Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3535

Ein interaktives Planungssystem für Ausbildungskurse

Reinhold Rumberger

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Plödereder
Betreuer/in:	Prof. Dr. Plödereder, Udo Bufler
Beginn am:	2013-08-14
Beendet am:	2014-05-27
CR-Nummer:	H.4.1, H.5.2, I.2.8, J.1

Kurzfassung

Im Rahmen dieser Diplomarbeit wird ein bestehendes Planungssystem namens *JVS Planung* komplett überarbeitet, erweitert und modernisiert, um seit der ursprünglichen Implementierung hinzugekommenen Anforderungen gerecht zu werden. Dabei sollen Einschränkungen an die Planung („Constraints“) automatisch geprüft werden, um den Anwender bei der Planung zu unterstützen. Am Ende einer erfolgten Planung werden offizielle Dokumente generiert, die an örtliche Behörden, Ämter und Schulen verteilt werden. Aus diesem Grund muss die Anwendung nicht nur benutzerfreundlich, sondern auch robust und praxistauglich sein.

Bei der Planung handelt es sich um die jährliche Planung von Terminen, an denen im Folge-Schuljahr die Schulklassen von Grund- und Förderschulen eine ihnen zugewiesene Jugendverkehrsschule (JVS) besuchen sollen. An diesen JVS der Kreisverkehrswacht Esslingen nehmen die Schüler an einer Fahrradausbildung teil.

Die überarbeitete Anwendung – *JVS2* – wird dynamisch Benutzereingaben prüfen, das Speichersystem komplett überarbeiten und die Fähigkeit besitzen, neue Constraints mit geringem Aufwand zu implementieren und in eine bestehende Anwendung einzufügen. Zusätzlich wird die Codebasis komplett neu implementiert und modernisiert.

In dieser Ausarbeitung wird beschrieben, wie welche Anforderungen erhoben wurden, welche Entscheidungen den Entwurf bedingt haben und wie daraus die konkrete Implementierung entwickelt wurde.

Abstract

In this Diplomarbeit, an exisiting planning software named *JVS Planung* is completely reengineered, extended and modernised to be able to fulfill requirements that have surfaced since the original implementation was finished. It needs to automatically check some constraints of the resulting plan to support the user in the planning process. The result of such a process is a set of official documents, which are distributed to local government offices and schools. This means, that the application has to be not only user friendly, but robust and usable in real-world scenarios.

The planning process is the yearly allotment of time slots at which school classes of local elementary schools will visit an assigned Jugendverkehrsschule (JVS) the following school year. They attend bicycle riding lessons in these JVS of the Kreisverkehrswacht Esslingen.

The reengineered application – *JVS2* – will dynamically verify user input, completely reengineer the saving system and have the ability to implement new constraints and add them to the application with little effort. Additionally, the code base will be completely re-implemented and modernised.

This document will describe which requirements were found, which decisions influenced the design and how the actual implementation was derived from these.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Da diese Arbeit auf der Diplomarbeit von Herrn Schwab[19] aufbaut, wird diese als Lektüre empfohlen.

In dieser Arbeit werden Grund- und Förderschulen als „Schulen“ bezeichnet. „Schulen“ beinhaltet jedoch nicht Jugendverkehrsschulen, die als „Jugendverkehrsschulen“ oder „JVS“ bezeichnet werden. Die im Rahmen dieser Diplomarbeit implementierte Software wird als *JVS2* bezeichnet. Das im Rahmen der Diplomarbeit von Herrn Schwab[19] implementierte System wird *JVS Planung* genannt.

1.1. Motivation

Ursprünglich wurde die Planung der Unterrichtstermine an den Jugendverkehrsschulen von Hand durchgeführt. Um diesen Vorgang zu erleichtern und die Randbedingungen der Planung zu prüfen, wurde die Software *JVS Planung*[19] erstellt. Im Betrieb haben sich bei dieser Software einige geänderte Anforderungen ergeben. Aus diesem Grund soll jetzt eine neue Version implementiert werden, welche diese Anforderungen erfüllt.

Eine dieser Anforderungen ergibt sich aus dem Fehlen einer einheitlichen Implementierung der „Constraints“. Dabei handelt es sich um vordefinierte Eigenschaften einer Planung, die bei deren Erstellung eingehalten werden müssen. Diese Constraints wurden in *JVS Planung* in zwei Kategorien unterteilt: „hart“ und „weich“. Diese Unterteilung hat sich als zu unflexibel erwiesen, weshalb *JVS2* eine feingliedrigere Unterteilung erlauben soll. Zusätzlich soll die Möglichkeit geschaffen werden, Constraints zentral zu implementieren und gegebenenfalls neue Constraints nachzuladen.

Eine weitere Anforderung ergibt sich aus dem gewählten Speicherformat. Hier wurde ein auf Javas Serialisierung basierendes Format gewählt. Dies hat zur Folge, dass schon kleinere Änderungen am Speicherformat oder der Code-Struktur alte Speicherstände unbrauchbar machen. Um dies zu umgehen, soll *JVS2* ein neues Speicherformat verwenden.

1.2. Aufgabenstellung

In dieser Diplomarbeit soll *JVS Planung* in enger Zusammenarbeit mit den Sachbearbeitern der Polizeidirektion Esslingen komplett überarbeitet, erweitert und dabei die Planungseffizienz verbessert werden.

Dabei soll das überarbeitete System auf eine aktuelle Windows-Version portiert werden, bezüglich der in Esslingen verwendeten Betriebssysteme portabel sein und eine höhere Robustheit aufweisen. Die Benutzerschnittstelle soll modernisiert werden, was vor allem die dynamische Rückmeldung der Gültigkeit von Eingaben betrifft. Einige von den Sachbearbeitern identifizierte Schwachstellen von *JVS Planung* sollen vermieden werden.

Nach Möglichkeit soll *JVS2* die Fähigkeit besitzen, weitere Constraints einzubinden, ohne große Eingriffe in den Programmcode vornehmen zu müssen.

Die Implementierung soll qualitativ hochwertig und real einsetzbar sein. Tests der Implementierung sollen anhand realer Anwendungsdaten aus den vergangenen Jahren erfolgen.

1.3. Gliederung

Die Gliederung dieser Arbeit orientiert sich an den Arbeitsschritten der Projektdurchführung. So werden zuerst die ermittelten Ziele und Anforderungen beschrieben, dann wird der Entwurf erläutert und schließlich die Implementierung erklärt. Im Anschluss wird die Planung und Durchführung der Tests erläutert, sowie deren Resultate vorgestellt.

Abschließend wird die Arbeit kurz zusammengefasst und ein Ausblick auf potentielle zukünftige Entwicklung gegeben.

Im Anhang A befindet sich eine Begriffserklärung für projektspezifische Fachbegriffe.

2. Anforderungen und Ziele

In diesem Kapitel werden die Anforderungen und Ziele an das Programm *JVS2* beschrieben. Wo angebracht, beschränkt sich diese Beschreibung auf die Unterschiede, die sich zu *JVS Planung* ergeben.

2.1. Ermittlung der Anforderungen und Ziele

Als erster Schritt wurde probehalber ein minimalistisches Planungsszenario mit *JVS Planung* durchgespielt und die beigefügte Dokumentation gelesen, um bisherige Funktionalität zu ermitteln. Danach wurde der Code analysiert, um das Verständnis für das Programm zu erhöhen und die Entscheidung treffen zu können, ob vorhandener Code weiterverwendet werden sollte, oder eine komplette Neuimplementierung stattfinden müsse. Schließlich wurde der Kunde nach seinen Anforderungen befragt.

Die hier aufgeführten Anforderungen und Ziele wurden während des gesamten Projekts ermittelt. Deshalb mussten Entwurfs- und Implementierungsdetails während des Projekts mehrmals angepasst werden. Dies hatte zur Folge, dass Teile der Planung und Implementierung suboptimal verliefen. Die nachträgliche Anpassung betraf indirekt verwandte Stellen, und es fehlte die Zeit, die komplette Planung und Implementierung zu überarbeiten, um diese Stellen zu ermitteln.

Der Kunde war an jedem Schritt der Anforderungsanalyse beteiligt, um ermittelte Anforderungen zu verifizieren und zu ergänzen. Daraus ergab sich das Ziel, dass Änderungswünschen des Kunden auch kurzfristig entsprochen werden müsste. Um dies zu ermöglichen, wurde sowohl beim Entwurf als auch bei der Implementierung darauf Wert gelegt, möglichst flexibel zu bleiben und Teile des Programms, die nicht logisch verwandt waren, voneinander abzugrenzen. Dies sollte es ermöglichen, für jede Änderung der Funktionalität nur möglichst kleine Teile des Programmcodes bearbeiten zu müssen und so den Aufwand zu begrenzen.

Hier musste jedoch auch darauf geachtet werden, möglichst große, logisch verwandte Teile des Programms zusammenzufassen um die Komplexität zu begrenzen und so die Verständlichkeit und Wartbarkeit von sowohl Entwurf als auch Implementierung möglichst hoch zu halten.

Um sowohl die möglichst große Anpassbarkeit zu gewährleisten als auch die Komplexität zu minimieren wurde das Programm schon im Entwurf anhand von Java-Packages in grobe Bereiche unterteilt. Während der Implementierung wurde dann darauf geachtet, dass die Menge der Daten, die innerhalb dieser Bereiche geteilt wurden, minimiert wurden. Dies

führt dazu, dass der Datenaustausch in logisch zusammenhängenden Teilen des Programms ohne Umwege möglich, jedoch auf das Nötigste beschränkt ist.

Um geänderten Anforderungen entgegenzukommen und neue Konzepte erproben zu können, wurde außerdem ein iteratives Entwicklungsmodell verwendet. Hierbei wurde kein klassisches Modell verwendet, da diese Modelle strenge Anforderungen an den Projektablauf stellen und großteils auf Entwicklerteams ausgelegt und somit auf dieses Projekt nicht anwendbar sind. Das gewählte Entwicklungsmodell hat keine formellen Einschränkungen an die Durchführung der einzelnen Projektphasen gemacht. Es wurden lediglich eine Reihe von zweiwöchigen Planungs- und Implementierungsphasen definiert, in denen angestrebt wurde, jeweils mindestens ein Kundentreffen zu organisieren, an dem der aktuelle Fortschritt vorgestellt und der weitere Projektverlauf geplant werden konnte. Gegen Ende des Projekts wurde eine große Testphase vorgesehen, für die mehrere Kundentreffen angestrebt wurden.

2.2. Ziele

2.2.1. Bedienung

Der Kunde wünscht ein Oberflächendesign basierend auf bekannten Bedienkonzepten. Dies bedeutet vor allem die Verwendung von Menüs, Auswahllisten und Knöpfen, wie sie aus Büroprogrammen und allgemeinen Windows-Anwendungen bekannt ist.

Allgemein soll das Programm Fehlbedienungen verhindern oder behandeln und „intuitiv“ bedienbar sein. Da diese Ziele abstrakt sind, und sich keine konkreten Anforderungen daraus ableiten lassen, wurde der Kunde vor und nach der Implementierung neuer Interaktionsmöglichkeiten nach seiner Meinung befragt. Hierbei wurde besonders darauf geachtet, dass die Bedienung für ihn verständlich und intuitiv sei.

2.2.2. Sicherheit

Programmsicherheit

JVS2 wird vor allem auf einem Laptop ausgeführt, der nur selten für den Download von Betriebssystem-Updates eine Internetverbindung aufbaut. Da nur eine begrenzte Menge Anwendungen installiert wird und der Laptop nicht für Internetbenutzung verwendet wird, ist die Programmsicherheit keine Priorität. Des weiteren hat *JVS2* keine Internet-Schnittstelle, die abgesichert werden müsste.

Eine konkrete Sicherheitslücke wurde, nach Rücksprache mit dem Kunden, als unwichtig eingestuft und ignoriert. Hierbei handelt es sich um die neue Funktionalität, dass Constraints durch JAR-Dateien nachgeladen werden können. Diese JAR-Dateien werden ohne Prüfung oder Benachrichtigung des Benutzers nachgeladen. Die in ihnen enthaltenen Constraints

können beliebigen Code ausführen und haben somit ein sehr hohes Schadenspotential. Allerdings ist das Risiko eines Angriffs gering genug, um vernachlässigt werden zu können.

Datensicherheit

Für die auf der Festplatte befindlichen Daten ist der Benutzer verantwortlich. Die Datenintegrität wird nicht vom Programm verifiziert. Sofern eine geladene Datei der vom Programm erwarteten Struktur entspricht, wird sie ohne weiterführende Prüfung verwendet.

Die Sicherung der gespeicherten Daten liegt im Verantwortungsbereich des Benutzers. Da diese Daten in einem lesbaren XML-Format vorliegen, können sie mit den meisten Backup-Programmen effizient gesichert werden.

Damit im Falle eines Programmabsturzes möglichst wenig Daten verloren gehen, werden sie nach jedem vom Benutzer bestätigten Arbeitsschritt auf die Festplatte gespeichert.

2.2.3. Anpassbarkeit

JVS2 soll so implementiert werden, dass geänderte Anforderungen möglichst wenig Code-Änderungen benötigen. Dies bedeutet für Stellen, an denen Änderungen erwartet werden, dass ein modularer Entwurf verwendet werden muss. Auch allgemein sollte der Code möglichst gut dokumentiert und verständlich geschrieben werden.

2.3. Anforderungen

2.3.1. Übernommene Anforderungen

Da ein Teil der Aufgabenstellung die Weiterentwicklung von *JVS Planung* ist, werden große Teile der Funktionalität und des Designs übernommen. Der Code wird jedoch komplett neu implementiert, um den geänderten Anforderungen gerecht zu werden und eine den geänderten Umständen entsprechende Architektur bereitstellen zu können.

Sofern im Folgenden nicht anders spezifiziert, wird alle Funktionalität unverändert von *JVS Planung* übernommen.

Wie auch *JVS Planung* wird *JVS2* dazu verwendet, Ferien-, JVS- und Schuldaten zu erfassen und mithilfe dieser Informationen einen Belegungsplan für die JVS der Kreisverkehrswacht Esslingen zu erstellen. Aus diesem Belegungsplan werden schließlich RTF-Dokumente erstellt, die den Belegungsplan enthalten und ausgedruckt werden können.

Analog zu *JVS Planung* stehen auch bei *JVS2* Benutzerfreundlichkeit, Bedienbarkeit und Robustheit als Ziele im Vordergrund. Dazu muss vor allem die Benutzerführung gut durchdacht sein. Portabilität soll auch gewährleistet sein.

Dem Benutzer ist es wichtig, die Dateneingabe und Planung in kleine Arbeitsschritte zu unterteilen und deren Fertigstellung zu bestätigen. Nach dieser Bestätigung sollen die Änderungen sowohl ins Datenmodell übernommen, als auch auf das Dateisystem gespeichert werden. Um dies zu bewerkstelligen, wird jede Dateneingabemaske mit zwei Knöpfen ausgestattet: „Zurücksetzen“ und „OK & Speichern“. Dieses Konzept wurde aus *JVS Planung* übernommen und ist vom Kunden explizit gewünscht.

2.3.2. Geänderte Anforderungen

Constraint-Warnungen

Es hat sich herausgestellt, dass die Warnungsmarkierungen bei verletzten weichen Constraints für den Benutzer zu kompliziert waren. Deshalb wurde dieses Konzept komplett verworfen und beim Entwurf der Constraints neu durchdacht.

Geblockte Termine

Im Sprachgebrauch der Verkehrspolizei Esslingen werden Zeitblöcke, die von der Planung ausgeschlossen werden sollen, als „Blocktermine“ bezeichnet. Da dieser Begriff verwirrend sein und mit einem „Block an Terminen“ verwechselt werden kann, wurde entschieden, diese Zeitblöcke als „geblockte Termine“ zu bezeichnen. Somit wird der bisherige Begriff „Ausschlusstermine“ durch „geblockte Termine“ ersetzt.

Leitsystem-Übersicht

Der Übersichtsdialog des bisherigen Leitsystems war rein informativ und enthielt nur die Namen und Beschreibungen der Arbeitsschritte. Hier wurde entschieden, dass man aus diesem Dialog direkt auf einen bestimmten Arbeitsschritt springen können soll.

Teilautomatische Planung

JVS Planung enthält eine rudimentäre Implementierung einer teilautomatischen Planung. Konkret können mit dieser Funktionalität Klassen und Schulen nach einem First-Come-First-Served-Prinzip eingeplant werden. Diese Möglichkeit verwendet der Kunde aber nicht, weshalb sie in *JVS2* nicht implementiert werden sollte.

Konkret wünscht der Kunde eine komplett automatische Planung, was aber bei näherer Betrachtung ein – im Rahmen einer Diplomarbeit – unlösbares Problem darstellt. Da der Kunde die teilautomatische Planung nicht als sinnvoll erachtet, wurde diese Funktionalität verworfen.

Modernisierung der Benutzeroberfläche

Eine der Haupt-Anforderungen war die Modernisierung der Benutzeroberfläche. Hier war es wichtig zu beachten, dass der Benutzer mit der vorhandenen Oberfläche gut vertraut war und ihre Bedienung beherrschte. Deshalb durfte das Aussehen und die Handhabung nicht vollständig neu entworfen werden.

Eine Möglichkeit, die Benutzeroberfläche zu modernisieren ohne ihr Aussehen oder ihre Bedienung stark zu verändern, war die Bedienelemente interaktiv auf Eingaben reagieren zu lassen. Das beinhaltet das dynamische (De-)Aktivieren von Schaltflächen und visuelles Feedback bei fehlerhaften[1] und potentiell fehlerhaften[2] Eingaben. Um dies einheitlich implementieren zu können, wurde ein System benötigt, Constraints auch interaktiv zu prüfen.

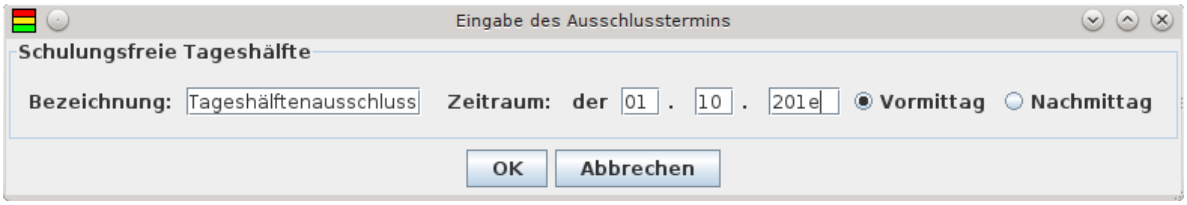


Abbildung 2.1.: Planung der interaktiven Constraints: Dialog aus *JVS Planung* **ohne** interaktive Eingabeprüfung

Beispielhaft wurde diese Änderung dem Kunden mittels eines Dialogs aus *JVS Planung* (Abbildung 2.1) erläutert. Dieser Dialog wurde mit Hilfe einer Graphiksoftware so angepasst, dass sie einem noch zu entwerfenden entsprechen könnte (Abbildung 2.2). Da dem Kunden dieses Konzept mit kleineren Änderungen sehr gefiel, wurde es weitestgehend übernommen (siehe Abbildung 3.19).

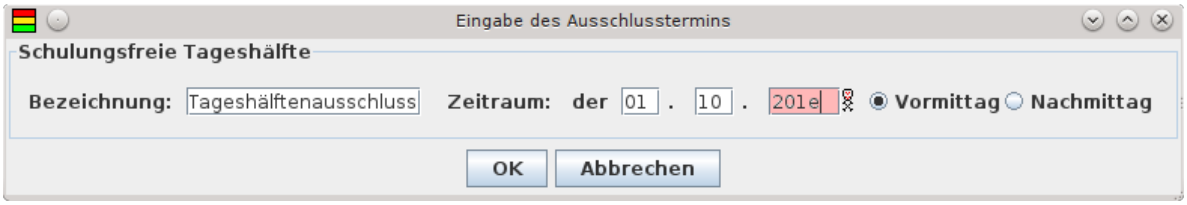


Abbildung 2.2.: Planung der interaktiven Constraints: Dialog aus *JVS Planung* **mit** interaktiver Eingabeprüfung

Eine andere Stelle, an der die Benutzeroberfläche modernisiert werden konnte, war die Datumseingabe. Hier wurde die Entscheidung getroffen, die Datumseingabe mittels Freitexteingabe in einem einfachen Textfeld sowie mittels eines Kalender-Popups – wie es von

[1] beispielsweise ein Text wo eine Zahl erwartet wird
[2] Eingaben, die korrekt sein können, es wahrscheinlich aber nicht sind - beispielsweise eine Klassen-Schülerzahl > 100

modernen Anwendungen bekannt ist – zu ermöglichen. An dieser Stelle wäre es möglich, Datumsfelder vom Programm mit Vorschlägen füllen lassen, die der Benutzer dann nur noch geringfügig anpassen müsste. Der Kunde bevorzugt jedoch leere Eingabefelder, damit er sehen kann, dass die jeweiligen Felder noch nicht bearbeitet wurden. Außerdem kann so die interaktive Constraint-Prüfung ermitteln, dass es sich um eine ungültige – da nicht vorhandene – Eingabe handelt.

Der Kunde wünscht für *JVS Planung* die Möglichkeit, die Schriftgröße anpassen zu können, da einige Bearbeiter ein eingeschränktes Sehvermögen aufweisen.

Oberflächensperrung bei Eingaben

JVS Planung hat fast die gesamte Oberfläche bei Eingaben gesperrt, bis der Benutzer entschied, die Eingaben zu verwerfen oder zu speichern. Leider wurde gesperrt, sobald ein Eingabefeld aktiviert wurde, auch wenn keine tatsächliche Eingabe stattfand. Dies hat sich im Betrieb als hinderlich herausgestellt, weshalb *JVS2* dieses Prinzip ersetzt. Der Benutzer kann andere Elemente nach einer Eingabe anwählen, wird aber gefragt, ob er eventuelle Änderungen verwerfen möchte. Erst wenn er dies bejaht, wird die Eingabemaske gewechselt.

Layout der generierten Dokumente

Bei den generierten Dokumenten gab es öfters Probleme mit den gedruckten Schulnamen, da diese nicht in den vorgesehenen Platz passten. Dies führte zu Layoutproblemen im Dokument, die später manuell korrigiert werden mussten. Um dieses Problem zu minimieren, wurden die gedruckten Schulnamen auf 20 Zeichen begrenzt und das Layout der RTF-Vorlage verbessert, indem einige überflüssige Leerzeichen entfernt wurden.

Es stellte sich an dieser Stelle heraus, dass das Layout auch mit diesen Modifikationen nicht korrekt war, da die generierten Seiten größer waren als eine DIN-A4-Seite. Aus diesem Grund musste die RTF-Vorlage komplett überarbeitet werden.

Inkonsistenzen der Benutzeroberfläche

In *JVS Planung* ist das Design der Benutzeroberfläche stellenweise inkonsistent und unhandlich. So muss bei der Festlegung der JVS-Ausschlusstermine zwischen zwei komplett verschiedenen Ansichten gewechselt werden, während man bei der Eingabe der JVS-Stammdaten nur eine Ansicht hat, in der die JVS durch Klick auf eine Liste gewechselt wird. Diese Inkonsistenz sollte behoben werden.

Bei der Belegungsplanung war eine Ansicht vorgelagert, in der man eine JVS auswählen konnte, für die man einen Plan bearbeiten bzw. erstellen wollte. Die resultierenden 3 ineinander verschachtelten Ebenen an Oberflächen waren für neue Benutzer schwer nachvollziehbar. Hier sollten die ersten beiden Ebenen zu einer kondensiert werden.

Verworfene Constraints

JVS Planung hatte die Fähigkeit, zu melden, wenn Schulen in den gleichen Zeitraum eingeplant wurden, in dem sie bereits zwei Jahre zuvor die zugewiesene JVS besuchten. Dies hat sich in der Praxis als nicht sinnvoll erwiesen, weshalb dieser Constraint in *JVS2* entfallen soll.

Die in *JVS Planung* vorhandene Möglichkeit, zu spezifizieren, dass Schulen vor bzw. nach den Herbstferien eingeplant werden wollen, wird auf Wunsch des Kunden verworfen. Diese Funktionalität war nicht flexibel genug, da die Schulen oft spezifischere Wünsche hatten und wurde adäquat durch den internen Vorplanungsprozess ersetzt.

Schulverwandte Änderung

In *JVS Planung* konnte die Schulart zwischen Grund- und Förderschule geändert werden. Der Kunde wünscht, dass diese Funktionalität für *JVS2* entfernt wird.

Der Kunde wünscht sich für *JVS2* eine farbliche Unterscheidung zwischen Grund- und Förderschulen. Konkret sollen Grundschulen blau und Förderschulen grün dargestellt werden.

Wo Listen mit Schulen angezeigt werden, sollen diese durchnummeriert werden. Dies dient der besseren Übersicht und gibt dem Benutzer eine zusätzliche Möglichkeit zu verifizieren, dass alle Schulen im Datensatz vorhanden sind, ohne diese selbst zählen zu müssen.

In *JVS Planung* konnte die Zahl der JVS-Besuche pro Klasse für Förderschulen individuell pro Schule eingestellt werden. Diese Möglichkeit soll für *JVS2* entfallen und durch eine konstante Anzahl an Klassenbesuchen von 6 Besuchen pro Förderschul-Klasse ersetzt werden.

Schulen sollen in *JVS2* nicht nach Schulart sortiert werden. In *JVS Planung* wurden Schulen alphabetisch nach ihrer Abkürzung sortiert. Dies hatte zur Folge, dass sie zuerst nach Schulart und dann nach ihrem Namen sortiert wurden. In *JVS2* soll die Schulart bei der Sortierung ignoriert werden. Da die Abkürzungen der Schulen nach dem Schema „<Schulart-Abkürzung> <gekürzter Name>“ aufgebaut sind, bedeutet dies konkret, dass die Abkürzung bis einschließlich dem ersten Leerzeichen für die Sortierung ignoriert werden soll.

Allgemeine Änderungen

JVS Planung zeigte an einigen Stellen - speziell bei der Eingabe von geblockten Terminen - einen Tab mit einer Kalenderansicht. Diese Kalenderansicht wurde standardmäßig angezeigt, weshalb man bei jeder Termineingabe vom Kalender-Tab zum Termineingabe-Tab wechseln musste. Da der Kunde die Kalenderansicht nicht verwendete und *JVS2* ein Kalender-Popup für die Termineingabe bereitstellt, soll die Kalenderansicht in *JVS2* entfallen.

JVS Planung wurde für Java 5 auf Windows XP entwickelt und getestet. *JVS2* soll für Java 7 auf Windows 7 entwickelt und getestet werden. Um eine möglichst große Plattformunabhängigkeit zu erreichen, soll *JVS2* zusätzlich auf Linux[3] entwickelt und getestet werden.

Bei der Belegungsplanung hat *JVS Planung* bei geblockten Terminen nur die jeweiligen Felder schwarz gefüllt. Dies machte es schwer, den Grund für den jeweiligen Block zu erfahren. *JVS2* soll hier, wo möglich, die Beschreibung des geblockten Termins anzeigen.

JVS Planung verlangt eine alte Planung, auf der aufgebaut wird, indem die JVS- und Schuldaten übernommen werden. Während diese Option für *JVS2* weiterhin gewünscht ist, um Arbeit zu sparen, soll die Möglichkeit bestehen, ein neues Planungsprojekt ohne alte Daten zu beginnen.

Da der Kunde diese Funktionalität nicht wünscht, und ihr Vorhandensein potentiell gefährlich sein kann, sollen einmal erstellte Belegungspläne in *JVS2* nicht einfach löschbar sein.

JVS Planung musste wegen seiner Architektur die parallele Ausführung mehrerer Programminstanzen verhindern. Die Architektur von *JVS2* soll hingegen die parallele Ausführung mehrerer Programminstanzen erlauben.

JVS Planung erlaubt die Anpassung der Zeitintervalle der individuellen Unterrichtsstunden. Diese Funktionalität könnte für den Benutzer verwirrend sein, weshalb sie in *JVS2* nicht vorhanden sein soll.

2.3.3. Neue Anforderungen

Anforderungen aus dem Betrieb

Im Betrieb ist aufgefallen, dass manchmal vergessen wurde, einige Schulklassen fertig einzuplanen. Da die Warnungen teilweise übersehen, teilweise ignoriert wurden, waren sie ineffektiv. Um dieses Problem zu umgehen, musste dies beim Entwurf der Constraints berücksichtigt werden. Außerdem wurden beim Einplanen unvollständige verplante Schulklassen farblich hervorgehoben.

Obwohl die „Faschingsferien“ keine offiziellen Ferien sind, werden sie doch von allen Schulen durch bewegliche Ferientage in der gleichen Woche implementiert. Um die Planung hier zu vereinfachen, wurden die Faschingsferien als einplanbare Ferien zwischen Weihnachts- und Osterferien vorgesehen.

JVS Planung hatte keine Programmeinstellungen, die über Sitzungen hinaus gespeichert wurden. So war einer der ersten Schritte des Benutzers nach dem Öffnen der Anwendung stets die Maximierung des Programmfensters. *JVS2* soll hier Größe und den Maximierungsstatus des Fensters speichern und beim nächsten Programmstart wieder herstellen. Außerdem soll bei Speicher- und Ladedialogen der zuletzt geöffnete Ordner vorausgewählt werden,

[3] ein jeweils aktuelles Debian Testing vom 15.08.2013 bis zum 27.05.2014

damit der Benutzer nicht jedes Mal manuell zu dem selben Ordner navigieren muss. Die stellte in *JVS Planung* kein Problem dar, da die Ordner zum Ausführungsverzeichnis statisch festgelegt waren – eine Einschränkung, die in *JVS2* nicht mehr existieren soll.

Überarbeitung des Speichersystems

JVS Planung verwendet Javas Serialisierung für das Speicher-Subsystem. Der Vorteil dieses Systems ist die schnelle und einfache Implementierung. Der Haupt-Nachteil ist der Kompatibilitätsverlust zwischen Speicherformaten verschiedener Versionen schon bei geringen Änderungen und jeder strukturellen Anpassung des Codes. Um diesen Kompatibilitätsverlust zu umgehen, können bestimmte Methoden implementiert werden, um alte Java-Klassen einzulesen. Damit sind aber die erwähnten Vorteile nicht mehr vorhanden. Selbst reines Umbenennen von Klassen oder Umstrukturieren der Paketstruktur machen die Vorteile zunichte.

Aus diesem Grund wurde entschieden, dass das Speichersystem komplett erneuert werden müsste, um diese Schwäche zu vermeiden.

Um Speicherstände aus *JVS Planung* auch in *JVS2* verwenden zu können, wird eine Hilfsanwendung bereitgestellt, die Projekte aus *JVS Planung* importieren und in einem für *JVS2* lesbaren Format abspeichern kann.

Überarbeitung des Constraint-Handlings

In *JVS Planung* sind alle Constraints direkt an den Stellen implementiert, an denen sie verwendet werden. Dies macht es schwer, neue Constraints hinzuzufügen oder vorhandene Constraints konsistent zu ändern. Deshalb soll das Constraint-Handling für *JVS2* komplett überarbeitet werden, um abstrakter zu sein und die Implementierung der Constraints an einer zentralen Stelle zu ermöglichen.

Zusätzlich sollen die Constraints in feinere Klassen unterteilt werden, damit nicht mehr nur zwischen harten und weichen Constraints unterschieden wird. Die ursprüngliche Unterscheidung kam durch ein Missverständnis zwischen Kunde und Entwickler zustande. Der Kunde versteht unter einem „harten“ Constraint etwas, das im Normalfall nicht ignoriert werden darf. In Ausnahmefällen möchte er jedoch die Möglichkeit haben, auch viele dieser „harten“ Constraints zu verletzen, was von *JVS Planung* jedoch nicht zugelassen wurde.

Es gibt jedoch einige Constraints, die nicht verletzt sein dürfen, damit eine sinnvolle Planung durchgeführt werden kann. Somit muss es auch weiterhin unverletzbare – „harte“ – Constraints geben. Deren Anzahl soll jedoch minimiert werden.

Um die Modernisierung der Benutzeroberfläche zu ermöglichen – speziell die Anforderung der interaktiven Eingabeprüfung – müssen verschiedene Constraint-Sorten unterschieden werden. Es müssen deshalb „interaktive“ Constraints definiert werden, die interaktiv Eingaben prüfen und „normale“ Constraints, die komplexere Prüfungen übernehmen.

Übernommen werden soll die Möglichkeit, Constraints ab- und anschalten zu können.

Um die Möglichkeit bereitzustellen, in Zukunft durch einen unabhängigen Entwickler weitere Constraints implementieren zu lassen, sollen Constraints aus JAR-Dateien nachgeladen werden können.

Encoding-Probleme

In *JVS Planung* wurde die Speicherung der Schulen mittels Serialisierung in Dateien realisiert, deren Namen die Abkürzung der jeweiligen Schule waren. Da einige Schulen Sonderzeichen wie „ß“ oder Umlaute im Namen hatten, konnte es beim Übertragen zwischen Dateisystemen mit verschiedenen Zeichenkodierungen zu Problemen kommen. Beispielsweise konnte es vorkommen, dass die Dateiverwaltung des Systems mit diesen Namen nicht zurechtkam oder dass später das Laden alter Speicherstände fehlschlug.

In der Praxis ist dieses Problem nie aufgetreten, da der Anwender alle Planungen auf dem gleichen Rechner durchgeführt hat.

Um dies zu adressieren, wurde das Speichersystem so konzipiert, dass die Namen in einer UTF-8-kodierten Datei gespeichert werden. Somit sind Kodierungsprobleme von vornherein ausgeschlossen. Außerdem wurden bei der Benennung der Java-Klassen alle Sonderzeichen vermieden.

Eine andere Stelle, an der Kodierungsprobleme auftreten konnten, war der CSV-Import neuer Grundschuldaten. Hier ist nicht vordefiniert, in welcher Kodierung die CSV-Dateien eingelesen werden. Da der Benutzer nichts von Zeichenkodierungen weiß, kann diese Information auch nicht abgefragt werden. Hier verwendet *JVS2* eine externe Bibliothek (ICU4J[18]) um die Kodierung der CSV-Datei zu ermitteln und korrekt zu öffnen.

Dadurch sollte es in Zukunft auf keiner Ebene des Programms zu Kodierungsproblemen kommen können.

3. Entwurf

In diesem Kapitel werden die Entwurfsentscheidungen dieser Diplomarbeit aufgeführt und erläutert. Dabei wird der Entwurf der graphischen Oberfläche separat behandelt, da sich die Arten der Entscheidungen hier fundamental unterscheiden.

3.1. Architekturentwurf

Zur Erstellung des UML-Entwurfs wurde Visual Paradigm for UML Community Edition[15] verwendet. Mit diesem Programm hatte der Entwickler gute Erfahrungen unter Linux, was Vollständigkeit der UML-Unterstützung und die Bedienung betraft. Andere Programme wurden auch in Betracht gezogen, jedoch war deren Bedienung zu schwerfällig, ihnen hat eine vollständige UML-Unterstützung gefehlt oder sie waren nicht kostenfrei verfügbar.

Die erste Implementierung fand auf Basis dieses Entwurfs statt. Jedoch musste aufgrund von Kundenwünschen und Designschwächen dieser Entwurf oft und umfassend verändert werden, sodass die aktuell vorliegende Software nur noch in Grundzügen diesem Entwurf gleicht. Aus diesem Grund wird der erste auf UML basierende Entwurf hier nicht aufgeführt.

3.1.1. Entwurfsmuster

Für *JVS2* wurde MVP[12] als Architekturmuster gewählt. Es ist von MVC[11] abgeleitet, trennt aber strikter zwischen View und Presenter und erlaubt eine engere Kopplung zwischen Model und Presenter. Diese Eigenschaft erlaubt es, weitreichende Modultests des Presenters zu erstellen, wenn eine Dummy-View bereitgestellt wird, die vordefinierte Daten liefert.

Die engere Kopplung zwischen Model und Presenter erlaubt es außerdem, viele Interfaces einzusparen und somit Komplexität zu vermeiden. Auf der anderen Seite erlaubt die Entkopplung von Presenter und View einfachere Programmzustände, da der Großteil des Zustands in der View zu finden ist. Der Presenter ist lediglich für Zustandsübergänge verantwortlich, die zum größten Teil in zwei Teile aufgeteilt werden können: die Behandlung und Prüfung der eingegebenen Daten, gefolgt von der Initialisierung und Anzeige der nächsten Oberfläche. Das Datenmodell ist zustandslos.

An manchen Stellen müssen Vorgaben des Architekturmusters ignoriert werden, um die Übersichtlichkeit des Programms zu wahren. Bei der Implementierung wurden diese Fälle individuell betrachtet und entschieden.

Um die View vom Presenter zu entkoppeln, wurde das „Abstract Factory“-Pattern[1] verwendet. Hierbei wird die erste View genutzt, die gefunden wird. Da die Anwendung gezielt für einen Kunden entwickelt wird, ist nicht zu erwarten, dass sich im Betrieb mehr als eine View finden wird. Diese Methode erlaubt jedoch, andere Views zu Testzwecken bereitzustellen, oder in Zukunft mit geringem Aufwand eine modernere View zu entwickeln und bereitzustellen.

MVP gibt eine grobe Trennung der Programmkomponenten vor, weshalb die Anwendung in drei Haupt-Unterprojekte aufgeteilt wurde: „model“, „view“ und „presenter“. Ein viertes Unterprojekt – „utils“ – wurde vorgesehen, um nicht-anwendungspezifischen, allgemein nützlichen Code bereitzustellen.

Um Hauptkomponenten des Programms voneinander abzugrenzen und hervorzuheben, wurden diese Unterprojekte in Pakete aufgeteilt. Interfaces für bestimmte Pakete werden in Unterpaketen mit dem Namen „`interfaces`“ bereitgestellt.

Im folgenden wird die Paketstruktur genauer erläutert.

model

Die Paketstruktur für das `model`-Unterprojekt sieht wie folgt aus (`interfaces`-Pakete sind der Übersichtlichkeit halber nicht aufgeführt):

- `constraints`
  Enthält Constraint-Implementierungen.
  - `factories`
    Enthält die Factories, die die Standard-Constraints bereitstellen.
- `school`
  Enthält die Implementierungen der verschiedenen Schularten.
- `serialisation`
  Enthält Implementierungen von Datei Ein-/Ausgaben.
- `time`
  Enthält verschiedene Zeit-Repräsentationen.

Das Model enthält die Geschäftslogik, und insbesondere die Implementierung des Speicherns und Ladens von Dateien. Dadurch können Erweiterungen und Änderungen, die nur die Logik betreffen, komplett im Model stattfinden. So braucht das Hinzufügen neuer Constraints keine projektspezifischen Abhängigkeiten außerhalb des Models.

view

Die Paketstruktur für das `view`-Unterprojekt sieht wie folgt aus (`interfaces`-Pakete sind der Übersichtlichkeit halber nicht aufgeführt):

- `adapters`
  Hier werden die Constraint-`Adapters` implementiert, wie unten in „3.1.2 Constraints: 3. Entwurf“ beschrieben.
- `main`
  Hier werden die Haupt-Views implementiert.
  - `panels`
    Dieses Unterpaket enthält die Panels, die von den Haupt-Views angezeigt werden.
- `message`
  Implementiert die graphischen Oberflächen für Benachrichtigungsdialoge.
- `util`
  Enthält Klassen, die in der gesamten View verwendet werden.

Die View ist komplett vom Rest des Projekts entkoppelt. Dadurch kann sie leicht komplett oder teilweise ausgetauscht werden. Sie wurde jedoch auf Swing hin konzipiert, so dass in nicht Swing-basierten Implementierungen einige Funktionen und Parameter durch Dummy-Implementierungen bereitgestellt werden müssen. Daraus folgt, dass die Kommunikation zur View gegen `null` und andere unerwartete Werte abgesichert werden muss.

presenter

Die Paketstruktur für das `presenter`-Unterprojekt sieht wie folgt aus (`interfaces`-Pakete sind der Übersichtlichkeit halber nicht aufgeführt):

- `constraints`
  Enthält die Programmlogik zum Auffinden und Zugänglich machen der Constraints.
- `listeners`
  Listeners, die auf Events der View hören.
- `phase`
  Implementiert die Steuerung der Phasen, die einzelne Arbeitsschritte im Leitsystem darstellen.

Im aktuellen Entwurf sind die Listener Swing-spezifisch, und können nicht unbedingt auf andere GUI-Frameworks angewendet werden. Dies stellt für die aktuelle Implementierung kein Problem dar, da sie vollständig auf Swing basiert. Der Vorteil dieses Entwurfs besteht in der Wiederverwendung des Codes, wobei Zeit eingespart und die Wartbarkeit erhöht wird, indem bekannte Konstrukte verwendet werden. In Zukunft sollte jedoch überlegt werden, ob eine Abstraktion sinnvoll sein könnte.

Der Einstiegspunkt der Anwendung befindet sich im Presenter, speziell in der `JVS2`-Java-Klasse. Diese Klasse enthält einige statische Methoden und Felder, die den Zustand der laufenden Anwendung enthalten. Dies erleichtert den Zugriff auf den aktuellen Zustand durch beliebige Bereiche des Presenters, da sie zentral und einfach zugänglich implementiert sind. Dieser Entwurf das potentielle Problem, dass keine zwei Anwendungen gleichzeitig in der gleichen JVM aktiv sein dürfen, da sie auf die selben Zustandsinformationen zugreifen und sich so gegenseitig beeinflussen würden. Da jedoch pro Anwendung immer eine eigene JVM gestartet wird, wird dies in der Praxis kein Problem darstellen.

utils

Das `utils`-Unterprojekt enthält nur ein Paket namens `xml`. Darin befinden sich Hilfsklassen für die (De-)Serialisierung. Das Hauptpaket dieses Unterprojekts enthält die eigentlichen Hilfsklassen, die verschiedene Funktionen implementieren, die im gesamten Projekt von Nutzen sind und auch in anderen Projekten verwendet werden können.

3.1.2. Constraints

Der Entwurf der Constraint-Unterstützung war eine der großen Herausforderungen dieses Projekts. Folgende Eigenschaften wurden gefordert:

- Constraints müssen an- und abschaltbar sein.
- Es sollte leicht sein, neue Constraints hinzuzufügen.
- Es gibt die Schweregrade „kritisch“, „Fehler“, „Warnung“ und „Information“.
- Constraints sollten zentral implementiert sein, nicht über den kompletten Code verteilt.
- Einige Constraints müssen während der Eingabe geprüft werden können.
- Die Verletzungen von Constraints muss in verschiedene Schweregrade unterteilbar sein.
- Verletzte Constraints müssen dem Benutzer den Grund der Verletzung mitteilen und ihm eine Möglichkeit kommunizieren können, wie die Verletzung behoben werden kann.
- Die Verletzung eines Constraints muss für einzelne Objekte ignoriert werden können, sofern die Verletzung nicht vom Schweregrad „kritisch“ ist. So soll der Benutzer beispielsweise eine Warnung für eine bestimmte JVS endgültig abschalten können.

Die Constraint-Unterstützung durchlief drei Entwurfs-Iterationen, bis ein Entwurf entwickelt war, der alle geforderten Eigenschaften abdeckte.

Constraints: 1. Entwurf

Der erste Entwurf unterschied nicht zwischen Constraints, deren Resultate interaktiv während der Eingabe geprüft, angezeigt und aktualisiert wurden („interaktive“ oder „ondemand“-Constraints) und solchen, die komplexere Verhältnisse prüften und erst bei der Bestätigung einer Eingabe aufgerufen wurden („normale“ Constraints). Dieses Problem machte den Entwurf unbrauchbar und und hatte zur Folge, dass die Kommunikation der Resultate nicht berücksichtigt wurde.

Ein Element des ersten Entwurfs wurde jedoch in den folgenden Iterationen übernommen: die „`ConstraintRegistry`“. Diese Klasse sollte darüber Buch führen, welche Constraints aktiv waren, bzw. für welche Objekte sie deaktiviert wurden. Leider wurde beim Entwurf dieser Klasse die Serialisierung nicht berücksichtigt, so dass diese Einstellungen im aktuellen Entwurf nicht gespeichert werden. Der Kunde akzeptierte diese Einschränkung, da sie Komplexität bei der Constraint-Verwaltung vermeidet: Werden bei einer Ausführung zu viele Constraints abgeschaltet, muss das Programm nur neu gestartet werden, um den Ursprungszustand wieder herzustellen.

Der Hauptgrund für den Entwurf der `ConstraintRegistry` war die Eigenschaft von *JVS Planung*, in vielen Situationen mehrfach redundant zu warnen. Die *ConstraintRegistry* kann verwendet werden, um bestimmte Constraints für bestimmte Objekte zu deaktivieren. So kann konzeptionell eine Warnung deaktiviert werden, die für eine Schulklasse bereits angezeigt wurde. Somit wird diese Warnung für diese Schulklasse in Zukunft nicht mehr angezeigt, was die Möglichkeit schafft, die Verbosität des Programms zu verringern.

Constraints: 2. Entwurf

Der zweite Entwurf unterschied bereits zwischen interaktiven und normalen Constraints. Allerdings sah er vor, dass interaktive Constraints mit der View registriert würden. Außerdem war eine direkte Kommunikation zwischen Model und View vorgesehen, was dem MVP-Pattern widerspricht und erfordert hätte, dass die View das Model kennt. Dieses kritische Problem wurde schnell erkannt und bewirkte den frühzeitigen Abbruch der weiteren Planung.

Der zweite Entwurf sah auch vor, Javas `InputVerifier`[1] zur interaktiven Constraint-Validierung zu verwenden. Leider kann dieser Mechanismus jedoch nur zwischen „korrekt“ und „inkorrekt“ unterscheiden, so dass er den Anforderungen nicht entsprach. Außerdem kann jeder Komponente nur ein `InputVerifier` zugeordnet werden. Dies hätte es erforderlich gemacht, eine weitere Abstraktionsebene einzuführen, um mehrere Constraints an eine Komponente zu binden.

[1] siehe http://docs.oracle.com/javase/7/docs/api/javax/swing/InputVerifier.html

Constraints: 3. Entwurf

An dieser Stelle wurde offensichtlich, dass eine weitere Recherche sinnvoll wäre. Bei dieser Recherche wurde „JGoodies Validation“[6] gefunden, welches eine interaktive Prüfung von Eingabedaten ermöglicht. Diese interaktive Prüfung ist sehr vollständig und beinhaltet die Möglichkeit, Eingabefelder asynchron zu prüfen.

Vom Aufbau her gibt es die Möglichkeit, Constraints zentral zu definieren, und mittels Validators zu komplexen Gebilden zu verbinden. So kann man angeben, dass nur einer von mehreren Constraints validieren muss, um eine Prüfung erfolgreich zu beenden (OR-Verknüpfung). Andere logische Verknüpfungen waren auch verwendbar. Leider gab es keine Möglichkeit, nicht-interaktive Constraints zu definieren, und auch die Dokumentation war nicht mehr zugänglich, da die Webseiten-Verknüpfungen nicht mehr gültig waren.

Validierungsergebnisse werden in JGoodies Validation nicht durch einfache boolesche Werte übertragen, sondern mittels spezieller Validierungsergebnis-Objekte, die auch einen Schweregrad angeben konnten. Leider waren nur die Schweregrade „Fehler“, „Warnung“ und „Information“ verfügbar. *JVS2* benötigte aber zusätzlich den Schweregrad „kritisch“.

Die von JGoodies Validation gebotene Komplexität wurde in *JVS2* nicht benötigt, dafür aber die Möglichkeit, normale Constraints zu implementieren. Das Konzept der Validators war für die Implementierung der interaktiven Constraints jedoch sehr sinnvoll, und auch die Validierungsergebnis-Klasse hatte deutliche Vorteile. Aus diesen Gründen wurde entschieden, zwar eine eigene Implementierung der Constraint-Validierung bereitzustellen, deren Entwurf jedoch stellenweise an JGoodies Validation anzulehnen.

Der endgültige Entwurf unterscheidet auf oberster Ebene nicht zwischen interaktiven und normalen Constraints. Dies erlaubt es, beide Constraintarten von einer `ConstraintRegistry` zu verwalten, die die (De-)Aktivierung von Constraints auf globaler Ebene erlaubt. Dies vereinfacht die Constraint-Verwaltung für den Benutzer. Auf dieser Ebene werden Eigenschaften definiert, die interaktive und normale Constraints teilen. Das sind `ID`, `Name`, `Beschreibung` und `Priorität`. Die Priorität ist eine Ganzzahl, für die gilt: je höher die Zahl desto höher die Priorität. Sie wird verwendet, um die Reihenfolge der Constraints bei ihrer Ausführung festzulegen.

Bei der Prüfung wird primär zwischen interaktiven und normalen Constraints unterschieden. Interaktive Constraints werden bei der Initialisierung der Anwendung in `Executors` zusammengefasst, die die tatsächliche Eingabe-Validierung verwalten. Diese `Executors` werden mittels eines `Bindings` an ein oder mehrere Oberflächenelemente gebunden. `Bindings` verbinden dabei mehrere einfache Eingabe-Datentypen zu einem komplexen Datentyp, der validiert wird. So wird es beispielsweise ermöglicht, zwei Datumseingaben zu einem Zeitraum zu verbinden, oder auch nur eine String-Eingabe vor der Validierung in einen anderen Datentyp zu parsen.

Die konkrete Überwachung der Oberflächenelemente ist großteils elementspezifisch. So werden für die Überwachung von Texteingaben andere Listener benötigt als für die Überwachung einer Liste. Um die Anzahl der nötigen `Bindings` nicht durch verschiedene Varianten

für verschiedene Listener aufzublähen, wurden `Adapters` als zusätzliche Abstraktionsebene eingeführt. Diese `Adapters` abstrahieren die verschiedenen Listener-Varianten und bieten den `Bindings` ein einheitliches Interface um auf Eingaben zu hören.

Der Aufbau einer Datensatzprüfung ist in Abbildung 3.1 dargestellt. Zu beachten ist hier, dass `Adapters` mehreren `Bindings` zugeordnet werden können. Dies erlaubt die Zuordnung eines Eingabefeldes zu mehreren Prüfungen.

Ein Beispiel hierfür ist die Eingabe des Startdatums eines Ferienzeitraums. Das Datum selbst muss gültig sein, der Zeitraum von Start- bis Enddatum muss gültig sein, und der Zeitraum vom Enddatum der vorherigen Ferien bis zum Startdatum des aktuellen Ferienzeitraums muss ebenfalls gültig sein. Somit ist dieses Startdatum an drei Prüfungen beteiligt und dem entsprechenden `Adapter` sollten drei `Bindings` zugewiesen werden.

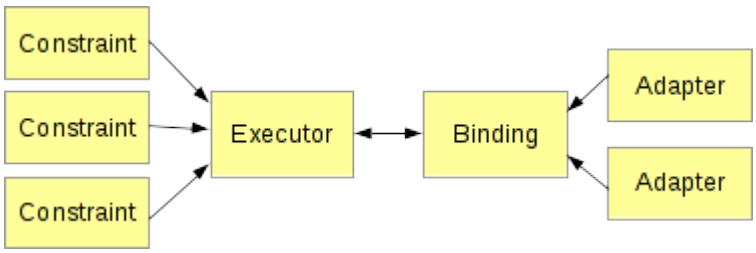


Abbildung 3.1.: Interaktive Constraints: Zuweisung der an der Prüfung eines Datensatzes beteiligten Elemente.

Der Arbeitsablauf einer Eingabeänderung sieht in diesem Modell wie folgt aus:

1. Ein `Adapter` ermittelt eine Eingabeänderung, normalerweise indem er als Listener ein Event abfängt.
2. Der `Adapter` kommuniziert den neuen Feldwert an die ihm zugewiesenen `Bindings`.
3. Jedes `Binding` verwendet den neuen Wert, um den zu validierenden Datensatz zu aktualisieren.
4. Jedes `Binding` kommuniziert dem ihm zugewiesenen `Executor`, dass geänderte Daten vorliegen.
5. Der `Executor` entscheidet, wann er den neuen Wert prüfen will und fragt zu diesem Zeitpunkt das `Binding` nach dem aktuellen Wert seines Datensatzes.
6. Der `Executor` iteriert über die ihm zugewiesenen interaktiven Constraints, wobei erst nach Schweregrad und dann nach Priorität sortiert wird. Je nach `Executor` wird entweder nach dem ersten Fehlschlag abgebrochen oder alle Constraints werden geprüft.
7. Der `Executor` teilt dem `Binding` das Resultat der Prüfung mit.
8. Das `Binding` teilt allen seinen `Adapters` das Resultat der Prüfung mit.

9. Die `Adapters` visualisieren das Resultat auf eine Weise, die zu dem überwachten Oberflächenelement passt.

Dieses Modell hat den Vorteil, dass die Prüfung bei Bedarf asynchron verlaufen kann und dass die Komplexität auf jeder Ebene minimiert wird, ohne Flexibilität einzubüßen.

Interaktive Constraints müssen sehr schnell reagieren. Konkret bedeutet das, dass die maximale Ausführungszeit aller Constraints eines Eingabefeldes unter 0,1 s liegen sollte. Um den Entwickler zu einer entsprechenden Implementierung zu ermutigen, wird bei den Prüfungergebnissen interaktiver Constraints nur zwischen „gültig“ und „ungültig“ unterschieden. Somit kann der Entwickler keine komplexen Ergebnisse ausdrücken und ist nicht versucht, mehr als ein Resultat pro Prüfung zu liefern.

Normale Constraints sind einfacher entworfen. Für sie existiert keine direkte Interaktion mit der View. Dadurch konnte das Konzept der `Executors` komplett ausgeschlossen werden, was die Komplexität stark verringert. Normale Constraints konzentrieren sich darauf, die Eigenschaften eines konkreten Objekts zu prüfen. Konzeptionell werden sie immer dann ausgeführt, wenn ein Objekt vollständig definiert wurde. In der Realität ist dieses Konzept nicht durchführbar, da der vorgegebene Arbeitsablauf bestimmte Klassen in mehreren Arbeitsschritten mit Daten füllt. Somit müssen an manchen Stellen Teilobjekte geprüft werden können.

Um diesen Gegebenheiten zu entsprechen, stellen alle normalen Constraints eine Methode bereit, die Eigenschaften eines kompletten Objekts zu prüfen und mehrere Methoden, um die einzelnen Eigenschaften individuell zu validieren.

Da normale Constraints nicht an bestimmte Oberflächenelemente gebunden sind, werden sie nicht während der Programminitialisierung einmal erstellt, sondern bei jeder Prüfung erneut abgerufen und direkt ausgeführt. Da sie immer am Ende eines Arbeitsschritts aufgerufen werden, müssen sie nicht so schnell reagieren wie interaktive Constraints. Hier ist es kein Problem, wenn die Ausführungszeit 2 - 3 s beträgt. Dies erlaubt komplexere Prüfungen.

Da der Benutzer an dieser Stelle nicht bei jeder Prüfung mit einem neuen Problem konfrontiert werden soll, ist es hier angebracht, alle vorhandenen Probleme zu ermitteln und dem Benutzer zu präsentieren. Um dem Benutzer an dieser Stelle irrelevante Informationen zu ersparen, werden die gefundenen Probleme nach Schweregrad gruppiert und präsentiert. Hier wurde entschieden, pro Schweregrad ein Dialogfeld zu öffnen. So kann der Benutzer entscheiden, wie er mit den schwereren Problemen umgehen will, ohne von niederen Schweregraden belästigt zu werden. Sollte er entscheiden, die Probleme des höheren Schweregrads zu ignorieren, werden die des jeweils nächsten Schweregrads angezeigt.

Um nachträglich neue Constraints nachladen zu können, wurde das „Abstract Factory“-Pattern[1] für das Auffinden der vorhandenen Constraints angewendet.

3.1.3. Speichersystem

Eine in *JVS2* vorhandene Funktionalität sollte die Möglichkeit sein, ab einem bestimmten Punkt der Planung weitere Planungschritte experimentell durchzuführen und bei einem minderwertigen Ergebnis auf einen früheren Zustand zurückgreifen zu können. Diese Funktionalität wurde aus Zeitgründen nicht explizit für *JVS2* geplant oder implementiert. Sie lässt sich jedoch emulieren, indem der Benutzer vor Experimenten einen neuen Speicherstand anlegt und mit diesem weiterarbeitet. Der Kunde ist mit dieser Möglichkeit zufrieden, da er das Verfahren von anderen Programmen – wie Microsoft Word – bereits kennt.

Die in *JVS Planung* verwendete Java-Serialisierung erfüllt nicht die Anforderungen an ein modernes Speichersystem und erschwert die Anpassung des Codes. Das neue Speichersystem soll diese Schwächen vermeiden.

Es wurden zwei Möglichkeiten ermittelt, das neue Speichersystem zu realisieren: basierend auf einer Datenbank oder auf einer XML-Repräsentation.

Datenbank

Die Verwendung einer Datenbank zur Datenverwaltung hätte eine Reihe von Vorteilen:

- Teilautomatisierte Aktualisierung des Datenbankschemas um Änderungen im Speicherformat zu reflektieren: Kleine Änderungen im Speicherformat können von gängigen Datenbank-Abstraktionsschichten ohne zusätzlichen Aufwand für Entwickler und Benutzer in die Datenbank übernommen werden.
- Beim Speichern müssen nur geänderte Daten modifiziert werden, was auch bei großen Projekten extrem schnelles Speichern ermöglicht.
- Transaktionen, um Änderungen atomar abzuarbeiten.
- Bei einigen Datenbanksystemen gehören Undo- und Checkpoint-Funktionalität zum Funktionsumfang und könnten so ohne zusätzlichen Aufwand angeboten werden.

Sie hätte aber auch den großen Nachteil, dass Datenbanken regelmäßig gewartet werden sollten, um Leistungs-Verringerung und Datenkorruption zu vermeiden. Dies kann dem Benutzer aber nicht zugetraut werden, was bedeuten würde, dass die Anwendung diese Aufgabe automatisieren müsste oder regelmäßige Wartungen durch den Entwickler durchgeführt werden müssten. Außerdem hat das Aufsetzen und Warten einer Datenbank erst bei größeren Datenmengen ein positives Preis-Leistungsverhältnis, als sie im Rahmen eines Planungsprojekts zu erwarten sind.

Die Verwendung einer Datenbank würde auch eine Möglichkeit zum Datenexport benötigen, um die Daten zwischen verschiedenen *JVS2*-Installationen teilen zu können. Dieser Datenexport würde wahrscheinlich auf XML basieren.

XML

Die Verwendung von XML zur Datenserialisierung vermeidet diese Probleme. Da nach Abschluss eines Projekts keine neuen Daten an eine XML-Datei angehängt werden, bleiben ihre Größe und die Leistung beim Zugriff auf die Daten konstant. Die zu erwartende Datenmenge (1 MB - 5 MB) eignet sich gut für eine XML-Datei, da sowohl die Serialisierung als auch die Deserialisierung sehr schnell verlaufen.

Die Vorteile von XML als Serialisierungsformat sind:

- Die Daten eines Projekts sind an einem Ort gespeichert und können beliebig gesichert und verschoben werden.
- Das Datenformat eignet sich sehr gut zur Komprimierung, so dass eine CD voraussichtlich für die Datensicherung von mehr als 100 Jahren[2] reicht[3].
- Projekte sind komplett voneinander getrennt und können sich nicht gegenseitig beeinflussen.
- Konstante Leistung nach Projektabschluss: Die Leistung verschlechtert sich nicht dadurch, dass andere Projekte auf dem gleichen Datenträger vorhanden sind.
- Geringer Overhead als bei der Verwendung einer Datenbank, da keine separate Anwendung zur Bereitstellung der Daten benötigt wird.

Da die Vorteile einer auf XML basierenden Implementierung – verglichen mit der auf einer Datenbank basierenden – überwogen, fiel die Wahl auf XML.

Als Alternativen für die Bereitstellung der XML-Serialisierungs-Implementierung wurden JAXB[4] und JiBX[7] identifiziert. Beide Projekte ermöglichen eine Abstraktion der XML-Darstellung von der tatsächlichen Implementierung. JAXB hatte jedoch den Vorteil, dass es im JRE[4] enthalten ist, von vielen Projekten eingesetzt wird und dem Entwickler bereits bekannt war.

Aus diesen Gründen fiel die Entscheidung auf JAXB.

3.1.4. Leitsystem & Phasen

Das Leitsystem ist ein aus *JVS Planung* übernommenes Konzept. Es ermöglicht dem Benutzer eine Verwendung des Programms, auch wenn er es zuvor noch nicht kannte. Hierbei wurde der typische Arbeitsablauf in eine Reihe von Arbeitsschritten aufgeteilt, die konzeptionell nacheinander abgearbeitet werden sollten. Die meisten dieser Arbeitsschritte sind nicht

[2] unkomprimierte Größe einer Projektdatei < 5 MB, Größe einer CD = 700 MB; $700 \div 5 = 140$

[3] Ein nach dem Projektabschluss durchgeführter Test offenbarte eine Dateigröße von 1,4 MB für die unkomprimierte Form der Projektdatei und eine Größe von 36 KB für eine ZIP-komprimierte Form, was diese Erwartung untermauert.

[4] Java Runtime Environment – siehe http://www.oracle.com/technetwork/java/javase/downloads/index.html

direkt voneinander abhängig, weshalb ihre Reihenfolge geändert werden könnte. Der Kunde wünscht diese Funktionalität jedoch nicht, weshalb die Reihenfolge fest vorgegeben wurde.

Das Leitsystem ermöglicht es dem Anwender, vom aktuellen Arbeitsschritt zum nächsten zu springen. Hierbei wurden folgende Arbeitsschritte identifiziert:

1. Eingabe der Projekt-Metadaten
2. Eingabe der Feriendaten
3. Eingabe der JVS-Stammdaten
4. Eingabe der geblockten Termine von JVS
5. Eingabe der Grund- und Förderschul-Stammdaten
6. Eingabe der geblockten Termine von Grund- und Förderschulen
7. Durchführung der Terminplanung
8. Generierung des Ausdrucks

Dieses Konzept wurde nicht verändert, lediglich die „Leitsystem-Übersicht“ angepasst. Hier wurde die Interaktivität erhöht, so dass der Benutzer sie verwenden kann, um zu einem beliebigen Arbeitsschritt zu springen. Diese Änderung benötigt eine Kommunikation zwischen der graphischen Oberfläche der Übersicht und dem Presenter. Sonstige Änderungen sind oberflächenspezifisch.

Das Leitsystem wird intern durch „Phasen“ repräsentiert, die den Arbeitsablauf unterteilen. Dabei sind die Phasen untereinander nur lose gekoppelt, so dass ihre Reihenfolge weitgehend anpassbar ist. Phasen haben weitgehend eine spezielle Ansicht in der Oberfläche. Sie sind nur in Presenter und View vertreten, und bieten jeweils eine individuelle Sicht auf den Datenbestand.

Phasen, die nicht in ihrer Reihenfolge verändert werden dürfen, sind die Erfassung der Projekt-Metadaten und die Generierung der Dokumente. Die Generierung der Dokumente besitzt außerdem keine spezielle Ansicht, da hier keine Daten dargestellt werden. Bei ihr handelt es sich deshalb um eine „virtuelle Phase“, die das Ende der Bearbeitung signalisiert.

3.1.5. Sprache

Der Kunde erwartet eine Anwendung mit einer deutschen Benutzeroberfläche. Somit muss mindestens deutsch verfügbar sein. Andere Sprachen könnten theoretisch auch unterstützt werden. Da es sich bei *JVS2* jedoch um ein Projekt handelt, dessen Verwendung ausschließlich in Deutschland geplant ist, wird auf eine Unterstützung weiterer Sprachen oder anders gearteter Internationalisierung verzichtet.

Da es sich bei diesem Projekt um ein deutsches Projekt mit deutscher Zielgruppe handelt, das viele Fachbegriffe enthält, die nicht leicht übersetzt werden können, wurde entschieden,

die Code-Dokumentation in deutscher Sprache zu halten. Der Code selbst soll englisch geschrieben werden, mit deutschen Begriffen, wo es sich um typisch deutsche Namen und Konzepte handelt.

3.1.6. Zeitdarstellung

JVS2 benötigt – wie auch *JVS Planung* – nur eine Darstellung des Datums und des belegten Unterrichtszeitraums. Somit ist eine Zeitdarstellung mit einer Genauigkeit von Tag, Monat und Jahr zuzüglich der Information zum Unterrichtszeitraum ausreichend. Um Kompatibilität zu verwendeten Bibliotheken zu wahren, wurde entschlossen Javas `Calendar` als Basis der Zeitdarstellung zu verwenden, genauere Informationen als das Datum jedoch zu ignorieren. Stattdessen wird über eine Enumerationsklasse namens `Slot` der gewählte Unterrichtszeitraum dargestellt.

3.2. Entwurf der graphischen Oberfläche

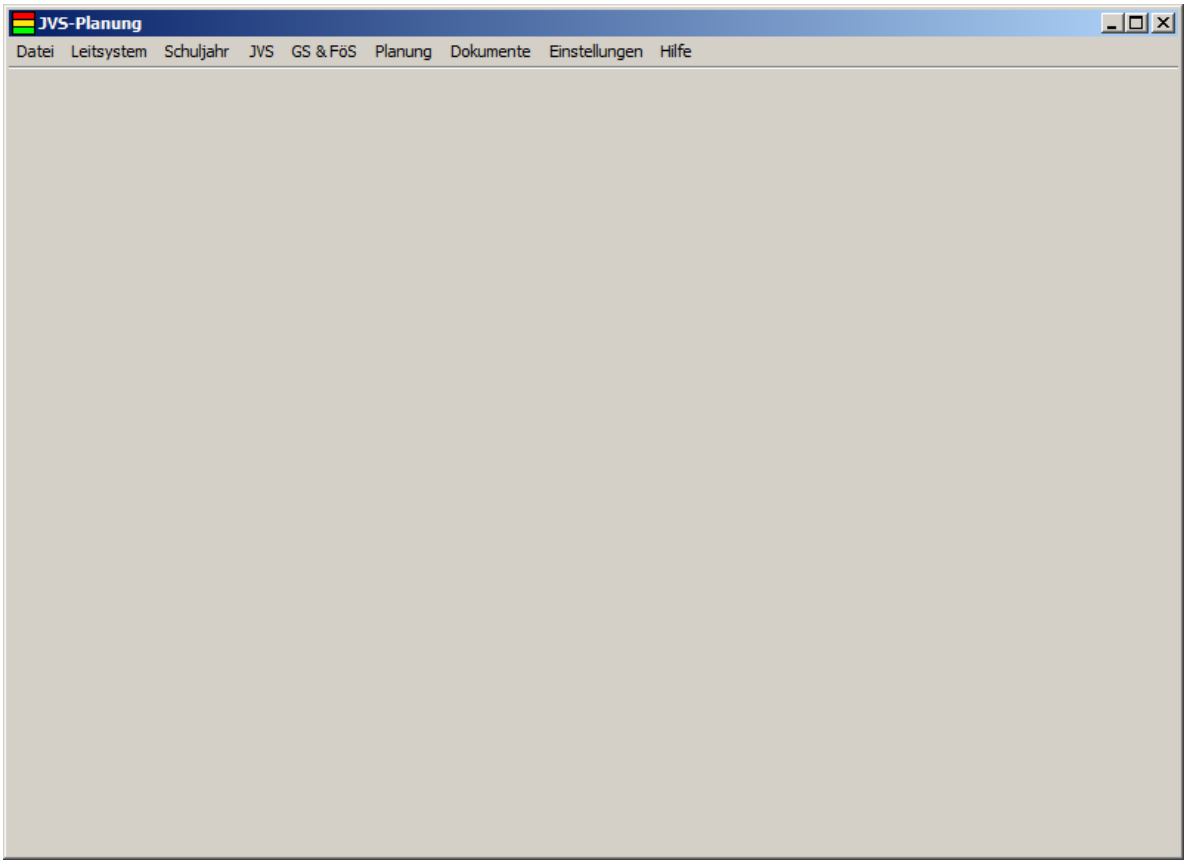


Abbildung 3.2.: *JVS Planung*: Ansicht nach Öffnen eines Projekts

In diesem Unterkapitel wird der Entwurf der graphischen Oberfläche erläutert. Wo angebracht, werden den neu entworfenen Oberflächen ihre Gegenstücke aus *JVS Planung* gegenübergestellt, um Ähnlichkeiten und Änderungen hervorzuheben. So lässt sich aus dem Vergleich von Abbildungen 3.2 und 3.3 erkennen, dass die Schriftgrößen in *JVS2* vergrößert wurden, während die Standard-Fenstergröße verringert wurde. Der Rest des Designs wurde übernommen.

Der Entwurf der graphischen Oberfläche wurde weitestgehend an *JVS Planung* angelehnt. Allerdings wurde konsequenter auf Swing gesetzt, und die interaktive Constraintprüfung berücksichtigt. An einigen Stellen wurde auch versucht, bereits vorhandene Entwurfskonzepte konsequenter einzusetzen. Im Gegensatz zu *JVS Planung* wird von *JVS2* eine minimale Fenstergröße von 700 x 400 px vorgeschrieben, da kleinere Fenster vom Layout nicht unterstützt werden. Dies verhindert, dass der Benutzer eine kleinere Fenstergröße wählt und durch Layoutprobleme gestört wird.

Abbildung 3.3.: *JVS2*: Ansicht nach Öffnen eines Projekts

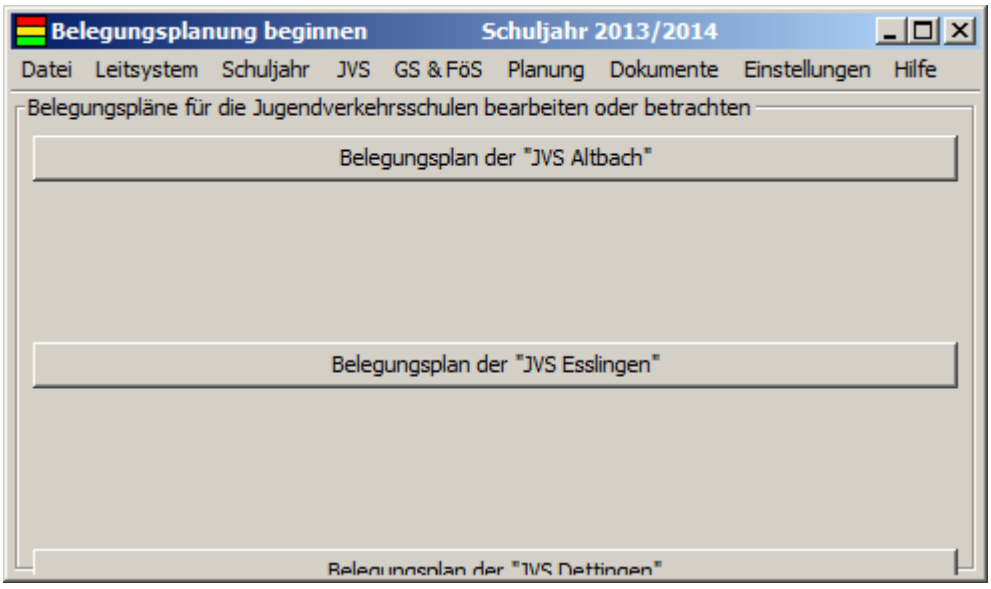


Abbildung 3.4.: *JVS Planung*: JVS-Auswahl bei kleinerem Fenster

Eine der identifizierten Schwächen von *JVS Planung* war die Tatsache, dass bei kleinen Fenstergrößen teilweise nicht mehr alle Oberflächenelemente sichtbar und erreichbar waren (siehe Abbildung 3.4). In *JVS2* sollen in diesen Situationen konsequent Scrollbalken auftauchen, damit der Benutzer immer auf alle Elemente zugreifen kann.

3.2.1. Kalender-Popup

Um eine modernere Benutzeroberfläche bereitstellen zu können, soll für Datumseingaben ein Kalender-Popup eingebunden werden. Eine Voruntersuchung identifizierte die Projekte „JCalendar“[5] (Abbildung 3.5) und „Microba controls“[9] (Abbildung 3.6) als potentielle Implementierungen dieser Funktionalität.

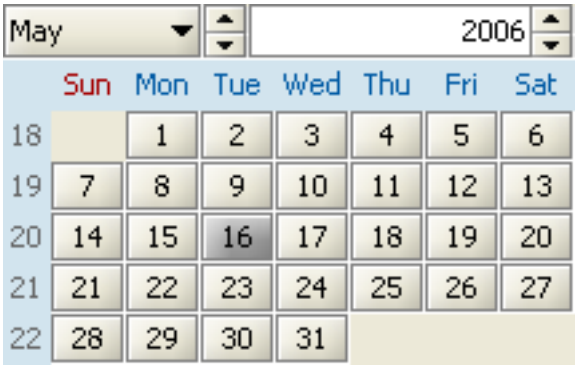


Abbildung 3.5.: JCalendar (Quelle: JCalendar-Webseite[5])

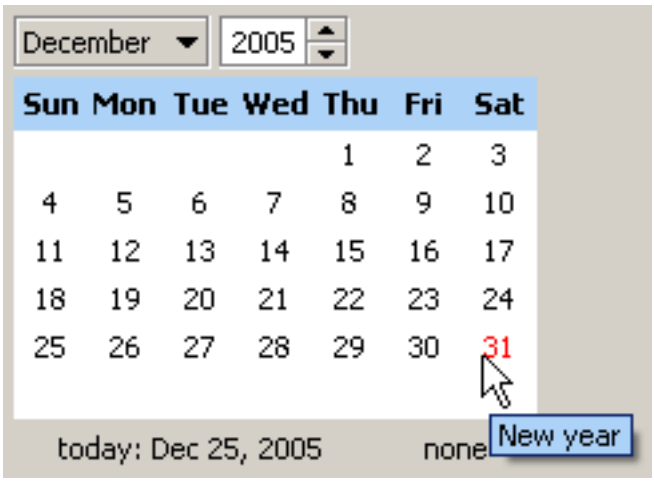


Abbildung 3.6.: Microba controls (Quelle: Microba-Screenshot-Webseite[10])

Da JCalendar für den Entwickler einfacher zu verwenden und in der Ausführung schneller sein soll[17], fiel die Wahl letztendlich auf JCalendar. Außerdem beinhaltet Microba controls eine Reihe weiterer Bedienelemente, die in *JVS2* nicht verwendet werden. Somit würde es das Programm nur unnötig vergrößern.

3.2.2. Projektstart

Zum Start des Projekts werden die Projektparameter abgefragt. Da dies in *JVS Planung* anhand einer Reihe von Dialogen geschah, wurde diese Eingabemaske komplett neu entworfen. So können nun alle relevanten Daten an einer Stelle eingegeben werden, was dem Benutzer eine bessere Übersicht ermöglicht. Es erzielt auch eine bessere Konsistenz der Bedienung. Die neue Eingabemaske ist in Abbildung 3.7 dargestellt.

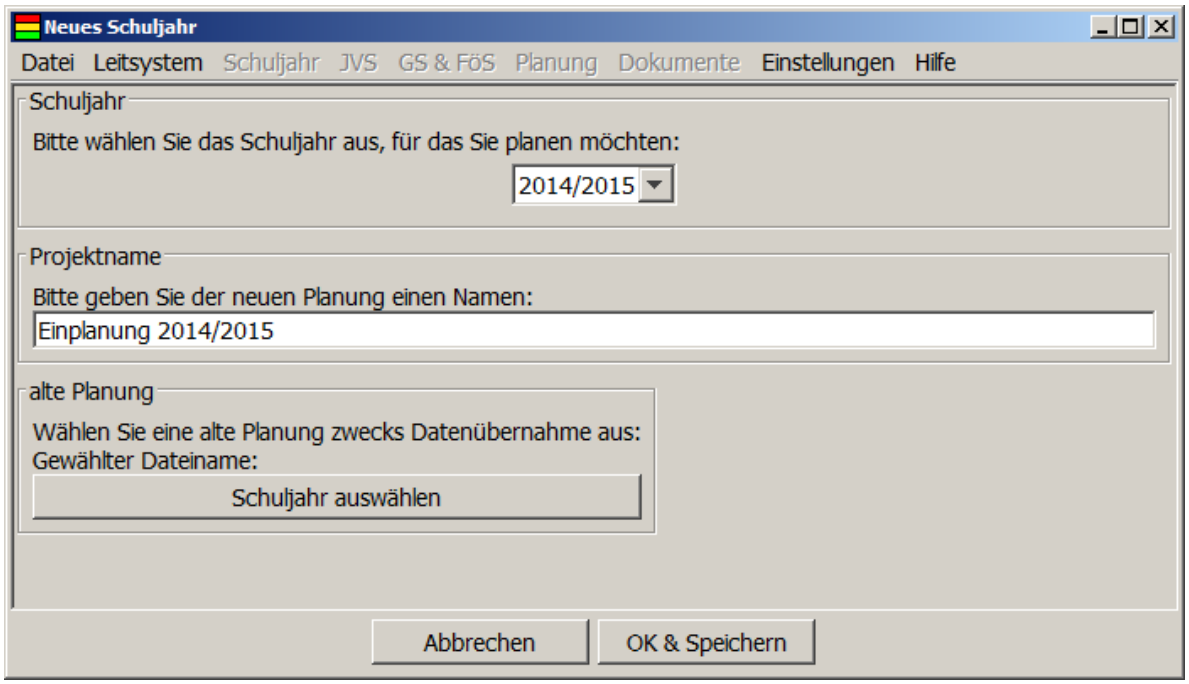


Abbildung 3.7.: Eingabe der Projektdaten

3.2.3. Ferieneingabe

Die Eingabe der Sommerferien wurde auf Kundenwunsch mit der Eingabe der restlichen Ferien in einer Maske zusammengeführt. Die Faschingsferien werden aus dem selben Grund zusätzlich erfasst. Sonst wurde die Oberfläche komplett von *JVS Planung* übernommen.

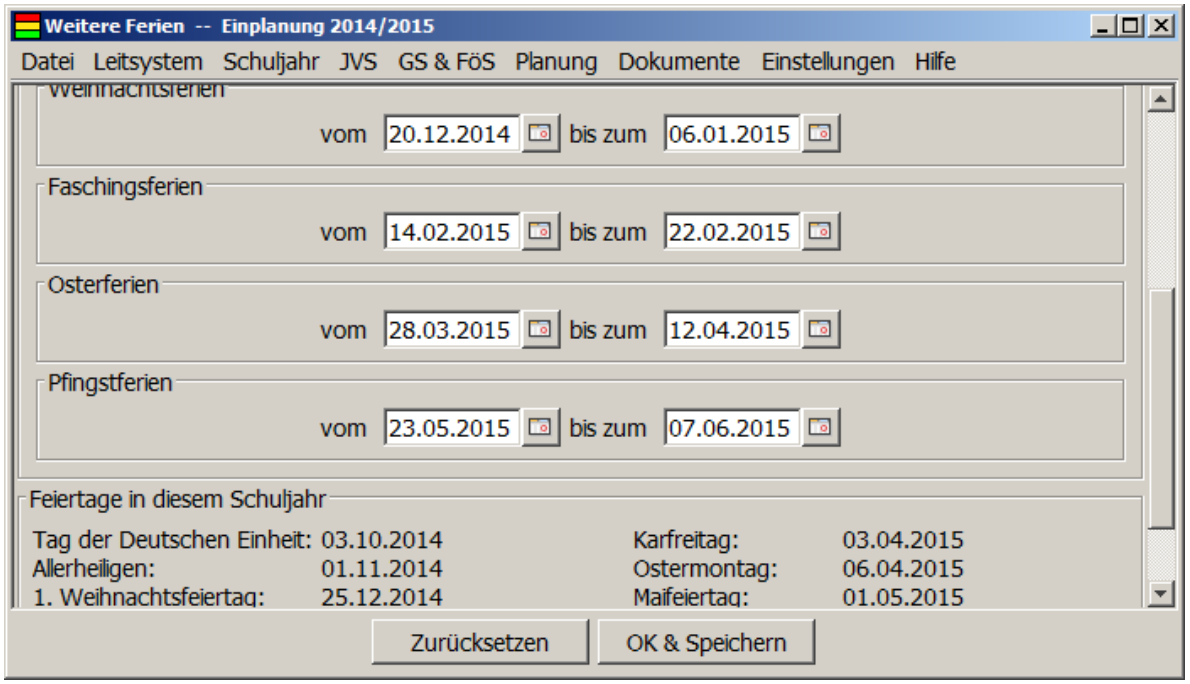


Abbildung 3.8.: Ferieneingabe und Anzeige der berechneten Ferien

3.2.4. Eingabe geblockter Termine

Die Eingabemasken für geblockte Termine wurden vereinfacht, indem die Termineingabe und -anzeige in der gleichen Maske realisiert wurde. Zusätzlich können Termine nun auch gelöscht werden.

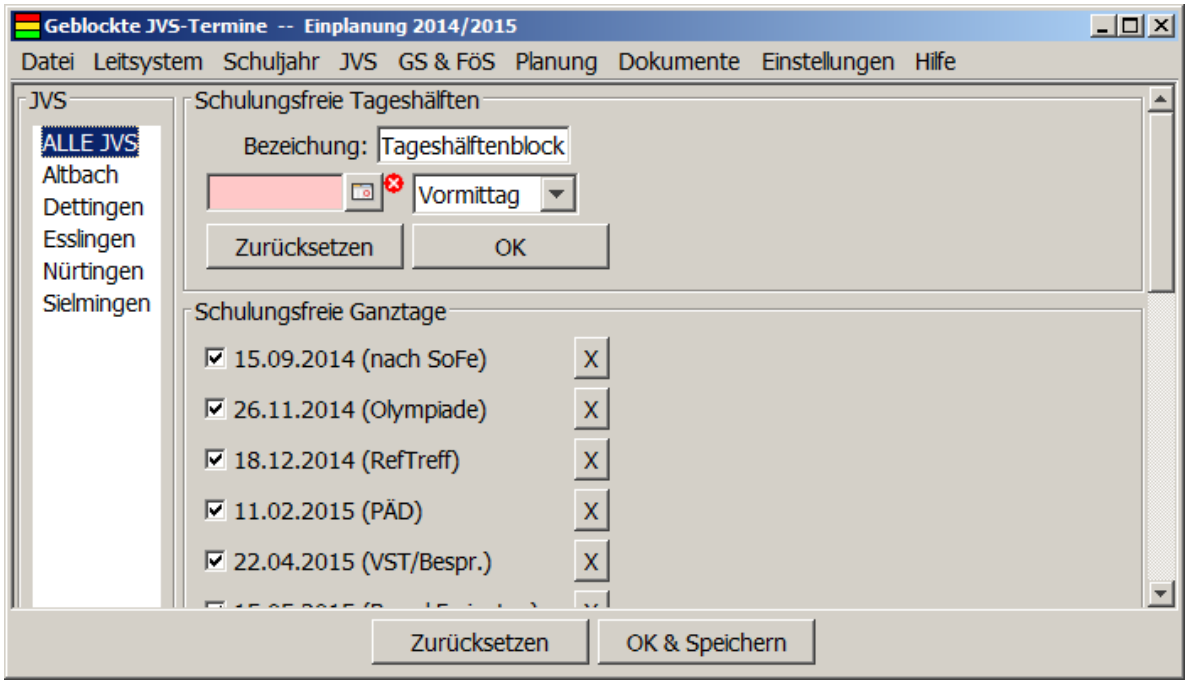


Abbildung 3.9.: JVS: Eingabe geblockter Termine

Die Ausnahme zu dieser Regel bildet die Eingabe von beweglichen Ferientagen, da diese mit anderen Schulen geteilt werden. Die Möglichkeit, bewegliche Ferientage zu löschen, würde zu viel Komplexität schaffen und deshalb wahrscheinlich zu Eingabefehlern führen. Deshalb wurde diese Funktionalität in Rücksprache mit dem Kunden an dieser Stelle deaktiviert.

Die Eingabemaske für geblockte Termine der JVS (Abbildung 3.9) wurde auch dahingehend vereinfacht, dass die JVS-Auswahl nun in der gleichen Maske stattfindet, wie die Anzeige der geblockten Termine. Dies verringert die Anzahl der ineinander verschachtelten Eingabemasken, die der Benutzer kennen muss und vereinheitlicht das Design der Anwendung.

Zum Vergleich enthalten die Abbildungen 3.10 und 3.11 die in *JVS Planung* verwendete Oberfläche, die – zusätzlich zu den geschilderten Eigenschaften – für die verschiedenen Termindarstellungen ein `GridLayout` als `LayoutManager` verwendete, was zu visuell wenig ansprechenden Darstellungen führen und die Übersichtlichkeit behindern konnte.

Abbildung 3.10.: *JVS Planung*: Eingabe geblockter JVS-Termine (1. Bildschirm)

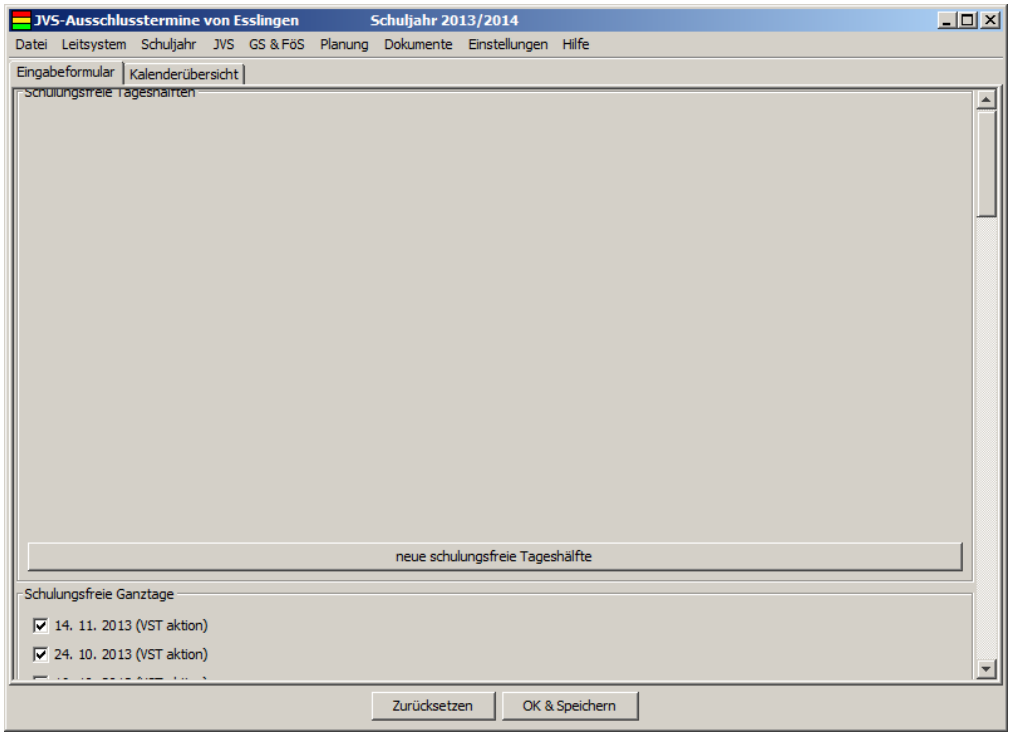


Abbildung 3.11.: *JVS Planung*: Eingabe geblockter JVS-Termine (2. Bildschirm)

3.2.5. Belegungsplanung

Schulwahl- und Detailansicht

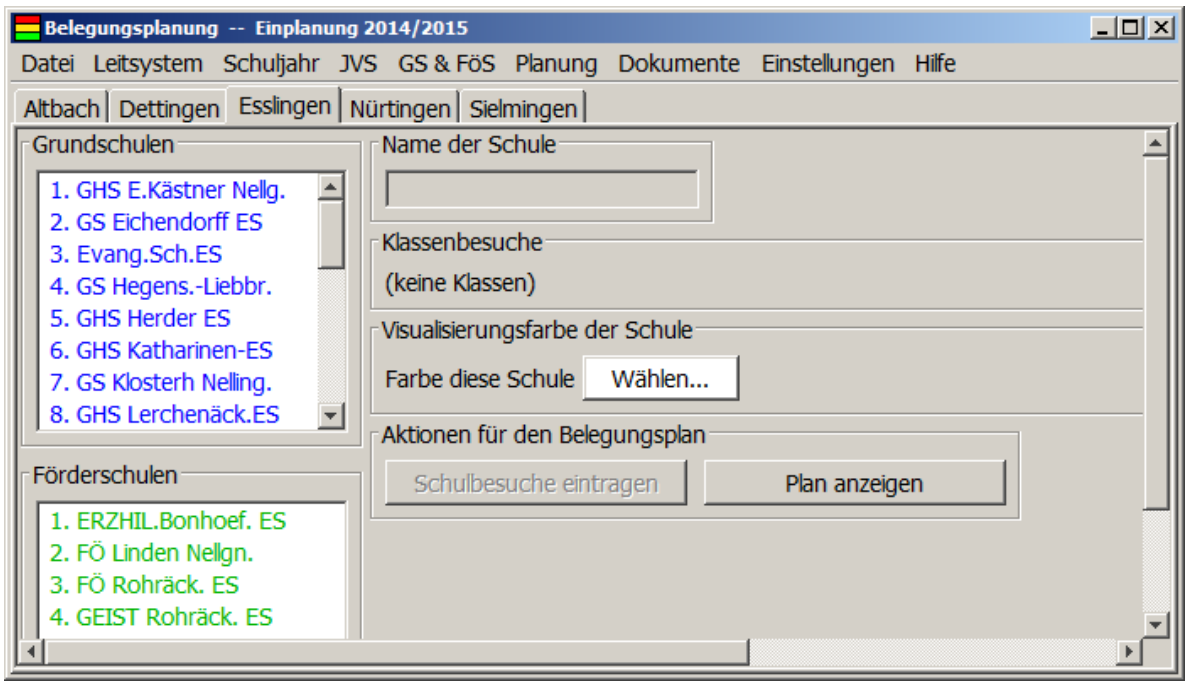


Abbildung 3.12.: Belegungsplanung: Detailansicht

Analog zu der Einsparung einer verschachtelten Eingabemaske (siehe Abbildung 3.13) bei der Eingabe geblockter JVS-Termine, wird auch bei der Belegungsplanung die Auswahl der JVS verändert. Wie in Abbildung 3.12 zu sehen ist, kann die JVS durch einen Tab am oberen Rand der Maske gewählt werden. Dieses Verfahren hat den Vorteil, dass es ein Bedienelement verwendet, das von jedem aktuellen Internet-Browser bekannt ist. Somit kann der Benutzer die Bedienung sofort intuitiv erfassen. Die Implementierung von *JVS Planung*, die Buttons für diese Funktionalität verwendete (siehe Abbildungen 3.13 und 3.14), war umständlicher und unflexibler.

Eine weitere Änderung stellt die Anzeige von lediglich zwei Auswahllisten – eine für Grund- und eine für Förderschulen – dar. Hier wurde die Unterscheidung zwischen Schulen mit und ohne verletzten Constraints eingespart, da in reellen Planungen fast alle Schulen mindestens einen Constraint verletzen. Zusätzlich wurden die Schulnamen eingefärbt, was eine einfachere Unterscheidung von Grund- und Förderschulen ermöglicht.

Die beschriebenen Änderungen werden durch den Vergleich der Abbildungen 3.14 und 3.12 verdeutlicht.

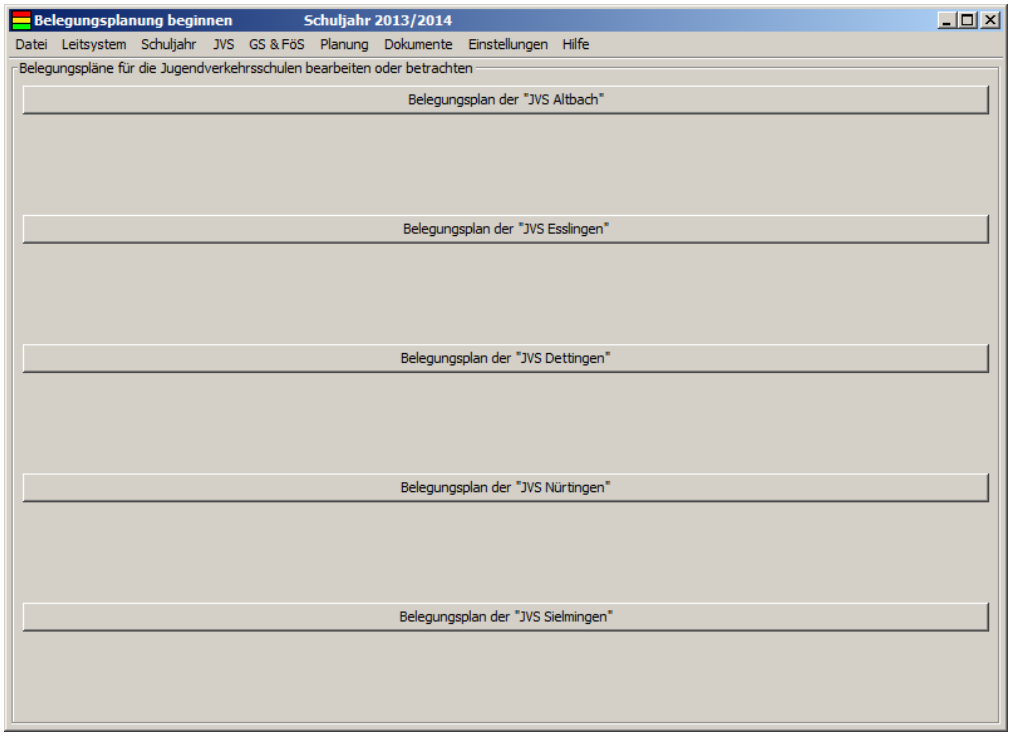


Abbildung 3.13.: *JVS Planung*: Belegungsplanung-Detailansicht (1. Bildschirm)

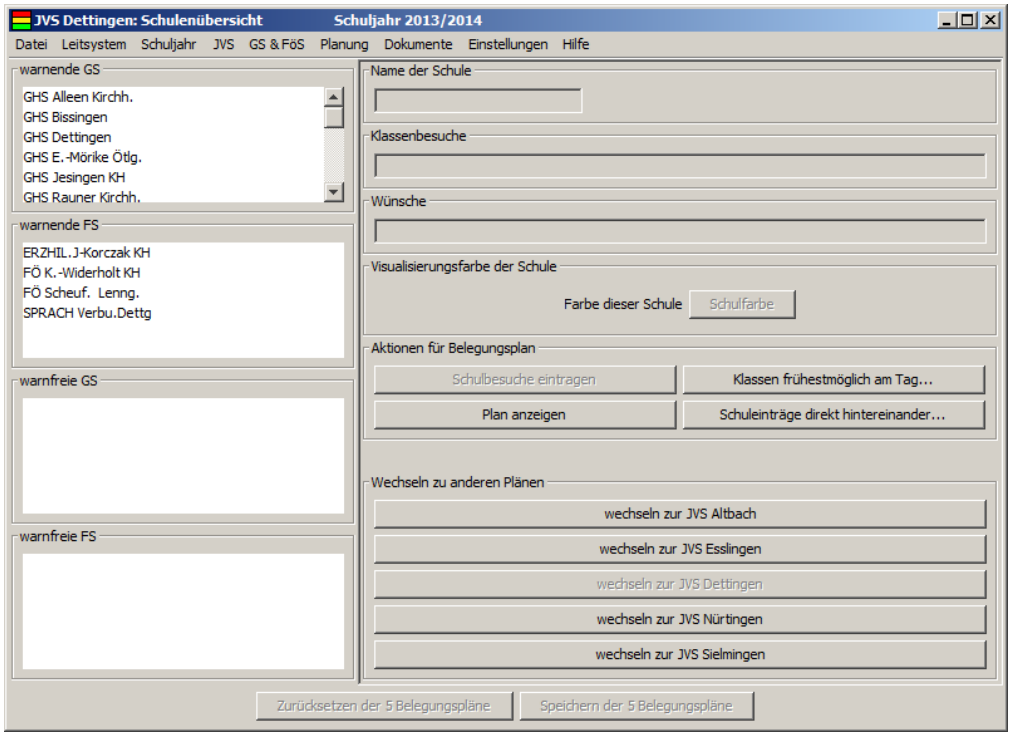


Abbildung 3.14.: *JVS Planung*: Belegungsplanung-Detailansicht (2. Bildschirm)

Wochenansicht

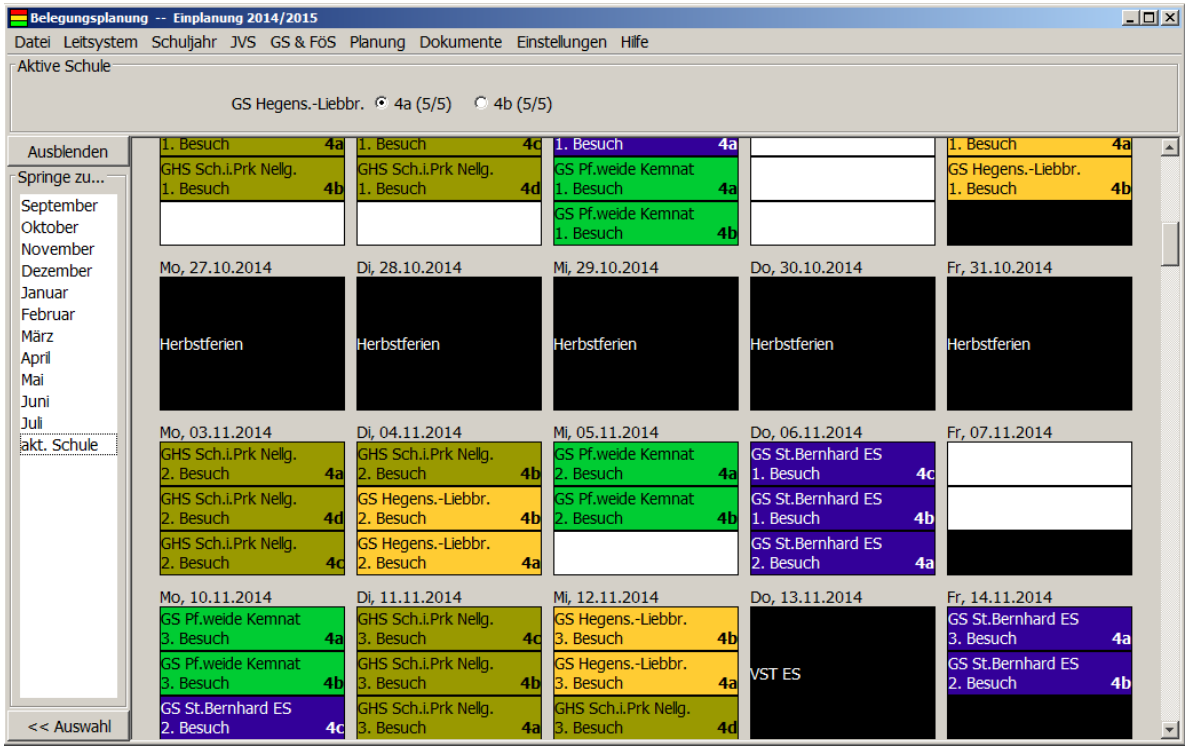


Abbildung 3.15.: Belegungsplanung: Wochenansicht

Die Wochenansicht, die für die konkrete Terminplanung vorgesehen ist, wurde minimal geändert. Hier skaliert die Größe der angezeigten Tagesdarstellungen lediglich mit der ausgewählten Schriftgröße und bei geblockten Terminen wird die Beschreibung dieser Termine angezeigt. Zusätzlich wird der Listeneintrag „akt. Schule“ (= aktive Schule) nur angezeigt, wenn aktuell eine Schule eingeplant wird und für diese Schule Termine festgelegt wurden. Die in *JVS Planung* vorhandenen Buttons zur teilautomatischen Planung wurden entfernt, da diese Funktionalität in *JVS2* entfernt wurde.

Diese Detailänderungen sind auf den ersten Blick kaum erwähnenswert, erleichtern dem Benutzer die Arbeit aber sehr, da sie Arbeitsschritte einsparen und die Übersichtlichkeit erhöhen.

3.2.6. Leitsystem

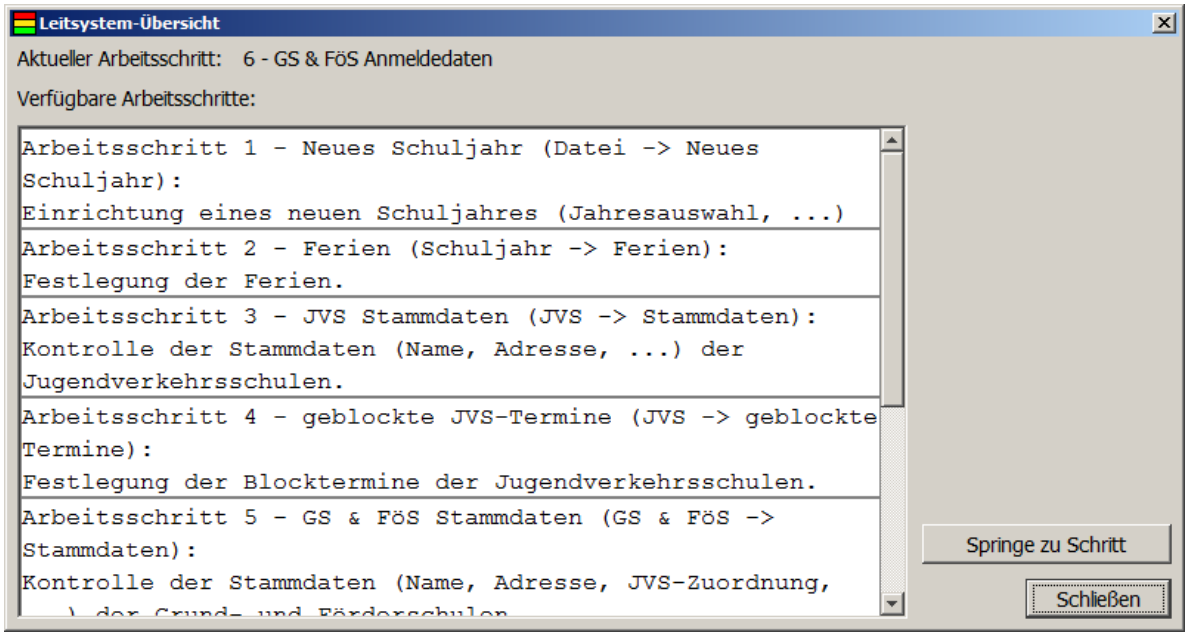


Abbildung 3.16.: *JVS2*: neue Leitsystem-Übersicht

Das Leitsystem wurde weitestgehend von *JVS Planung* übernommen. Die größte Änderung betrifft die Leitsystem-Übersicht (Abbildung 3.16). Dieser Dialog wurde komplett überarbeitet, um seine Übersichtlichkeit und seinen Nutzen zu erhöhen. In der neuen Fassung bekommt der Benutzer die Möglichkeit, einen Arbeitsschritt aus einer Liste auszuwählen und so direkt anzusteuern.

Als Vergleich wird in Abbildung 3.17 die Leitsystem-Übersicht dargestellt, wie sie in *JVS Planung* implementiert war.

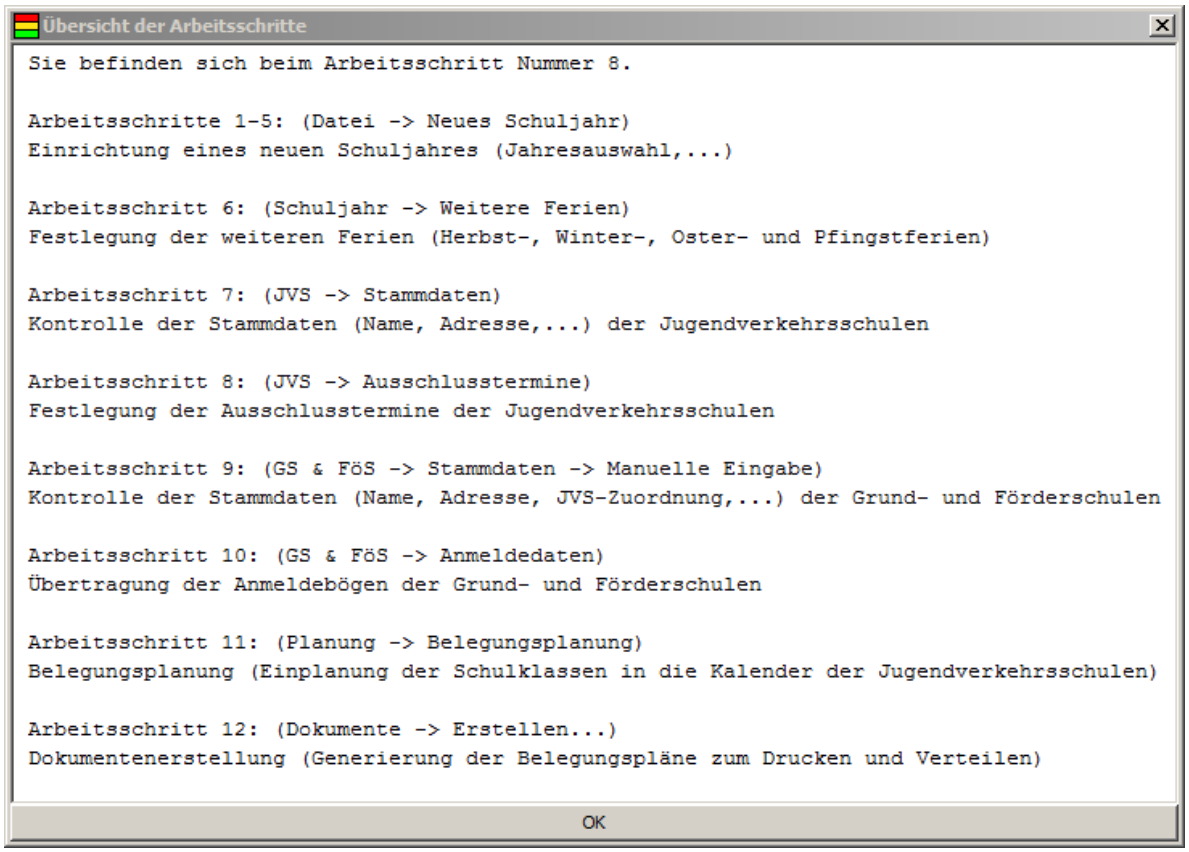


Abbildung 3.17.: *JVS Planung*: alte Leitsystem-Übersicht

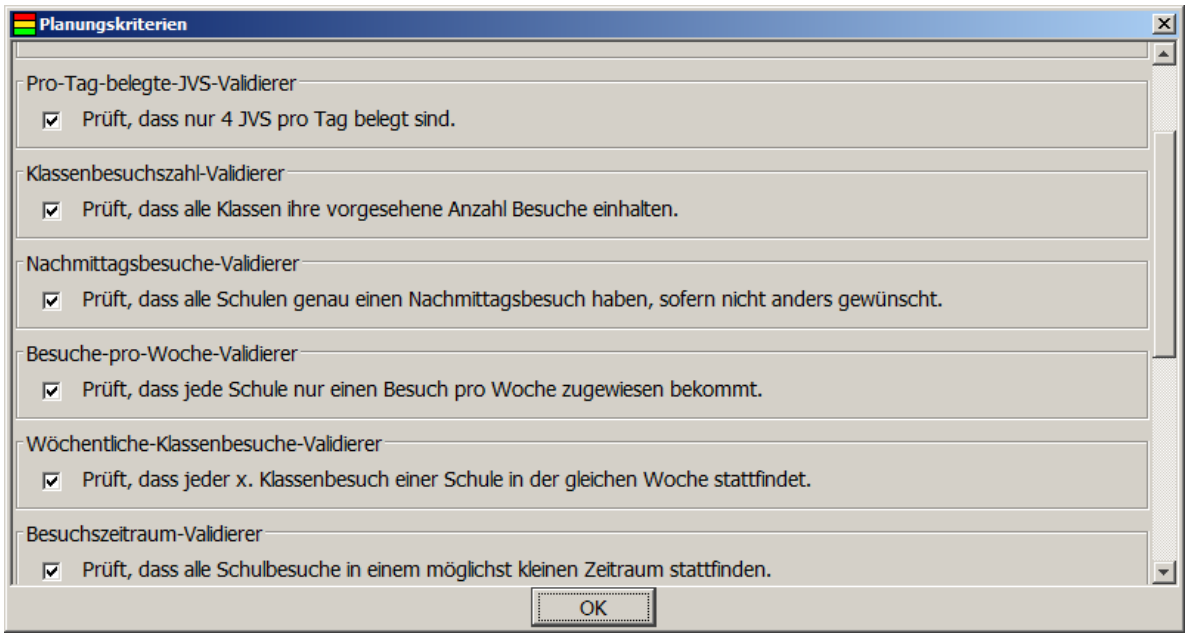


Abbildung 3.18.: Constraints: Einrichtungsdialog

3.2.7. Constraints

Der Entwurf und die Darstellung der Constraints wurden in *JVS2* komplett überarbeitet. Ein Entwurfselement, das beibehalten wurde, ist die Fähigkeit, Constraints zu (de-)aktivieren (siehe Abbildung 3.18).

Interaktive Constraints

Interaktive Constraints reagieren auf jede sie betreffende Eingabe. Hierbei wird immer der höchste Schweregrad für die Visualisierung der Verletzungsanzeige verwendet. Je nach Eingabefeld können Verletzungen anders visualisiert werden – so werden Text-Eingabefelder eingefärbt und mit einem Symbol versehen, während Knöpfe deaktiviert werden. Anschaulich dargestellt ist dies in Abbildung 3.19. Hier ist bei Eingabefeld 3.19 a ein kritischer Constraint verletzt, bei Eingabefeld 3.19 b ein wichtiger und in Eingabefeldern 3.19 c wird vor einer potentielle Fehleingabe gewarnt.

Nicht jede Visualisierung findet direkt an dem Eingabefeld statt, das die Verletzung ausgelöst hat. So kann der „OK & Speichern“-Knopf bei kritischen Verletzungen ausgegraut werden, während die Verletzung durch eines der Felder der Eingabemaske verursacht wurde. Bei Zeitraumeingaben werden die beiden am Zeitraum beteiligten Eingabefelder eingefärbt und markiert, auch wenn nur eines von ihnen eine Verletzung auslöst.

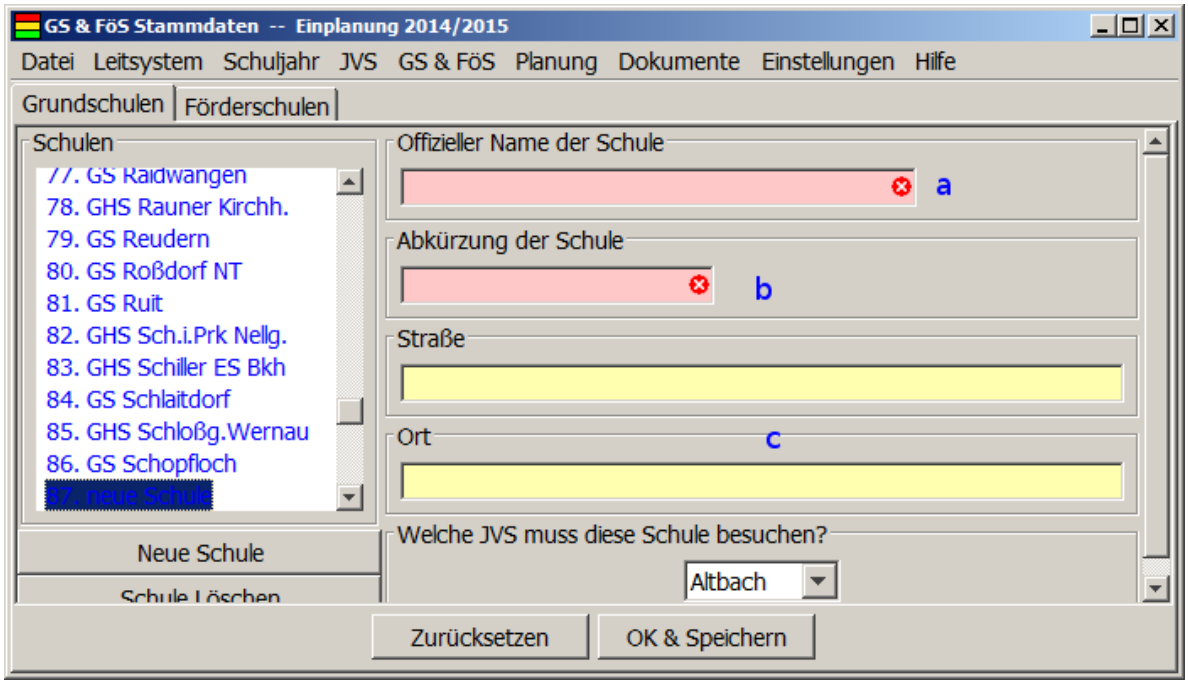


Abbildung 3.19.: Constraints: interaktive Constraints

Im Gegensatz zu *JVS Planung* wird ein Eingabefeld nicht zurückgesetzt, wenn ein Constraint verletzt wurde. Die Verletzung wird lediglich markiert.

Normale Constraints

Normale Constraints werden immer ausgeführt, wenn eine Eingabe bestätigt wird. Da bei der Belegungsplanung jede Eingabe, die den Plan ändert, gleichzeitig auch eine Bestätigung darstellt, wird in diesem Fall nach jeder solchen Eingabe geprüft.

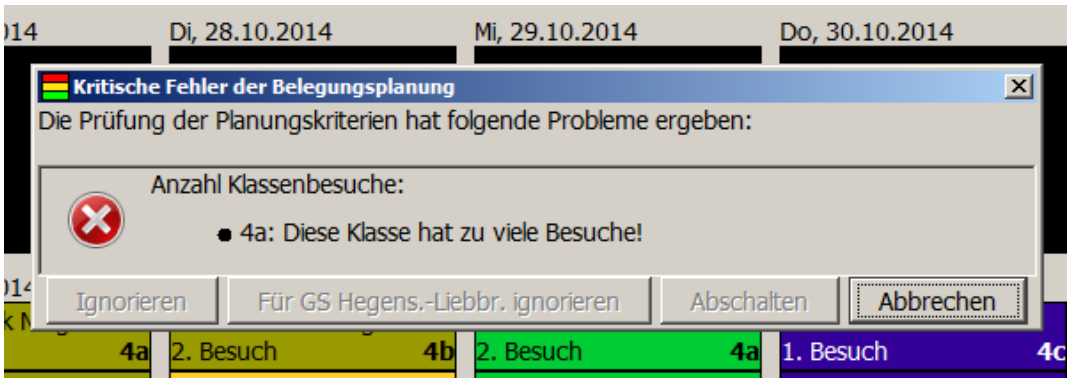


Abbildung 3.20.: Constraints: kritische Fehlermeldung

Wenn mehrere Verletzungen verschiedener Schweregrade gefunden werden, werden diese nach Schweregrad gruppiert und angezeigt. Dabei wird pro Schweregrad ein Dialog (siehe Abbildung 3.21) angezeigt, damit sich der Benutzer auf die wesentlichen Informationen konzentrieren kann. Der Benutzer kann sich dann jeweils entscheiden, die Verletzungen dieses mal, für das Objekt, das untersucht wurde, oder immer zu ignorieren. Er kann sich auch entscheiden, die Verletzung zu beheben, indem er auf den „Abbrechen“-Knopf drückt. In diesem Fall werden keine weiteren Dialoge angezeigt, und die aktuelle Aktion wird abgebrochen.

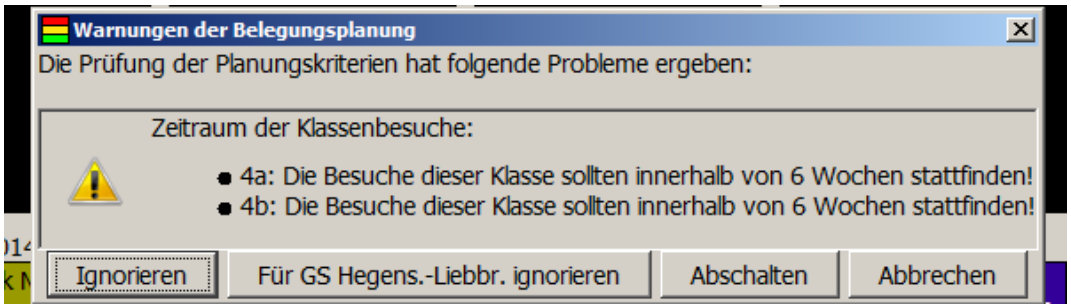


Abbildung 3.21.: Constraints: Warnungsmeldung

Bei kritischen Constraint-Verletzungen wird die Option nicht angeboten, diesen Constraint zu ignorieren (siehe Abbildung 3.20). Hier muss die Verletzung auf jeden Fall behandelt werden.

4. Implementierung

Dieses Kapitel erläutert wichtige Details der Implementierung, sowie Besonderheiten, die im Laufe der Implementierung aufgefallen sind. Es enthält viele fachliche Details und Begriffe, weshalb es stärker auf Entwickler ausgerichtet ist als andere Kapitel dieser Arbeit.

Während der Implementierung wurde das aktuelle Zwischenprodukt immer wieder dem Kunden vorgestellt, um Feedback zu erhalten und Implementierungsentscheidungen zu verifizieren.

4.1. Wahl der Programmiersprache

Es wurde entschieden, Java als Programmiersprache zu verwenden. Die Gründe hierfür waren:

- *JVS Planung* hat bereits Java eingesetzt, was eventuelle Code-Wiederverwendung vereinfacht.
- Eines der Hauptziele dieser Diplomarbeit war es, ein Programm zu erstellen, das portabel ist. Dies ist in Java einfacher als in den meisten anderen Programmiersprachen.
- Die graphische Oberfläche sollte möglichst wenige Änderungen aufweisen, was dank der Verwendung von Swing sowohl in *JVS Planung* als auch *JVS2* einfacher war.
- Java erfüllt alle Anforderungen, die sich aus der Analyse ergaben.
- In Java ist es relativ einfach, schnell kompakten und fehlerarmen Code zu produzieren.
- Der Entwickler hatte bereits viele Jahre Erfahrung mit Java-Entwicklung.
- *JVS Planung* beweist, dass Java performant genug für die Aufgabenstellung ist.

4.2. Abhängigkeitsauflösung

Zur Auflösung und Bereitstellung der Abhängigkeiten des Projekts wurde Maven[16] gewählt. Um das Projekt möglichst modular aufzubauen, die Implementierung von Constraints unter Verwendung einer minimalen Anzahl an Abhängigkeiten zu ermöglichen und die Übersichtlichkeit zu erhöhen, wurde das Projekt in mehrere Maven-Unterprojekte aufgeteilt.

Es wurde ein Basisprojekt namens „base“ erstellt, das Informationen und Abhängigkeiten definiert, die alle Unterprojekte teilen.

Die Unterprojekte „model“, „presenter“ und „view“ leiteten sich aus dem Entwurfsmuster MVP[12] ab. Zusätzlich wurden alle geteilten Interfaces in ein separates „interfaces“-Projekt abgespalten, um die Abhängigkeiten zwischen den Unterprojekten zu minimieren. Da es von den anderen Projekten komplett unabhängig ist, wurde das im Entwurf definierte „utils“-Unterprojekt nicht von „base“ abgeleitet. Schließlich wurde ein Projekt namens „integration“ definiert, welches die anderen Teilprojekte integriert und die endgültige Anwendung baut.

Da alle externen Abhängigkeiten des Projekts von Maven bereitgestellt werden, musste kein separates Projekt für externe Abhängigkeiten erstellt werden.

4.3. Allgemein

Das im Entwurf verwendete „Abstract Factory“-Pattern[1] wurde mittels Javas `ServiceLoader`[2] implementiert. Dies hatte den Vorteil, dass neue Module hinzugefügt werden konnten, ohne in den Programmcode eingreifen zu müssen. Außerdem wird dieses Framework im JRE mitgeliefert, weshalb es sehr gut getestet und überall verfügbar sein sollte.

JVS2 wurde so implementiert, dass es den ersten Kommandozeilenparameter als Projektdatei interpretiert und so beim Start direkt ein Projekt laden kann. So wird es möglich, den Dateityp der Projektdateien mit *JVS2* so zu verknüpfen, dass ein Projekt direkt aus dem Dateimanager heraus geladen wird.

In Java werden unbehandelte Exceptions standardmäßig nur an die Standard-Fehlerausgabe weitergeleitet, wo sie für den Benutzer unbemerkt bleiben können und auch nicht im Logging-System vermerkt werden. Dies hat den Effekt, dass das Programm unerklärliches und unberechenbares Verhalten aufweisen kann, ohne dass für den Benutzer ein Grund erkennbar ist.

Um dieses Problem zu adressieren, implementiert und registriert *JVS2* einen `UncaughtExceptionHandler`. Dieser protokolliert unbehandelte Exceptions im Logging-System und weist den Benutzer auf das Problem hin. Zusätzlich bittet er den Benutzer, den Entwickler zu kontaktieren und ihm alle relevanten Informationen zukommen zu lassen.

Um die vom Kunden gewünschte Sortierung der Schulen zu implementieren, wird von allen Grund- und Förderschulen das `Comparable`-Interface implementiert. Zum Vergleichen der Schulen werden ihre Abkürzungen verwendet. Hier wird im ersten Schritt des Vergleiches die Abkürzung bis einschließlich des ersten Leerzeichens ignoriert, da dieser Teil der Abkürzung die Schulart enthält. Erst wenn der Rest der Abkürzung gleich ist, wird der volle Name verglichen.

Um die Benutzeroberfläche möglichst robust zu implementieren, wurde ein Interface namens „Resettable“ definiert. Dieses Interface definiert zwei Methoden `reset()` und `resetFocus()`. Es wird von allen Hauptansichten implementiert und erlaubt es, die Ansicht vor dem Laden neuer Informationen in einen definierten Zustand zu bringen. Dies kostet zwar

Rechenleistung, vereinfacht aber den Code und erhöht so dessen Verständlichkeit. Der benötigte Rechenaufwand ist auf aktuellen Systemen vernachlässigbar gering, so dass die Vorteile dieser Implementierung überwiegen.

4.3.1. Anpassung der Schriftgröße

Um die Schriftgröße aller Schriften der Anwendung anpassen zu können, iteriert die Anwendung bei der Initialisierung über alle in Swing bereitgestellten Schriften und ändert ihre Größe um einen spezifizierten Wert. Da der Benutzer keine Anpassbarkeit zur Laufzeit wünschte, wird diese Einstellung in die Konfigurationsdatei geschrieben, ohne eine graphische Oberfläche für ihre Anpassung bereitzustellen.

4.3.2. Logging-System

Als Logging-System wurde LOGBack[8] mit SLF4J[14] als Backend eingesetzt. Dies hat den Vorteil, dass ein einheitliches Logging-System für die gesamte Anwendung bereitsteht, das ohne Eingriffe in den Code konfiguriert werden kann.

Das Logging-System hat die Funktion, dem Entwickler Details über die Verwendung der Anwendung zu liefern, die die Fehlersuche erleichtern und eventuell unbemerkte Fehler offenbaren. Hierbei werden Fehler und potentielle Probleme, die sich nicht auf kritische Weise auf den Arbeitsablauf auswirken, ausschließlich über das Logging-System protokolliert. Fehler, die sich auf den Arbeitsablauf auswirken, werden sowohl protokolliert als auch über die graphische Oberfläche dem Benutzer gemeldet.

Ein Beispiel für eine Meldung, die im Protokoll vermerkt, dem Benutzer jedoch nicht angezeigt wird: Wird keine Implementierung der Constraints gefunden, wird diese Tatsache im Protokoll vermerkt. Sie wird dem Benutzer jedoch nicht angezeigt, da das Programm weiterhin verwendet werden kann und dieser Zustand möglicherweise beabsichtigt ist.

Wie bei modernen Logging-Systemen üblich, unterstützt auch das Verwendete verschiedene „Loglevels“. So können unwichtige Meldungen leicht von den wichtigen getrennt werden, und auch für das Debugging relevante Meldungen über das gleiche System ausgegeben werden.

4.3.3. Ladebildschirm

In *JVS Planung* war der Ladebildschirm eine reine Attrappe, die dem Benutzer ein Gefühl von Fortschritt liefern sollte, während das Programm lädt. Da er jedoch selbst erst geladen wurde, nachdem der Rest der Anwendung schon initialisiert war und dann fest eine Sekunde angezeigt wurde, hat er lediglich den Start der Anwendung verzögert.

In *JVS2* wird die eigentliche Funktion des Ladebildschirms implementiert. Er läuft in einem separaten Thread, der parallel zur Initialisierung der eigentlichen Anwendung läuft, und

wird beendet, sobald die Initialisierung abgeschlossen ist. Damit der Benutzer im Falle eines extrem schnellen Starts das Vorhandensein des Ladebildschirms nicht vermisst, wird er immer minimal 0,25 s angezeigt. Dies gibt dem Benutzer ein Gefühl von Sicherheit, dass die Anwendung wie erwartet startet.

Diese Implementierung hat den Effekt, dass die Anwendung 0,75 s - 1 s schneller starten kann, was im Wahrnehmungsbereich eines Menschen liegt.

4.3.4. JCalendar

Während der Implementierung der Kalender-Popups ist aufgefallen, dass die Popup-Variante, die einen `JSpinner` verwendet, um das Datum anzuzeigen und einzugeben, jegliche manuelle Benutzereingabe ignoriert. Dieses Problem ließ sich umgehen, indem die Variante verwendet wurde, die ein einfaches `JTextField` zur Darstellung benutzt.

JCalendar-Popups haben auch die Eigenschaft, dass sie bei leerem Anzeigefeld immer das aktuelle Datum vorselektieren. Dies macht es unmöglich, dem Benutzer eine sinnvolle Vorauswahl zu bieten, ohne das Anzeigefeld zu füllen. Der Kunde wünscht jedoch leere Anzeigefelder, weshalb der Benutzer bei jeder Eingabe zum richtigen Datum navigieren muss.

4.4. Speichersystem

Bei der Implementierung der Serialisierung wurde entschieden, anstatt den in den Schulen enthaltenen Referenzen auf JVS, die komplette JVS zu serialisieren. Dies hat den Vorteil, dass JVS eingelesen werden können, wann immer sie gebraucht werden, ohne einen zweiten Durchlauf zu benötigen. Es hat aber auch den Nachteil, dass die Dateigröße stark aufgebläht wird und ein `XmlAdapter` beim Einlesen benötigt wird, der die verschiedenen JVS-Instanzen zu einem JVS-Objekt zusammenfasst.

Würden statt der kompletten JVS-Instanzen nur Referenzen auf an anderer Stelle vorhandene JVS gespeichert, könnte beim Einlesen dieser Referenzen nicht sichergestellt werden, dass das zugehörige JVS-Objekt bereits eingelesen und initialisiert wurde. So müsste in einem ersten Durchlauf die XML-Referenz-ID in ein Java-Objekt eingelesen werden und erst in einem zweiten Durchlauf könnten die Java-Referenzen gesetzt werden. Dies könnte jedoch potentiell zu ungültigen Zuständen führen, wenn bei zukünftigen Implementierungen die Möglichkeit einer ungültigen JVS-Referenz nicht berücksichtigt würde.

Im Vergleich dazu ist die Bereitstellung eines Adapters, der beim Einlesen doppelte JVS-Instanzen konsolidiert, wesentlich einfacher und zukunftssicherer. Der zusätzlich benötigte Speicherplatz ist relativ gering und bei heutigen Festplatten zu vernachlässigen.

Diese Implementierung kann jedoch in Zukunft ein Problem verursachen, wenn eine Referenz auf eine ältere Version des Datensatzes in den Daten gesetzt werden soll. Diese Konstellation kann in der aktuellen Implementierung nicht erfolgreich deserialisiert werden, da die

Abkürzungen der JVS zusätzlich als einziges Identifikationsmerkmal dienen. Eine ältere Version des Datensatzes würde jedoch mit großer Wahrscheinlichkeit die gleichen JVS-Abkürzungen verwenden, weshalb nach einer Deserialisierung die ältere und die aktuelle Version des Datensatzes die gleichen JVS teilen würden. Dieser Fehler würde erst auffallen, wenn in einer Version des Datensatzes die JVS modifiziert würde, da sich diese Modifikation fälschlicherweise auf alle anderen Versionen auswirken würde.

Dieses Problem kann jedoch leicht umgangen werden, indem ein anderes Identifikationsmerkmal – beispielsweise eine GUID[3] – verwendet wird. Die aktuelle Implementierung benötigt diese Komplexität nicht, sie kann jedoch mit geringem Aufwand nachgerüstet werden.

Die Java-Klassen `java.awt.Dimension` und `java.awt.Color` lassen sich nicht fehlerfrei (de-)serialisieren, weshalb eine alternative Implementierung angeboten werden muss.

`java.awt.Dimension` wurde nicht ausreichend für JAXB annotiert, weshalb die Methode `Dimension.getSize()` als separate Eigenschaft „size“ serialisiert wird[13]. Sie gibt jedoch das aktuelle `Dimension`-Objekt zurück, so dass beim Serialisieren eine endlose Rekursion auftritt. Als alternative Implementierung wurde eine Klasse `SerialisableDimension` implementiert, die die `Dimension`-Eigenschaften „width“ und „height“ enthält und serialisiert. Diese Klasse kann mit einem `Dimension`-Objekt initialisiert werden und bietet eine Methode `public Dimension retrieveDimension()` an, so dass sie bei der Serialisierung als Alternative zu `java.awt.Dimension` verwendet werden kann.

Die Klasse `java.awt.Color` wird in ihrer Standardimplementierung von JAXB immer mit einem leeren Wert serialisiert. Um dieses Problem zu umgehen, wurde ein `XmlAdapter` namens `ColorAdapter` implementiert, der den Farbwert beim Speichern in einen `String` kodiert und beim Laden wieder ausliest.

4.5. Constraints

Die Constraints werden konzeptionell über ein Plugin-System angeboten. Jedoch sind die Constraints, mit denen die Anwendung ausgeliefert wird, im Unterprojekt „model“ enthalten. So steht immer eine Constraint-Implementierung bereit und es wird ein zusätzliches Unterprojekt eingespart. Diese Implementierung bedeutet auch, dass das „model“-Unterprojekt gleichzeitig ein Constraint-Plugin ist.

Die Implementierung sowohl der Constraints als auch des Codes, der Constraints anwendet, muss gegen `null` und ungültige Rückgabewerte abgesichert sein. Die aktuelle Implementierung sichert nicht gegen Exceptions, da diese dann – je nach Implementierung – entweder unbemerkt bleiben oder den Benutzer mit für ihn nichtssagenden Fehlermeldungen belästigen würden. In beiden Fällen würden Fehler nicht behoben, und wahrscheinlich nicht gemeldet werden. Die aktuelle Implementierung hat jedoch zur Folge, dass der begonnene Arbeitsschritt nicht erfolgreich beendet werden kann, was den Benutzer zwingt, das Problem zu beachten.

5. Tests

Ursprünglich wurde geplant, alle möglichen Klassen im Model und Presenter mit Unit-Tests zu prüfen. Leider wurde im Verlauf des Projekts die Zeit knapp, so dass zugunsten von System- und Kundentests keine Unit-Tests implementiert wurden. Lediglich die Klassen `Time` und `TimePeriod` wurden von Projektbeginn mittels Unit-Tests getestet.

Von den geplanten drei Kundentests konnte aus Zeitgründen nur einer durchgeführt werden. Als Ersatz für einen der ausgefallenen Tests musste eine konkrete Planung dienen, die somit einen Praxistest darstellte. Dies hat den Vorteil, dass die geforderte Praxistauglichkeit des Systems unter Beweis gestellt wurde.

5.1. Modul- und Integrationstests

Als Framework für Unit-Tests wurde JUnit4 gewählt, und die Tests wurden regelmäßig als Teil der Continuous-Integration-Builds auf einem Jenkins-Server ausgeführt. Dieser Server führte im Rahmen dieser Builds auch automatisch Integrationstests durch, indem die einzelnen Unterprojekte zu einer ausführbaren Applikation verbunden wurden. Die Builds wurden automatisch angestoßen, sobald neue Änderungen im Projektarchiv gefunden wurden.

Selbst die vorhandenen Unit-Tests halfen, an kritischen Stellen Fehler schon während der Implementierung zu finden, was spätere Tests der vollständigen Anwendung erleichterte und Fehler vermied. Eine umfassendere Implementierung von Unit-Tests wäre wünschenswert gewesen, musste jedoch aus Zeitgründen unterlassen werden.

5.2. Kundentests

Es wurde ein großer Kundentest durchgeführt, da aus Zeitgründen keine weiteren Kundentests vor dem Einsatz im laufenden Betrieb möglich waren. Bei diesem Test wurde eine kleine Planung durchgeführt. Konkret bedeutet das, dass Projekt- und Feriendaten vollständig und mit realen Daten gefüllt wurden. Die JVS-Stammdaten wurden geprüft und einige geblockte Termine wurden probehalber bei den JVS eingetragen. Im Anschluss wurden die Stamm- und Anmeldedaten der Grund- und Föderschulen auf verletzte Constraints geprüft und diese korrigiert. Auch hier wurden probehalber einige geblockte Termine eingetragen. Schießlich wurden die Klassen einiger Schulen probehalber in den Belegungsplan eingetragen.

Während dieses Tests wurden durchgeführte Testfälle mit ihrem Ergebnis protokolliert. Sofern Fehler außerhalb der Testfälle gefunden wurden, wurden auch diese notiert. Zusätzlich wurde jedem positiven Testfall sowie jedem zufällig gefundenen Fehler eine Priorität von 0 bis 3 zugeteilt. Hierbei hatten die einzelnen Prioritäten die in Tabelle 5.1 gelisteten Bedeutungen. Tabelle 5.1 listet auch die Anzahl der Vorkommnisse der einzelnen Prioritäten. Hierbei sei angemerkt, dass zwei der kritischen Fehler falsch positiv waren.

Insgesamt wurden 60 Testfälle identifiziert und geprüft. Es wurden acht Fehler außerhalb der geprüften Testfälle gefunden für einen Testfall wurden drei verwandte Fehler identifiziert. Insgesamt wurden 23 Fehler identifiziert, zwei davon falsch positiv. Daraus lässt sich schließen, dass 47 Testfälle negative Ergebnisse hatten.

Priorität	Bedeutung	Vorkommnisse
1	Kritischer Fehler	14
2	Problematischer Fehler	6
3	Unkritischer Fehler	1
0	Sollte erledigt werden.	2

Tabelle 5.1.: Kundentest: Fehler-Prioritäten

Der Kunde hat diesen Test mit Vorbehalt akzeptiert, unter der Bedingung, dass die gefundenen kritischen Fehler behoben würden. Das Testprotokoll – inklusive der geprüften Testfälle – ist in Anhang B dargestellt.

5.2.1. Vollständige Planung: Praxistest

Im Rahmen dieser Arbeit wurde die Anwendung zum ersten Mal im laufenden Betrieb eingesetzt. In diesem Zusammenhang wurde die Planung für das Schuljahr 2014/2015 erstellt und das Programm musste seine Praxistauglichkeit unter Beweis stellen. Zusätzlich kam bei diesem Test eine Kollegin des Kunden hinzu, die bei der Entwicklung von *JVS2* nicht beteiligt war, und die auch keinen Kontakt zu *JVS Planung* hatte.

Während dieses Praxistests wurden einige weitere Fehler identifiziert, die teilweise nicht durch den Kundentest offenbart wurden und teilweise durch die Korrektur der dort gefundenen Fehler neu hinzukamen. Diese Fehler wurden, soweit möglich, vor Ort behoben.

Desweiteren offenbarte der Praxistest folgendes:

- Die Geschwindigkeit und Funktionalität sind zufriedenstellend, abgesehen von den noch vorhandenen Fehlern.
- Die Ladezeit der Anwendung ist schneller, auch weil *JVS Planung* den Ladebalken auf eine Sekunde hartkodiert hatte, während er in *JVS2* nur angezeigt wird, bis die Oberfläche geladen wurde. Die verkürzte Ladezeit gefällt dem Kunden.

- Die Leitsystem-Übersicht ist in der überarbeiteten Version wesentlich nützlicher, da sie verwendet werden kann, um zu einem bestimmten Arbeitsschritt zu springen, ohne den entsprechenden Menüpunkt auswendig lernen zu müssen.
- Die aktuelle Implementierung der Constraints zeigt Warnungen zu häufig und aufdringlich an.
- Bei der Einplanung werden schon leichte Mausbewegungen von einem Pixel als Start einer Drag&Drop-Aktion gewertet, weshalb die Anwendung manchmal nicht auf Eingaben zu reagieren scheint.
- Die Auslieferung als JAR-Datei macht ein Software-Upgrade wesentlich kundenfreundlicher als die in *JVS Planung* verwendete Alternative, den Quellcode auszuliefern und vor Ort zu kompilieren.
- Die Größe der Projektdatei einer typischen Einplanung der Kreisverkehrswacht Esslingen ist zwischen ein und zwei MB.
- *JVS2* ging auch in Fehlerfällen nie in einen undefinierten Zustand über und konnte somit ohne Neustart weiterverwendet werden.

6. Zusammenfassung und Ausblick

6.1. Zusammenfassung

Das Programm *JVS2* erfüllt die gestellten Aufgaben und liefert ein befriedigendes Ergebnis. Das Projekt wurde zur Zufriedenheit des Kunden fertiggestellt, obwohl der Zeitrahmen knapp bemessen war und es einige unerwartete Schwierigkeiten während der Implementierung gab. Wegen des Zeitmangels konnten nur wenige Tests durchgeführt werden.

Im Betrieb ist aufgefallen, dass bei verletzten Constraints zu viele Warnmeldungen angezeigt wurden. Außerdem wird nicht gegen fehlerhafte Constraint-Implementierungen abgesichert. Aus Zeitgründen konnten diese Probleme nicht mehr adressiert werden. Sie stellen aber auch kein akutes Problem dar, so dass sie im Rahmen dieser Arbeit nicht adressiert werden mussten.

6.2. Ausblick

Das vorliegende Programm könnte an einigen Stellen ergänzt oder umgestaltet werden, um den Arbeitsablauf zu vereinfachen und modernisieren.

6.2.1. Frequenz der Constraint-Meldungen

Während eines Tests des Programms ist aufgefallen, dass die Meldungen bei verletzten Constraints während der Planung zu oft angezeigt werden. Dies ist auf einen Mangel im Entwurf zurückzuführen, da der nicht vorsieht, einen Constraint nur temporär abzuschalten. Dieser Mangel ist nicht schwer zu beheben und könnte in einer zukünftigen Progammversion mit geringem Aufwand entfernt werden. Dies hätte eine große Auswirkung auf die Benutzbarkeit, da Constraint-Warnungen nicht mehr einfach nur weggeklickt würden, wenn sie nicht mehr ständig angezeigt werden.

6.2.2. Automatische Planung

Eine offensichtliche Möglichkeit, das Programm aufzuwerten, wäre eine qualitativ hochwertige automatische Einplanung der Klassenbesuche. Allerdings handelt es sich hier um ein sehr schweres Problem, da das Programm selbständig ermitteln müsste, welche Constraints

in welchen Situationen ignoriert werden sollen. Eine Planung ohne Verletzung einiger Constraints ist mit im verfügbaren Zeitraum bei der Anzahl der einzuplanenden Schulklassen leider unmöglich. Aus diesem Grund ist diese Funktionalität kaum implementierbar.

Eine Möglichkeit, dem Benutzer wenigstens ein wenig Arbeit abzunehmen, wäre die automatische Planung, soweit sie ohne Verletzung von Constraints machbar ist. Das Ergebnis dieser Planung wäre unvollständig, könnte aber den ersten Schritt der manuellen Planung ersetzen, in dem Klassen nach einem Schema in den vorhandenen Zeitraum eingeplant werden, solange sich keine Konflikte ergeben.

Diese zweite Möglichkeit könnte realisierbar sein, würde aber viel Arbeit und Feinjustierung benötigen, um ein verwendbares Ergebnis zu liefern. Sie könnte sich als Thema einer Folgearbeit eignen.

6.2.3. Umgestaltung zu einer Internetapplikation

Ein großer Teil des bisherigen Planungsprozesses wird nicht vom Programm erfasst oder unterstützt: die Vorplanung. Hier werden die Wünsche und Daten der Schulen auf einem Formular erfasst, das von den Schulen ausgefüllt und an die Planer geschickt wird. Diese Formulare werden in der Vorplanung mühsam sortiert und geprüft, ohne dass *JVS2* dabei unterstützen könnte.

Eine Methode, diesen Schritt der Planung zu vereinfachen, wäre ein Online-System. Die Schulen könnten ihre Daten in dieses System eintragen. Problematische Eintragungen könnten zu großen Teilen automatisch identifiziert werden, und den jeweiligen Parteien mitgeteilt werden. Fehlende Eingaben könnten automatisch angemahnt werden. Der endgültige Datensatz könnte schließlich als Eingabe für das vorliegende Programm dienen, ohne mühsam manuell erfasst werden zu müssen. Schließlich könnten die relevanten Teile der vollständigen Planung automatisch an die jeweiligen Empfänger verteilt werden.

Nachträgliche Änderungen könnten außerdem automatisch den betroffenen Stellen mitgeteilt werden, was in der aktuellen Fassung des Programms ein mühsamer und fehlerträchtiger Prozess ist.

Außer den offensichtlichen Änderungen müsste bei einem solchen System allerdings beachtet werden, dass Daten- und Programmsicherheit bei der Planung und Implementierung berücksichtigt werden. Es handelt sich bei den Daten teilweise um vertrauliche Informationen, die nicht frei zugänglich sein dürfen. Es würde somit ein komplexes Authenzifizierungs- und Rechtevergabesystem benötigt, das in dieser Form bisher nicht existiert. Außerdem müsste eine Prüfung der Datenintegrität bei der Kommunikation zwischen den Systemkomponenten ermöglicht und durchgeführt werden.

Wegen der Komplexität einer solchen Plattform ist es fraglich, ob sich dieses Thema für eine Diplom-, Bachelor- oder Masterarbeit eignet. Es könnte aber möglich sein, das Thema auf mehrere Arbeiten zu verteilen, um so das gewünschte System zu erstellen.

A. Begriffserklärung

Abkürzung von Schulen und JVS Jede Grund-, Förder- und Jugendverkehrsschule besitzt in *JVS2* eine Abkürzung, die diese Schule bzw. JVS eindeutig identifiziert. Diese Abkürzung kodiert den Namen und bei Grund- und Förderschulen die Schulart.

Constraint Eine Einschränkung an die Planung, die durch *JVS2* geprüft wird.

der Ausdruck Die von *JVS Planung* und *JVS2* generierten Dokumente, die der Kunde ausdrucken kann. Da manchmal Nachbearbeitungen nötig werden, und die Dokumente teilweise auch per E-Mail verschickt werden sollen, werden keine konkreten Druckaufträge erstellt.

Jugendverkehrsschule Ein Ort, an dem Schüler das Fahrradfahren in Theorie und Praxis erlernen. Befindet sich unter der Verwaltung der Kreisverkehrswacht Esslingen.

JVS Die Abkürzung für „Jugendverkehrsschule“.

Kunde Die Kreisverkehrswacht Esslingen, vertreten durch Herrn Bufler von der Verkehrspolizei Esslingen.

Planungskriterium Ein anderer Begriff für „Constraint“.

Vorplanungsprozess Ein Prozess, der die Belegungsplanung mittels *JVS* vorbereitet.

B. Protokoll des Kundentests

Definierte Testfälle:

0 Allgemein

a) Menüs/OK-Knöpfe werden bei verletzten critical Constraints ausgegraut.

b) In einer Liste auf ein anderes Element wechseln verursacht eine Beschwerde bei ungespeicherten Daten.

c) Mit OK & Speichern geht es an manchen Stellen im Leitsystem weiter.

d) Über Leitsystem-Menü geht es im Leitsystem weiter.

e) Mit der Leitsystem-Übersicht kann man an beliebige Punkte im Leitsystem springen.

f) Neues Projekt erstellen warnt, wenn bereits ein Projekt geöffnet ist.

g) „Speichern“ und „Speichern unter“ funktionieren wie erwartet.

h) Beim Versuch, ein Projekt zu überschreiben, wird gewarnt.

i) Die Geschwindigkeit der Anwendung ist gut.

1 Neues Schuljahr

a) Ein leerer Projektname wird nicht akzeptiert.

b) Langer Projektname (50+ Zeichen) wird akzeptiert.

1.1 Projekt auf altem Projekt aufbauen

a) Alte Sommerferien werden übernommen.

b) Alle anderen Ferien sind ursprünglich leer.

c) Keine JVS-Blocktermine außer Fr. nachmittags.

d) Keine Schul-Blocktermine.

e) Keine Schul-Ferientage.

f) Keine Belegungen.

2 Ferieneingabe

a) Ferien müssen zwischen d. Sommerferien liegen.

b) Sommerferien müssen plausibel sein.

c) Ferien müssen im richtigen Jahr liegen.

d) Ferien müssen chronologisch sein.

e) Ferien müssen gültige Zeitintervalle sein.

f) Ungültige Eingaben bei nicht-Sommerferien werden mit Gemecker akzeptiert.

g) Gültige Ferien können eingetragen werden.

3 JVS-Dateneingabe

a) Eine leere Abkürzung wird nicht akzeptiert.

b) Leere JVS-Namen werden akzeptiert, aber nur mit Hinweis (rot).

c) Sonstige leere Felder werden akzeptiert, aber nur mit Hinweis (gelb).

4 JVS-Blocktermin-Eingabe

a) Termine können nur im gültigen Zeitraum eingegeben werden.

b) Alle nötigen Termine können angegeben werden.

c) Termine können (de-)aktiviert werden.

d) Termine können gelöscht werden.

e) Termine erscheinen in dem JVS-Plan, wenn sie nicht deaktiviert/gelöscht sind, aber nicht in anderen JVS-Plänen.

5 Schuldateneingabe

a) Eine leere Abkürzung wird nicht akzeptiert.

b) Leere Schulnamen werden akzeptiert, aber nur mit Hinweis (rot).

c) Sonstige leere Felder werden akzeptiert, aber nur mit Hinweis (gelb).

d) Schulen löschen/hinzufügen funktioniert.

6 Schul-Anmeldebogeneingabe

a) „keine Klasse“ angeben funktioniert.

b) Kein Klassenname: kritischer Fehler

c) nicht-positive Schülerzahl: Warnung

d) Klassenname > 4 Zeichen: Warnung

e) Schülerzahl 100+: Warnung

6.1 Schul-Blocktermin-Eingabe

a) Termine können nur im gültigen Zeitraum eingegeben werden.

b) Alle nötigen Termine können angegeben werden.

c) Termine können (de-)aktiviert werden.

d) Termine können gelöscht werden.

e) Termine erscheinen in dem Schulplan, wenn sie nicht deaktiviert/gelöscht sind, aber nicht in anderen Schulplänen.

f) Ferientage können (de-)aktiviert, aber nicht gelöscht werden.

g) Neue Ferientage werden in anderen Schulen deaktiviert angezeigt.

7 Belegungsplanung

a) Es können nur 5 oder 6 Klassen eingeplant werden.

b) Alle erwarteten Eigenschaften werden erfüllt.

c) Schulfarbe ändern funktioniert und wird angepasst.

d) Termine verschieben funktioniert.

e) Termine kopieren funktioniert.

f) Termine einplanen funktioniert.

g) März -> Scrollen -> März funktioniert.

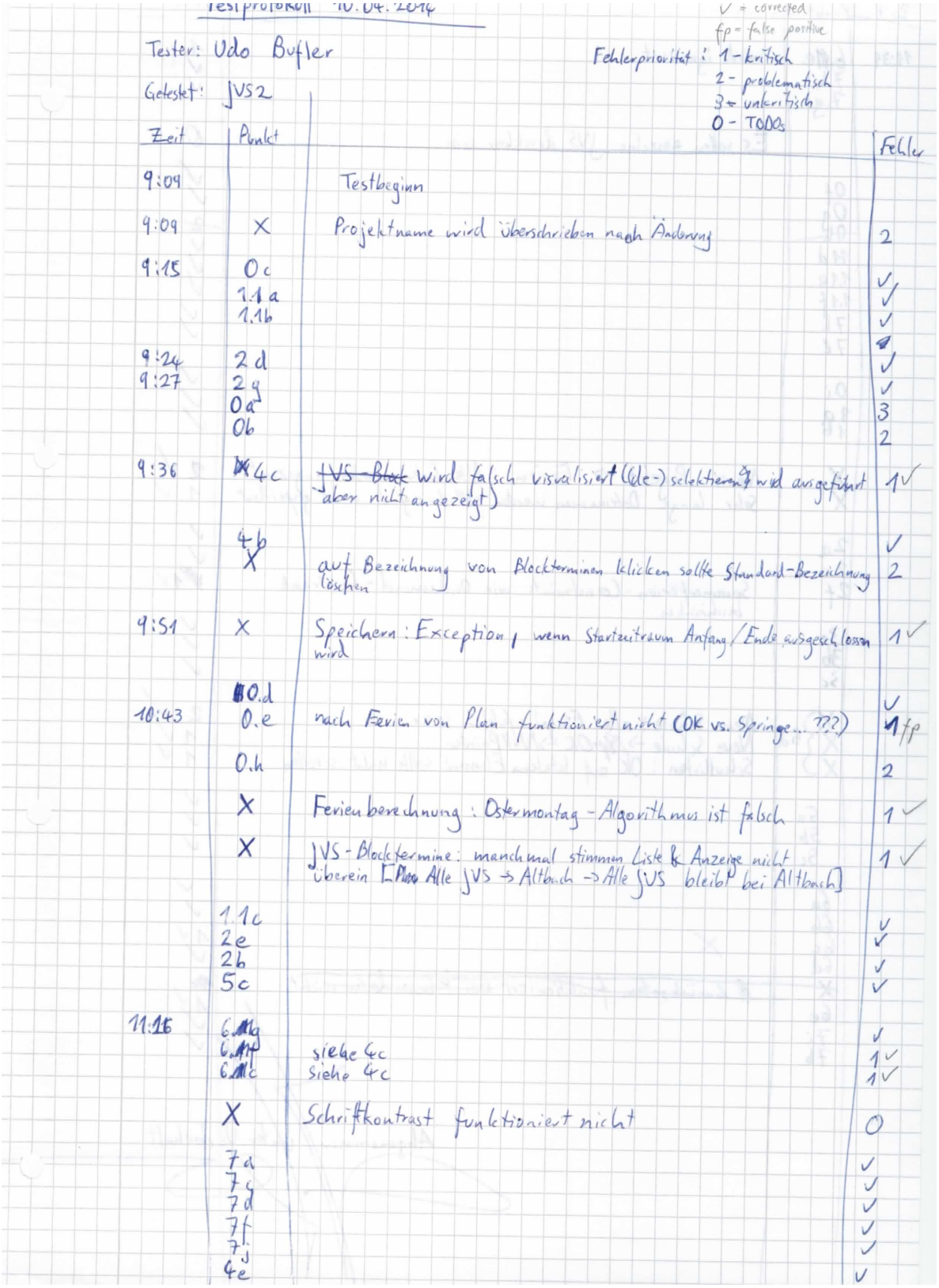
h) Zur Schule springen funktioniert.

i) 2 Termine in eine Woche einplanen: Fehlermeldung

j) 2 Nachmittagstermine: Warnung

k) 2 Termine 7 Wochen (Grundschule) bzw. 8 Wochen (Förderschule) auseinander: Warnung

l) x. Besuche in verschiedenen Wochen: Warnung



Testprotokoll 10.04.2014

Tester: Udo Bufler

Getestet: JVS2

✓ = corrected
fp = false positive

Fehlerpriorität: 1 - kritisch
2 - problematisch
3 - unkritisch
0 - TODOs

Zeit	Punkt		Fehler
9:09		Testbeginn	
9:09	X	Projektname wird überschrieben nach Änderung	2
9:15	0c		✓
	1.1a		✓
	1.1b		✓
9:24	2d		✓
9:27	2g		✓
	0a		3
	0b		2
9:36	~~X~~ 4c	~~JVS-Block~~ wird falsch visualisiert ((de-)selektieren wird ausgeführt aber nicht angezeigt)	1✓
	4b		✓
	X	auf Bezeichnung von Blockterminen klicken sollte Standard-Bezeichnung löschen	2
9:51	X	Speichern: Exception, wenn Startzeitraum Anfang/Ende ausgeschlossen wird	1✓
	~~X~~ 0.d		✓
10:43	0.e	nach Ferien von Plan funktioniert nicht (OK vs. Springe... ???)	1fp
	0.h		2
	X	Ferienberechnung: Ostermontag-Algorithmus ist falsch	1✓
	X	JVS-Blocktermine: manchmal stimmen Liste & Anzeige nicht überein [Plan Alle JVS → Altbach → Alle JVS bleibt bei Altbach]	1✓
	1.1c		✓
	2e		✓
	2b		✓
	5c		✓
11:15	6.~~X~~g		✓
	6.~~X~~f	siehe 4c	1✓
	6.~~X~~c	siehe 4c	1✓
	X	Schriftkontrast funktioniert nicht	0
	7a		✓
	7g		✓
	7d		✓
	7f		✓
	7.j		✓
	4e		✓

Abbildung B.1.: Testprotokoll Seite 1

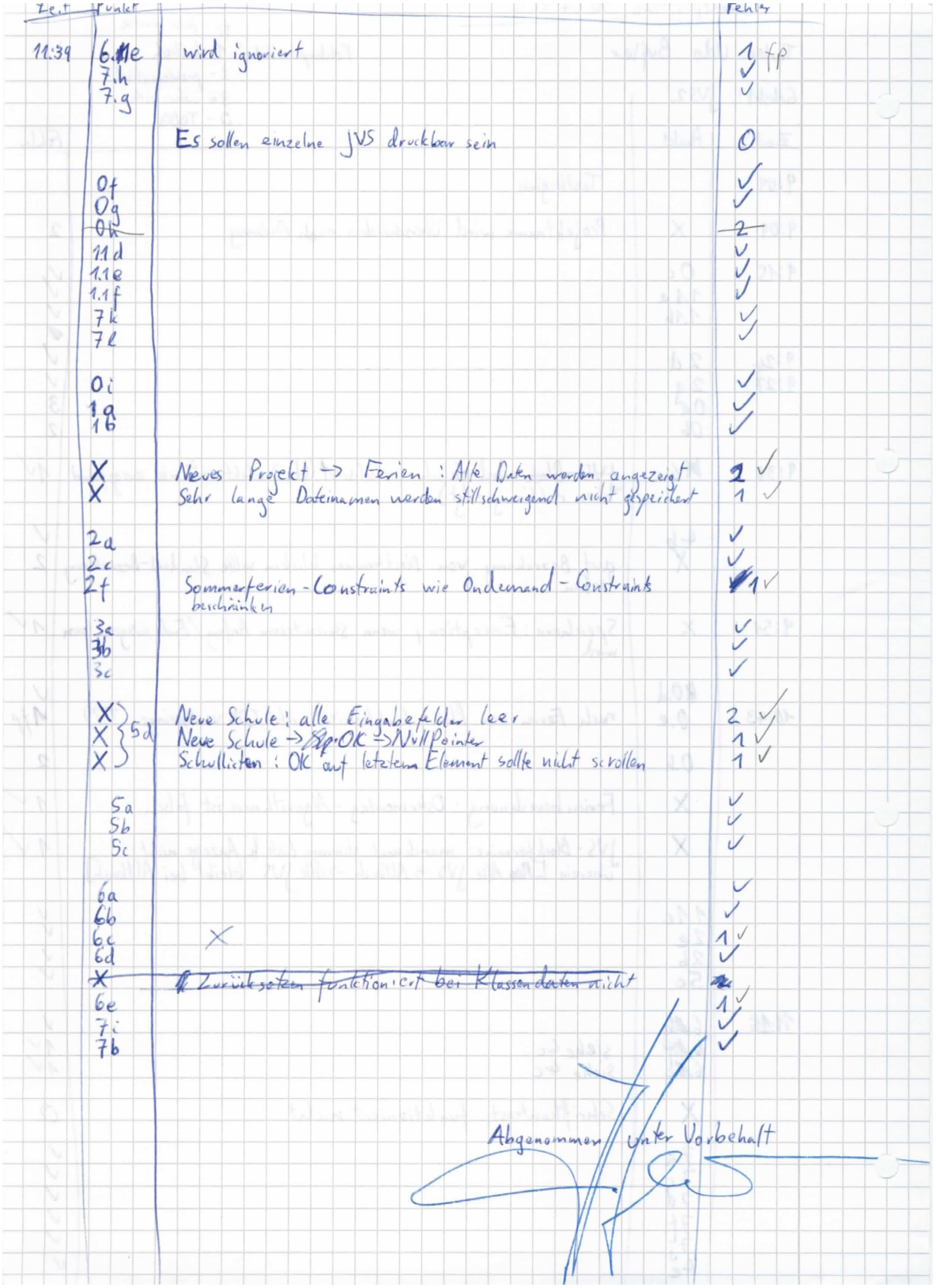


Abbildung B.2.: Testprotokoll Seite 2

C. Inhalt und Aufbau des beigelegten Datenträgers

Der beigelegte Datenträger ist wie folgt aufgebaut:

ausarbeitung Der Ordner *Ausarbeitung* enthält dieses Dokument.

programm Die ausführbare JAR-Datei, die das Produkt dieser Arbeit enthält.

quelltext Der Quelltext, der im Rahmen dieser Arbeit entstanden ist.

- **eclipse** Dieser Unterordner enthält komprimierte Archive des Eclipse-Workspace, der für die Implementierung verwendet wurde. Der Inhalt der Archive ist – soweit vom Archiv-Format unterstützt – identisch. Unter den enthaltenen Projekten befindet sich eines namens „JVS“. Dieses Projekt enthält eine modifizierte Version von *JVS Planung*, die im Dateimenü einen Eintrag enthält, der ein geladenes Projekt in das von *JVS2* verwendete Format exportiert. Dieses Unterprojekt wird auf expliziten Kundenwunsch nicht an den Kunden ausgeliefert.

Literaturverzeichnis

[1] Abstrakte Fabrik. URL: https://de.wikipedia.org/wiki/Abstrakte_Fabrik [cited 2014-04-03]. (Zitiert auf den Seiten 22, 28 und 48)

[2] Class ServiceLoader<S>. URL: http://docs.oracle.com/javase/7/docs/api/index.html?java/util/ServiceLoader.html [cited 2014-05-05]. (Zitiert auf Seite 48)

[3] Globally Unique Identifier. URL: https://de.wikipedia.org/wiki/Globally_Unique_Identifier [cited 2014-05-07]. (Zitiert auf Seite 51)

[4] Java Architecture for XML Binding. URL: https://de.wikipedia.org/wiki/JAXB [cited 2014-05-05]. (Zitiert auf Seite 30)

[5] JCalendar. URL: http://toedter.com/jcalendar/ [cited 2014-05-05]. (Zitiert auf Seite 35)

[6] JGoodies Validation. URL: http://www.jgoodies.com/freeware/libraries/validation/ [cited 2014-05-23]. (Zitiert auf Seite 26)

[7] JiBX. URL: https://de.wikipedia.org/wiki/JiBX [cited 2014-05-05]. (Zitiert auf Seite 30)

[8] LOGBack. URL: http://logback.qos.ch/ [cited 2014-05-05]. (Zitiert auf Seite 49)

[9] Microba controls. URL: http://microba.sourceforge.net/ [cited 2014-05-05]. (Zitiert auf Seite 35)

[10] Microba controls screenshots. URL: http://microba.sourceforge.net/screenshots.html [cited 2014-05-05]. (Zitiert auf Seite 35)

[11] Model View Controller. URL: https://de.wikipedia.org/wiki/Model_View_Controller [cited 2014-04-03]. (Zitiert auf Seite 21)

[12] Model View Presenter. URL: https://de.wikipedia.org/wiki/Model_View_Presenter [cited 2014-04-03]. (Zitiert auf den Seiten 21 und 48)

[13] RE: DEFAULT MARSHALLING OF JRE CLASSES. URL: https://java.net/projects/jaxb/lists/users/archive/2006-10/message/106 [cited 2014-05-05]. (Zitiert auf Seite 51)

[14] Simple Logging Facade for Java (SLF4J). URL: http://www.slf4j.org/ [cited 2014-05-05]. (Zitiert auf Seite 49)

[15] Visual Paradigm Community Edition. URL: https://www.visual-paradigm.com/editions/community.jsp [cited 2014-05-05]. (Zitiert auf Seite 21)

[16] Welcome to Apache Maven. URL: https://maven.apache.org/ [cited 2014-05-23]. (Zitiert auf Seite 47)

[17] What are good Java date-chooser Swing GUI widgets? URL: https://stackoverflow.com/questions/1339354/what-are-good-java-date-chooser-swing-gui-widgets [cited 2014-05-05]. (Zitiert auf Seite 35)

[18] International Business Machines Corporation et al. ICU - International Components for Unicode. URL: http://site.icu-project.org/ [cited 2014-04-03]. (Zitiert auf Seite 20)

[19] Magnus Schwab. Ein Planungssystem für örtlich verteilte Ausbildungskurse. Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, August 2005. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2307&engl=1. (Zitiert auf Seite 9)

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift