[Zusammenfassung 2](#_Toc518330183)

[Einleitung 2](#_Toc518330184)

[Architekturmuster und -prinzipien 2](#_Toc518330185)

[MVC 2](#_Toc518330186)

[Model 2](#_Toc518330187)

[View 3](#_Toc518330188)

[Controller 3](#_Toc518330189)

[SOLID 3](#_Toc518330190)

[Single-Responsibility-Prinzip 3](#_Toc518330191)

[Open-Closed-Prinzip 3](#_Toc518330192)

[Liskovsches Substitutionsprinzip 4](#_Toc518330193)

[Interface Segregation 4](#_Toc518330194)

[Dependency-Inversion-Prinzip 4](#_Toc518330195)

[Eine SOLID View 4](#_Toc518330196)

[Single-Responsibility-Prinzip im Viewmodul 4](#_Toc518330197)

[Open-Closed-Prinzip im Viewmodul 5](#_Toc518330198)

[Liskovsches Substitutionsprinzip im Viewmodul 5](#_Toc518330199)

[Interface-Segregation-Prinzip im Viewmodul 6](#_Toc518330200)

[Dependency-Inversion-Prinzip im Viewmodul 6](#_Toc518330201)

[Ein SOLID Controller 6](#_Toc518330202)

[Single-Responsibility-Prinzip im Controllermodul 6](#_Toc518330203)

[Open-Closed-Prinzip im Controllermodul 6](#_Toc518330204)

[Liskovsches Substitutionsprinzip im Controllermodul 7](#_Toc518330205)

[Interface Segregation im Controllermodul 7](#_Toc518330206)

[Dependency Inversion im Controllermodul 7](#_Toc518330207)

[Ein SOLID Model 7](#_Toc518330208)

[Single Responsibility im Model 7](#_Toc518330209)

[Open-Closed-Prinzip im Model 8](#_Toc518330210)

[Liskovsches Substitutionsprinzip 8](#_Toc518330211)

[Interface Segregation 8](#_Toc518330212)

[Dependency Inversion 8](#_Toc518330213)

[Fazit 8](#_Toc518330214)

# Zusammenfassung

Dieses Dokument befasst sich mit dem Model-View-Controller (MVC) Architekturmuster. Es befasst sich weiterhin mit den fünf SOLID-Architekturprinzipien Single Responsibility, Open-Closed, Liskovsches Substitionsprinzip, Interface Segregation und Dependency Inversion. Das Architekturmuster wird auf die Architekturprinzipien hin überprüft und angepasst. In Beispielimplementierungen wird die Praxistauglichkeit nach den Anpassungen untersucht.

Das Liskovsche Substitionsprinzip kann als einziges Prinzip beim Entwurf der neuen Architektur nicht beachtet werden, sondern erst wenn die abstrakten Klassen implementiert und erneut erweitert werden.

Die neue Struktur bildet ein Grundgerüst, das aus den drei Komponenten des Architekturmusters sowie aus deren Unterkomponenten besteht. Die Komponenten und Unterkomponenten sind lose gekoppelt. Das Grundgerüst ist flexibel einsetzbar, stabil und erweiterbar.

# Einleitung

Seit der Entwicklung des Architekturmusters MVC hat sich in den Programmentwürfen für Benutzerschnittstellen die Trennung von Daten und View immer mehr durchgesetzt. Dabei wird der Quelltext einer Anwendung in die drei Klassen Programmlogik (\gls{model), View (\gls{View) und Steuerung (\gls{controller) unterteilt. Diese Struktur macht den Quelltext für \gls{Views und \gls{controller austauschbar und wiederverwendbar. Außerdem erleichtern die Abstrahierung und die Modularisierung die Zusammenarbeit mehrerer Entwickler. Solange die Schnittstellen unverändert bleiben, können die verschiedenen Klassen flexibel und voneinander unabhängig bearbeitet werden.

Zwei der wichtigsten Argumente der solid--Prinzipien sind Wiederverwendbarkeit und Flexibilität. Außerdem soll durch deren Anwendung die Lesbarkeit und die Erweiterbarkeit des Quelltexts sichergestellt werden, und Entwicklerteams sollen effizienter zusammenarbeiten können. Die solid--Prinzipien sind in dem Buch „Agile Software Development, Principles, Patterns, and Practices“ von Robert C. Martin ausführlich beschrieben.

Wir werden zuerst #ref die Komponenten des MVC-Architekturmusters genau betrachten, anschließend #ref die fünf SOLID-Architekturprinzipen. Die drei Komponenten des MVC-Architekturmusters werden wir #ref jeweils auf die Relevanz der fünf SOLID-Prinzipien untersuchen und entsprechend anpassen. Beispielimplementierungen (#ref) werden die Praxistauglichkeit überprüfen.

# Architekturmuster und -prinzipien

## MVC

Das Architekturmuster MVC S.~26—49 KraPo88 sieht für interaktive Anwendungen die Trennung in ein Modul für die Programmlogik (Model), ein Viewmodul (View) und ein Steuerungsmodul (Controller) vor. Jedes dieser Module steht mit jedem anderen in einer bestimmten Relation.

MVC wurde 1978 als eine Designlösung zu einem bestimmten Problem entworfen. Das oberste Ziel war es, das mentale Modell des Nutzers der relevanten Informationsraums zu unterstützen und es dem Benutzer zu ermöglichen, diese Informationen zu untersuchen und zu bearbeiten. ööReens03

MVC was conceived in 1978 as the design solution to a particular problem. The top level goal was to support the

user's mental model of the relevant information space and to enable the user to inspect and edit this information.

Diese Strategie wird gewählt, um (1) ein spezielles Set von Systemkomponenten zu kreieren, die für einen hoch interaktiven Softwareentwicklungsprozess benötigt werden und (2) ein allgemeines Set von Systemkomponenten zu beschaffen, das Programmierern ermöglicht, portable interaktive grafische Anwendungen leicht zu erstellen. ööKraPo88

### Model

Im Model befindet sich die eigentliche Programmlogik. Ein typisches Szenario ist eine Interaktion des Benutzers mit der View, die durch den Controller an das Model vermittelt wird. Nach Abschluss einer Verarbeitung der Interaktion durch das Model benachrichtigt es die View über geänderte Daten. Die View aktualisiert schließlich die Darstellung der Daten. Danach (\*fn oder zwischendrin) können weitere Interaktionen stattfinden. Das Model ist normalerweise einzigartig. Views und Controller kennen also nur ein Model.

### View

Die Relationen der drei Module werden durch verschiedene Entwurfsmuster realisiert. Die View als Observer S.~293—305 GoF94 des Models wird über Änderungen am Model benachrichtigt. Hierfür registriert die View einen oder mehrere Callbacks (eine Methode oder eine Funktion), der immer dann aufgerufen wird, wenn sich an den anzuzeigenden Daten etwas ändert. Man sagt, dass die View das Model kennt, das Model die View aber nicht – obwohl aus programmatischer Sicht dem Model dadurch eine Schnittstelle der View bekannt gemacht wird. Mehrere Views können außerdem untereinander in Relation stehen. Mit dem Composite-Pattern S.~163—175 GoF94 können Teilkomponenten der View zum Gesamtmodul zusammengefasst werden. Sowohl das Gesamtmodul als auch die Teilkomponenten können dann auf Änderungen am Model reagieren.

### Controller

It is a controller's job to handle the control or manipulation

(editing) functions of a model and a particular view. KraPo88

Benutzereingaben werden vom Controller interpretiert. In Form des Strategy-Patterns S.~315—325 GoF94 wird der View ein Controller zugewiesen. Eine View kann einen oder mehrere Controller besitzen. In der Regel werden aber sogenannte View-Controller-Paare erzeugt, also eine View mit einem Controller. Der Controller der View kann sich jederzeit ändern, auch zur Laufzeit. Wir betrachten in refiwas ein Beispiel, in dem gleiche Benutzereingaben aufgrund geänderter Zustände unterschiedlich interpretiert werden. Controller besitzen außerdem die Fähigkeit, andere Views aufzurufen. Ein Controller benötigt deshalb Zugriff auf entsprechende Schnittstellen der Laufzeitumgebung. Außerdem muss ein Controller das Datenmodell der Ansicht manipulieren können. Wir sehen uns das in refinochwas genauer an.

## SOLID

Das Ziel der Prinzipien ist die Erstellung von Softwarestrukturen der mittleren Ebene, die

* Änderungen tolerieren,
* Leicht zu verstehen sind und
* Die Grundlage für Komponenten sind, die in vielen Softwaresystemen genutzt werden können.

Der Begriff „mittlere Ebene“ bezieht sich auf die Tatsache, dass diese Prinzipien von Programmierern angewendet werden, die auf der Modulebene arbeiten. Sie werden genau oberhalb der Codeebene angewendet und helfen, die Arten der Softwarestrukturen innerhalb der Module und Komponenten zu definieren. ööMartin17

### Single-Responsibility-Prinzip

Das Single-Responsibility-Prinzip besagt, dass es \textit{für eine Klasse nur einen Grund zur Änderung geben sollte \autocite[S.~95]{Marti13. Nehmen wir an, dass eine Datenbankanwendung um eine Filtermöglichkeit nach betriebsspezifischen Kennzahlen -- etwa dem Anteil eines Verkaufsartikels am Umsatz -- erweitert werden soll. Um konkrete Werte für gefilterte Abfragen eingeben zu können, müssen der View entsprechende Steuerelemente hinzugefügt werden. Jede Änderung, die außerdem an der View vorgenommen werden muss, weist auf eine Verletzung des Single-Responsibility-Prinzips hin. Es ist zum Beispiel denkbar, dass der Ergebnistabelle im Viewmodul eine Spalte mit der neuen Kennzahl hinzugefügt werden soll. Offensichtlich müssen die Steuerelemente zur Eingabe der Abfragewerte und die Darstellung des Abfrageergebnisses in unterschiedlichen Viewklassen implementiert werden, wenn sichergestellt werden soll, dass jede Klasse eine einzige Verantwortlichkeit besitzt.

### Open-Closed-Prinzip

Open-Closed sind \textit{Klassen, Module, Funktionen etc., die für Erweiterungen offen, aber für Modifikationen verschlossen sind \autocite[S.~99]{Marti13. Ein Anwendungsgerüst muss flexibel erweiterbar sein, aber auch eine unveränderliche Grundfunktionalität bieten, die von Erweiterungen nicht beeinträchtigt wird.

Eine Hauptaufgabe von Controllerklassen ist es, Benachrichtigungen entgegenzunehmen und weiterzuleiten. Weil die Stabilität des bestehenden Quelltexts davon abhängt, dass der Controller diese Aufgaben zuverlässig ausführt, muss die Controllerklasse für Änderungen an ihren Funktionen verschlossen sein. Neue Controller-Implementierungen müssen zusätzlich weitere Aufgaben ausführen. Das Modul muss deshalb für solche Erweiterungen offen sein.

### Liskovsches Substitutionsprinzip

Vereinfacht gesagt schreibt das Liskovsche Substitutionsprinzip vor, dass \textit{Obertypen durch Untertypen ersetzbar sein sollen \autocite[S.~111]{Marti13.

Die Beziehung soll sicherstellen, dass jede Eigenschaft, die Oberklassenobjekten zugeschrieben werden kann auch für Objekte ihrer Unterklasse gelten.ööLiswi94

Angenommen, einer von mehreren Teilkomponenten des Viewmoduls soll ein Textfeld hinzugefügt werden. Die Teilkomponente ohne das zusätzliche Textfeld soll aber weiterhin zur Verfügung stehen. Es ist naheliegend, für diesen Zweck eine neue Viewklasse von der bestehenden abzuleiten. Bei Bedarf kann dann ein Objekt der Basisklasse durch ein Objekt der abgeleiteten Klasse ersetzt werden. Wird das Liskovsche Substitutionsprinzip befolgt, kann das Objekt der abgeleiteten Klasse verwendet werden, ohne das Viewmodul zu verändern. Ein wichtiger Aspekt ist dabei das neue Datenfeld, das dem Viewmodul unbekannt ist.

### Interface Segregation

Interface Segregation bedeutet, dass eine Klasse einer anderen genau die Schnittstelle zur Verfügung stellt, die für den jeweiligen Zweck vorgesehen ist. In MVC verwenden sowohl die View als auch der Controller Schnittstellen des Models. Die View muss sich als Empfänger für Benachrichtigungen über Änderungen registrieren. Der Controller muss das Model über Benutzereingaben benachrichtigen. Daraus ergeben sich zwei verschiedene Schnittstellen zum Model, von denen die View und der Controller auch jeweils nur eine kennen sollten.

### Dependency-Inversion-Prinzip

Das Dependency-Inversion-Prinzip schreibt vor, dass Module nicht von anderen Modulen, die sich niedriger in der Modulhierarchie befinden, abhängig sein sollten S.~127 Marti13. Wir werden dieses Prinzip bei der Implementierung des Viewmoduls und seiner Teilkomponenten berücksichtigen.

# Eine SOLID View

## Single-Responsibility-Prinzip im Viewmodul

Ein Viewmodul beinhaltet wenig Programmlogik. Es reagiert auf Änderungen an den Anwendungsdaten und macht diese sichtbar. Für Benachrichtigungen über Änderungen ist das Observer-Entwurfsmuster vorgesehen. Ein Observer besitzt eine Verantwortlichkeit im Sinne des Single-Responsibility-Prinzips. Die Registrierung am Model und der Callback sollten sich deshalb in einer eigenen Klasse befinden.



\ref{preslog zeigt die Klassenstruktur des Viewmoduls. Die Klasse \texttt{ModelObserver behandelt die Benachrichtungen über Änderungen im Datenmodell der Anwendung, indem sie ein Attribut der Klasse \texttt{PresentationState synchronisiert. In \texttt{PresentationState werden außerdem die Zustände von Steuerelementen gespeichert. In der Klasse \texttt{Presentation befinden sich Methoden, mit denen die View abhängig von den Zustandsattributen dynamisch aktualisiert werden kann. Zum Beispiel könnte das Datum der letzten Aktualisierung angezeigt werden. Die Klasse \texttt{View ist für die Darstellung der Daten und Zustände verantwortlich. Diese Klassenstruktur befolgt das Single-Responsibility-Prinzip besser als eine einzige \texttt{View--Klasse, in der alle diese Aspekte implementiert sind.

## Open-Closed-Prinzip im Viewmodul

Das Viewmodul ist für Erweiterungen an seiner Präsentationslogik offen. Die Klasse \texttt{ModelObserver ist keiner anderen Klasse bekannt und ist deshalb vollständig implementiert. Die Klasse View ist abstrakt. Es sind keine Änderungen am Viewmodul nötig, um neue Views zu implementieren. Die Vorgaben des Open-Closed-Prinzips sind eingehalten.

## Liskovsches Substitutionsprinzip im Viewmodul

Bisher gibt es im Viewmodul keine Vererbungshierarchie. Für die Erweiterungen des Datenmodells und der Präsentationslogik werden von den bestehenden Klassen neue abgeleitet.

Nennen wir die Klasse, die wir von \texttt{Presentation ableiten, \texttt{Form. Ein Formular kann Pflichtfelder enthalten, und Formularfelder können bereits mit einem Wert gefüllt sein. Diese Informationen werden vom Model geliefert und sind über das Attribut \texttt{state, das die neue Klasse erbt, verfügbar. \texttt{Form erhält zwei neue Methoden. Eine Methode füllt Formularfelder mit Standardwerten und die zweite markiert noch ungespeicherte Änderungen. Die geerbte abstrakte Methode \texttt{updateView wird implementiert, um die beiden neuen Methoden aufzurufen. Die von \texttt{PresentationState abgeleitete Klasse nennen wir \texttt{FormState. Sie erhält für jedes Formularfeld ein Flag, das eine Änderung an dem Wert des jeweiligen Felds anzeigt. Von der Klasse \texttt{View leiten wir ebenfalls eine Klasse ab, sie bekommt den Namen \texttt{FormView und Attribute für die Formularfelder. Bei jeder Änderung wird im Objekt der Klasse \texttt{FormState das Änderungs-Flag des jeweiligen Formularfelds gesetzt und geänderte Formularfelder werden hervorgehoben.

Wir fügen den drei Klassen \texttt{Form, \texttt{FormState und \texttt{FormView jeweils noch eine Erweiterung hinzu, indem wir von jeder der drei Klassen wieder eine neue Klasse ableiten. Wir nennen die Klassen \texttt{ExtForm, ExtFormState und \texttt{ExtFormView. Mit ihnen wird ein erweitertes Formular realisiert, in dem bestimmte Formularfelder abhängig von einem Radiobutton ein- oder ausgeblendet werden. Das Attribut \texttt{appData erhält Optionen für den Radiobutton und für jedes Formularfeld eine Zuordnung zu einer Radiobutton-Option. \texttt{ExtForm erhält eine Methode, die Formularfelder ein- und ausblendet. Die Methode \texttt{updateView wird noch einmal überschrieben, um die Basisklassenversion und die neue Methode aufzurufen. In \texttt{ExtFormState wird in einem zusätzlichen Attribut die Information gespeichert, ob es versteckte Formularfelder mit ungespeicherten Änderungen gibt, beziehungsweise welchen Radiobutton-Optionen diese zugeordnet sind. In \texttt{ExtFormView werden diese Radiobutton-Optionen hervorgehoben.

Objekte der Klassen \texttt{ModelObserver und \texttt{Form funktionieren zusammen mit Objekten der Klassen, die direkt von den abstrakten Basisklassen abgeleitet sind, genauso wie mit Objekten der Ext-Klassen, die von den konkreten Implementierungen erneut abgeleitet wurden. Die Forderung des Liskovschen Substitutionsprinzips ist erfüllt.

## Interface-Segregation-Prinzip im Viewmodul

Ein Konstruktorparameter für die Klasse \texttt{ModelObserver ist nötig, damit das Interface-Segregation-Prinzip eingehalten ist. Der Parameter gibt an, über welche Änderungen der Observer benachrichtigt werden soll.

## Dependency-Inversion-Prinzip im Viewmodul

Weil sie die Geschäftslogik der View enthalten, stellen die Klassen, die direkt oder indirekt von \texttt{Presentation abgeleitet sind, die «High~Level~Modules» im Sinne des Dependency-Inversion-Prinzips dar. Das Prinzip besagt, dass diese Klassen nicht von Implementierungsdetails in den anderen Klassen abhängig sein dürfen. Dadurch soll verhindert werden, dass Änderungen an Implementierungsdetails die Geschäftslogik in ihrer Gesamtheit beeinträchtigen. Wie schon beim Open-Closed-Prinzip ermöglicht Abstraktion, diese Anforderung zu implementieren. Keins der Attribute der Klasse \texttt{Presentation ist von einem konkreten Typ, und die Klasse besitzt keine konkreten Methoden. Die Implementierungsdetails befinden sich in abgeleiteten Klassen, ohne die ungewünschte Abhängigkeit zu erzeugen.

# Ein SOLID Controller

## Single-Responsibility-Prinzip im Controllermodul

Der Controller in einer MVC-Architektur hat zwei Aufgaben. Er interpretiert Benutzereingaben, überprüft zum Beispiel Pflichtfelder eines Formulars, und er benachrichtigt gegebenenfalls andere Module über relevante Benutzereingaben.



Das Controllermodul besteht aus einer Observer-Implementierung für die View, einer Klasse für die Prüfung der Benutzereingaben, einer Datenstruktur für verschiedene Zustände der Ansicht und der Controllerklasse, wie in \autoref{inputval} zu sehen ist. Die Aufgabe der Klasse \texttt{Controller} ist es, andere Komponenten über Benutzereingaben zu benachrichtigen.

Das Single-Responsibility-Prinzip ist mit dieser Klassenstruktur eingehalten.

## Open-Closed-Prinzip im Controllermodul

Im Controllermodul befindet sich die Klasse Validator, die Benachrichtigungen über Benutzereingaben entgegennimmt und unmittelbare Rückmeldungen gibt. Sie ist die einzige konkrete Klasse, ist im Controllermodul (\*fn der view ist er bekannt) keiner anderen Klasse bekannt und für Änderungen verschlossen. Die Klassen Validator und Controller sind abstrakt, also für Änderungen offen. Die Forderungen des Open-Closed-Prinzips sind erfüllt.

## Liskovsches Substitutionsprinzip im Controllermodul

Es gibt im Controllermodul wie auch im Viewmodul zunächst keine Vererbungshierarchie. Erweitern wir die abstrakten Basisklassen, sind die Forderungen des Liskovschen Substitutionsprinzips einzuhalten.

Damit Benutzereingaben auf ihre Gültigkeit überprüft werden können, muss eine konkrete Implementierung der Klasse Validator geschrieben werden. Validator ist eine generische abstrakte Klasse, deren Implementierungen den Typparameter ViewState oder eine von ViewState abgeleitete Klasse haben müssen. In der Vererbungsstruktur muss für die Implementierungen und Überschreibungen einer Implementierung die Invarianz oder Kovarianz des Rückgabetyps von validate beachtet werden.

In der Schnittstelle der Klasse Controller sind keine Übergabe- oder Rückgabeparameter deklariert. Das Liskovsche Substitutionsprinzip ist immer eingehalten, solange die Schnittstelle nicht verändert oder erweitert wird.

## Interface Segregation im Controllermodul

Das Controllermodul ist nur der View bekannt. Sie erhält Zugriff auf die Klasse Validator, um Benutzereingaben zu senden und unmittelbar Rückmeldungen zu erhalten. Die Klasse Controller erhält selbst eine Schnittstelle der View. Views sind je nach Art der Anwendung sehr unterschiedlich strukturiert. Der Controller benötigt mindestens die Methode update, um die View über Änderungen am ViewState zu benachrichten. Dieser einfache Anwendungsfall kann über das Observer-Entwurfsmuster #ref realisiert werden. In Erweiterungen des Controllers und der View können weitere Beziehungen gebildet werden, zum Beispiel um aufgrund von Benutzeraktionen Elemente der View ein- und auszublenden.

## Dependency Inversion im Controllermodul

Wie in #refDIP angedeutet, ist Dependency Inversion in MVC-Architekturen vor allem im Viewmodul zu beachten. Das Controllermodul definiert zwar auch eine Abhängigkeit der Klasse Validator von Controller. Weil Controller eine abstrakte Klasse ist, wird das Prinzip der Dependency Inversion dadurch aber bereits eingehalten.

# Ein SOLID Model

## Single Responsibility im Model

Das Model ist ein noch etwas abstrakteres Merkmal des MVC-Architekturmusters. Während View und Controller in MVC immer demselben Zweck dienen, hängt die Funktion des Models von der Art der Anwendung ab. (\*fn Beispiel Banking:Notizbuch) Die Verantwortlichkeit des Models ist es, die View über Änderungen der Daten, die dem Benutzer angezeigt werden sollen, zu informieren.



Das Klassendiagramm des Modells sieht dem der View sehr ähnlich. Ein ControllerObserver wird über Änderungen informiert, die der Controller an das Model weiterleitet. Änderungen an den Daten, die dem Benutzer angezeigt werden sollen, werden an die View gesendet. Diese Verantwortlichkeit befindet sich in der Klasse Model.

## Open-Closed-Prinzip im Model

Die Nachrichtenübermittlung vom Controller an das Model und vom Model an die View sind essentielle Bestandteile des Models. Sie sind verschlossen für Veränderungen. Denkbare Erweiterungen am Model im Sinne des MVC-Architekturmusters sind mehrere Ansichten oder weitere Observer für Datenquellen, die keine Benutzeroberflächen sind. Dieses Dokument behandelt diese Fälle nicht; das Open-Closed-Prinzip ist wegen der abstrakten Natur des Begriffs Model nur bedingt anwendbar.

## Liskovsches Substitutionsprinzip

Das Model im MVC-Architekturmuster ist nicht austauschbar #ref. Das Substitutionsprinzip ist deshalb für das Model nicht relevant.

## Interface Segregation

Das Model stellt eine Schnittstelle zur Verfügung, durch die es vom Controller über Benutzereingaben informiert wird. Wie in den anderen beiden Komponenten wird die Beziehung durch das Observer-Entwurfsmuster realisiert. Die Methode notify erhält als Argument eine Datenstruktur, die das Model als Nachricht verarbeiten kann. Das Model besitzt selbst genau die Schnittstelle der View, durch die es diese über Änderungen informiert.

## Dependency Inversion

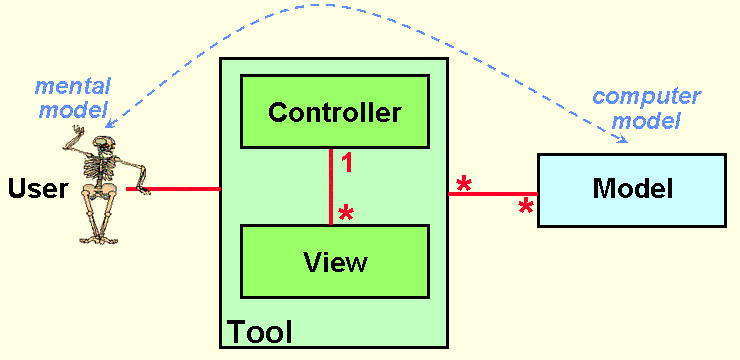
Die Klasse ControllerObserver besitzt eine Abhängigkeit von der abstrakten Klasse Model. Das Prinzip der Dependency Inversion ist dadurch bereits eingehalten.

# Fazit

Die SOLID-Prinzipien beim Entwurf einer MVC-Architektur zu beachten führt zu kompakten Klassendiagrammen der einzelnen Komponenten. Außerdem entkoppelt es die Komponenten und sogar die Unterkomponenten, wie etwa den Validator des Controllers, voneinander. Die Austauschbarkeit der Kompenenten wird dadurch erhöht, und die Abhängigkeiten werden auf ein Minimum reduziert.

Das Single-Responsibility-Prinzip hat den größten Einfluss auf die MVC-Architektur. Das MVC-Architekturmusters wird normalerweise nur durch die drei Komponenten Model, View und Controller beschrieben; weitere Klassen werden nirgends genannt (Abb x, Krapo88 s.28f). Durch die Beschränkung einer Klasse auf genau eine Verantwortlichkeit erweitert sich das Controllermodul auf zwei Unterkomponenten, nämlich auf den Validator und den Controller. Die Gesamtarchitektur besteht nun aus sechs statt vorher drei (Unter-)Komponenten.





Reens97

Das Open-Closed-Prinzip zwingt uns, die neue Architektur als Gesamtheit zu betrachten. Es ist ein architektonisches Grundgerüst, das erst zu einer Anwendung erweitert werden muss. Für das Grundgerüst war zu beachten, dass bestimmte Eigenschaften einer MVC-Anwendung immer vorhanden sein müssen. Andere Eigenschaften sind nur für bestimmte Anforderungen zu implementieren. Die Architektur kann um diese anderen Eigenschaften erweitert werden, ohne die Komponenten zu verändern.

Das Grundgerüst besitzt eine einfache Struktur und kann an sich nicht das Liskovsche Substitutionsprinzip verletzen. Das Prinzip kann aber in MVC-Anwendungen berücksichtigt werden, die das Grundgerüst nutzen. In den Beispielimplementierungen #ref irgendwas.

Interface Segregation ist das zweite Prinzip, das erheblichen Einfluss auf die Struktur der neuen MVC-Architektur hat. Weil sich die Komponenten alle untereinander Nachrichten senden, muss für jede Beziehung zwischen den Komponenten eine entsprechende Schnittstelle vorhanden sein. Die Schnittstellen werden in zwei Fällen als Observer realisiert und in einem Fall als Facade #ref.

Dependency Inversion wird in allen drei Komponenten berücksichtigt. Es werden Abstraktionen konstruiert, wodurch Abhängigkeiten von Implementierungen vermieden werden.