁵ macht. Darüber hinaus stellt **osglecture** aber auch eigene Funktionen bereit, etwa ein dokumentübergreifendes Referenzsystem, gemeinsame Dokumentbefehle und einige Formatierungsumgebungen.
- **langselect** ist ein Paket, das zur (sprachabhängigen) Auswahl von in das Ziel-PDF zu übernehmenden Text, ähnlich wie es auch **comment**⁶ oder **multilang**⁷ macht. Die tatsächliche genutzte Sprache kann auch von außen festgelegt werden.
- **osglecture-modes** generalisiert die Idee von Darstellungsmodi, wie man sie aus **beamer**⁸ oder **ltx-talk**⁹ kennt, macht also etwas ähnliches wie **langselect**, aber nicht auf Sprach- sondern auf Strukturebene. Damit können Modi in der aus **beamer** oder **ltx-talk** gewohnten `<>`-Syntax auch außerhalb von Präsentationen genutzt werden.
- **tagpax** dient dazu, vollständige Seiten getaggtter PDF-Dokumente in ein wiederum getaggttes PDF-Dokument zu übernehmen. Dabei bleiben neben der Taggingstruktur auch Verknüpfungen, Sprungziele und Annotationen erhalten. Im Kontext von **OSG Lecture**

3. <https://github.com/tuc-osg/osgbeamer>

4. <https://ctan.org/pkg/beamer>

5. auf CTAN als **beamer-switch**: <https://ctan.org/tex-archive/macros/latex/contrib/beamer-switch>

6. auf CTAN als **comment**: <https://ctan.org/tex-archive/macros/latex/contrib/comment>

7. auf CTAN als **multilang**: <https://ctan.org/tex-archive/macros/latex/contrib/multilang>

8. auf CTAN als **beamer**: <https://ctan.org/tex-archive/macros/latex/contrib/beamer>

9. auf CTAN als **ltx-talk**: <https://ctan.org/tex-archive/macros/latex/contrib/ltx-talk>

dient es dazu, unabhängige Übersetzungen z. B. einzelner Skriptkapitel zu einem Gesamtskript zu ermöglichen, ohne einen „großen“ Übersetzungslauf¹⁰ durchführen zu müssen.

- **OLLM** ist ein weiteres L^AT_EX-Build-Tool, aber speziell auf **OSG Lecture** zugeschnitten. Es arbeitet eng mit **osglecture** und den anderen genannten Paketen zusammen. Intern ruft **OLLM** **latexmk** auf (es ist aus einem rc-File für **latexmk** entstanden), das die meiste Arbeit erledigt.

Während diese genannten Pakete den Kern des Bundles bilden, gibt es noch einige Pakete, die sich für die Erstellung von Lehrmaterialien als nützlich erwiesen haben. Diese interagieren durchaus mit den Kernpaketen, tragen aber mehr zur direkten Gestaltung der Dokumente bei, als zur Realisierung des Polymorphieansatzes, den sie aber nutzen.

- **osgstyler** versucht die L^AT_EX-Kernidee der Trennung von Inhalt und Gestaltung über eine zentrale semantische Konfiguration zu stärken. Dinge wie Farben, Schriften, Abstände, Hervorhebungen und Blöcke können damit einerseits einfacher konsistent und andererseits leichter austauschbar gestaltet werden. Es kann gut mit **osglecture-modes** zusammenwirken.
- **ltxtalk-theme** stellt ein gemeinsames Gerüst für Themen der Klasse **ltx-talk** bereit. Für Gestaltungsaspekte innerhalb der Folien nutzt es **osgstyler**.
- **lththeme-tuc-2019** ist die (inoffizielle) Umsetzung des offiziellen Präsentationslayout meiner Universität für **ltx-talk**. Es nutzt **ltxtalk-theme**.
- **semcat** markiert Inhalte mit frei festlegbaren semantischen Kategorien, etwa „Hintergrund“, „technische Details“ oder „Wiederholung“. Kategorien können im Fließtext, als eigener Absatz oder als hervorgehobener Block erscheinen und durch Farben, Symbole oder QR-Codes kenntlich gemacht werden.
- **ansiterm** stellt zwei semiverbatim Umgebungen für die Darstellung von Computerterminalfenstern zur Verfügung, die ANSI-Escapesequenz¹¹ interpretieren können. Terminalmitschnitte können vorgegeben oder zur Übersetzungszeit in einer Shell erzeugt und für weitere Übersetzungen zwischengespeichert werden. Auch in getaggtten Dokumenten bleibt der dargestellte Text zugänglich.
- **osglisings** setzt Quellcode auf der Grundlage von **minted**.¹² Es unterstützt verschiedene Darstellungen, externe Quellen sowie inhaltliche Markierungen und Erläuterungen im Code; die Ausgabe kann auch in getaggtten Dokumenten verwendet werden.
- **tagbridge** ergänzt Tagging-Unterstützung für verwendete Fremdpakete, solange diese selbst noch keine ausreichende PDF/UA-Unterstützung besitzen. Derzeit betrifft dies insbesondere QR-Codes und TikZ-Grafiken.

10. Bei meinem Lehrskripten kann so ein großer Lauf durchaus auch mal über eine Stunde dauern.

11. Siehe z. B. <https://de.wikipedia.org/wiki/ANSI-Escapesequenz>

12. auf CTAN als **minted**: <https://ctan.org/tex-archive/macros/latex/contrib/minted>

- `tagdirtree` setzt Verzeichnisbäume als echte verschachtelte Listen statt als Grafik. Dateien, Verzeichnisse und Archive bleiben dadurch lesbarer Text, und ihre Hierarchie steht auch unterstützender Technik zur Verfügung.

1.3 Polymorphie in OSG Lecture

Es ist eine alte T_EX-Idee, Texte je nach Kontext unterschiedlich darzustellen. Die T_EX-Variante ConT_EXt leitet beispielsweise ihren Namen auch davon ab. OSG Lecture versucht die Anwendung dieses Konzept noch ein Stück weit zu erweitern. Dazu ist `osglecture` eine *Metaklasse*: Sie sagt (anders als die meisten L^AT_EX-Klassen) nicht, wie die fertig gesetzten Dokumente aussehen sollen,¹³ sondern stellt ein gemeinsames Interface zur Verfügung und verteilt dann die Arbeit an Klassen wie `article`, `scrarticle`, `beamer` oder `ltx-talk`.

Das Folgende ist ein Mini-Showcase der zeigen soll, wie aus einer Quelle verschiedene Instanzen erzeugt werden. Es dient lediglich dazu, dass der Nutzer einen kleinen Eindruck von den Möglichkeiten des kontextabhängigen Polymorphismus in OSG Lecture gewinnen kann – die Diskussion über Funktionsweise und die verwendeten Mechanismen erfolgt später.

```

1 \begin{frame}
2   \frametitle{\lende{One Source, Many Presentations}{Eine Quelle, viele Darstellungen}}
3   \begin{presitemize}
4     \item \lende{a unit contains the subject-matter content}%
5       {eine Lehreinheit enthält den fachlichen Inhalt}
6     %
7     \item \lende*{target @1 produces a presentation}%
8       {das Target @1 erzeugt eine Präsentation}{\emph{talk}}%
9     %
10    \anditem \lende*{target @1 produces a long form}%
11      {das Target @1 erzeugt eine Langform}{\emph{script}}%
12    %
13    \thereforeitem \lende{we \finite{need} to maintain changes only once}%
14      {wir \finite{müssen} Änderungen nur an einer Stelle pflegen}%
15    %
16    \item \lende{language-independent data needs to be entered at one place}%
17      {sprachunabhängige Daten müssen nur einmal angegeben werden}%
18    %
19    \only<article>{\lende{since they can be inserted into the text using @-parameters}%
20      {da sie über @-Parameter in den Text eingefügt werden können}.%
21    }%
22  \end{presitemize}%
23 \end{frame}
24 \lende*{Text written outside the @1 environment is reserved exclusively for the long form.}%
25 {Außerhalb der @1-Umgebung geschriebener Text ist exklusiv der Langform vorbehalten.}{\texttt{frame}}
```

Abbildung 1: Gemeinsamer Quellcode

13. Es gibt aber Pakete im OSG Lecture-Bundle, die so etwas machen

1 Einführung

One Source, Many Presentations

- A unit contains the subject-matter content
- Target *talk* produces a presentation
- Target *script* produces a long form
- We need to maintain changes only once
- Language-independent data needs to be entered at one place

Eine Quelle, viele Darstellungen

- Eine Lehreinheit enthält den fachlichen Inhalt
- Das Target *talk* erzeugt eine Präsentation
- Das Target *script* erzeugt eine Langform
- Wir müssen Änderungen nur an einer Stelle pflegen
- Sprachunabhängige Daten müssen nur einmal angegeben werden

(a) Folien auf English

(b) Folien auf Deutsch

1

Eine Lehreinheit enthält den fachlichen Inhalt. Das Target *talk* erzeugt eine Präsentation und das Target *script* erzeugt eine Langform, weshalb wir Änderungen nur an einer Stelle pflegen müssen. Sprachunabhängige Daten müssen nur einmal angegeben werden, da sie über @-Parameter in den Text eingefügt werden können. Außerhalb der **frame**-Umgebung geschriebener Text ist exklusiv der Langform vorbehalten.

(c) Ausgabe für script-Taget

Abbildung 2: Alle Varianten sind aus der Quelle in Abb. 1 erstellt: Bei den Folien in unterschiedlichen Sprachen (a) und (b) bleibt Gliederung, Aussagenfolge und Hervorhebung jeweils gleich, die Ausgabe für das Skript (c) ist inhaltlich äquivalent, jedoch ausführlicher und in Satzform

Dass auch gewöhnlich Textauszeichnungen unterschiedlich gestaltet und damit polymorph sein können, ist in L^AT_EX ein alter Hut. **OSG Lecture** unterstützt dies aber komfortabel durch spezifische Profile.

- 1 Ein `\emph{Seitenfehler}` unterbricht den normalen Kontrollfluss.
- 2 Das Betriebssystem muss anschließend `\textbf{den fehlenden Seitenrahmen}`
- 3 bereitstellen}.

Abbildung 3: Auch gewöhnliche Autorenauszeichnungen können polymorph sein

Ein *Seitenfehler* unterbricht den normalen Kontrollfluss. Das Betriebssystem muss anschließend **den fehlenden Seitenrahmen bereitstellen**.

Ein *Seitenfehler* unterbricht den normalen Kontrollfluss. Das Betriebssystem muss anschließend **den fehlenden Seitenrahmen bereitstellen**.

(a) Bildschirmprofil

(b) Druckprofil

Abbildung 4: Umsetzung des Codes aus Abbildung 3: Gleiche semantische Auszeichnung, unterschiedliche Profile.

Warum man solche Polymorphie überhaupt *braucht*, wird im folgenden Kurztutorial demonstriert.¹⁴

¹⁴ Die Idee, ein Tutorial als Geschichte zu schreiben, ist durch Till TANTAU und sein Tutorial für **Beamer** und **TikZ** inspiriert.

2 Kurztutorial

Dieses Tutorial begleitet Nora. Nora hält im kommenden Semester erstmals die Vorlesung „Betriebssysteme“. Die erste Unterrichtseinheit soll über Speicher gehen. Nora will sofort Speichervirtualisierung behandeln. Die ist vielleicht etwas optimistisch, aber das soll uns hier nicht aufhalten.

Der Studiengang, in dem der Kurs stattfindet, ist zwar deutschsprachig, aber Nora weiß, dass viele Gaststudenten teilnehmen werden, die besser Englisch als Deutsch beherrschen. Daher will sie Folien jeweils in beiden Sprachen vorbereiten. Außerdem hat sie versprochen, ein kurzes Skript bereitzustellen. Für die Lehrveranstaltung selbst möchte sie außerdem eine Vortragendenfassung mit der aktuellen und der nächsten Folie sowie eigenen Anmerkungen verwenden.

Mehrere Dokumentinstanzen für zunächst eine einzige Stunde Lehre: Nora ahnt bereits, dass getrennte Quelldateien keine besonders gute Idee wären.

Das Tutorial beschreibt nicht jede Möglichkeit von [OSG Lecture](#). Es zeigt vielmehr die Dinge, die Nora für ihren ersten kleinen Kurs tatsächlich benötigt.¹⁵ Am Ende entstehen aus einem einzelnen Quelldokument einer Lehreinheit Folien, eine Vortragendenfassung und ein Skript, jeweils in zwei Sprachen.

Noras Problem. Nora beginnt so, wie man vermutlich intuitiv ohne [OSG Lecture](#) beginnen würde. Sie legt `slides-de.tex` an, füllt es mit Inhalt, kopiert die Datei nach `slides-en.tex` und beginnt mit der Übersetzung. Für das Skript fügt sie später noch `script-de.tex` hinzu. Das funktioniert. Allerdings fällt kurze Zeit später ein, dass sie bei der Beschreibung des Unterschieds zwischen *Paging* und *Segmentierung* etwas Wichtiges vergessen hat. Jedoch klingelt bei der Bearbeitung ihr Telefon, und nach dem Anruf muss Nora überlegen, in welche der Dateien sie die Korrektur übernommen hat.

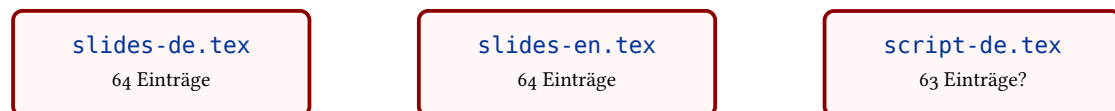


Abbildung 5: Kopien sind bequem anzulegen und unbequem zu pflegen.

Nora entscheidet deshalb, [OSG Lecture](#) zu nutzen und nur noch eine fachliche Quelle zu pflegen. Die Sprachen sollen darin nahe beieinanderstehen; Folien und Skript sollen ebenfalls aus dieser Quelle entstehen. Genau dies wird im Folgenden eingerichtet.

Ein Zuhause für den Kurs. Zunächst braucht der Kurs ein Verzeichnis. Nora könnte alle Dateien dort ablegen. Sie entscheidet sich aber gleich für zwei Unterverzeichnisse: eines für gemeinsame Angaben und eines für die erste Lehreinheit. Dass die Einheit die Nummer 010 statt 001 erhält, erscheint ihr zunächst unnötig großzügig. Später kann sie dafür eine neue Einheit

¹⁵ Wer lieber sofort in die Details gehen will, kann auch direkt im Abschnitt 3 weiterlesen.

005-Overview dazwischenschieben, ohne alles umzubenennen. In das Verzeichnis für die erste Lehreinheit kommt die Quelldatei. Nora legt erst mal eine leere Datei mit dem Namen `main.tex` an, da dies bei *OSG Lecture* der Standardname für die L^AT_EX-Hauptdatei einer Lehreinheit ist.

```
mkdir -p Noras-os/Include Noras-os/010-Memory
cd Noras-os
touch 010-Memory/main.tex
```

Noch ist das Ergebnis nicht besonders eindrucksvoll:



Immerhin ist nun klar, was zum gesamten Kurs und was nur zur Einheit „Virtueller Speicher“ gehört.

Nora bestellt Ergebnisse Nora hat nun Verzeichnisse, aber noch kein Weg, das aus ihrer einen Quelle die gewünschten Dokumente erzeugt. Diese Aufgabe übernimmt *OLLM*, der *OSG LaTeX Lecture Maker*. *OLLM* ist das Buildwerkzeug des Bundles: Es wählt Dokumenttyp und Sprache, startet den passenden L^AT_EX-Lauf und verwaltet die entstandenen Dateien. Nora wird *OLLM* deshalb nicht den Inhalt ihrer Vorlesung erklären, wohl aber, welche Ergebnisse sie erwartet.

Eine konkrete Wahl aus Lehreinheit, Target und Sprache heißt im Folgenden *Instanziierung*. Diesen Begriff kennt Nora schon aus der Objektorientierten Programmierung. Aber anders als dort ist das Ergebnis einer Instanziierung kein Objekt sondern eine *Dokumentinstanz*. Die Kombination „Unterrichtseinheit memory, Target slides, Sprache en“ erzeugt beispielsweise die englische Folieninstanz dieser Einheit. Damit ist präzise benannt, was *OLLM* später bauen soll.

Nora möchte *OLLM* am liebsten sofort mit `ollm slides` starten. Das Werkzeug kann jedoch noch nicht wissen, welche Sprachen der Kurs anbietet oder ob Nora z. B. überhaupt ein Skript haben möchte. Diese Angaben kommen in ein Manifest namens `ollmconfig.toml` in der Projektwurzel. Man kann es als Bestellzettel lesen: Nora bestellt zwei Sprachfassungen der Folien und zwei des Skripts.

```
schema = 2
bundle_preset = "OSG lecture/1"
```

```
[project]
id = "os-in-60"
```

```
[targets.defaults]
languages = ["de", "en"]
default_language = "de"
presentation_profile = "ltx-talk"
longform_profile = "book"
document_metadata = "enabled"

[targets.slides]
[targets.screen]
[targets.script]
```

Nora ist versucht, hier auch Titel, Autorin und Übersetzungen einzutragen. TOML kann schließlich Text speichern. Sie lässt es dennoch bleiben: Das Manifest beschreibt die zu erzeugenden Dokumentinstanzen, nicht den Inhalt des Dokuments. Titel und Sprachabbildung sind L^AT_EX-Angelegenheiten und folgen im nächsten Schritt.

Bevor sie weiterschreibt, prüft Nora den Bestellzettel:

```
ollm check
```

Bei ihrem ersten Versuch hatte sie `language` statt `languages` geschrieben. OLLM weist den unbekannten Schlüssel zurück. Das ist Nora lieber als stillschweigend Kursmaterial in der falschen Sprache zu erhalten. Nachdem sie den Tippfehler korrigiert hat, ist das Manifest gültig. Auch falls ihre Buildumgebung selbst Probleme bereitet, schließlich erwartet OSG Lecture eine relativ moderne L^AT_EX-Installation, kann OLLM helfen: `ollm doctor` liefert die passendere Diagnose.

Was für alle Lehreinheiten gilt. Nora möchte ihren Namen und den Kurstitel nicht in jeder späteren Einheit wiederholen. Sie legt deshalb `Include/projectconfig.tex` an. Im Manifest hat sie für Präsentationen bereits `ltx-talk` und für das Skript die Standardklasse `book` gewählt. Beide Profile arbeiten mit der neuen PDF-Verwaltung von L^AT_EX; damit kann Nora ihre Dokumente als Tagged PDF bereitstellen. Das passende Präsentationsthema kann sie unabhängig davon auswählen.

Die Wahl der Klassen ist eine Projektentscheidung; die einzelnen Fachtexte in den Lehreinheiten müssen von diesen Klassennamen nichts wissen. Außerdem lernt Nora, dass Dokumente, die nicht zu den Präsentationen oder Handouts gehören, in OSG Lecture „Langformen“ genannt werden. Ihr Skript ist eine solche Langform.

Sie schreibt in `/Include/projectconfig.tex`

```
\title{\ldeen{Betriebssysteme}{Operating Systems}}
\author{Nora Example}
```

```
\LectureProjectSetup{
  languages={
    map={de=ngerman,en=british},
    load babel
  }
}
```

Die Profilwahl steht im Manifest und nicht in `projectconfig.tex`, weil die Basisklasse und ihr Bedarf an `\DocumentMetadata` feststehen müssen, bevor die Klasse überhaupt startet. Für die frühe PDF-Konfiguration legt Nora deshalb zusätzlich `Include/documentmetadata.tex` an:

```
\DocumentMetadata{
  lang = \OsgLectureRequestedLanguage,
  tagging = on
}
```

`OLLM` bindet diese Datei bei allen drei Targets vor `\documentclass` ein. Die sichtbaren Dokumentangaben bleiben dagegen in `projectconfig.tex`.

Die verbleibende Sprachkonfiguration enthält zwei verschiedene Arten von Namen. `OLLM` arbeitet mit den kurzen, portablen Kennungen `de` und `en`, wie sie in der Norm ISO 639-1 definiert sind. Das verbreitete Sprachpaket `babel` benötigt dagegen die konkreteren Namen `ngerman` und `british`. Der Schlüssel `languages` reicht dieses Mapping an `langselect` weiter und lädt `babel`. Welche Sprachen das Projekt überhaupt anbietet, steht bereits im Manifest – `OSG Lecture` reicht die Liste von dort an `langselect`.

Endlich etwas Fachliches. Nun schreibt Nora `010-Memory/main.tex`. Sie beginnt bewusst mit wenig Inhalt. Ein kleines, erfolgreich gebautes Dokument ist eine angenehmere Ausgangsbasis als zwanzig Seiten, bei denen eine schließende Klammer fehlt.

```
\documentclass{osglecture}

\begin{document}
\lecture[Memory]{\ldeen{Virtueller Speicher}{Virtual Memory}}{lec:memory}

\begin{frame}
  \frametitle{\ldeen{Warum virtueller Speicher?}{Why virtual memory?}}
  \ldeen{Jeder Prozess sieht einen eigenen Adressraum.}
  {Each process sees a private address space.}
  \Note{\ldeen{Hier an die Analogie mit Zimmernummern erinnern.}
    {Recall the analogy with room numbers here.}}
\end{frame}
```

```

\begin{frame}
  \frametitle{\ldeeen{Seitentabellen}{Page tables}}
  \ldeeen{Eine Seitentabelle ordnet virtuelle Seiten physischen
    Seitenrahmen zu.}
  {A page table maps virtual pages to physical page frames.}
\end{frame}
\end{document}

```

Der Befehl `\lecture` dient Nora dazu, eine Unterrichtseinheit zu beginnen. Das erste Pflichtargument ist der Titel, das optionale Argument seine Kurzform. Das zweite Pflichtargument von `\lecture`, hier `lec:memory`, ist das Label der Unterrichtseinheit. Dieses etabliert die Identität der Einheit. Nora darf den sichtbaren Titel später verbessern und sogar das Verzeichnis umbenennen. Die Identität sollte sie aber dabei nicht beiläufig ändern; Querverweise und Buildzustand verwenden diesen Wert, um die richtige Lehreinheit zu finden.

Die `frame`-Umgebung kennt Nora schon aus ihrer früheren Arbeit mit `beamer`. Sie hat dort zwar häufig den Titel in ein Umgebungsargument geschrieben, aber das darf sie bei `OSG Lecture` nicht; dafür gibt es den Befehl `\frametitle`. Den Befehl `\ldeeen` kennt Nora noch nicht. Sie findet auch bei einer kurzen Suche mit `grep` in ihrem L^AT_EX-Verzeichnis kein Paket, das ihn definiert. Dennoch ist Nora intuitiv klar, was er macht: Er stellt die beiden Sprachvarianten, die Nora bestellt hat, nebeneinander dar.

`\Note` enthält dagegen eine Anmerkung für Nora selbst. Sie gehört in den Frame, auf den sie sich bezieht. Auf den gewöhnlichen Folien und im Skript erscheint sie nicht; die Vortragendenfassung sammelt sie später auf ihrer Presenter-Seite.

Der erste Build. Nora wechselt in die Lehreinheit und versucht aus alter Gewohnheit zuerst `lualatex main.tex`. Das ist zwar ein verständlicher Versuch, umgeht aber den Bestellzettel: Target und Sprache fehlen. Für einen Serienkurs ist `OLLM` der richtige Eingang. Nora baut also zunächst genau das Ergebnis, das sie gerade sehen möchte:

```

cd 010-Memory
ollm build slides --language=de

```

Das Ergebnis ist eine deutsche Folie. Sie liegt nicht neben `main.tex`; `OLLM` hält erzeugte Dateien unter `.osglecture/build` von den Quellen getrennt. Nora sucht kurz im falschen Verzeichnis, findet die Trennung dann aber überzeugend: Ein beschädigter Buildordner kann entfernt werden, ohne dass dabei Lehrmaterial verschwindet.

Die englische Folie ist jetzt fast enttäuschend einfach:

```

ollm build slides --language=en

```

Nur die gewählte Formulierung ändert sich. Struktur und Formel bleiben dieselben. Genau das wollte Nora erreichen.

Eine Fassung für Nora. Für die Lehrveranstaltung baut Nora außerdem das Target screen:

```
ollm build screen --language=de
```

Diese Dokumentinstanz verwendet dieselben Folien, macht jede Seite aber doppelt so breit. Links liegt die Folie für den Projektor; rechts sieht Nora die aktuelle und die nächste Folie sowie ihre mit \Note notierte Erinnerung. Bei einem erweiterten Desktop kann sie die beiden Hälften auf Projektor und eigenen Bildschirm verteilen. Die Vortragendenfassung bleibt damit Teil derselben Quelle, ohne selbst eine Fassung für die Studierenden zu sein.

Auch diese Instanz wird mit eingeschaltetem Tagging gebaut. Die zusätzliche Presenter-Seite ist jedoch eine Bedienhilfe für Nora und gehört nicht zur Lesereihenfolge der eigentlichen Folie.



(c) Die Vortragendenfassung ergänzt rechts Vorschau und Anmerkung.

Abbildung 6: Drei Dokumentinstanzen aus derselben Quelle.

Abbildung 6 macht den Unterschied der drei Targets sichtbar. Die Anmerkung gehört ausschließlich in Noras Vortragendenfassung; das Skript übernimmt dagegen beide Frames als fortlaufenden Text.

Von Stichpunkten zu einem Satz Nora ergänzt zwei weitere Aussagen. Eine gewöhnliche `itemize`-Liste wäre auf der Folie passend. Im Skript möchte sie jedoch keinen Absatz, der nur aus drei Stichpunkten besteht. Sie könnte dafür zwei Fassungen schreiben, aber das würde ihr ursprüngliches Problem in kleinerem Maßstab wiederholen. Stattdessen verwendet sie die `presitemize`-Umgebung:

```
\begin{frame}
  \frametitle{\ldeeen{Warum virtueller Speicher?}{Why virtual memory?}}
  \begin{presitemize}
    \item \ldeeen{Jeder Prozess sieht einen eigenen Adressraum}
      {Each process sees a private address space}
    \anditem \ldeeen{die Hardware übersetzt virtuelle Adressen}
      {the hardware translates virtual addresses}
    \thereforeitem \ldeeen{Prozesse \finite{sind} voneinander isoliert}
      {processes \finite{are} isolated}
  \end{presitemize}
\end{frame}
```

Auf der Folie bleiben dies drei Einträge. In der Langform werden daraus verbundene Aussagen. Das markierte `finite` Verb hilft dabei, im deutschen Nebensatz aus „Prozesse sind voneinander isoliert“ die Formulierung „weshalb Prozesse voneinander isoliert sind“ zu bilden. Nora findet die Markierung etwas ungewohnt, aber immer noch angenehmer als einen zweiten Absatz, der später vergessen werden kann. Und sie ist froh, dass English in dieser Beziehung ein bisschen regelmäßiger ist und die Verben nicht so herumschubst.

Etwas mehr Platz im Skript. Für die Folie genügt Nora die knappe Begründung. Im Skript möchte sie zusätzlich erklären, wo die Abbildung gespeichert wird. Diesmal unterscheidet sich nicht bloß die Formulierung, sondern der gewünschte Inhalt. Nora ergänzt deshalb nach dem Frame einen sogenannten *Modusabschnitt*:

```
\lecturemode<longform>{
  \ldeen*{Eine Seitentabelle speichert die Abbildung @1.}
    {A page table stores the mapping $v\mapsto @1.}{$v\mapsto p$}
}
```

Sie hätte auch `<script>` schreiben können. Jedoch sagt `<longform>` genauer, was sie meint: Der Absatz passt in jede ausführliche Langform, nicht nur in den heute konfigurierten Skripttyp. Falls sie später von `scrbook` zu einem anderen Langformprofil wechselt, muss sie diese Stelle nicht wiederfinden.

Noch längere Erläuterungstexte kann Nora außerhalb der `frame`-Umgebung schreiben. Dann braucht sie für die Langform auch keinen extra Modus: Die dortigen Texte werden bei den Präsentationen ignoriert.

Nun baut Nora das deutsche Skript:

```
ollm build script --language=de
```

Beim Vergleich sieht sie erstmals die beiden Dimensionen unabhängig voneinander: `--language` wählt die Sprache, das Target `slides`, `screen` oder `script` die Dokumentgestalt. Beides verändert die Ausgabe, aber keines verlangt eine Kopie der Lehrinheit.

Hat Nora wirklich an alles gedacht? Während des Schreibens erzeugt Nora gewöhnlich nur die Dokumentinstanz, an der sie gerade arbeitet. Das ist schnell, birgt aber eine kleine Gefahr: Eine fehlende Klammer kann ausschließlich im englischen Zweig stecken; ein nur in Langformen aktiver Befehl kann auf den Folien unbemerkt bleiben. Vor der Veröffentlichung lässt Nora daher alle für diese Einheit bestellten Kombinationen bauen:

```
ollm build --all
```

Vom Build zur Veröffentlichung. Sechs erfolgreiche Dokumentinstanzen im Buildverzeichnis sind erfreulich, aber Noras Studierende wissen davon noch nichts. Die beiden Vortragendefassungen bleiben bei Nora; Folien und Skripte möchte sie in ein Verzeichnis `release` kopieren, das später etwa von der Lernplattform oder einem Webserver übernommen werden kann. Sie könnte die Dateien von Hand kopieren und umbenennen. Nach dem dritten Termin würde sie dabei vermutlich mindestens einmal die englische Fassung unter dem deutschen Namen ablegen.

OLLM trennt deshalb *Bauen* und *Deployment*. Ein Build erzeugt und prüft eine Dokumentinstanz im internen Buildbereich. Ein Deployment kopiert eine bereits erfolgreich gebaute Instanz unter einem vereinbarten Namen an einen Veröffentlichungsort. Es startet selbst keinen neuen L^AT_EX-Lauf.

Nora ergänzt am Ende von `ollmconfig.toml` zwei Deploymentregeln:

```
[deployment]
series = "units"

[deployment.types.slides]
paths = ["release/slides"]
filename = "{ordinal}-{unit}-{lang}.pdf"

[deployment.types.script]
paths = ["release/notes"]
filename = "{ordinal}-{unit}-{lang}.pdf"
```

Die Platzhalter werden für jede Dokumentinstanz gefüllt. Für die aktuelle Einheit entstehen so beispielsweise `010-memory-de.pdf` und `010-memory-en.pdf`. Nora nimmt die Sprache absichtlich in den Namen auf; ein scheinbar schönerer Name `010-memory.pdf` wäre für zwei Sprachfassungen nicht eindeutig.

OLLM legt Deploymentverzeichnisse nicht selbst an. Das ist zunächst überraschend, verhindert aber, dass ein Tippfehler im Manifest unbemerkt einen zweiten Veröffentlichungsort erzeugt. Nora legt die vereinbarten Verzeichnisse daher einmalig an und deployt anschließend alle gebauten Instanzen der aktuellen Einheit:

```
mkdir -p ../release/slides ../release/notes
ollm deploy --all
```

Nora befindet sich noch in 010-Memory; deshalb bezieht sich das Deployment standardmäßig auf diese Einheit. **OLL**M kopiert nur vorhandene, erfolgreich abgeschlossene Builds. Hat Nora nach einer Textänderung noch nicht neu gebaut, ist `deploy` also kein versteckter Ersatz für `build`. Diese Trennung gefällt ihr: Vor dem Kopieren kann sie die Dokumentinstanzen in Ruhe prüfen.

Beim nächsten Deployment liegt möglicherweise bereits eine ältere Datei unter demselben Namen im Ziel. Ist sie identisch, bleibt sie unberührt. Ist sie verschieden, verlangt die voreingestellte Sicherheitsregel eine ausdrückliche Freigabe mit `--overwrite`. Nora sollte diese Option erst verwenden, nachdem sie sich vergewissert hat, dass sie tatsächlich die veröffentlichte Fassung ersetzen möchte.

Aus einer Einheit wird eine Serie. Für die zweite Unterrichtsstunde legt Nora neben 010-Memory das Verzeichnis 020-Processes an. Dessen `main.tex` folgt dem bereits bekannten Muster. Entscheidend ist nicht die Nummer im Verzeichnisnamen, sondern die neue logische Unit-ID `processes`:

```
\documentclass{osglecture}

\begin{document}
\lecture[Processes]{\ldeen{Prozesse}{Processes}}{processes}
\section{\ldeen{Prozessplanung}{Scheduling}}\label{sec:scheduling}

\begin{frame}
  \frametitle{\ldeen{Wer läuft als Nächstes?}{Who Runs Next?}}
  \ldeen{Der Scheduler wählt einen ausführbaren Prozess aus.}
  {The scheduler selects a runnable process.}
\end{frame}
\end{document}
```

Die beiden Verzeichnisse sind nun nicht bloß zufällige Nachbarn. Das Manifest und ihre Unit-IDs machen sie zu Einheiten derselben *Serie*. Damit kann **OLLM** ihre Dokumentinstanzen gemeinsam auswählen, ihren Zustand verwalten und Abhängigkeiten zwischen ihnen auflösen. Nora baut beispielsweise die deutschen Skriptinstanzen der ganzen Serie aus der Projektwurzel:

```
ollm build script --language=de --scope=series
```

Der Scope `series` bedeutet dabei nicht, dass bereits ein einziges großes PDF entsteht. Zunächst werden zwei selbstständige Skriptinstanzen gebaut. Das ist nützlich, weil Nora einzelne Kapitel weiterhin getrennt bereitstellen kann.

Ein Verweis über die Unit-Grenze. Im Kapitel über Speicher möchte Nora nun auf die spätere Erläuterung der Prozessplanung verweisen. Ein gewöhnliches `\ref` kennt nur Ziele der aktuellen Dokumentinstanz. Für den serieninternen Verweis nennt Nora deshalb zusätzlich die logische Ziel-Unit:

```
\ldeen*{Mehr dazu in @1.}
    {More on this in @1.}
    {\olautoref[processes]{sec:scheduling}}
```

Ohne weitere Angaben wählt `\olautoref` im Ziel dieselbe Sprache und denselben Dokumenttyp wie im Ausgangsdokument. Der deutsche Skriptverweis führt also zum deutschen Skript, der englische Folienverweis zu den englischen Folien. Soll ein Verweis bewusst etwa von einer Folie in das deutsche Skript führen, kann Nora die Adresse als `[processes,type=script,lang=de]` schreiben. Seitenverweise über Dokumentgrenzen vermeidet sie: Die Seitenzahl des anderen PDFs kann sich unabhängig ändern.

Das Ziel muss mindestens einmal erfolgreich gebaut worden sein, damit **OLLM** seine Unit-ID und Referenzziele kennt. Nach späteren Änderungen lässt Nora mit `--resolve` die davon abhängigen Instanzen bis zu einem konsistenten Stand neu bauen:

```
ollm build script --language=de --scope=series --resolve
```

Die Kapitel zu einem Gesamtskript verbinden. Für das Gesamtskript legt Nora eine dritte Unit an. Sie trägt die Rolle `i` für *Integration* im Verzeichnisnamen und enthält keinen weiteren Fachtext, sondern nur das Rahmendokument. Die Reihenfolge der `\includeunit`-Befehle bestimmt die Reihenfolge im Ergebnis:

```
mkdir -p 900-i-collection
```

In `900-i-collection/main.tex` schreibt sie:

```

\documentclass{osglecture}

\begin{document}
\includeunit{memory}
\includeunit{processes}
\end{document}

```

`\includeunit` fügt nicht die TeX-Quellen ein, sondern die bereits gebauten Dokumentinstanzen derselben Sprache und desselben Dokumenttyps. So bleiben die Kapitel einzeln nutzbar, während das Rahmendokument sie zu einem Gesamtskript verbindet. Nora baut zuerst die Kapitel und anschließend die eindeutig passende Integrationsunit:

```

ollm build script --language=de --scope=series --resolve
ollm build script --language=de --scope=collection --resolve
ollm check --scope=series

```

Der abschließende Check baut nichts neu, sondern meldet unter anderem fehlende oder veraltete Referenz- und Integrationszustände. Für ein echtes Projekt würde Nora dieselben Schritte natürlich auch für die englische Skriptinstanz ausführen. Alle Varianten der Referenzbefehle beschreibt Abschnitt 4.10; die Regeln für Integration stehen in Abschnitt 5, und die vollständigen Buildoptionen in Abschnitt 8.6.

Damit ist der kleine Kurs natürlich noch nicht fertig. Nora fehlen etwa weitere Lehreinheiten und vermutlich mehr als zwei Stunden Inhalt. Aber das Grundmuster steht: Das Manifest beschreibt die gewünschten Ergebnisse, die Projektkonfiguration enthält gemeinsame Entscheidungen, und jede Lehreinheit bewahrt den fachlichen Inhalt an einer Stelle. Neue Einheiten kann Nora nun nach demselben Muster hinzufügen, ohne neue Sprach- oder Dokumentkopien anzulegen.

Hinweis

Wenn Sie dem Tutorial mit eigenen Dateien folgen, beginnen Sie ebenfalls mit einer einzigen kleinen Einheit und einer Dokumentinstanz. Erst wenn diese gebaut wird, ergänzen Sie die zweite Sprache, die Langform und schließlich `--all`. Fehler bleiben so dem Schritt zugeordnet, der sie verursacht hat.

3 Konzepte

Nach dem kurzen Tutorial sollen nun die Begriffe und Konzepte von **OSG Lecture** etwas systematischer und ausführlicher besprochen werden.

Dabei ist zu beachten, dass der Name „Osglecture“ in diesem Bundle zwei Bedeutungen hat: Zum einen bezeichnet er das gesamte Bundle, zum anderen dessen Hauptklasse. Zur besseren Unterscheidung wird in diesem Manual für den ersten Fall die Schreibweise **OSG Lecture** verwendet, für den zweiten dagegen **osglecture**.

3.1 Projekt

Als ein „Projekt“ wird hier eine Sammlung zusammengehöriger Materialien verstanden. Im Kontext unseres Anwendungsfalls bedeutet dies Lehrmaterialien wie Vorlesungsfolien oder Skripts, auch wenn **OSG Lecture** auf andere Anwendungsfälle übertragbar ist. Mit anderen Worten: Ein **OSG Lecture**-Projekt ist ein *Kurs*.

Alle Kursmaterialien stehen in einem Verzeichnis, welches in der Regel ein spezielles Manifest enthält. Eine Ausnahme von dieser Regel bilden sogenannte *Standalone*-Projekte, die ursprünglich für einen der oben angesprochenen anderen Anwendungsfälle gedacht waren, nämlich Konferenzbeiträge. Diese Ausnahme wird später gesondert betrachtet. Im Folgenden gehen wir aber von Kursen aus.

3.2 Lehreinheit

Ein Kurs ist aus Sicht von **OSG Lecture** eine Serie von zusammenhängenden Lehreinheiten, z. B. Vorlesungen oder Übungen (oder beidem), die typischerweise im Laufe eines Semesters gehalten werden. Eine Lehreinheit ist hier eher ein didaktisches Konzept als ein zeitliches Konzept: Es ist durchaus möglich, dass sich eine Lehreinheit über mehrere Termine streckt oder dass zu einem Termin mehrere (eher kurze) Einheiten behandelt werden. Welche Lehrmaterialien für einen Kurs benötigt werden, kann recht unterschiedlich sein. In der OSG gibt es einige typische Lehrszenarien:

1. Es gibt Präsentationen, die die Lehrenden benutzen, während die Studierenden dazu passende Handouts erhalten.
2. Wie 1, aber die Handouts gibt es in einer englischen und einer deutschen Sprachfassung.
3. Wie 2, aber statt der Handouts pro Lehrthema gibt es jeweils ein Gesamtskript pro Sprache.

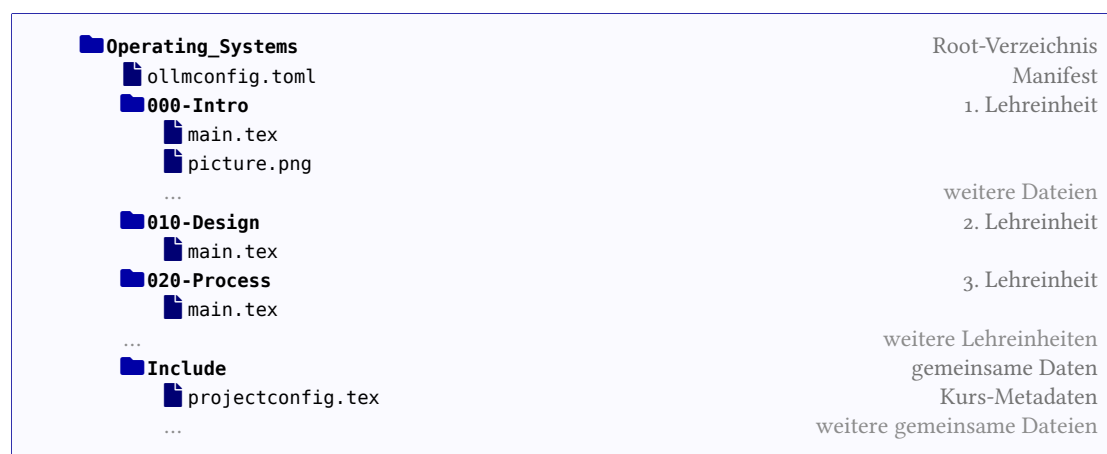
All diese und noch viel mehr Szenarien werden durch **OSG Lecture** unterstützt. Dabei spielt **osglecture** die Rolle einer *Metaklasse*: Sie organisiert im Wesentlichen, welche konkrete Klasse geladen wird (z. B. `article`, `book`, `ltx-talk` oder `beamer`) und sorgt dafür, dass es ein gemeinsames Interface gibt. Dabei werden Materialien, die nicht so unmittelbar präsentationsbezogen sind

wie die eigentliche Präsentation oder Handouts, also beispielsweise Skripte oder Lehrbücher, unter dem Begriff „Langform“ zusammengefasst.

3.3 Verzeichnisstruktur

Die Materialien jedes Kurses befinden sich in einem eigenen Verzeichnis, dem sogenannten (Kurs-)Projektverzeichnis oder (bezogen auf das Projekt) Rootverzeichnis. Dabei bildet die Verzeichnisstruktur die Struktur des Kurses nach.

Im Folgenden ist eine mögliche Verzeichnisstruktur für ein einfaches Szenario zu sehen:



Der Verzeichnisname einer Lehreinheit besteht mindestens aus einer dreistelligen Nummer und einer durch einen Bindestrich abgetrennten Themenbeschreibung. Die Nummer muss dreistellig sein (führende Nullen sind erlaubt) und legt lediglich die Reihenfolge der Lehreinheiten fest, nicht etwa Kapitelnummern. Daher bietet es sich mindestens in der Entwicklungsphase eines Kurses an, eine größere Schrittweite zu wählen, um später Kapitel ohne großen Aufwand umorganisieren zu können.¹⁶

Die Themenbeschreibung im Verzeichnisnamen ist frei wählbar und soll nur als Gedächtnisstütze dienen, was der Inhalt der jeweiligen Einheit ist. Mehrere Worte können durch Unterstrich oder Gedankenstrich getrennt werden, aber in der Regel empfiehlt es sich, ein einzelnes aussagekräftiges Wort zu nutzen.

Der Verzeichnisname kann aber noch zusätzliche Informationen beinhalten. Insgesamt hat der Verzeichnisname einer Lehreinheit das folgende Format:

$$\langle number \rangle [\langle target \rangle] - [\langle role \rangle] - \langle topic \rangle$$

Die Teile in eckigen Klammern sind jeweils optional. Sie haben folgende Bedeutung:

¹⁶ Ältere Programmierer kennen dieses Prinzip vielleicht noch von Programmiersprachen wie BASIC.

⟨target⟩ Beschreibt das Ziel dieser Einheit. Manchmal gibt es Material, das nur für bestimmte Arten von Materialien sinnvoll ist. Beispielsweise werden häufig die organisatorischen Inhalte der Auftaktvorlesung nicht mit im Skript stehen. Dies kann mit dem Targetbereich beschrieben werden. Es ist ein Code aus ein oder zwei Buchstaben:

- a Es ist nur Material eine sogenannte Langform (also ein Skript, ein Buch, etc.) in dem Verzeichnis enthalten.
- b Das Verzeichnis enthält Material für eine Präsentation.

Ein zusätzlicher zweiter Buchstabe kann den Bereich weiter einschränken. Beispielsweise steht „bh“ für ausschließlich Handouts, es werden also keine Slides erstellt.

Ohne eine Target-Spezifikation enthält das Verzeichnis Material für *alle* konfigurierten Dokumentarten.

Dieser Zielbereich ist erweiterbar, siehe die Dokumentation zu `ollm`.

⟨role⟩ Rolle, die die Einheit einnimmt. Dies wird stets durch einen einzelnen Buchstaben beschrieben:

- a Die Einheit ist ein Anhang.
- c Dies ist synonym zu einer ausgelassenen Rollenangabe und steht für eine normale Einheit.
- e Die Einheit ist eine Ergänzung. Damit ist gemeint, dass sie zur vorhergehenden Einheit gehört, aber eine gewisse Eigenständigkeit besitzt. Beispielsweise könnte es ein Übungsblatt oder Ergänzungsmaterial sein.
- i Diese Einheit integriert andere Einheiten. Beispielsweise würde die Integrationseinheit eines Vorlesungsskripts alle Skriptkapitel (die anderen Einheiten) zusammenfügen und ein Titelblatt, gemeinsames Inhaltsverzeichnis etc. hinzufügen. Jede Dokumentenart darf nur eine Integrationseinheit haben.

Achtung

Bitte beachten Sie, dass der Aufbau der Verzeichnisnamen nicht nur Konvention ist: Sowohl die Hauptklasse `osglecture` als auch das Tool `OLLM` werten die Namen aus.

In jedem Lehreinheitenverzeichnis befindet sich eine $\text{\LaTeX}$ -Datei. Sie verwendet die Dokumentklasse `osglecture` und trägt in der Standardkonfiguration den Namen `main.tex`. Neben dieser Datei kann sich beliebiges weiteres Material in dem Verzeichnis befinden, etwa Bilder, einzubindende Dateien oder private `.sty`-Dateien, die nur in dieser Einheit benötigt werden.

Dagegen sollten private `.sty`-Dateien oder andere Materialien, die in *allen* Lehreinheiten benötigt werden, im Include-Verzeichnis stehen.

3.4 Modi

Wer schon einmal mit [beamer](#) oder [ltx-talk](#) gearbeitet hat, kennt Modi und Overlayspezifikationen. Beide erlauben es, aus derselben Quelle unterschiedliche Darstellungen zu setzen, beantworten aber verschiedene Fragen.

Für alle, die noch nicht damit gearbeitet haben: Beides wird (auch gemischt) in spitzen Klammern als erstes Argument eines Befehls angegeben. Ein *Modus* entscheidet bei der Übersetzung, ob der Befehl und damit sein Inhalt mit übersetzt wird. So kann Textmaterial, das nur auf den Folien, aber z. B. nicht im Skript stehen soll, entsprechend gekennzeichnet werden. Bei Overlays können aus einer gestalteten Folie mehrere Seiten im PDF erzeugt werden, die alle leicht unterschiedlich aussehen, aber die gleiche Foliennummer haben. Damit können z. B. Animationen wie das zeilenweise Aufdecken oder Markieren von Inhalt erreicht werden. Angenommen, eine Folie zeigt zunächst nur eine Frage und nach dem nächsten Klick auch die Antwort. Das ist ein Overlay: Beide Ansichten gehören zum selben PDF, Target und Sprachbuild. Soll die ausführliche Begründung dagegen nur im Skript stehen und auf Folien ganz fehlen, ist das eine Modusentscheidung. Sie führt zu unterschiedlichem Inhalt verschiedener Dokumentinstanzen.

Modi und Overlays werden mitunter verwechselt, was auch daran liegt, dass sie bei [beamer/ltx-talk](#) häufig gemischt eingesetzt werden und eine gemeinsame Syntax haben. Tabelle 1 stellt die Unterschiede noch einmal kompakt dar.

Tabelle 1: Overlay und Modus

Frage	Mechanismus	Beispiel
Gehört der Inhalt in dieses Ergebnis?	Modus	longform, presentation
Wann erscheint er innerhalb der Präsentation?	Overlay	<2->

Das Paket [osglecture-modes](#) stellt das gemeinsame Modussystem bereit, das unabhängig von [beamer](#) und [ltx-talk](#) genutzt werden kann. Die konkreten Modi der Präsentationsklassen bleiben verfügbar, sollten aber nur abgefragt werden, wenn tatsächlich eine Besonderheit dieser Klasse gemeint ist.

Bei `\lecturemode<longform>` ist es ein Modus. Bei einem von der Basisklasse bereitgestellten `\item<2->` ist es ein Overlay. Ein Overlayausdruck ist keine portable Modusbedingung.

Modi bilden in [OSG Lecture](#) keinen flachen Schaltervorrat, sondern einen gerichteten Graphen vom Speziellen zum Allgemeinen. Der konkrete Modus `handout` hat beispielsweise die Eltern `presentation` und `print`; `script` führt über `article` zu `longform` und `print`. Eine Abfrage nach einem Elternmodus trifft damit alle passenden Spezialisierungen. Das ist der Grund, warum Autoren meist ihre Absicht als `presentation`, `longform`, `slides` oder `print` formulieren sollten, statt eine Basisklasse abzufragen.

Target, Dokumenttyp und Profil wirken dabei nacheinander. Das Target wählt die gewünschte Ausgabe, der Dokumenttyp aktiviert deren fachlichen Blattmodus, und das Profil ergänzt technische Modi der Satzumgebung. Das [beamer](#)-Profil ergänzt beispielsweise `beamer`; ein anderes

Präsentationsprofil muss diesen technischen Modus nicht besitzen. Der Elterngraph sorgt dennoch dafür, dass beide Profile dieselbe portable Abfrage `presentation` erfüllen. Profile dürfen außerdem zusätzliche Modi und eine Modus-Setup-Datei mitbringen. Sie erweitern damit die Fähigkeiten einer Satzumgebung, ohne die Bedeutung des Dokumenttyps zu ersetzen.

Overlays bleiben dagegen Aufgabe der gewählten Präsentationsklasse. [OSG Lecture](#) kann gebräuchliche Overlaybefehle anbinden, aber es verspricht nicht, dass jede klassen- oder themespezifische Animation einen Profilwechsel überlebt. Wer Portabilität zwischen [beamer](#) und [ltx-talk](#) benötigt, sollte den gemeinsamen Strukturwortschatz verwenden und Overlays auf die von beiden Profilen unterstützte Teilmenge begrenzen. Die konkrete Inhaltsauswahl wird in Abschnitt 4.5 fortgesetzt.

Erweiterbarkeit ist dann sinnvoll, wenn ein Projekt eine semantisch neue Ausgabe benötigt, nicht bloß ein anderes Layout. Ein betreutes Selbststudium könnte beispielsweise den Typ `studyguide` mit eigenen aktiven Modi und einem Langformprofil einführen. Ein bloßer Wechsel von [book](#) zu [scrbook](#) benötigt dagegen nur ein anderes Profil für den vorhandenen Typ `script`. Diese Unterscheidung verhindert, dass technische Satzentscheidungen unnötig viele Dokumentinstanzen erzeugen. Das vollständige Profilbeispiel steht in Abschnitt 7; neue OLLM-Targets werden über die dort beschriebenen Targetdefinitionen registriert.

3.5 Dokumententypen und Profile

Ein *Dokumenttyp* (*doctype*) bezeichnet die Ausgabeform, die angefordert werden kann: beispielsweise `slides`, `handout`, `script` oder `article`. Denken Sie dabei an die Frage „Was soll für wen entstehen?“ und nicht an eine konkrete L^AT_EX-Klasse.

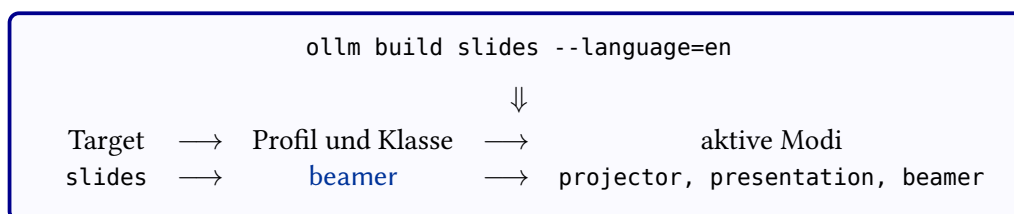


Abbildung 7: Von der Nutzerwahl zur konkreten Darstellung.

Im normalen Arbeitsablauf begegnet Ihnen der Dokumenttyp als Targetname im [OLLM](#)-Befehl. [OLLM](#) prüft im Manifest, ob dieses Target und die gewünschte Sprache für das Projekt vorgesehen sind. Anschließend übergibt es die normalisierte Auswahl an [osglecture](#).

Der anschließende Build *instanciiert* die gemeinsame Quelle für genau diese Auswahl. Das konkrete Ergebnis heißt *Dokumentinstanz*. Zwei Instanzen können also denselben fachlichen Inhalt besitzen und sich dennoch in Sprache, Dokumenttyp oder beidem unterscheiden.

Die Klasse wählt zu jedem Target ein *Profil*. Das Profil legt die konkrete Basisklasse und deren Adapter fest. So kann `slides` in einem Projekt mit [beamer](#), in einem anderen mit [ltx-talk](#) gesetzt

werden, ohne dass die Lehreinheiten umgeschrieben werden. Targetspezifische Ausnahmen sind ebenfalls möglich, etwa `beamer` für Folien und `ltx-talk` für Handouts.

Die *Basisklasse* ist die eigentliche L^AT_EX-Klasse, die den Satz ausführt, beispielsweise `beamer`, `ltx-talk`, `book` oder `scrbook`. Ein *Adapter* übersetzt die einheitliche Schnittstelle von `osglecture` in die Befehle und Konventionen dieser Basisklasse. Dadurch kann der fachliche Quelltext unabhängig von der gewählten Basisklasse bleiben.

Für Präsentationen wählen Sie das Profil projektweit mit der Option `presentation-profile`, für Langformen mit `longform-profile` in `Include/projectconfig.tex`. Soll nur ein einzelnes Target abweichen, verwenden Sie :

```
\LectureTargetSetup{<target>}{profile=<profile>}
```

Im Standalone-Betrieb (siehe Abschnitt 6) steht dafür die Klassenoption `profile` zur Verfügung. Die Profilwahl betrifft mehr als das Erscheinungsbild. `beamer` bietet beispielsweise ein großes Ökosystem vorhandener Themes; `ltx-talk` ist dagegen von Grund auf auf Tagged PDF und damit auf automatisches Struktur-Tagging ausgelegt. Bei Langformen unterscheiden sich `book` und `scrbook` unter anderem in ihren Klassenoptionen und Anpassungsmöglichkeiten. Ein Profilwechsel sollte deshalb durch einen vollständigen Build der betroffenen Targets geprüft werden.

Dokumenttypen und Klassen sind daher absichtlich getrennt. Die Trennung hält Ihre Quellen stabil: Sie entscheiden im Projekt, *wie* Folien gesetzt werden; in der Lehreinheit schreiben Sie nur, *was* auf Folien und in Langformen erscheinen soll.

slides bildschirmorientierte Präsentation für die Lehrveranstaltung;

handout druckorientierte Präsentationsfassung für das Publikum;

screen Vortragendenfassung der Folien: Jede Seite ist doppelt so breit und zeigt rechts neben der Folie wahlweise die aktuelle und die nächste Folie, die mit `\Note` notierten Anmerkungen und eine Uhr. Bei erweitertem Desktop liegt die linke Hälfte auf dem Projektor;

script zusammenhängende Langform, typischerweise als Kapitel oder Gesamtskript;

article eigenständige Langform, beispielsweise für einen Beitrag.

Einige Profile unterscheiden sich außerdem darin, ob sie die neue PDF-Verwaltung von L^AT_EX benötigen. Insbesondere `ltx-talk` verlangt einen `\DocumentMetadata`-Aufruf, der technisch vor `\documentclass` stehen muss. In einer Serie kann die Unit ihn nicht selbst passend voranstellen, weil Sprache und Target erst durch den Buildauftrag feststehen. Deshalb gehört der frühe Aufruf in `Include/documentmetadata.tex`:

```
\DocumentMetadata{
  lang=\OsgLectureRequestedLanguage,
  tagging=on
}
```

Das ist bewusst von sichtbaren Angaben wie `\title` und `\author` getrennt: `documentmetadata.tex` aktiviert PDF-Infrastruktur, während `projectconfig.tex` den fachlichen und gestalterischen Projektkontext beschreibt.

Diese Liste ist erweiterbar. Für den Einstieg sollten Sie jedoch die mitgelieferten Typen verwenden: Sie sind in [OLLM](#) registriert und besitzen bereits passende Profile, Modi und Anwendungsbereiche.

Profil	Basisklasse	Dokumenttypen	Kennzeichen
beamer	beamer	slides, handout, screen	große Theme-Auswahl
ltx-talk	ltx-talk	slides, handout, screen	Tagged PDF erforderlich
book	book	script, article	Standardklasse
scrbook	scrbook	script, article	KOMA-Script

Tabelle 2: Mitgelieferte Dokumentprofile.

4 Inhalte gestalten

4.1 Titelei und Metadaten

Auch wenn jede Lehreinheit *physisch* in einem eigenen Verzeichnis ist, gibt es einen Befehl für die *logische* Gliederung:

```
\lecture[⟨Kurzname⟩]{⟨Name⟩}{⟨Label⟩}
```

Beginnt eine logisch benannte Lehreinheit und erzeugt in Präsentationsformen deren Titelfolie.

`\lecture` legt in der Kurzform eine Titelfolie an, in der typischerweise neben dem Namen der Vorlesungseinheit auch die Metadaten noch einmal wiedergegeben werden. Die genaue Gestaltung hängt vom Präsentationsthema ab, falls Sie ein solches verwenden.

Alle Lehreinheiten eines Projekts gehören zum gleichen Kurs und haben daher den gleichen Kursnamen, den oder die gleichen Autoren und ggf. weitere gemeinsame Metadaten. Damit diese nicht für jede Lehreinheit wiederholt werden müssen, werden diese Metadaten in der Datei `Include/projectconfig.tex` hinterlegt. Dafür stehen Ihnen die folgenden Befehle zur Verfügung:

```
\course<⟨Modus⟩>[⟨Kurzname⟩]{⟨Name⟩}
```

Setzt den Kursnamen für alle oder für den angegebenen Modus.

```
\event<⟨Modus⟩>[⟨Kurzname⟩]{⟨Name⟩}
```

Setzt den Veranstaltungsnamen für alle oder für den angegebenen Modus.

Dieselben Signaturen besitzen `\title`, `\subtitle`, `\author`, `\institute` und `\date`. Das Profil entscheidet über `course-target` und `event-target`, ob Kurs und Veranstaltung in Titel, Untertitel oder gar nicht übernommen werden. Die vollständigen Setter und Abfragebefehle sind in Abschnitt 7.7 aufgeführt.

Die drei Argumentarten dürfen nicht vertauscht werden. Die spitzen Klammern wählen einen Modus, die eckigen Klammern enthalten den Kurztext, und die geschweiften Klammern den verpflichtenden Langtext. Beispielsweise setzt:

```
\course<presentation>[OS]{Betriebssysteme}
```

```
\course<longform>[Betriebssysteme]{Betriebssysteme und Systemsoftware}
```

damit je nach Dokumentinstanz unterschiedliche Kurz- und Langwerte. Ohne Modusangabe gilt der Wert für alle Instanzen; ohne Kurzform wird der Langwert zugleich als Kurzwert verwendet.

4.2 Präsentationen

Wenn Sie Präsentationen entwickeln, stellt **OSG Lecture** unabhängig vom gewählten Profil die gemeinsame Struktur aus `frame`, `\frametitle` und `\lecture` bereit. Overlays und themenspezifische Befehle stammen dagegen von der gewählten Basisklasse. Ein auf **beamer** zugeschnittenes Theme ist daher nicht automatisch mit **ltx-talk** verwendbar. Portable Inhaltsauswahl sollte über die im folgenden Abschnitt beschriebenen Modi erfolgen.

Standardmäßig bindet `patch-overlays=true` Ausdrücke aus registrierten Modi in die nativen Modus- und Overlaybefehle von **beamer** und **ltx-talk** ein. Eine Spezifikation wie `<longform>` oder `<slides|handout>` wird dann portabel ausgewertet; bei `<slides:2->` wertet **OSG Lecture** den Modusteil aus und reicht der Präsentationsklasse nur den Overlaybereich `<2->`. Eine rein numerische Spezifikation wie `<2->` bleibt ein klassenabhängiges Overlay. `|` und `,` innerhalb von `<...>` gehören stets zur Syntax der Präsentationsklasse; `||` ist der portable Oder-Operator. Unbekannte Ausdrücke werden unverändert an die Basisklasse weitergereicht. Wer diese Integration nicht möchte, setzt `patch-overlays=false`; `\lecturemode` und `\IfLectureModeTF` funktionieren weiterhin.

4.3 Anmerkungen und Quellenangaben

Mit `\Note{<Anmerkung>}` kann zu einem Frame ein Hinweis für die Vortragenden notiert werden. In einer Dokumentinstanz des Typs `screen` erscheinen die Anmerkungen neben der Folie; auf den Folien für das Publikum und in Langformen werden sie nicht gesetzt. Der Aufruf steht innerhalb des Frames, auf den sich die Anmerkung bezieht.

```
\begin{frame}
  \frametitle{Virtueller Speicher}
  Seitentabellen ordnen virtuelle Seiten physischen Seitenrahmen zu.
  \Note{Hier die Abbildung schrittweise erläutern.}
\end{frame}
```

Quellenangaben, etwa unmittelbar unter einer Abbildung, werden mit `\sourceref` gesetzt. Das optionale Argument enthält den Bezeichner, das Pflichtargument die Quelle. Der Bezeichner lautet sprachabhängig zunächst „Quelle“ beziehungsweise „Source“; ein leeres optionales Argument unterdrückt ihn. Die Sternform setzt eine eigene Quellenzeile rechtsbündig. In einem getaggtten Dokument ist die Quellenangabe regulärer Inhalt und bleibt Teil der Lesereihenfolge.

```
\includegraphics[width=.7\linewidth]{page-table}
\sourceref{Eigene Darstellung}
```

Für einen Einwurf neben dem eigentlichen Textstrom steht `\osgannotation{<Inhalt>}` zur Verfügung. Typischerweise enthält das Argument eine mit TikZ über die Folie gelegte handschriftliche Anmerkung. Der Aufruf gehört im Quelltext an die Stelle, an der die Anmerkung gelesen werden soll: Diese Stelle bestimmt bei getaggtter Ausgabe die Lesereihenfolge, nicht die gezeichnete Lage auf der Seite.

4.4 Mehrsprachige Dokumente

In einem **OSG Lecture**-Projekt können mehrere Sprachfassungen eines Inhalts erstellt werden. Dabei teilen sich OLLM und das Paket **langselect** die Arbeit: Das Manifest sagt, *welche* Sprachfassungen gebaut werden dürfen; OLLM wählt eine davon; **langselect** setzt in der Quelle die passende Formulierung. Diese Aufgabenteilung verhindert, dass die Buildlogik in den Fachtext wandert.

Tragen Sie jede Projektsprache zuerst im Manifest ein (`[targets.defaults].languages`, in fester Reihenfolge) und ordnen Sie sie anschließend in `Include/projectconfig.tex` einem von **babel** verstandenen Namen zu. Für Deutsch und britisches Englisch sieht die praktische Konfiguration so aus:

```
\LectureProjectSetup{
  languages={
    map={de=ngerman,en=british},
    load babel
  }
}
```

Die auswählbaren Sprachen selbst werden hier nicht mehr wiederholt – **OSG Lecture** übernimmt sie aus dem Manifest.

Ohne expliziten Unterschlüssel `targetlang` verwendet **OSG Lecture** automatisch `\OsgLectureLanguage`, also die vom Build gewählte Projektsprache. `targetlang` bleibt für einen bewussten Override verfügbar.

In den Lehreinheiten verwenden Sie danach die bereitgestellten Sprachmakros. Bei der Reihenfolge `selectable = {de,en}` ist `\ldeen{deutsch}{English}` die naheliegende Form. Schreiben Sie kurze, vollständige Varianten; Satzzeichen gehören in die jeweilige Sprachfassung, wenn sie sich unterscheiden können.

```
\section{\ldeen{Synchronisation}{Synchronization}}
\ldeen{Ein Semaphore schützt den kritischen Abschnitt.}
{A semaphore protects the critical section.}
```

Werte, Formeln, Makros oder Verweise, die in beiden Sprachen gleich sind, sollten Sie nicht kopieren. Nutzen Sie dafür die Sternform und Platzhalter. So steht ein später geänderter Wert weiterhin nur an einer Stelle:

```
\ldeeen*{Der Scheduler wählt alle @1 eine Aufgabe aus.}
    {The scheduler selects a task every @1.}
    {$10\,\mathrm{ms}$}
```

Die Platzhalter $\langle @1 \rangle$, $\langle @2 \rangle$ usw. werden in jeder Sprachfassung durch dieselben nachfolgenden Argumente ersetzt. Das ist besonders nützlich für Fachbegriffe, Querverweise und Zahlen. Wenn eine Sprache eine andere Wortstellung benötigt, platzieren Sie denselben Platzhalter einfach an der passenden Stelle.

Achtung

Starten Sie Builds über OLLM und wählen Sie die Sprache mit `--language`. Setzen Sie die Zielsprache **nicht** zusätzlich von Hand im Quelldokument; sonst können Manifest, Name der Ausgabedatei und tatsächlich gesetzte Sprache auseinanderlaufen.

Ob eine Übersetzung vollständig ist, prüfen Sie am zuverlässigsten mit aller vorgesehenen Dokumentinstanzen der Einheit. Eine erfolgreiche deutsche Fassung sagt nichts darüber aus, ob im englischen Zweig ein Argument oder ein Befehl fehlt.

```
ollm build --all
```

4.5 Inhalte mit OSG Lecture-Modi auswählen

Die begriffliche Trennung von Modi und Overlays sowie das Zusammenspiel mit Targets und Profilen wurde in Abschnitt 3.4 eingeführt. Hier geht es um die konkrete Autorenschnittstelle: Modi wählen Inhalte und Metadaten einer Dokumentinstanz aus; sie steuern nicht die Klickfolge innerhalb einer Präsentation. Ein *Modus* ist eine benannte Eigenschaft der gerade erzeugten Dokumentinstanz. Er beantwortet eine einfache Frage wie „Ist dies eine Präsentation?“, „Ist dies eine Langform?“ oder „Ist dies eine Druckausgabe?“. Der Befehl `\lecturemode<Modus>{Inhalt}` gibt seinen Inhalt nur aus, wenn die aktuelle Instanz diese Eigenschaft besitzt.

Wer `beamer` oder `ltx-talk` kennt, kennt bereits ähnlich geschriebene Modusbefehle. `\lecturemode` ist die portable Schnittstelle von `OSG Lecture`: Sie fragt den von Dokumentklasse und Profil unabhängigen Modusgraphen ab und funktioniert deshalb beim Wechsel zwischen beiden Präsentationsklassen ebenso wie in Langformen. Die spitzen Klammern sind hier eine Modusauswahl und keine Beamer-Overlayangabe.

Beginnen Sie bei angepassten Inhalten mit der allgemeinsten Absicht. Soll ein Absatz ausschließlich in einer Langform erscheinen, wählen Sie den Verhaltensmodus `longform`; soll er nur auf

Folien stehen, wählen Sie `slides`. Eine Festlegung auf die konkrete Basisklasse ist selten nötig und erschwert einen späteren Profilwechsel.

`OSG Lecture` unterscheidet ausgewählte Blattmodi und deren aktive Elternhülle. `\LectureModes` nennt die konkret gesetzten Modi; `\ActiveLectureModes` enthält zusätzlich alle über den Modusgraphen erreichbaren allgemeineren Modi. Bei Beamer-Folien kann die erste Liste beispielsweise `projector`, `beamer` enthalten, während in der zweiten außerdem `presentation` aktiv ist. `slides` ist dabei der benutzerfreundliche Alias für `projector`. Für die Inhaltsauswahl ist fast immer die Elternhülle gemeint; genau diese prüft `\lecturemode`.

```
\lecturemode<longform>{
  \ldean{Ausführliche Herleitung der Laufzeit:  $T(n)=\Theta(n\log n)$ .}
    {Detailed derivation of the running time:  $T(n)=\Theta(n\log n)$ .}
}
\lecturemode<slides>{
  \centering  $T(n)=\Theta(n\log n)$ 
}
```

Modi sind hier Eigenschaften der Ausgabe. Ein konkreter Dokumenttyp aktiviert zugleich allgemeinere Modi: Ein Handout ist beispielsweise eine Präsentationsform und eine Druckausgabe. Dadurch kann derselbe Inhalt mit `<presentation>` für Folien und Handouts oder mit `<print>` für alle druckorientierten Ergebnisse freigeschaltet werden. Mehrere Alternativen lassen sich mit `||` verbinden.

Dasselbe Modusargument steht auch an Metadatenbefehlen wie `\course<longform>` und an bedingt geladenen Präambelfragmenten zur Verfügung. Dadurch bleibt die Auswahlregel auf allen Ebenen gleich: Erst wird der Modusgraph aus Dokumenttyp und Profil fertiggestellt, danach werden modusabhängige Werte und Inhalte ausgewertet. Eigene Modi müssen deshalb vor dieser Finalisierung deklariert werden; die Projektkonfiguration dafür steht in Abschnitt 7.6.

4.6 Aufzählungen als Fließtext

Für Aufzählungen, die im Skript besser als Fließtext lesbar sind, ist `presitemize` die nutzerfreundlichere Wahl. Die Einträge beschreiben Aussagen; die Einleitungsbefehle geben deren logische Beziehung an.

```
\begin{presitemize}
  \item Die Übersetzung \finite{funktioniert}
  \anditem das Programm \finite{liefert} ein Ergebnis
  \butitem wir \finite{haben} vereinfacht
  \thereforeitem wir \finite{müssen} das Ergebnis prüfen
\end{presitemize}
```

In Präsentationen bleibt daraus eine Liste. In der Langform verbindet `presitemize` die Aussagen sinngemäß zu: „Die Übersetzung funktioniert und das Programm liefert ein Ergebnis, aber wir haben vereinfacht, weshalb wir das Ergebnis prüfen müssen.“ `\finite` markiert das finite Verb, damit es insbesondere im deutschen Nebensatz an die richtige Stelle rücken kann.

Für reine Interpunktionsfortsetzungen stehen `\commaitem` und `\colonitem` zur Verfügung. Sie erzeugen in der Langform „,“ beziehungsweise „:“ ohne ein zusätzliches Verbindungswort; in Präsentationen erscheinen sie als gewöhnliche Items. Gleichwertig sind `\presitem[relation=comma]` und `\presitem[relation=colon]`.

Verwenden Sie `presitemize` für flache Folgen kurzer Aussagen. Für verschachtelte Listen, Tabellen oder deutlich verschiedene Erklärwege sind explizite Modusvarianten verständlicher. Als Faustregel gilt: Ändert sich nur die sprachliche Verbindung, genügt `presitemize`; ändert sich die didaktische Struktur, verwenden Sie `\lecturemode`.

Hinweis

Die automatische Fließtextverknüpfung von `presitemize` besitzt derzeit Sprachprofile für Deutsch, Englisch und Französisch einschließlich üblicher Varianten wie `ngerman`, `british` und `fr-CA`. Bei einer anderen Sprache warnt das Paket und verwendet für die Verbindungswörter ersatzweise das englische Profil. Das ist nur dann sinnvoll, wenn die betreffende Ausgabe tatsächlich englischen Text enthält.

Für weitere Sprachen formulieren Sie die Langform explizit und wählen mit `\lecturemode` zwischen ihr und der Präsentationsliste. So bleiben Wortstellung, Flexion und Interpunktion unter Ihrer Kontrolle; die einzelnen Formulierungen können darin weiterhin mit den von `langselect` erzeugten Sprachmakros ausgewählt werden.

4.7 Zweispaltige Inhalte

Was auf einer breiten Folie nebeneinander steht, etwa eine Abbildung und ihre Erläuterung, liest sich auf der schmaleren Seite einer Langform meist besser untereinander. Die Umgebung `twocolumns` trifft diese Unterscheidung selbst: Sie setzt zwei durch `\nextcolumn` getrennte Teile nebeneinander, wenn ein Modusausdruck erfüllt ist, und sonst nacheinander. Ohne Angabe lautet der Modusausdruck `presentation`.

```
\begin{twocolumns}[t,.46]
  \includegraphics[width=\linewidth]{page-table}
  \nextcolumn
  Seitentabellen ordnen virtuelle Seiten physischen Seitenrahmen zu.
\end{twocolumns}
```

Im optionalen Argument bestimmt eine Zahl zwischen null und eins den Anteil der ersten Spalte an der verfügbaren Breite (Vorgabe `.5`). Ein Wert `t`, `c` oder `b` richtet beide Spalten oben, mittig

oder unten aus; zwei solche Werte richten die Spalten einzeln aus. `sep` setzt den Steg zwischen den Spalten, etwa `sep=1cm`, und ersetzt die Vorgabe `\columnsep`. Der Modusausdruck steht in spitzen Klammern vor dem optionalen Argument: `<all>` erzwingt Spalten in jedem Modus, `<longform>` nur in Langformen und `<presentation|longform>` in beiden Verhaltensklassen.

4.8 Querverweise

Für Verweise innerhalb derselben Dokumentinstanz können Sie die üblichen Befehle von `\hyperref` verwenden. `OSG Lecture` stellt daneben `\olref` und `\olpageref` bereit. Ohne optionale Dokumentadresse verhalten sie sich wie `\ref` und `\pageref`:

```
\section{Scheduling}\label{sec:scheduling}
Siehe Abschnitt~\olref{sec:scheduling} auf Seite~\olpageref{sec:scheduling}.
```

Das optionale Argument vor dem Label bezeichnet eine (andere) logische Lehreinheit derselben Serie. Sie erinnern sich: Diese logische Identität wird im dritten Parameter des `\lecture`-Befehls festgelegt. Standardmäßig verwendet der Verweis dabei denselben Dokumenttyp und dieselbe Sprache wie das aktuelle Dokument. Der folgende Verweis aus der Unit `memory` führt daher in einem deutschen Skript zum deutschen Skript der Unit `processes`, in englischen Folien dagegen zu deren englischen Folien:

```
\olref[processes]{sec:scheduling}
```

Mit den Schlüsseln `type` und `lang` im optionalen Argument lässt sich diese Dokumentinstanz ausdrücklich überschreiben. Das ist dann ein dokumenttypübergreifender Verweis: Er kann beispielsweise von Folien auf die ausführliche Herleitung im Skript zeigen:

```
\olautoref[processes,type=script,lang=de]{sec:scheduling-proof}
```

Ein schlüsselloser Eintrag der Adresse ist stets die Unit-ID. Verwenden Sie stabile logische IDs statt Verzeichnisnamen; OLLM löst die gewünschte Kombination aus Unit, Dokumenttyp und Sprache über seinen Referenz-Snapshot auf. Das Zieldokument muss mindestens einmal erfolgreich gebaut worden sein; erst dann stehen seine Referenzen anderen Units zur Verfügung. Nach Änderungen an abhängigen Units ist `ollm build --resolve` der bequemste Weg, Produzenten und Konsumenten bis zu einem konsistenten Zustand neu zu bauen.

4.9 Literatur

Für Literaturangaben nutzt [OSG Lecture biblatex](#) mit biber. Die Literaturverwaltung wird in [Include/projectconfig.tex](#) eingeschaltet; die Datenbank liegt üblicherweise ebenfalls in [Include](#), wo [OLLM](#) sie für biber auffindbar macht:

```
\LectureProjectSetup{
  bibliography = {resources = lecture.bib}
}
```

Der voreingestellte Stil alphabetic erzeugt Labels wie [Knu69]; [style](#) wählt einen anderen [biblatex](#)-Stil, [options](#) reicht weitere Ladeoptionen weiter. Zitiert wird mit den üblichen Befehlen wie [\cite](#). Ein Literaturverzeichnis gibt [\printbibliography](#) aus; welche Dokumenttypen eines erhalten, steuern Sie wie alles andere über Modi, etwa nur im Skript:

```
\mode<longform>{\printbibliography}
```

Ein Zitat verlinkt nur dann auf das Literaturverzeichnis, wenn dieses den Eintrag im selben Dokument tatsächlich ausgibt. In Folien ohne Verzeichnis bleibt das Label daher reiner Text statt eines Links ins Leere. Themen- oder kapitelweise Verzeichnisse erhalten Sie mit den Filtern von [biblatex](#), etwa `keyword=`, `type=` oder `category=`.

Ein serienweites Literaturverzeichnis gibt [\printseriesbibliography](#) aus: alle Einträge, die irgendeine Unit der Serie mit gleichem Dokumenttyp und gleicher Sprache zitiert, gefiltert wie bei [\printbibliography](#). Es kann in jeder Unit stehen, etwa in einer eigenen Literatur-Unit oder am Ende eines Kapitels. Zitiert eine Unit Neues, meldet `ollm check` das Verzeichnis als veraltet; `ollm build --resolve` baut es neu. Nennt die Projektkonfiguration diese Unit als [series-target](#), verlinken Zitate anderer Units auf ihre Einträge, wenn das zitierende Dokument kein eigenes Verzeichnis mit dem Eintrag hat; bei der Integration werden diese Links intern:

```
\LectureProjectSetup{
  bibliography = {resources = lecture.bib, series-target = literature}
}
```

Eine kommentierte Leseliste stellt ausgewählte Einträge vollständig dar, gegliedert in Gruppen:

```
\begin{literaturelist}
  \litgroup{Grundlagen}
  \litentry{Knu:Art-1:1969}
  \litcomment{Das Standardwerk; umfassend, aber anspruchsvoll.}
  \litgroup{Entwurf von Algorithmen}
  \litentry{Ski:AlgorithmDesign:2008}
\end{literaturelist}
```

`\litcomment` kommentiert die vorangehenden Einträge; ein optionales Argument setzt ein Label vor den Kommentar. Das optionale Argument der Umgebung legt die Breite der Labelspalte fest.

4.10 Befehlsreferenz

Die folgenden Übersichten nennen die vollständige öffentliche Schnittstelle für die in diesem Kapitel behandelten Aufgaben. Befehle zur Definition neuer Profile oder Adapter stehen dagegen in Abschnitt 7.7; sie werden beim gewöhnlichen Schreiben von Lehreinheiten nicht benötigt.

Lehreinheit und Metadaten.

`\lecture`[*<short title>*]{*<title>*}{*<unit ID>*}

beginnt die logisch identifizierte Lehreinheit.

`\course`<*<mode>*>[*<short value>*]{*<long value>*}

`\event`<*<mode>*>[*<short value>*]{*<long value>*}

`\title`<*<mode>*>[*<short value>*]{*<long value>*}

`\subtitle`<*<mode>*>[*<short value>*]{*<long value>*}

`\author`<*<mode>*>[*<short value>*]{*<long value>*}

`\institute`<*<mode>*>[*<short value>*]{*<long value>*}

`\date`<*<mode>*>[*<short value>*]{*<long value>*}

setzen ein Metadatenfeld; Modus und Kurzform sind optional.

`\OsgLectureSet`<*<Field>*><*<mode>*>[*<short value>*]{*<long value>*}

einheitlicher Settername für jedes vordefinierte oder deklarierte Metadatenfeld.

`\OsgLecture`<*<Field>*>

`\OsgLectureShort`<*<Field>*>

expandierbare Abfrage des Lang- beziehungsweise Kurzwerts.

Modi und bedingte Inhalte.

`\lecturemode`<*<mode expression>*>{*<content>*}

gibt den Inhalt aus, wenn mindestens einer der mit Komma oder || getrennten Modi aktiv ist; ohne Modusausdruck wird der Inhalt immer ausgegeben.

`\IfLectureModeTF`{*<expression>*}{*<true>*}{*<false>*}

vollständige Verzweigung.

`\IfLectureModeT`{*<expression>*}{*<true>*}

nur der Wahr-Zweig.

`\IfLectureModeF{<expression>}{<false>}`
 nur der Falsch-Zweig.

`\LectureModes`

`\ActiveLectureModes`

expandierbare kommaseparierte Listen der gewählten Blattmodi beziehungsweise ihrer vollständigen Elternhülle.

Die Befehle zur Erweiterung des Modusgraphen sind in Abschnitt 7.6 zusammengefasst; sie gehören zur Projekt- und Profilkonfiguration, nicht zum gewöhnlichen Schreiben einer Unit.

Präsentationslisten.

`\begin{presitemize}[<language, punctuation, whitespace>]`
 erzeugt in Präsentationen eine Liste und in Langformen einen grammatisch verbundenen Absatz. `language` überschreibt die erkannte Sprache; `punctuation` akzeptiert `auto` oder `manual` und respektiert bei `auto` vorhandene Satzzeichen; `whitespace` akzeptiert `normalize` (Vorgabe) oder `preserve`.

`\presitem[<relation, initial>]<overlay>[<label>]`
`relation`: sentence, and, but, therefore, comma, colon; `initial`: auto, keep.

`\anditem<overlay>[<label>]`

`\butitem<overlay>[<label>]`

`\thereforeitem<overlay>[<label>]`

`\commaitem<overlay>[<label>]`

`\colonitem<overlay>[<label>]`

Kurzformen für die entsprechenden Relationen; jede Form akzeptiert eine Overlayangabe und ein optionales Label wie `\item`.

`\finite{<text>}`
 markiert den finiten Teil einer untergeordnet angeschlossenen Formulierung.

`\prespar`
 beginnt in der Langform einen neuen Absatz und schließt den laufenden Satz ab; in Präsentationen wirkungslos.

`\longform`

`\slides`

`\longslides`

`\prescase`

`\preskeepcase`

niedrigstufige Auswahl- und Großschreibungshelfer für eigene Listenmakros.

Zweispaltige Inhalte.

`\begin{twocolumns}<\mode expression>[<alignment, share, sep=length>]`

setzt zwei Teile nebeneinander, wenn der Modusausdruck erfüllt ist (Vorgabe presentation), sonst nacheinander.

`\nextcolumn`

trennt die beiden Teile von twocolumns.

Verweise.

`\olref*[\langle address \rangle]{\langle label \rangle}`

Nummer wie `\ref`.

`\olpageref*[\langle address \rangle]{\langle label \rangle}`

Seite wie `\pageref`.

`\olnameref*[\langle address \rangle]{\langle label \rangle}`

Name wie `\nameref`.

`\olautoref*[\langle address \rangle]{\langle label \rangle}`

automatischer Verweis wie `\autoref`.

Die Adresse besteht aus einer schlüssellosen Unit-ID und optional `type = {doctype}` sowie `lang = {language}`. Ohne Adresse bleiben alle vier Befehle lokal. Die Sternform unterdrückt wie bei den entsprechenden Standardbefehlen den Hyperlink.

Literatur.

`\begin{literaturelist}[\langle label width \rangle]`

kommentierte Leseliste; verfügbar, wenn die Literaturverwaltung eingeschaltet ist.

`\printseriesbibliography[\langle options \rangle]`

serienweites Literaturverzeichnis; `\langle options \rangle` wie bei `\printbibliography`.

`\litentry{\langle key \rangle}`

vollständiger Eintrag mit Zitatlabel.

`\litgroup{\langle title \rangle}`

beginnt eine Gruppe mit Überschrift.

`\litcomment[\langle label \rangle]{\langle text \rangle}`

Kommentar zu den vorangehenden Einträgen.

Zitier- und Verzeichnisbefehle sind die von `biblatex`, etwa `\cite` und `\printbibliography`.

5 Serien

Eine Serie ist mehr als eine Menge von Verzeichnissen. Ihre Einheiten können voneinander abhängen: Kapitelnummern sollen fortlaufen, ein Querverweis benötigt ein Ziel aus einer anderen Unit, und ein Gesamtdokument benötigt die gebauten Einzelkapitel. **OLL**M speichert dafür den Zustand erfolgreicher Builds und kann veraltete Produzenten mit `--resolve` neu bauen. Ohne diesen Serienzustand könnte ein lokaler L^AT_EX-Lauf nur die aktuelle Quelldatei sehen.

5.1 Zähler fortsetzen

Jede Unit wird als eigenständiges Dokument gebaut. Das löst das Problem getrennter Ausgabe und inkrementeller Builds, führt aber zunächst zu einem neuen Problem: L^AT_EX würde Kapitel, Abschnitte und Seiten in jeder Unit wieder bei eins beginnen lassen. Für ein später integriertes Skript wären dann etwa mehrere Kapitel 1 vorhanden. Die Fortsetzungsfunktion importiert deshalb ausgewählte Endstände der vorherigen, für den Dokumenttyp passenden Unit.

Welche Zähler fortlaufen, ist eine Darstellungsentscheidung. Folien benötigen meist keine durchgehenden Seitenzahlen; ein Skript dagegen häufig schon. Die Einstellung gehört deshalb zum Modus longform in `Include/projectconfig.tex`:

```
\OsgLectureSetup{
  mode/longform={
    continuation={counters={page,chapter,section}}
  }
}
```

Angenommen, 010-introduction endet mit Abschnitt 2. Dann beginnt der erste `\section`-Aufruf in 020-processes mit Abschnitt 3. Ein nicht genannter Zähler wie `figure` beginnt weiterhin neu. Diese explizite Auswahl ist wichtig: Lokale Zähler sollen nicht versehentlich eine Bedeutung über Unit-Grenzen erhalten. Soll eine Langform überhaupt nichts fortsetzen, kann `continuation=none` verwendet werden.

Die vorherige Instanz muss mindestens einmal erfolgreich gebaut worden sein, damit ihr Zählerstand vorliegt. Nach dem Einfügen eines neuen Abschnitts in eine frühe Unit aktualisiert folgender Lauf die abhängigen deutschen Skriptinstanzen:

```
ollm build script --language=de --scope=series --resolve
```

5.2 Integration

Die Dokumente der Lehreinheiten einer Serie können einzeln verwendet werden. Dies ist in der Regel bei Präsentationen sinnvoll. Sie können jedoch auch zu einem gesamten Dokument integriert werden. So könnte jede Lehreinheit ein Kapitel eines Skripts bilden, die dann zu einem Gesamtskript integriert werden. Es kann aber z. B. auch eine Handoutsammlung angelegt werden. Ein Gesamtskript entsteht in *OSG Lecture* in der Regel nicht durch `\include` der TeX-Quellen, sondern durch Zusammenfügen bereits übersetzter Dokumente. Dafür muss es eine eigene Integrationseinheit mit einem Rahmendokument geben. Diese Einheit ist durch den Buchstaben „i“ beim Rollenelement gekennzeichnet. Die laufende Nummer des Verzeichnisses ist unerheblich, weil sie bei der Zählung nicht berücksichtigt wird. Es empfiehlt sich aber, hierfür entweder die kleinste oder die größte Nummer zu wählen, so dass die zu integrierenden Einheiten aufeinander folgen können.

```
course/
  ollmconfig.toml
  Include/
    projectconfig.tex
    documentmetadata.tex
  010-introduction/main.tex
  020-processes/main.tex
  900-i-collection/main.tex
```

Der normale Workflow der Integration ist inkrementell: Zunächst bauen Sie beim Schreiben die zu integrierenden Dokumente und erst zuletzt das Rahmendokument. Es empfiehlt sich, vor dem Bau des Rahmendokuments mit `ollm check` zu überprüfen, ob alle zu integrierenden Dokumente erstellt und alle Referenzen aufgelöst sind. Falls nicht, müssen Sie ggf. weitere Übersetzungsläufe anstoßen. Auch hier kann Sie `ollm` unterstützen: Mit Hilfe der Option `--resolve` werden Abhängigkeiten bis zu einer vorgegebenen Tiefe erkannt und aufgelöst.

```
cd 020-processes
ollm script --language=de
ollm script --language=de --scope=series --resolve
ollm script --language=de --scope=collection --resolve
ollm check --scope=series
```

5.3 Rahmendokumentunit

Die Klasse erkennt diese Rolle am Verzeichnisnamen und lädt den Integrationsdienst automatisch. Das Dokument nennt die einzubindenden logischen Units in der gewünschten Reihenfolge:

```

\documentclass{osglecture}

\begin{document}
\includeunit{introduction}
\includeunit{processes}
\includeunit{synchronization}
\end{document}

```

`\includeunit` wählt automatisch dieselbe Sprache und denselben Dokumenttyp wie das Rahmendokument. Es importiert also bei einem deutschen Skript genau die deutschen Skriptinstanzen. OLLM baut die Collection mit `--scope = {collection}`; existieren keine oder mehrere passende Integrationsunits für den Dokumenttyp, ist die Auswahl nicht eindeutig und der Build schlägt fehl.

Jede einzubindende Dokumentinstanz muss zuvor mindestens einmal gebaut sein. Der Integrationslauf protokolliert den tatsächlich verwendeten Buildstand als Abhängigkeit. Nach Änderungen genügt deshalb normalerweise ein erneuter Lauf mit `--resolve`. Die Reihenfolge der `\includeunit`-Befehle ist die Reihenfolge im Ergebnis; sie wird nicht aus den Verzeichnisnummern erraten.

5.4 Tags, Links und Fallback

Sind Rahmendokument und importierte Units als Tagged PDF gebaut, übernimmt `tagpax` Seiten, Strukturbaum, Ziele, Links, Inhaltsverzeichniseinträge und Lesezeichen. Dadurch bleiben Barrierefreiheitsstruktur und Navigation über Unit-Grenzen erhalten. Dafür muss das Target `\DocumentMetadata` erlauben beziehungsweise verlangen, und `documentmetadata.tex` muss Tagging aktivieren.

```

\DocumentMetadata{
  lang=\OsgLectureRequestedLanguage,
  tagging=on
}

```

Für ungetaggte Dokumentinstanzen verwendet die Integration einen Fallback über `pdfpages` und `newpax`. Links und Gliederung bleiben dabei weitgehend funktionsfähig, die importierten Seiten werden jedoch als Artefakt eingebunden und besitzen keine erschlossene Lesereihenfolge oder Überschriftenstruktur. Gemischte getaggte und ungetaggte Units zwingen für den betroffenen Dokumenttyp den ungetaggten Pfad und erzeugen eine deutliche Warnung. Eine vollständig getaggte Serie ist daher nicht nur qualitativ besser, sondern auch vorhersehbarer.

5.5 Referenzübersicht

`\includeunit{⟨unit ID⟩}`

bindet in einer Integrationseinheit die bereits gebaute Instanz derselben Sprache und desselben Dokumenttyps ein; die Aufrufreihenfolge bestimmt die Dokumentreihenfolge.

Integration ist nur in einer Serie mit genau einer passenden Unit der Rolle `i` pro Dokumenttyp verfügbar. Sie setzt erfolgreich gebaute Quelldokumente voraus. Serienübergreifende Referenzen, Counter-Fortsetzung und Integration werden über OLLMs Zustands- und Abhängigkeitsmodell aufgelöst; lokale LaTeX-Läufe ohne diesen Zustand können sie nicht rekonstruieren.

6 Standalone

Neben dem Serienbetrieb, den wir bisher betrachtet haben, besitzt **OSG Lecture** noch eine zweite Betriebsart, den Standalone-Betrieb. Er dient dazu, ein einzelnes Dokument in der durch ein **OSG Lecture**-Profil bestimmten Weise zu übersetzen, ohne dafür ein Kursmanifest und eine Verzeichnisserie anzulegen. Ein typischer Anwendungsfall ist der Foliensatz für einen Vortrag auf einer wissenschaftlichen Konferenz.

Im Standalone-Betrieb beschreibt das Dokument seine Ausgabe selbst und besitzt weder Buildauftrag noch Serienmanifest. Die fachlichen Befehle bleiben weitgehend gleich; unterschiedlich ist vor allem, woher Kontext und Defaults stammen.

Ein Standalone-Dokument hat weder Buildauftrag noch Serienmanifest und muss dies mit der Klassenoption **standalone** ausdrücklich erklären. Alternativ versteht **OLLM** den Komfortaufruf `ollm standalone main.tex`. Der direkte Klassenmodus ist meist einfacher: Ohne eine Menge automatisch zu erzeugender Dokumentinstanzen, Serienabhängigkeiten oder Deployment gibt es für **OLLM** wenig zu koordinieren, und `latexmk main.tex` genügt.

Hinweis

Die Wahl ist eine Frage des Problems, nicht der Dokumentlänge: Ein vierseitiges Handout, das in drei Sprachen zusammen mit zwölf Lehreinheiten gebaut wird, gehört in eine Serie. Ein hundertseitiger einmaliger Tagungsband ohne Nachbardokumente kann dagegen Standalone sein.

Dokumenttyp und Sprache, die in einer Serie von **OLLM** kommen, werden nun mit **doctype** und **lang** gewählt. Mit **profile** kann das Dokumentprofil überschrieben werden:

```
\documentclass[
  standalone,
  doctype=script,
  profile=scrbook,
  lang=de,
  class-options={11pt,oneside}
]{osglecture}
```

In diesem Beispiel ist **script** die semantische Ausgabeform, **scrbook** die technische Satzklasse und **de** die unmittelbar im Dokument gewählte Sprache. In einer Serie kämen diese drei Entscheidungen aus Manifest und Projektkonfiguration; Standalone macht sie absichtlich am Dokument sichtbar.

doctype ist ohne Angabe **slides**; für die eingebauten Dokumenttypen existieren Defaultprofile. Explizite Werte machen ein Standalone-Dokument dennoch verständlicher und reproduzierbarer. **class-options** reicht Optionen direkt an die vom Profil gewählte Basisklasse weiter. Eine

gleichzeitig vorhandene OLLM-Auftragsdatei und die Option `standalone` sind ein Fehler, weil sonst zwei Quellen denselben Buildkontext bestimmen würden.

Ohne Manifest wird `Include/projectconfig.tex` nicht automatisch gefunden. Gemeinsam genutzte Pakete, Sprachkonfiguration, Titel und eigene Makros müssen daher im Dokument geladen oder ausdrücklich aus einer Datei eingelesen werden. Ein eigenständiges mehrsprachiges Skript kann beispielsweise so beginnen:

```
\documentclass[standalone,doctype=script,profile=scrbook,lang=de]
  {osglecture}
\usepackage[
  languages={de,en},
  targetlang=de,
  map={de=ngerman,en=british},
  load babel
]{langselect}
\title{\ldeeen{Betriebssysteme}{Operating Systems}}
```

Profile wie `ltx-talk` verlangen `\DocumentMetadata` bereits vor `\documentclass`. In einem Standalone-Dokument gibt es keine von OLLM vorgeschaltete `documentmetadata.tex`; der Aufruf muss daher wirklich am Dateianfang stehen:

```
\DocumentMetadata{lang=en,tagging=on}
\documentclass[
  standalone,doctype=slides,profile=ltx-talk,lang=en
]{osglecture}
```

Sie können ein solches Dokument direkt mit Lua^AT_EX beziehungsweise latexmk bauen. Der kompatible Komfortaufruf `ollm standalone main.tex` verwendet ebenfalls die Klassenoptionen; Target-, Sprach- und Matrixoptionen von OLLM sind dabei nicht zulässig. Ohne Serienzustand gibt es keine externen Unit-Referenzen, Counter-Continuation oder `\includeunit-`Auflösung. Lokale Referenzen und alle modusabhängigen Darstellungsbefehle funktionieren unverändert.

7 Projekteinstellungen

Die Konfiguration von **OSG Lecture** liegt absichtlich nicht in einer einzigen Datei. Verschiedene Ebenen beantworten verschiedene Fragen: **OLLM** entscheidet, *was* gebaut werden darf und wo Ergebnisse liegen; **L^AT_EX** entscheidet, *wie* eine ausgewählte Dokumentinstanz gesetzt wird. Eine Einstellung gehört auf die niedrigste Ebene, auf der sie für alle betroffenen Dokumente gilt.

Nutzerkonfiguration wählt persönliche Defaults, derzeit insbesondere das bevorzugte Bundle-Preset.

ollmconfig.toml definiert Identität, Dokumentinstanzen, Sicherheit, Auflösung und Deployment des Projekts.

Preset- und Targetdefinitionen geben den Namen eines Targets seine Bedeutung: Dokumenttyp, Profilkategorie, Metadatenvertrag und zulässige Unit-Scopes.

Include/projectconfig.tex enthält die gemeinsame semantische **L^AT_EX**-Konfiguration, etwa Profile, Sprachen, Metadaten und Klassenoptionen.

Include/documentmetadata.tex enthält den frühen `\DocumentMetadata`-Aufruf für Targets, die Tagged PDF oder andere frühe PDF-Eigenschaften verlangen.

Lehreinheit kann projektweite Metadaten für ihren eigenen Inhalt konkretisieren; erzwungene Projektvorgaben bleiben verbindlich.

7.1 Das Projektmanifest

Die Datei **ollmconfig.toml** markiert die Projektwurzel. **OLLM** sucht sie vom Arbeitsverzeichnis aus nach oben. Das Manifest beschreibt eine zu erzeugenden Dokumentinstanzen und enthält keinen Lehrtext. Ein kompaktes Projekt mit zwei Sprachen und drei Targets beginnt beispielsweise so:

```
schema = 2
bundle_preset = "OSG lecture/1"

[project]
id = "operating-systems"

[targets.defaults]
languages = ["de", "en"]
default_language = "de"

[targets.slides]
[targets.handout]
[targets.script]
```

`schema` bezeichnet das Dateischema, nicht die Version des Bundles. `project.id` ist die dauerhafte technische Serien-ID und geht in Jobnamen und Dokumentidentitäten ein. Ändern Sie sie nach dem ersten Build nur bewusst.

`[targets.defaults]` sammelt, was jedes Target erbt, solange es nichts Eigenes setzt: `languages` listet in fester Reihenfolge alle Projektsprachen und ist zugleich der Prüfkanker (die `languages` eines Targets müssen eine Teilmenge sein); `default_language` wird gebaut, wenn die CLI keine Sprache nennt. Ein leeres `[targets.slides]` erbt alles; ein weggelassenes Target existiert schlicht nicht. Die Zuordnung von `de` zu `ngerman` oder von `en` zu `british` ist dagegen eine L^AT_EX-Angelegenheit und gehört als `map` von `langselect` in `projectconfig.tex` – die Sprachliste selbst muss dort nicht wiederholt werden.

Ein Target darf `languages`, `default_language`, `profile` und `document_metadata` überschreiben. `profile` wählt das Dokumentprofil für genau dieses Target; `[targets.defaults]` kann stattdessen `presentation_profile` und `longform_profile` setzen, die je nach Profilkategorie des Targets greifen. Fehlt beides, gelten die Profil-Defaults des Bundle-Presets. `document_metadata` mit `enabled` oder `disabled` ist eine bewusste Ausgabeentscheidung und wirkt nur dort, wo das aufgelöste Profil den Metadatenvertrag offen lässt (`supported`); widerspricht sie einem Profil mit erzwungenem Vertrag, weist `OLLM` den Build vor dem Lauf ab.

`[project]` kann zusätzlich `tex_directory` und `tex_config` setzen (Standard Include bzw. `projectconfig.tex`). Der Verzeichnisname bleibt projektwurzel-relativ, `tex_config` ist nur ein Dateiname:

```
[project]
id = "operating-systems"
tex_directory = "Include"
tex_config = "projectconfig.tex"
```

Weitere Manifestbereiche steuern `[security]` (`shell_escape`, `overwrite`), die maximale Laufzeit von `build --resolve` und das Deployment.

7.2 Bundle-Preset

Das Bundle-Preset ist ein versionierter Vertrag zwischen dem Projekt und den mit `OLLM` installierten Definitionen. `OSG lecture/1` wählt das Vokabular und die Regeln dieser Bundle-Generation. Es ist nicht mit einem Dokumentprofil zu verwechseln: Das Preset erklärt `OLLM` die Buildwelt, während ein Profil später die konkrete L^AT_EX-Klasse und Darstellung festlegt.

Die mitgelieferten Targetdefinitionen registrieren `slides`, `handout`, `screen` und `script`. Eine Definition legt `doctype`, `profile_class` und `unit_scopes` fest. Das Preset bringt außerdem die Profil-Defaults `presentation_profile` und `longform_profile` mit. Profilbeschreibungen (`document-metadata`, `doctypes` und mehr) liegen im mitgelieferten Verzeichnis

definitions/profiles/; projektlokale Definitionen können über `.ollmconfig.local.toml` ergänzt werden und ersetzen gleichnamige eingebaute nicht stillschweigend.

Ein eigenes Target ist beispielsweise sinnvoll, wenn Studierende neben dem normalen Skript einen didaktisch anders aufgebauten Studienleitfaden erhalten sollen. Dazu registriert Nora zunächst den Suchpfad in `.ollmconfig.local.toml`:

```
schema = 1

[definitions]
paths = ["Definitions"]
```

Dann legt sie `Definitions/targets/studyguide.toml` an:

```
schema = 1
kind = "target"
name = "studyguide"
version = "1.0"
doctype = "studyguide"
profile_class = "longform"
unit_scopes = ["a", "as"]
```

Erst diese Definition gibt dem Namen `studyguide` eine Bedeutung: Er ist eine Langform und gilt für die angegebenen Unit-Scopes. Danach kann das Manifest `[targets.studyguide]` aufnehmen. Zusätzlich benötigt L^AT_EX ein Profil, das den neuen Dokumenttyp unterstützt; dessen Profilbeschreibung legt Nora als `Definitions/profiles/<name>.toml` neben die `.def`. Ein neuer Name allein genügt also absichtlich nicht; Buildauswahl und Satz müssen denselben Vertrag verstehen.

7.3 Standardeinstellungen für Nutzer

Wer mehrere Projekte mit demselben Bundle bearbeitet, kann sein bevorzugtes Preset persönlich vorbelegen. Unter Unix liegt die Datei gewöhnlich in `$XDG_CONFIG_HOME/ollm/config.toml` oder `$HOME/.config/ollm/config.toml`, unter Windows in `%APPDATA%/ollm/config.toml`. `OLLM_USER_CONFIG` kann einen anderen Pfad wählen:

```
schema = 1
bundle_preset = "OSG lecture/1"
```

Ein explizites `bundle_preset` im Projektmanifest hat Vorrang. Die Nutzerdatei ist deshalb ein Komfortdefault und keine Möglichkeit, die reproduzierbare Projektkonfiguration unbemerkt zu überschreiben.

7.4 L^AT_EX-Konfiguration

Die gemeinsame L^AT_EX-Konfiguration liegt standardmäßig in `Include/projectconfig.tex`. Sie beschreibt die fachliche und gestalterische Semantik des Projekts – Profilwahl und Sprachliste stehen dagegen im Manifest. Ein typischer Kern ist:

```
\LectureProjectSetup{
  languages={
    map={de=ngerman,en=british},
    load babel
  }
}

\OsgLectureSetup{
  class/<slides>={aspectratio=169},
  class/scrbook={twoside,open=right},
  mode/longform={continuation={counters={page,chapter,section}}}}
}

\title{\ldeen{Betriebssysteme}{Operating Systems}}
\author{Nora Example}
```

Die auswählbaren Sprachen kommen für einen OLLM-Build aus dem Manifest; `languages` nennt hier nur noch das `langselect`-Mapping. `OSG Lecture` reicht die Sprachliste (in Manifestreihenfolge) aus der Buildauftragsdatei an `langselect` weiter.

`class/NAME` enthält unveränderte Optionen für die wirkliche Basisklasse und wird vor deren Laden ausgewertet: `class/beamer` adressiert die Basisklasse namentlich, `class/<slides>` den Modus (der *fertige* Modusgraph steht schon vor `\LoadClass` bereit, ein einzelner Modusname wird gegen die aktiven Modi geprüft). `mode/NAME` enthält dagegen Einstellungen für eine `osglecture`-eigene Ausgabeeigenschaft und wird nach Finalisierung des Modusgraphen von allgemein nach spezifisch angewandt.

Frei programmierter Präambelcode gehört vorzugsweise in getrennte Fragmente, die mit `\IncludeOsgLecturePreamble{<name>}` eingebunden werden. Sie dürfen Paketladungen und eigene Makros enthalten und können durch einen Modusausdruck oder `class = {<class>}` eingeschränkt werden. Damit bleibt `projectconfig.tex` eine lesbare Sammlung semantischer Projektentscheidungen.

Metadaten. Felder wie Titel und Autor können modusabhängige Werte besitzen. Eigene Felder werden mit `\DeclareOsgLectureMetadataField` registriert. Die normale Priorität lautet Fallback, Profil, Projekt und schließlich Unit. `\EnforceLectureProjectConfiguration` macht eine Projektvorgabe verbindlich, sodass eine Unit sie nicht überschreiben kann.

```

\author<presentation>[N. Example]{Nora Example}
\author<longform>[Example]{Dr. Nora Example}
\DeclareOsgLectureMetadataField{supervisor}
\supervisor<longform>{Ada Professor}

```

`documentmetadata.tex` hat eine andere Aufgabe: Sein `\DocumentMetadata`-Aufruf muss vor der Dokumentklasse erfolgen und steuert PDF-Eigenschaften, nicht die sichtbaren fachlichen Metadaten. `OLLM` bindet die Datei ein, wenn die aus dem Profil abgeleitete Metadatenpolicy `enabled` ergibt – also bei einem `required`-Profil immer, bei einem `supported`-Profil, wenn `document_metadata = {enabled}` im Manifest steht. Dabei stehen Symbole wie `\OsgLectureRequestedLanguage` zur Verfügung.

7.5 Dokumentenprofile

Ein Dokumentprofil verbindet einen fachlichen Dokumenttyp mit einer technischen Satzumgebung. Es benennt Basisklasse, Klassenanbindung, unterstützte Dokumenttypen, zusätzliche Modi, Klassenoptionen und den `\DocumentMetadata`-Vertrag. Mitgeliefert werden:

beamer für `slides`, `handout` und `screen`; Metadaten vor der Klasse sind verboten (Preset-Default für `presentation_profile`).

ltx-talk für `slides`, `handout` und `screen`; `\DocumentMetadata` ist erforderlich.

book für `script` und `article` auf Basis der Standardklasse `book`.

scrbook für `script` und `article` auf Basis von KOMA-Script; `\DocumentMetadata` ist verboten, weil KOMA-Script es noch nicht unterstützt, es entsteht also kein Tagged PDF (Preset-Default für `longform_profile`).

Das Profil wird im Manifest gewählt (`profile` am Target oder `presentation_profile/longform_profile` in `[targets.defaults]`), nicht in `projectconfig.tex` – denn sein `\DocumentMetadata`-Vertrag muss feststehen, bevor die Klasse überhaupt läuft. Für ein Standalone-Dokument ohne Manifest wählt die Klassenoption `profile = {\langle name \rangle}`.

Ein eigenes Profil liegt üblicherweise als `osglecture-profile-NAME.def` im TeX-Suchpfad. Es kann ein vorhandenes eine vorhandene Klassenanbindung wiederverwenden; eine neue Basisklasse benötigt zusätzlich eine passende Anbindung. Die drei Pflichtangaben sind `backend`, `class` und `doctypes`:

```

\ProvidesExplFile{osglecture-profile-mybook.def}{2026-08-28}{1.0}
  {Project book profile}
\DeclareOsgLectureProfile{mybook}{
  backend=scrbook,
  adapter=scrbook,
  class=scrbook,

```

```
capabilities={longform,print},
doctypes={script,article},
class-options={11pt},
document-metadata=supported,
course-target=title,
event-target=subtitle
}
\DeclareOsgLectureProfileClassOptions{mybook}{script}{open=right}
```

Damit [OLLM](#) den `\DocumentMetadata`-Vertrag schon vor dem Lauf kennt, liegt neben der `.def` eine maschinenlesbare Profilbeschreibung `osglecture-profile-mybook.toml` (bzw. unter `Definitions/profiles/`):

```
schema = 1
kind = "profile"
name = "mybook"
version = "1.0"
profile_class = "longform"
document_metadata = "supported"
doctypes = ["script", "article"]
```

Nach Installation oder Aufnahme des gemeinsamen Verzeichnisses in den TeX-Suchpfad genügt `longform_profile = {mybook}` in `[targets.defaults]`. [OSG Lecture](#) lädt die `.def` bei der ersten Auswahl automatisch und prüft, ob der angeforderte Dokumenttyp unterstützt wird. [ollm/testfiles](#) bzw. `definitions/profiles.t` bewacht, dass `.def` und `.toml` nicht auseinanderlaufen.

7.6 Den Modusgraphen erweitern

Ein eigener Modus ist angebracht, wenn eine neue Ausgabe-eigenschaft inhaltlich abgefragt werden soll. Ein neues Farbschema ist kein Modus; es ändert nur die Darstellung. Ein Studienleitfaden, der zusätzliche Selbstkontrollfragen enthält und zugleich eine druckbare Langform ist, benötigt dagegen sinnvollerweise den Blattmodus `studyguide` mit den Eltern `longform` und `print`.

Die Deklaration gehört in eine Modus-Setup-Datei, die ein Profil vor der Finalisierung des Graphen lädt:

```
% osglecture-mode-studyguide.tex
\DeclareLectureMode[parents={longform,print}]{studyguide}
\AssignLectureModeBehavior{studyguide}{longform}
```

Die Eltern beschreiben fachliche Zugehörigkeit: Eine Abfrage nach `studyguide`, `longform` oder `print` ist für diese Instanz wahr. Die Verhaltensklasse beantwortet eine andere Frage: Welche konkrete Implementierung sollen polymorphe Befehle für diesen Blattmodus verwenden? Durch die Zuordnung zu `longform` kann der neue Typ die vorhandenen Langformimplementierungen wiederverwenden. Nur wenn tatsächlich neues Verhalten benötigt wird, sollte mit `\DeclareModeBehaviorClass` eine neue Verhaltensklasse eingeführt werden; dann müssen alle dafür geltenden Befehlsverträge vollständig implementiert sein.

Das zugehörige Profil verbindet den fachlichen Modus mit einer Satzumgebung. Gegenüber dem vorigen Profilbeispiel kommen der neue Dokumenttyp und die Setup-Datei hinzu:

```
\DeclareOsgLectureProfile{studyguide-scrbook}{
  backend=scrbook,
  adapter=scrbook,
  class=scrbook,
  doctypes={studyguide},
  mode-setup-file=osglecture-mode-studyguide.tex,
  document-metadata=supported
}
```

Der Lebenszyklus ist absichtlich streng: Zuerst werden Modi, Elternkanten und Aliase deklariert; danach aktiviert `OSG Lecture` die aus dem Dokumenttyp und dem Profil stammenden Blattmodi; schließlich finalisiert es den Graphen. Erst dann werden modusabhängige Konfiguration und Inhalte ausgewertet. Nach der Finalisierung sind Deklarationen und Änderungen Fehler. Dadurch sehen alle späteren Abfragen denselben geprüften, zyklensfreien Graphen.

Die `mode-setup-file` eines Profils wird nur für dessen Dokumenttypen gelesen. Nennt gemeinsamer Quelltext einen projekteigenen Modus – etwa `\mode<solutions>` in einer Unit, die auch als script gebaut wird –, muss der Modus in jedem Target bekannt sein, denn ein unbekannter Modus ist ein Fehler. Dafür liest `\LectureProjectSetup{mode-setup-file=<Datei>}` in `projectconfig.tex` eine Datei in jedem Target, vor der des Profils. Aktiv wird der Modus weiterhin nur dort, wo Dokumenttyp oder Profil ihn wählen. Das Beispiel `examples/series-modern` nutzt das für ein Übungsblatt mit Lösungsfassung.

`\DeclareLectureModeParents{<mode>}{<parents>}` ergänzt oder ersetzt die Elternangabe getrennt von der Deklaration; `\DeclareLectureModeAlias{<alias>}{<mode>}` stellt einen alternativen Namen bereit. `\SetLectureModes` und `\AddLectureModes` wählen Blattmodi, `\FinalizeLectureModes` schließt den Graphen. In gewöhnlichen `osglecture`-Projekten werden die letzten drei Schritte von Klasse und Profil ausgeführt; Projektquellen sollten sie nicht vorwegnehmen.

7.7 Konfigurationsreferenz

Klassens Optionen.

standalone manifestfreien Betrieb aktivieren.

doctype = {name} Dokumenttyp wählen; im Standalone-Betrieb standardmäßig slides.

profile = {name} Dokumentprofil ausdrücklich wählen (Standalone; für OLLM-Builds kommt es aus dem Manifest).

lang = {tag} Dokumentsprache wählen.

class-options = {list} Optionen an die Basisklasse weiterreichen.

patch-overlays = {true|false} portable Modi in native Overlaybefehle integrieren.

\LectureProjectSetup{<key-value list>}.

class-options gemeinsame Basisklassensoptionen.

mode-setup-file = {file} Moduldeklarationen, die in jedem Target vor der Moduldatei des Profils gelesen werden.

presentation-profile, longform-profile, identity-profile Profile für Standalone-Dokumente ohne Manifest; für OLLM-Builds ist die Profilwahl abgekündigt und kommt aus dem Manifest.

theme, numbering, references projektweite semantische Policies.

bibliography = {{resources, style, options, series-target}} schaltet die Literaturverwaltung mit biblatex und biber ein (Vorgabestil alphabetic).

languages eingebettete Konfiguration von langselect.

Unter **languages** sind die Schlüssel **targetlang, prefix, auto, trim, map, load babel, load polyglossia** und **unified shorthands** zulässig. **selectable** kommt für einen OLLM-Build aus dem Manifest (ein Wert hier wird mit Warnung verworfen) und ist nur für Standalone gedacht. Wird **targetlang** ausgelassen, setzt OSG Lecture dafür automatisch **\OsgLectureLanguage** ein. Ein expliziter Wert überschreibt diese Konvention.

Weitere Projektbefehle.

\OsgLectureSetup{<areas>}

weist Basisklassensoptionen zu, adressiert nach Klassenname (class/NAME) oder aktivem Modus (class/<MODE>), und mit mode/NAME osglecture-eigene modusabhängige Einstellungen.

\LectureTargetSetup{<target>}{<settings>}

akzeptiert **filecolor**; **profile** wird für OLLM-Builds ignoriert (Profilwahl im Manifest).

\IncludeOsgLecturePreamble*<modes>[<class=list>]{<name>}

registriert ein bedingtes Präambelfragment; die Sternform lädt sofort und erlaubt deshalb keine Bedingungen.

`\LectureProjectEnforce{<settings>}`

setzt Projekt-Policies mit erzwungener Priorität.

`\EnforceLectureProjectConfiguration{<code>}`

führt beliebigen Konfigurationscode mit erzwungener Priorität aus.

`\DeclareOsgLectureMetadataField{<name>}`

deklariert ein Metadatenfeld samt Setter und Abfragebefehlen.

Vordefinierte Metadatenfelder sind title, subtitle, author, institute, date, course und event. Zu jedem Feld existieren ein Setter `\OsgLectureSet{<Field>}`, ein Langwert `\OsgLecture{<Field>}` und ein Kurzwert `\OsgLectureShort{<Field>}`. Die Setter akzeptieren `<mode>[<short value>]{<long value>}`.

Profile definieren. `\DeclareOsgLectureProfile{<name>}{<settings>}` akzeptiert vollständig engine, capabilities, backend, adapter, class, doctypes, class-options, setup-file, modes, mode-setup-file, course-target, event-target und document-metadata. Die letzten drei Wahlwerte sind title|subtitle|none, title|subtitle|none beziehungsweise supported|required|forbidden. Pflichtfelder sind backend, class und doctypes; engine ist derzeit luatex. Typspezifische Klassenoptionen werden mit `\DeclareOsgLectureProfileClassOptions{<profile>}{<doctype>}{<options>}` ergänzt.

Referenzdienst konfigurieren. `\OsgLectureReferencesSetup{<settings>}` akzeptiert legacy = {true|false} und replace. replace kann eine Liste aus ref, pageref, nameref, autoref oder den Sammelwert all enthalten und ersetzt die entsprechenden Standardbefehle durch die serienfähigen Varianten.

8 OLLM

OLLM ist die ausführbare Buildschnittstelle des Bundles. Es liest das Manifest, entdeckt Units, normalisiert Target und Sprache, erzeugt einen isolierten Buildauftrag und lässt ihn von `latexmk` mit `LuaLATEX` ausführen. Dadurch müssen Quelldateien weder Jobnamen zerlegen noch den Projektzustand erraten. Die allgemeine Befehlsform lautet:

```
ollm [globale Optionen] [Aktion] [Target] [Aktionsoptionen]
```

Ohne Aktion wird `build` angenommen. Die mitgelieferten Targets `slides`, `handout`, `screen` und `script` dürfen als nacktes Wort stehen; projektdefinierte Targets werden eindeutig mit `--target = {NAME}` gewählt:

```
ollm slides --language=de
ollm build --target=script --language=en
```

8.1 Bauen: von einer Instanz bis zur Serie

Für die tägliche Arbeit in einer Unit genügt meist ein einzelner Build. Ohne `--language` verwendet OLLM die Projektsprache, sofern das Target sie unterstützt, sonst dessen erste konfigurierte Sprache. `--source` ist standardmäßig `main.tex`:

```
ollm build script --language=en
ollm build --target=slides --source=main.tex
```

Der Scope bestimmt, wie weit die Auswahl reicht. `current` baut eine Dokumentinstanz der aktuellen Unit. `unit` baut deren zulässige Target-/Sprachmatrix. `series` baut eine Dokumentinstanz aller passenden normalen Units, während `collection` die eindeutig passende Integrationsunit des Dokumenttyps baut. Die letzten beiden Scopes funktionieren auch aus der Projektwurzel:

```
ollm build --all --scope=unit
ollm handout --language=de --scope=series
ollm script --language=de --scope=collection
```

`--all` erweitert die Matrix nur innerhalb des gewählten Scopes; es bedeutet nicht automatisch „das ganze Projekt“. `--rebuild` erzwingt einen vollständigen `latexmk`-Neubau. `--dry-run` zeigt die normalisierte Auswahl, ohne `LATEX` zu starten, und eignet sich zusammen mit `--format = {json}` für Werkzeuge und CI.

Serieninterne Referenzen und Integrationen bilden Abhängigkeiten. `--resolve` baut veraltete Produzenten und Konsumenten in mehreren Runden bis zu einem Fixpunkt. Eine logische Unit, die noch nie direkt gebaut wurde, muss zunächst einmal in der benötigten Dokumentinstanz gebaut werden, damit OLLM ihre ID dem Verzeichnis zuordnen kann.

```
ollm build script --language=de --resolve
```

8.2 Prüfen und Diagnostizieren

report, check und doctor beantworten drei verschiedene Fragen. report beschreibt den bekannten Zustand: Units, Dokumentinstanzen, Fassungen, Ausgabedateien und Abhängigkeiten. Befunde ändern seinen Exitstatus nicht, solange der Zustand lesbar ist. check liest dieselben Daten, bewertet Inkonsistenzen aber als Fehler und eignet sich daher für eine Freigabeprüfung. Beide Befehle bauen nichts.

Ein typischer Fehler macht den Unterschied greifbar: Fehlt auf einem neuen Rechner das Paket `tagpax`, hilft doctor, weil die Werkzeugkette unvollständig ist. Wurde dagegen 020-processes nach einem erfolgreichen Build verändert und das Gesamtskript verwendet noch dessen alten Buildstand, meldet check den veralteten Projektzustand. report eignet sich zuvor zur Ursachenforschung, weil es die beteiligten Buildstände und Abhängigkeiten zeigt, ohne den Befund als Freigabefehler zu werten.

```
ollm report --scope=series
ollm check --scope=current --target=script --language=de
ollm check --format=json
```

doctor prüft dagegen die Werkzeugkette: Perl, latexmk, LuaL^AT_EX, kpsewhich, den TOML-Parser und die benötigten TeX-Pakete. In einem Projekt kommen Manifest, Definitionen, Schreibbarkeit, Zustand und Shell-Escape-Policy hinzu. Verwenden Sie doctor, wenn bereits die Umgebung fraglich ist, und check, wenn ein vorhandener Projektzustand widersprüchlich erscheint.

```
ollm doctor
ollm doctor --project-root=/path/to/project --format=json
```

8.3 Aufräumen und Zustände pflegen

OLLM trennt Buildverzeichnisse von den in die Serie übernommenen Dokumentinstanzen. clean --level=aux entfernt Hilfsdateien, behält aber das lokale PDF; build entfernt das isolierte Buildverzeichnis; state entfernt diese Dokumentinstanzen; all kombiniert die letzten beiden. Mit --dry-run sehen Sie zuerst den Löschanplan. Scopes sind hier current, unit und series.

```
ollm clean --level=aux
ollm clean --level=build --scope=unit --dry-run
ollm clean --level=state --scope=series
```

prune beseitigt aufgegebene temporäre Zustände und ersetzt ältere Buildstände. Zustände physisch entfernter Units werden nur mit `--stale-units` einbezogen. Das ist absichtlich vorsichtiger als ein pauschales Löschen von `.osglecture`, bei dem auch die bekannten Zuordnungen logischer Unit-IDs verloren gehen.

```
ollm prune --dry-run
ollm prune --stale-units
```

8.4 Bereitstellen

deploy kopiert bereits erfolgreich gebaute PDFs an die im Manifest konfigurierten Zielorte. Es startet keinen Build und legt keine Zielverzeichnisse an. Verwenden Sie daher zuerst build oder `build --resolve` und danach deploy. In der Projektwurzel gilt standardmäßig Scope series, in einer Unit current.

```
ollm deploy handout --language=de --scope=series
ollm deploy --target=script --scope=collection
ollm deploy --all --overwrite
```

Dateinamenschablonen können unter anderem Serien-ID, Unit-ID, Ordnungskennung, Kapitel, Dokumenttyp, Sprache und Rolle verwenden. Eine vorhandene identische Datei bleibt unverändert. Ob eine abweichende Datei `--overwrite` verlangt, bestimmt `security.overwrite`.

Die bereitgestellten Kopien können mit eingeschränkten PDF-Rechten versehen werden, etwa ohne Druckerlaubnis. Die Einschränkungen stehen in `[deployment.permissions]` als Default für alle Dokumenttypen oder in `[deployment.types.<doctype>.permissions]`; eine eigene Tabelle eines Dokumenttyps ersetzt den Default vollständig, `permissions = {}` hebt ihn also auf. Nicht genannte Rechte bleiben erlaubt.

```
[deployment.permissions]
print = "none"      # full | low | none
extract = false     # copy text and graphics
modify = false      # change page content
annotate = false    # add comments
form = false        # fill in forms and sign
assemble = false    # insert, delete, rotate pages

[deployment.types.handout]
permissions = {}    # handouts are meant to be printed
```

`--permissions` ersetzt für einen Lauf die konfigurierten Rechte aller Dokumenttypen, etwa `--permissions = {print=none,extract=false}`; `--permissions = {all}` deployt ohne Einschränkungen.

Für eingeschränkte Kopien benötigt OLLM das externe Programm `qpdf`; `ollm doctor` meldet, ob es gefunden wird. Die Kopie wird mit AES-256 verschlüsselt und öffnet ohne Kennwort; das Eigentümerkennwort ist zufällig und wird nicht gespeichert, sofern Sie nicht mit `--owner-password = {KENNWORT}` selbst eines angeben. Das gebaute Original bleibt unverändert. Solche Einschränkungen lassen sich mit üblichen Werkzeugen leicht entfernen; sie sind ein Hinweis an die Lesenden, keine Zugriffskontrolle.

Fehlt `qpdf`, fragt `deploy` im Terminal, ob die PDFs ohne Einschränkungen kopiert werden sollen. Mit `--non-interactive` oder ohne Terminal, etwa in CI, bricht es dagegen mit Exitcode 69 ab.

8.5 Projekt anlegen und allgemeine Optionen

`ollm newproject` verlangt die Wahl der Profile mit `--profiles = {taggable|classic}` und erzeugt ein neues `ollmconfig.toml` sowie, falls noch nicht vorhanden, `Include/projectconfig.tex` mit Dummy-Metadaten. `taggable` stellt `ltx-talk` und `book` mit eingeschaltetem Tagging ein und legt `Include/documentmetadata.tex` an; `classic` stellt `beamer` und `scrbook` ohne Tagged PDF ein. Fehlt die Option, erklärt der Befehl beide Varianten und schreibt nichts. Einzelne Targets lassen sich danach im Manifest von Hand mischen. Der Befehl überschreibt keine bestehende Datei. Für die genaue Syntax einer installierten Version sind `ollm --help` und `ollm --version` maßgeblich. Häufige globale Optionen sind `--config = {DATEI}`, `--project-root = {VERZEICHNIS}`, `--format = {text|json}`, `--color = {auto|always|never}` und `--non-interactive`.

Für Builddiagnosen wählen `--debug = {tex}`, `ollm` oder `tex+ollm` die Ausgabebeschicht; `--warnings = {all}`, `important` oder `none` steuert die Warnungsmenge. Unbekannte kompatible Minusoptionen werden beim Build an `latexmk` weitergereicht; Optionen, die den kontrollierten Buildvertrag umgehen würden, weist OLLM zurück.

8.6 Vollständige Kommandoübersicht

build baut eine oder mehrere Instanzen einer Serie; dies ist die Standardaktion.

standalone baut eine manifestfreie Quelldatei anhand ihrer Klassenoptionen.

report beschreibt Units, Instanzen, Buildstände, Ausgabedateien und Abhängigkeiten, ohne sie zu bewerten.

check prüft denselben Zustand und liefert bei Inkonsistenzen einen Fehlerstatus.

clean entfernt Daten gemäß `--level` und `--scope`.

prune entfernt ersetzte ältere Buildstände und optional den Zustand fehlender Units.

doctor prüft Werkzeugkette und, innerhalb eines Projekts, dessen grundlegende Konfiguration.

deploy kopiert bereits gebaute PDFs nach den Deploymentregeln.

convertproject konvertiert ein altes `ollmconfig.pl` nach TOML und übernimmt erkennbare Metadaten aus `Include/lectdates.tex`.

newproject legt Manifest und TeX-Projektkonfiguration an und verwendet vorhandene Legacy-Dateien als Konvertierungsquelle.

Targets und Aliase. Die eingebauten nackten Targetnamen sind `slides` mit den Aliasen `beamer` und `presentation`, `handout`, `screen` sowie `script` mit dem Alias `article`. Andere registrierte Targets werden mit `--target = {TARGET}` gewählt.

Alle OLLM-Optionen.

`--help, -h` Hilfe ausgeben.

`--version` Version ausgeben.

`--format = {text|json}` menschen- oder maschinenlesbare Ausgabe.

`--color = {auto|always|never}` Farbausgabe steuern.

`--non-interactive` interaktive Aktionen verbieten.

`--profiles = {taggable|classic}` Profilsatz des erzeugten Manifests; Pflicht für `newproject`, optional für `convertproject` (dort Vorgabe `classic`).

`+enforce+`, `--enforce+` das Präfix `+` für Aktions- und Targetwörter verlangen.

`--legacy` ausdrücklich die alte `ollmconfig.pl` verwenden.

`--overwrite` Ersetzen erlauben, sofern die Projektpolicy dies zulässt.

`--target = {TARGET}` Target wählen.

`--language = {LANG}` Sprache wählen; kompatibel sind auch `lang=LANG` und `+lang=LANG`.

`--source = {FILE}` Hauptquelle wählen.

`--all` alle zulässigen Target-/Sprachkombinationen im Scope wählen.

`--resolve` Abhängigkeiten bis zum Fixpunkt neu bauen.

`--rebuild` vollständigen `latexmk`-Neubau erzwingen.

`--dry-run` normalisierten Plan anzeigen, ohne ihn auszuführen.

`--level = {aux|build|state|all}` Ebene für `clean` wählen.

`--scope = {current|unit|series|collection}` Reichweite für Build, Inspektion, Pflege oder Deployment wählen; nicht jede Aktion erlaubt jeden Wert.

--stale-units bei prune auch physisch fehlende Units berücksichtigen.

--debug = {tex|ollm|tex+ollm} Diagnoseebene wählen.

--warnings = {all|important|none} Warnungsmenge wählen.

--config = {FILE} Manifest ausdrücklich wählen.

--project-root = {DIR} Projektwurzel ausdrücklich wählen.

Unbekannte kompatible Minusoptionen werden nur bei Builds an `latexmk` weitergegeben. Optionen, die Engine, Ausgabeformat oder kontrollierte Sicherheitsparameter umgehen würden, werden abgewiesen. Für `standalone` sind außer **--config** und **--project-root** keine projektbezogenen Auswahloptionen zulässig.

Index

Symbols

+enforce+ 56
 --all 18, 52, 56
 --config 57
 --dry-run 52 f., 56
 --enforce+ 56
 --help 56
 --language 15, 29, 52
 --legacy 56
 --level 55
 --non-interactive 55 f.
 --overwrite 16, 54, 56
 --permissions 55
 --project-root 57
 --rebuild 52, 56
 --resolve 17, 37 ff., 52, 56
 --scope 55
 --source 52
 --stale-units 54, 57
 --version 56
 -h 56

A

\ActiveLectureModes 30, 35
 adapter 51
 \AddLectureModes 49
 \anditem 35
 ansiterm (Paket) 5
 article (Klasse) 6
 \author 25 f., 34
 auto 50
 \autoref 36

B

babel (Paket) 11, 28
 backend 47, 51
 Beamer (Bundle) 7
 beamer (Klasse) 6, 12, 19, 22–25, 27, 29, 47, 55
 beamer (Paket) 4
 beamerswitch (Paket) 4
 biblatex (Paket) 33, 36, 50
 bibliography 50
 book (Klasse) 10, 19, 23 ff., 47, 55
 bundle_preset 45
 \butitem 35

C

capabilities 51
 \cite 33, 36
 class 46 f., 51
 class-options 41, 50 f.

class/<slides> 46
 class/beamer 46
 class/NAME 46
 \colonitem 31, 35
 \columnsep 32
 \commaitem 31, 35
 comment (Paket) 4
 \course 26, 30, 34
 course-target 26, 51
 CTAN 4 f.

D

\date 26, 34
 \DeclareLectureModeAlias 49
 \DeclareLectureModeParents 49
 \DeclareModeBehaviorClass 49
 \DeclareOsgLectureMetadataField 46, 51
 \DeclareOsgLectureProfile 51
 \DeclareOsgLectureProfileClassOptions 51
 default_language 44
 doctype 41, 44, 50
 doctypes 44, 47, 51
 document-metadata 44, 51
 document_metadata 44, 47
 \documentclass 11, 24, 42
 \DocumentMetadata 11, 24, 39, 42 f., 47 f.

E

\EnforceLectureProjectConfiguration 46, 51
 engine 51
 \event 26, 34
 event-target 26, 51

F

filecolor 50
 \FinalizeLectureModes 49
 \finite 31, 35
 frame (Umgebung) 12, 14, 27
 \frametitle 12, 27

H

hyperref (Paket) 32

I

identity-profile 50
 \IfLectureModeF 35
 \IfLectureModeT 34
 \IfLectureModeTF 27, 34
 \include 38
 \IncludeOsgLecturePreamble 46, 50
 \includeunit 17 f., 39 f., 42

INDEX

`initial` 35
`\institute` 26, 34
`\item` 22, 35

L

`lang` 32, 36, 41, 50
`langselect` (Paket) 4, 11, 28, 31, 44, 46, 50
`language` 10, 35
`languages` 10 f., 44, 50
`\ldeen` 12, 28
`\lecture` 12, 26 f., 32, 34
`\lecturemode` 22, 27, 29 ff., 34
`\LectureModes` 30, 35
`\LectureProjectEnforce` 51
`\LectureProjectSetup` 49 f.
`\LectureTargetSetup` 24, 50
`legacy` 51
`\litcomment` 34, 36
`\litentry` 36
`literaturelist` (Umgebung) 36
`\litgroup` 36
`load babel` 50
`load polyglossia` 50
`\LoadClass` 46
`\longform` 35
`longform-profile` 24, 50
`longform_profile` 44, 47 f.
`\longslides` 35
`lttheme-tuc-2019` (Paket) 5
`ltx-talk` (Klasse) .. 4 ff., 10, 19, 22–25, 27, 29, 42, 47, 55
`ltx-talk` (Paket) 4 f.
`ltxtalk-theme` (Paket) 5

M

`map` 44, 50
`minted` (Paket) 5
`mode-setup-file` 49 ff.
`mode/NAME` 46
`modes` 51
`multilang` (Paket) 4

N

`\nameref` 36
`newpax` (Paket) 39
`\nextcolumn` 31, 36
`\Note` 12 f., 24, 27
`numbering` 50

O

`\olautoref` 17, 36
`\olnameref` 36
`\olpageref` 32, 36
`\olref` 32, 36
`options` 33

`\osgannotation` 28
`\OsgLecture` 34, 51
`osglecture` (Klasse) 4 ff., 19, 21, 23 f., 49
`osglecture-modes` (Paket) 4 f., 22
`\OsgLectureLanguage` 28, 50
`\OsgLectureReferencesSetup` 51
`\OsgLectureRequestedLanguage` 47
`\OsgLectureSet` 34, 51
`\OsgLectureSetup` 50
`\OsgLectureShort` 34, 51
`osglisings` (Paket) 5
`osgstyler` (Paket) 5
`overwrite` 44

P

`\pageref` 32, 36
`patch-overlays` 50
`patch-overlays=false` 27
`patch-overlays=true` 27
`pdfpages` (Paket) 39
`prefix` 50
`\prescase` 35
`presentation-profile` 24, 50
`presentation_profile` 44, 47
`\presitem` 31, 35
`presitemize` (Umgebung) 35
`presitemize` (Paket) 14, 30 f.
`\preskeepcase` 36
`\prespar` 35
`\printbibliography` 33, 36
`\printseriesbibliography` 33, 36
`profile` 24, 41, 44, 47, 50
`profile_class` 44
`project.id` 44
`punctuation` 35

R

`\ref` 17, 32, 36
`references` 50
`relation` 35
`replace` 51

S

`schema` 44
`scarticle` (Klasse) 6
`scribook` (Klasse) 14, 23 ff., 41, 47, 55
`\section` 37
`security.overwrite` 54
`selectable` 28, 50
`semcat` (Paket) 5
`series-target` 33
`\SetLectureModes` 49
`setup-file` 51
`shell_escape` 44

INDEX

<code>\slides</code>	35	<code>\thereforeitem</code>	35
<code>\sourceref</code>	27	<code>TikZ</code> (Bundle)	7
<code>standalone</code>	41 f., 50	<code>\title</code>	25 f., 34
<code>style</code>	33	<code>trim</code>	50
<code>\subtitle</code>	26, 34	<code>twocolumns</code> (Umgebung).....	31, 36
T		<code>type</code>	32, 36
<code>tagbridge</code> (Paket)	5	U	
<code>tagdirtree</code> (Paket).....	6	<code>unified shorthands</code>	50
<code>tagpax</code> (Paket)	4, 39, 53	<code>unit_scopes</code>	44
TANTAU, Till.....	7	W	
<code>targetlang</code>	28, 50	<code>whitespace</code>	35
<code>tex_config</code>	44	WRIGHT, Joseph	3
<code>tex_directory</code>	44		
<code>theme</code>	50		