

# Die Plug-In Technologie des Apache Web-Severs

## Das Modul mod\_so

Jan Kechel, Sebastian Kuhle  
Seminar Systemmodellierung 2003  
Hasso-Plattner-Institut für Softwaresystemtechnik

### Abriss

Es gibt verschiedene Techniken um einem bestehenden System Funktionalität hinzuzufügen. Hier beschäftigen wir uns mit der vom Apache Webserver verwendeten Technik und erklären diese im Detail.

**Schlüsselwörter:** Apache, Plug-In, Dynamic Loading, FMC, mod\_so

## 1. Einführung

Ein programmiertes und laufendes System hat im allgemeinen das Problem, dass man zum Zeit-punkt der Erstellung nicht alle in der Zukunft möglicherweise auftretenden Probleme erfassen und Lösungen bereitstellen kann. Somit kann es zur Laufzeit des betrachteten Systems zu Aufgabenstellungen kommen, die es notwendig machen, den Funktionsumfang des Systems zu erweitern.

## 2. Plug-In Technik

### 1.1. Allgemein

Als Plug-In bezeichnet man ein Stück Software, welches ein Kernsystem um Funktionalität er-weitert.

Zunächst betrachten wir die allgemeinste Darstellung, die man sich vorstellen kann, wenn man von Plug-Ins redet. In Abbildung 1 sieht man einen Kernsystemakteur und einen Speicher der dem Kernsystem zugehörig ist. Ausserdem findet man mehrere Plug-Ins. Alle Plug-Ins können auf den dem Kernsystem zugeordneten Speicher zugreifen. Zwischen dem Kernsystemakteur und den Plug-Ins gibt es Kanäle, über welche der Kernsystemakteur Funktionalität eines Plug-Ins aufrufen und somit benutzen kann.

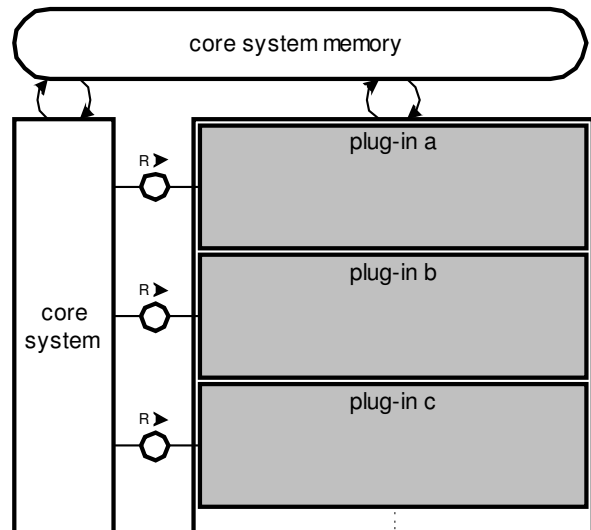
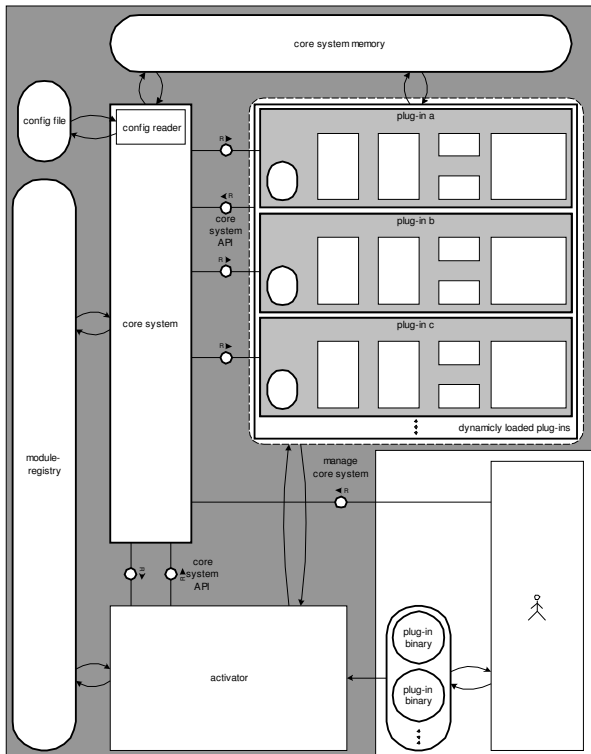


Abbildung 1: Aufbau eines durch Plug-Ins erweiterten Systems

Abbildung 2 stellt eine Erweiterung des in Abbildung 1 gezeigten Aufbaus dar. Es sind verschiedene Akteure, Speicher und Kanäle hinzugekommen, die im folgenden kurz erklärt werden.

Der Mensch steuert das System, indem er über das Kernsystem verschiedene Management-funktionalität nutzt. Ausserdem hat er Zugriff auf einen Speicher, in welchem Plug-In Binaries liegen.

Bei den Plug-In Binaries handelt es sich um in irgendeiner Form ausführbaren Programmcode. Auf die Plug-In Binaries hat der Aktivierer lesenden Zugriff. Der Aktivierer kapselt Funktionalität, um die Plug-In Binaries für das Kernsystem nutzbar zu machen. Dafür hat der Aktivierer ein Aktionsfeld, auf welchem dann die Plug-In Binaries aktiviert werden. Der Aktivierer hat Verbindungen zum Kernsystem, um Funktionalität beiderseitig nutzen zu können.



**Abbildung 2: Detailliertes Aufbaubild eines um Plug-Ins erweiterten Systems**

Bei den Aufrufen, die Plug-In-Akteure an das Kernsystem richten, handelt es sich um API (Application Programmiers Interface) Aufrufe. Diese API stellt einen Teil der Schnittstellendefinition dar.

Im Vergleich zum Einstiegsbild wurde ein Speicher mit Konfigurationsinformation mit dazugehörigem Auswertungsakteur hinzugefügt. Hier findet man beispielsweise Informationen, welche Plug-Ins geladen werden sollen.

Anstelle der hier gezeigten zentralen Konfiguration kann man sich auch eine dezentrale Konfiguration vorstellen. Dort würden dann die einzelnen Module selbst die für ihre Einbindung notwendigen Schritte ausführen.

Die letzte Änderungen gegenüber dem Einstiegsbild sind die die Modulregistrierungsdaten. Hierbei handelt es sich um Speicher, der zur Verwaltung von Plug-Ins und Plug-In-Funktionalität dient. Hier findet beispielsweise das Kernsystem Datenstrukturen, deren Informationen angeben, ob ein Plug-In geladen ist und welche Funktionalität es von diesem Plug-In erwarten kann.

Denkt man über das Thema von Plug-Ins

nach, so gibt es mehrere Diskussionspunkte, über die nachzudenken ist, wenn man Plug-Ins einsetzen möchte. Im folgenden werden Problemfragen gestellt und eine mögliche Lösung erläutert.

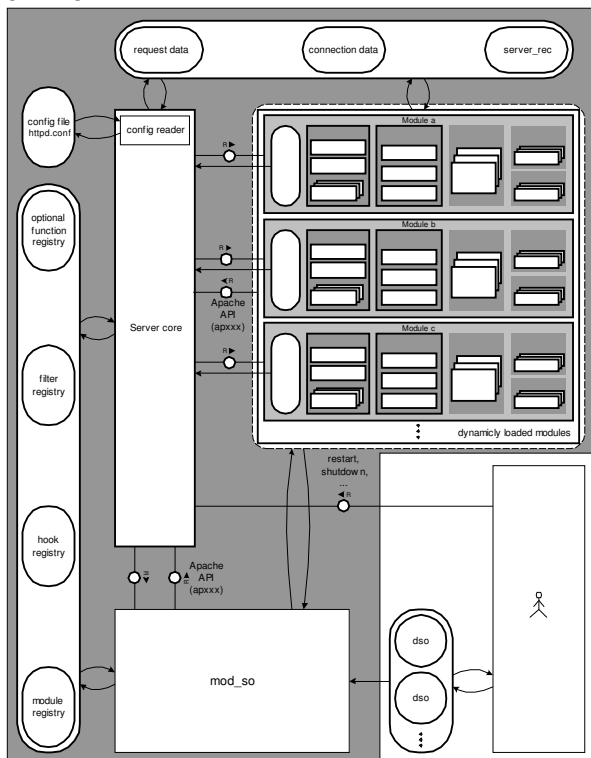
- Wie kann das Kernsystem mit dem Plug-In kommunizieren?  
Will ein Entwickler Module, die dynamisch geladen werden sollen schreiben, so muss er sich bei der Struktur des Plug-Ins und bei den Funktionen daran halten, dass es Funktionen und Datenstrukturen gibt, die pflichtmäßig vorgeschrieben sind. Über diese Daten-strukturen und Funktionen kann das Kern-system das Modul nutzen.
- Wie kann das Plug-In mit dem Kernsystem kommunizieren?  
Bei häufig auftretenden Aufgabestellungen, die ein System bearbeiten soll wird Funktionalität in einer API zusammengefasst. Mittels dieser API kann nun auch das Plug-In Funktionalität des Kernsystems nutzen.
- Wie wird bekannt gemacht, welche Funktionalität das Plug-In dem Kernsystem zur Verfügung stellt?  
Das Kernsystem muss wissen, was vom Modul zu erwarten ist. Diese Information kann mittels einer festgelegten Datenstruktur, die am Ende des Modulcodes liegt bereitgestellt werden. In dieser Struktur stellt der Modulprogrammierer Informationen für das Kernsystem zusammen und das Kernsystem und mod\_so lesen diese Informationen beim Laden des Moduls.
- Wie erhält das Plug-In Zugriff auf die Daten des Kernsystems?  
Bei datenverarbeitenden Systemen im allgemeinen werden bei den Funktionalitätserweiterungen mittels Plug-Ins auch solche hinzugefügt, die auf eben diesen Daten operieren sollen. Auf Daten des Kernsystem kann beispielsweise mit API-Funktionen zugegriffen werden.
- Wo im Ablauf des Kernsystems soll die Modul- Funktionalität benutzt werden?  
Die Lösung im Fall des Apache Webservers ist das Hook-Konzept. Dieses Konzept soll im folgenden als eine mögliche Lösung erklärt werden. Hooks sind Punkte im Ablauf des Apache Webservers, die schon beim Erstellen festgelegt wurden. Ein Modul kann sich beim Laden durch

mod\_so bei einem oder mehreren Hooks registrieren und wenn der Apache im Ablauf an diesem Hook „vorbeikommt“, dann wird Funktionalität dieses registrierten Moduls aufgerufen. Zur Vertiefung verweise ich auf einen Vortrag der gleichen Seminarveranstaltung. Der Vortrag hat den Titel „Apache for Module developers“.

Als Konsequenz der betrachteten Fragen lässt sich feststellen, dass bei Plug-In Techniken allgemein eine klare Schnittstelle definiert sein muss.

## 1.2. Die Plug-In Technik des Apache Webservers

Am Anfang der Betrachtung des Sachverhaltes in Bezug auf den Apache Webserver steht ein Aufbau bild. Dieses Aufbau bild (Abbildung 3) ist der Struktur nach dem Aufbau bild eines allgemeinen durch Plug-Ins erweiterten System sehr ähnlich.



**Abbildung 3: Aufbau bild des um Plug-Ins erweiterten Apache Webserver**

Im oberen Teil des Bildes kann man die verschiedenen Speicher erkennen, die der Apache Webserver benutzt. Bei der Konfigurationsdatei handelt es sich um die

Datei httpd.conf, in der die zu ladenden Module aufgelistet sind.

Der Akteur, der zuvor als Aktivierer bezeichnet wurde heißt im Kontext des Apache mod\_so. Dabei handelt es sich um eine Modul, welches beim Kompilieren des Apache mit in das Apache Binary hineinkompiliert werden muss. Dann ist es möglich, Plug-Ins dem Apache hinzuzufügen.

Mod\_so hat lesenden Zugriff auf die beispielsweise auf der Festplatte liegenden DSO's. DSO steht für dynamic shared object. Der DSO Mechanismus stellt eine Art dar, die Plug-In Idee umzusetzen. In diesem Fall lädt ein Aktivierer eine kompilierte Datei in den Adressbereich der um Funktionalität zu erweiternden Software.

Deshalb unterscheidet man auf der einen Seite die noch nicht geladenen shared objects und die geladenen shared objects. Im Aufbau bild findet man die noch nicht geladene shared objects unten in dem Speicher, auf den mod\_so lesenden Zugriff hat. Die geladenen shared objects, die mittels eines Abwicklers zu Akteuren geworden sind befinden sich zentral im oberen Teil.

Der Modulregistrierungsspeicher wurde verfeinert, wie im Bild links zu sehen ist. Die Benutzung und die Details werden später erläutert.

## 1.3. Statisch oder dynamisch?

Der Apache Webserver lässt es zu, dass man Module statisch oder dynamisch hinzufügen kann. Statisch bedeutet, dass der Modulcode zusammen mit dem Kernsystemcode kompiliert wird, sozusagen in ein Binary. Das Modul muss also schon beim Kompilieren des Apache vorliegen.

Dynamische Module sind unabhängig vom Zeitpunkt des Kompilierens des Kernsystems. Der Apache benutzt auf der Plattform Linux den DSO Mechanismus. Das verpflichtend eingebundene Modul mod\_so stellt die Funktionalität zur Verfügung, die notwendig ist für das dynamische Laden von Modulen. Das Einbinden eines dynamischen Moduls erfordert nicht das Neukompilieren des Apache.

Benutzt man dynamische Module, so ist man zur Ausführungszeit etwa fünf Prozent langsamer als im statischen Fall. Beim Starten sind es 20%.

Zusammenfassend kann man sagen, dass

beim Benutzen von dynamisch ladbaren Modulen Flexibilität der Vorteil ist. Denn bei einem System, wie es der Apache Webserver ist, liegt es auf der Hand, neue Funktionalität ohne grosse Anstrengungen dem System hinzuzufügen zu wollen.

## 3. Apache DSO-Unterstützung

### 1.4. Apache Konfiguration für DSO-Unterstützung

Der Apache Quellcode wird vor der Kompilation mit dem Skript `configure` vorkonfiguriert. `configure` überprüft u.a. bei der Ausführung die vorhandenen Bibliotheken und lokalen Systemvoraussetzungen um eine erfolgreiche Kompilation auf möglichst vielen unterschiedlichen Plattformen gewährleisten zu können. Zusätzlich können an `configure` sehr viele Commandozeilen-Optionen übergeben werden um den Apache den eigenen Wünschen anzupassen. Uns interessiert dabei die Option `--enable-so` bzw. `--disable-so`.

Dabei ist `--enable-so` der Default-Wert und braucht nicht angegeben zu werden. Mit

`--disable-so` wird die DSO-Unterstützung des Apache ausgeschaltet und somit `mod_so` nicht mit in den Server kompiliert.

`configure` erstellt bei jedem Aufruf unter anderem die Datei `modules.c` in der sich 2 Arrays befinden:

```
module *ap_prelinked_modules[] =
{&core_module, ..., &so_module, NULL};
```

und

```
module *ap_preloaded_modules[] =
{ &core_module, ..., &so_module, NULL};
```

Diese beiden Arrays enthalten wenn DSO aktiviert wurde einen Verweis auf `so_module`. Dieser Verweis vollführt die statische Einbindung des Moduls `mod_so` in den Apache bei der Kompilation.

Eine der Aufgaben von `mod_so` ist es diesen Verweis auf die Modul-Struktur, für dynamisch zu ladende Module, zur Laufzeit zu Erstellen.

## 1.5. Das Modul `mod_so`

Das Modul `mod_so` registriert sich durch seine Modul-Struktur `so_module` für die beiden Slots `server-config` und `command-table`.

Der Funktionszeiger `so_sconf_create` in dem Slot für `server-config` zeigt auf eine Funktion die nur Speicher für die Konfiguration anlegt. Viel interessanter ist der Zeiger `so_cmds` in dem Slot `command-table`, der auf ein Array von `command_rec` Strukturen verweist:

Die `command_rec` Struktur gibt Modulen die Möglichkeit weitere Konfigurations-Direktiven des Apache, die in der `httpd.conf` Datei angegeben werden, einzuführen. Die `httpd.conf` wird von Apache nach der Kompilation bei jedem Neustart des Programms eingelesen und ermöglicht so die endgültige Konfiguration des Apache zur Laufzeit.

`Mod_so` registriert mittels dieser Struktur zwei neue Direktiven: `LoadModule` und `LoadFile`.

`LoadModule` wird mit der Funktion `load_module` verknüpft welche bei einem entsprechenden Eintrag in der `httpd.conf`-Datei aufgerufen wird. Mit der Direktive `LoadModule` kann nun ein Apache-Modul in den Speicher geladen und aktiviert werden.

`LoadModule` erwartet 2 Parameter, den Pfad zu dem Modul, sowie den Namen der Modul-Struktur des Moduls.

Die Direktive `LoadFile` lädt eine Shared-Library nur in den Speicher. Der Inhalt der Bibliothek wird nicht weiter von `mod_so` beachtet. Dies wird z.B. dafür benötigt falls ein Modul geladen werden soll, welches selber eine weitere externe Bibliothek verwendet, die nicht in den Apache gelinkt wurde.

## 1.6. Das Dynamische Laden eines Moduls am Beispiel

Wir nehmen folgenden Eintrag in der `httpd.conf` Datei an mit der das Modul `mod_info` dynamisch geladen werden soll:

```
LoadModule info_module mod_info.so
```

Das Info-Modul erweitert den Apache Server um die Fähigkeit Informationen über diesen und die geladenen Module auszugeben.

### 3.1.1. Einlesen der Konfig-Datei httpd.conf

Der Apache-Server liest die Konfigurations-Datei httpd.conf beim Starten sowie bei jedem Neustart des Apache-Servers neu ein (siehe Abbildung 4).

Dabei geht er wie in Abbildung 5 gezeigt wird Zeile für Zeile durch die Datei und ruft die entsprechenden Funktionen auf die für die jeweiligen Direktiven registriert wurden.

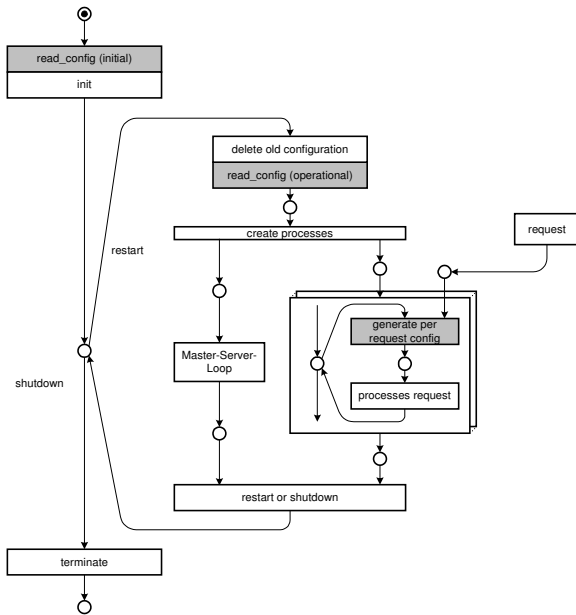


Abbildung 4: Hier liest der Apache die Konfigurations-Datei.

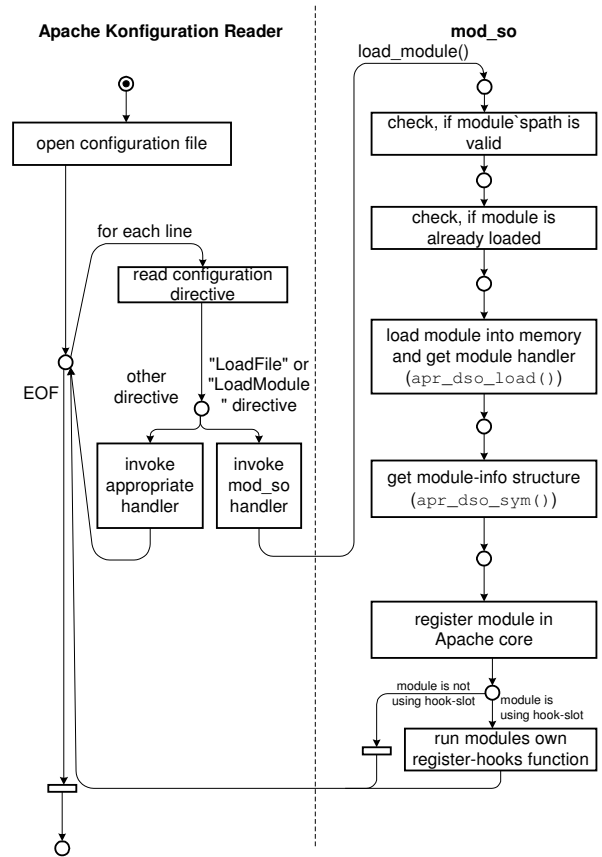


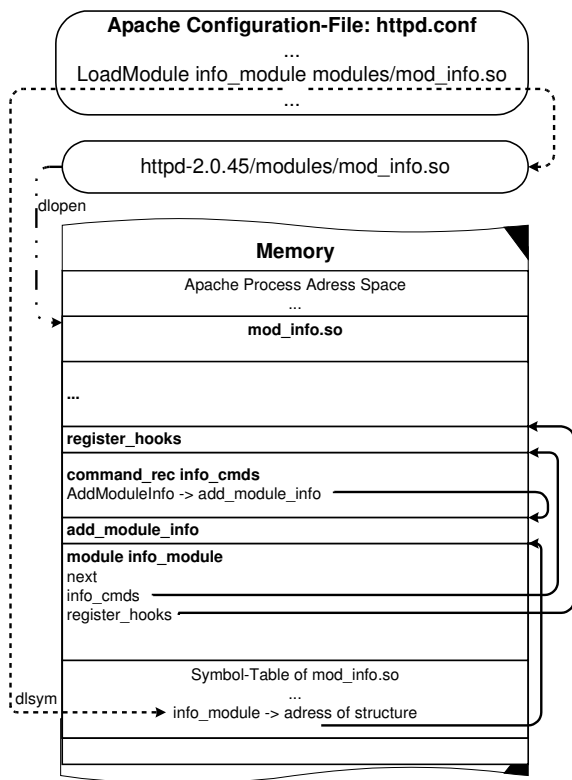
Abbildung 5: Apache Konfiguration und dynamisches laden eines Moduls.

### 3.1.2. Ausführung von load\_module

Für unser Beispiel `info_module` nehmen wir an der Apache Konfigurations-Prozess ist gerade bei der Zeile mit dem Eintrag `LoadModule` für `mod_info.so`.

Daraufhin wird die Funktion `load_module` aufgerufen. Diese überprüft zuerst ob der angegebene Pfad gültig ist und ob nicht schon ein Module mit dem gleichen Pfad und Dateinamen geladen wurde.

Wenn beide Überprüfungen erfolgreich waren wird das Modul mit der Funktion `apr_dso_sym` geladen. Diese Funktion ist Betriebssystem-spezifisch und wird unter Linux auf die Betriebssystem-Funktion `dlopen` übersetzt. `dlopen` veranlasst das Betriebssystem diese Datei in den Prozessadressraum des aufrufenden Prozesses zu laden und einen Zeiger auf die vom Betriebssystem verwendete Anfangs-Adresse des geladenen Moduls zurückzugeben. Dieser Vorgang wird in der Abbildung 6 durch den mit `dlopen` beschrifteten Pfeil dargestellt.



**Abbildung 6: Darstellung der beim laden eines Moduls beteiligten Verweise auf Dateien, Strukturen und Funktionen.**

Als nächstes, da es sich um ein Apache-Modul handeln soll, wird mit `apr_dso_sym` nach dem in der Konfig-Datei angegebenen Symbolnamen gesucht. Diese Funktion ist ebenfalls Betriebs-systemspezifisch und wird und Linux auf die Betriebssystem-Funktion `dlsym` übersetzt.

`Dlsym` sucht in der Symbol-Tabelle, die jeder geladenen dynamischen Bibliothek anhängt, nach eben diesem Bezeichner "info\_module" und liefert den entsprechenden Zeiger darauf zurück. Dieser Vorgang wird mit dem mit `dlsym` beschrifteten Pfeil dargestellt.

Da die Modul-Struktur eine fest definierte Form hat, kann der Apache ab nun ausgehend von der Anfangs-Adresse auf `info_module` auf den gesamten Inhalt des Moduls "seiner Bestimmung entsprechend" zugreifen. `Mod_so` führt nun noch verschiedene Registrierungen des dynamisch geladenen Moduls durch um eine möglichst einfache und schnelle Verwendung und Einbindung des Moduls in den Apache zu gewährleisten. Diese Registrierungen werden von der Funktion `ap_add_module`

durchgeführt. Vorher wird noch die Speicherstelle des neu geladenen Moduls überprüft, die in der Modul-Struktur mit `MODULE_MAGIC_COOKIE` bezeichnet wird überprüft. Dort muss die Zeichenfolge "AP20" stehen. Dies wird sozusagen als kleine Sicherheitsüberprüfung durchgeführt damit nicht eine überhaupt nicht für den Apache geeignete Datei geladen wird.

`Ap_add_module` überprüft als erstes die Apache-Version, für die das Modul programmiert wurde, anhand der Module-Magic-Number-Major, die hier 2 ist. Daraufhin schreibt der Apache direkt in den neu dazugekommenen Adressraum des Moduls und setzt dort einen Zeiger "next" auf das zuletzt geladene Modul. Danach wird das aktuelle Modul als das zuletzt geladene vermerkt.

Nachdem noch die Zähler der insgesamt geladenen Module, sowie der Zähler der insgesamt dynamisch geladenen Module inkrementiert wurde wird zum ersten mal tatsächlich Code aus dem Modul selber ausgeführt, sofern dieses eine Funktion für den Slot `register_hooks` angegeben hat.

Obiges Info-Modul registriert hooks, und somit wird in der Funktion `ap_register_hooks` die Funktion `register_hooks` aus `mod_info.so` aufgerufen. Diese Funktion verwendet selber Aufrufe aus der Apache-API um die eigenen Hooks zu registrieren.

Somit wurde das Modul vollständig geladen und steht nun dem Server in seiner ganzen Funktionalität zur Verfügung.

## Literaturverzeichnis

[AP] The Apache Project: <http://httpd.apache.org>

[APS] Apache Source Code

[AMP] The Apache Modelling Project:  
[apache.hpi.uni-potsdam.de](http://apache.hpi.uni-potsdam.de)

[IECL] [www.iecc.com/linker/linker10.html](http://www.iecc.com/linker/linker10.html)

[UMP] Unix man pages