Implementierung

Praxis der Softwareentwicklung

Entwicklung einer Software zur Berechnung der Mandatsverteilung im Deutschen Bundestag

Gruppe 1

Philipp Löwer, Anton Mehlmann, Manuel Olk, Enes Ördek,
Simon Schürg, Nick Vlasoff

WS 2013 / 14

Inhaltsverzeichnis

1 Einleitung

Die dritte Phase unseres Projektes - die Implementierung. Unser Ziel in dieser Phase ist es, die bisherigen Errungenschaften in der Pflichten- und Entwurfsphase als ausführbares Programm umzusetzen und dabei möglichst wenig von den bisherigen Entwürfen auszuweichen. Hierbei ist uns jedoch aufgefallen, dass Veränderungen am Entwurf unumgänglich sind. Der Grund hierfür ist, dass bestimmte Sachen einfach nicht beachtet wurden, und somit übersehen worden sind.
Dieses Dokument wird das fertige Programm mit seinen Funktionen erläutern und alle Veränderungen mit den zugehörigen Entwurfsentscheidungen ausführlich erklären. Anschließend werden wir eine Einblick in die nächste Phase geben.

Dieses Dokument ist im Zuge der Implementierungsphase entstanden. Ziel dieser Phase, ist die Umsetzung der in der Pflichten- und Entwurfsphase festgelegten Strukturen und Prozessabläufe unter Berücksichtigung gegebener Rahmenbedingungen, Regeln und Zielvorgaben.
Da sich jedoch während der Implementierung Sachverhalte ergeben, die mit dem eigentlichen Entwurf nicht vollständig zu vereinbaren sind, ist es notwendig einige Änderungen bzw. Anpassungen, aber auch Erweiterungen vorzunehmen. Diese, vom eigentlichen Plan abweichenden Entscheidungen, werden im Folgenden erläutert.
Abschließend wird ein kurzer Ausblick auf die nächste Phase gegeben.

1.1 Notationshinweise

`Klassennamen` werden in diesem Dokument textuell hervorgehoben, indem sie **fett** und in einer anderen Schriftart geschrieben werden.
`Methodennamen` werden hervorgehoben, indem sie *kursiv* und ebenfalls in einer anderen Schriftart geschrieben werden.
Außerdem wird Bundestagswahl im gesamten Entwurfsdokument durch BTW abgekürzt.

2 Pakete

2.1 Datenmodell

Da die verwendeten Listen im Datenmodell durch die Berechnungen und Zuweisungen recht groß wurden, wurde zusätzliche Funktionalität in das Datenmodell gebracht. Dies erleichterte den Zugriff auf die benötigten Daten für verschiedene Komponente wie zum Beispiel **`Mandatsrechner`**, **`Wahlgenerator`** und **`GUI`**. Zudem wurde die Klasse **`BerichtDaten`** für die Klasse **`Sitzverteilung`** erstellt. Dies war notwendig, damit die Berichttabelle in der GUI korrekt befüllt werden kann. Dabei hält die Klasse **`BerichtDaten`** fünf Listen die jeweils die dazugehörigen Spalten befüllen. Kandidaten haben nun einen Namen und einen bestimmten Platz in der Landesliste.

2.1.1 Bundestagswahl

Die Kernklasse unseres Programmes wurde mit geringen Änderungen umgesetzt. Die Deep-Copy Funktion wurde hierbei mithilfe von Serialisierung umgesetzt. Eine native Funktion hat den Aufwand hierfür übertroffen, was eigentlich geplant war. Mit diesem Weg, haben wir dies mit einem überschaubaren Aufwand umsetzen können.

2.1.2 Partei

Den Parteien wurden Überhang- und Ausgleichsmandate als Zahlenwerte hinzugefügt. Unser Versuch war es anfangs, Überhang- und Ausgleichsmandate als eindeutige Mandate in die Sitzverteilung einzubauen. Jedoch ist uns erst im Laufe der Umsetzung aufgefallen, dass dies nicht möglich ist.
Die Klasse Partei besitzt Funktionen, die Überhang- und Ausgleichsmandate setzen, zurücksetzen und verändern.

2.1.3 Kandidat

Was für uns in der Pflichten- und Entwurfsphase gegen Ablehnung gestoßen hatte, wir es jedoch dennoch Umgesetzt haben, sind Namen, Vornamen und das Geburtsjahr der Kandidaten.
Ursprünglich wollten wir Anonyme Kandidaten und Zufällig-gefüllte Landeslisten für unseren Mandatsrechner verwenden. Jedoch ist uns früh aufgefallen, dass für eine genaue Berechnung, die genaue Anzahl der Listenkandidaten benötigt wird.

2.2 Import/Export

Die Import-Export-Komponente wurde im Laufe der Implementierung stark angepasst. Anders als im Entwurf, haben wir das Exportieren vom Importieren getrennt. Da die Namen und der feste Platz in der Ladenliste mitgespeichert werden, muss eine zusätzliche .csv-Datei importiert werden. Diese Datei ist ebenfalls aus der Bundeswahlleiter-Seite und werden von uns als "Wettbewerber-Datei"bezeichnet. Mithilfe dieser Wahlbewerber-Datei werden die Landeslisten und die vorher ausgelesenen Kandidaten befüllt.
Die Wahlbewerber-Datei für die Bundestagswahl 2013 wird im Programm mit übergeben

und kann im Notfall auch für Bundestagswahl 2009 genutzt werden. Es ist leider keine solche Datei für die Bundestagswahl 2009 verfügbar. Sie wurde von der Bundeswahlleiter-Seite aufgrund von Datenschutzgründen entfernt.
Zudem wurde eine Config-Datei(ebenfalls im .csv-Format) hinzugefügt, die die Einwohnerzahl der Bundesländer und die Farben der Partei beinhaltet. Dadurch müssen diese Werte nicht mehr im Programmcode gespeichert und können durch das Editieren der Datei einfach angepasst werden. Hierzu kommt im nachfolgenden Bericht weiteres.
Der endgültige Crawler erfüllt die Funktionen, die in der Entwurfs- und Pflichtenphase genannt wurden.

2.3 GUI und GUI-Logik

2.3.1 Programmfenster

Das **Programmfenster** ist der eigentliche Eintrittspunkt in das Programm, d.h. es enthält die Main-Methode und wird beim Start als Erstes ausgeführt. Dies bietet sich an, da das **Programmfenster** das Erste sein soll, was der Benutzer sieht, da er damit ja interagieren muss.
Wie bereits im Entwurf festgehalten, enthält das **Programmfenster** eine Liste von **Wahlfenstern**. Diese werden mithilfe einer **TabLeiste**, die ebenfalls vom **Programmfenster** gehalten wird, realisiert.
Zusätzlich besitzt es ein **Menu**, welches dem Benutzer ermöglicht, den gewünschten Befehl auszuwählen und ausführen zu lassen, ohne genaue Steuerbefehle kennen und anwenden zu müssen.

2.3.2 WahlFenster

Im **WahlFenster** wurde die Methode *bundestagswahlDarstellen()* nicht implementiert, da der Code dieser in dem Konstruktor der Klasse Anwendung fand. Die *wechsleAnsicht()*-Methode aber, wurde implementiert, wobei als Parameter anstatt einer **Ansicht**, die im Konstruktor erstellt wird und für das ganze **Wahlfenster** immer das selbe Objekt ist, eine **Gebiet**s-Objekt, welches angezeigt werden soll, übergeben wird.
Jedes **WahlFenster** hat einen Namen, als String gespeichert, eine zugehörige Bundestagswahl, sowie eine **Ansicht** und eigene *GUISteuerung*, was auch leicht vom Entwurf abweicht.

2.3.3 Ansicht

Die **Ansicht** ist die Hauptkomponente des **WahlFenster**s. Sie enthält die im Späteren näher erläuterten **Tabellen-**, **Diagramm-** und **Kartenfenster**. Anders als im Entwurf festgelegt, haben wir uns entschieden nur eine Ansicht zu implementieren. Hauptgrund dafür war, dass eine **Ansicht** ausreichend ist, da bei Ansichtswechsel nur ein neues **DiagrammFenster** und ein neues **TabellenFenster** erzeugt werden müssen, das **KartenFenster** bleibt, dank JTree, das selbe. Immer wieder das selbe **KartenFenster**-Objekt zwischen den drei verschiedenen Ansichten hin und her zu schieben wäre weit aus

aufwendiger als einfach nur eine universale **Ansicht** einzuführen.
Die im Entwurf spezifizierte Methode *zeigeKomponenten()* wurde in zwei Methoden (*Initialisieren()* und *ansichtAendern()*) aufgespalten. Dies war von Nöten, weil bei der erstmaligen Ansichtserstellung alle drei Fenster angelegt werden müssen, bei einer Ansichtsänderung aber nur **Diagramm-** und **Kartenfenster** neu erstellt werden müssen.
Eine weitere Abweichung vom Entwurf ist die *berechnungNotwendig()*, welche festlegt, dass eine Stimme in einem Wahlkreis geändert wurde. Wurde eine Stimme geändert werden keine Diagramme angezeigt, sondern ein Berechne-Knopf an der **DiagrammFenster** Stelle angezeigt. Der Hauptgrund für diese Änderung ist, dass es dadurch möglich ist mehrere Stimmen nacheinander zu ändern, ohne dass nach jeder Änderung eine neue Berechnung durchgeführt werden muss.

2.3.4 TabellenFenster

Das **TabellenFenster** ist das erste der drei Komponenten der **Ansicht**.
Nicht wie im Entwurf vorgeschlagen in einer Klasse, haben wir diese in mehrere Klassen unterteilt. Da es drei Arten von Tabellen gibt (Land, Bundesland, Wahlkreis) gibt es zu jeder Art zwei Klassen, eine Daten-Klasse und eine TabelModel-Klasse. Dies hat die im Entwurf vorgeschlagene **Tabellenzellen**-Klasse zur Auslese von geänderten Stimmen abgelöst, da man dadurch viel leichter an die, in der Tabelle geänderten Stimmen kommt.
Das **TabellenFenster** an sich erstellt die Tabellen wie im Entwurf vorgeschlagen mit der *tabellenFuellen()*-Methode, wobei diese für die drei Gebietsarten überladen ist. Die Erstellung der Klasse **GUIPartei** war notwendig, um Daten wie Sitze, Direktmandate, etc. festzuhalten.

2.3.5 DiagrammFenster

Das **DiagrammFenster** ist das zweite der drei Komponenten. Die Klasse an sich wurde fast genauso implementiert wie im Entwurfsdokument festgelegt. Das einzige was noch hinzugefügt wurde waren die verschiedenen Diagramm-Klassen, die die Diagramme darstellen sollen. Die Methode *erstelleDiagramm()* wurde überladen, weil die drei Arten von Diagrammen in den vorher genannten Klassen erstellt werden. Die Methode *zeigeSitzplatzverteilung()* öffnet das **BerichtsFenster**.

2.3.6 KartenFenster

Das **KartenFenster** ist die letzte Komponente der **Ansicht**. Wie schon im Pflichtenheft festgelegt, ist es als Tabfenster implementiert und wie im Entwurf festgelegt gibt es die Methode *zeigeInformationen()*, die die Karte erstellt und eine Verzeichnisstruktur anlegt.
Das einzige was vom Entwurf abweicht ist das Weglassen des Zurück-Knopfes, welches unnötig wurde, da man sich in der Verzeichnisstruktur von Ansicht zu Ansicht navigieren

kann.

2.3.7 BerichtsFenster

Im **BerichtsFenster** werden Daten visualisiert, die veranschaulichen sollen, woher Mandate der Abgeordneten kommen. Dieses wurde als Tabelle implementiert, ähnlich wie das **TabellenFenster**, um die hohe Menge an Daten möglichs übersichtlich zu halten. Zu dem **BerichtsFenster** gehören die Klassen **BerichtTableModel** und **BerichtDaten**.

2.3.8 VergleichsFenster

Das **Vergleichsfenster** wurde wie im Pflichtenheft und im Entwurf beschrieben implementiert, wobei ein weiteres Diagramm hinzugefügt wurde, welches die Sitzdifferenzen der zwei Wahlen anzeigt. Dies fördert die Verdeutlichung der Unterschiede zwischen zwei Wahlen.

2.3.9 GUISteuerung

Die **GUISteuerung** ist dazu da, um das Programmfenster aktuell zu halten, d.h. Ansichten zu ändern, Stimmenänderung einzuleiten und, wenn notwendig, einen Wahlvergleich zu starten.
Die Implementierung wurde fast genauso durchgeführt wie im Entwurf festgelegt. Die einzigen Sachen die sich verändert haben sind, dass die Methoden *aktualisiereWahlfenster()* einen Parameter **Gebiet** erhält und *vergleichen()* zwei **Bundestagswahlen**, was sich notwendig für deren Funktion ist.
Eine weitere Sache ist, dass die Stimmänderung, anders als im Entwurf spezifiziert, nicht mehr vom **Tabellenfenster** direkt zur **Steuerung** geht sondern zuerst über die **GUISteuerung**, was die Abhängigkeit des Programms von der GUI verringern soll.
Und zuletzt wurden die Parameter **Steuerung** und **Wahlvergleich** entfernt, da die Klasse Steuerung auch ohne ein Attribut ansprechbar ist und der Wahlvergleich der *vergleiche()*-Methode übergeben wird.

2.3.10 Dialoge

Im Dialogepaket sind die Dialogeklassen enthalten die wir in unserem Programm verwenden. Diese wurden im Entwurf nicht erwähnt.
Zu diesen gehören die Klassen:

- **AboutDialog** startet ein About-Fenster
- **ExportDialog** visualisiert die Auswahl eines Verzeichnisses, um Wahlen zu exportieren

- **GeneratorDialog** es können **Stimmenanteile** vom Anwender für Parteien hinzugefügt werden
- **HandbuchDialog** enthält das Handbuch
- **ImportDialog** visualisiert die Suche nach zu importierenden Dateien
- **LizenzDialog** öffnet ein Lizenz-Fenster
- **VergleichsDialog** stellt die Auswahl einer Vergleichswahl zur aktuellen dar

2.4 Mandatsrechner

Die Klasse **Mandatsrechner2009** berechnet die Sitzverteilung nach Sainte-Laguë/Schepers ohne Ausgleichsmandate und **Mandatsrechner2013** berechnet die Sitzverteilung ebenfalls nach Sainte-Laguë/Scheper, aber mit Ausgleichsmandate. Da der **Mandatsrechner2013** dadurch den **Mandatsrechner2009** zur Berechnung nutzt, fällt die Notwendigkeit der Oberklasse **Mandatsrechner** weg. Deswegen hält der **Mandatsrechner2013** ein Objekt der Klasse **Mandatsrecher2009**. Zudem wurde noch in **Mandatsrechner2009** das Verteilungsverfahren nach d'Hondt implementiert, damit eine alternative Berechnung der Sitzverteilung möglich ist. Die Überladung der Methoden *bechne(Gebiet gebiet)* wurde aufgehoben, da die Berechnung nicht nach Gebieten sondern nach Ober- und Unterverteilung orientiert ist. Damit der **Mandatsrechner2013** möglichst viel wieder verwendet werden kann, wurden Bereiche die in beiden Berechnungsklassen Verwundung finden ausgelagert. Für die Implementierung des Entwurfsmuster Einzelstück wurden möglichst wenig globale Variablen, die vor jeder Berechnung neu initialisiert werden, verwendet, damit in dem Mandatsrechner nicht gewollte Zustände ausgeschlossen werden können.

2.5 Wahlgenerator

Die Klasse **AbstrakterWahlgenerator** bildet die Oberklasse für die Klasse **Wahlgenerator** und enthält alle Attribute und Methoden, die grundsätzlich für die Generierung von Wahlen wichtig sind.
In der Klassse **Wahlgenerator** wird die konkrete Implementierung der Methoden *erzeugeBTW()* und insbesondere von *verteileStimmen()* vorgenommen. Es gibt zusätzlich noch die privaten Methoden *getAnteileVonPartei()*, *getParteienOhneAnteile()*, *verteileRestAnteile()* und *hatParteiStimmanteile()* die intern verwendet werden.

In der Entwurfsphase wurde die Anforderung festgelegt, vollständige Stimmanteile von Parteien auf Bundesebene zu bestimmen und eine Bundestagswahl auf dieser Grundlage zu erzeugen.
Während der Implementierungsphase gab es darüber hinaus den Wunsch auch unvollständige Stimmanteile anzugeben. Damit ist gemeint, dass man beispielsweise nur festlegt,

dass eine Partei keine Stimmen bekommen soll oder dass eine Partei einen bestimmten Anteil von Erst- und/oder Zweitstimmen bekommen soll, unabhängig davon an welche Parteien die restlichen Anteile verteilt werden. Oder auch vollständig zufällige Bundestagswahlen zu generieren. Diese Funktionalität ist in der privaten Methode `verteileRestAnteile()` implementiert, welche von `erzeugeBTW()` verwendet wird.

Der Simulator für das negative Stimmgewicht benötigt außerdem nicht wie ursprünglich gedacht eine eigene Implementierung eines konkreten Wahlgenerators, sondern operiert auf bereits vorhandenen Bundestagswahlen, die bereits in der Anwendung zur Verfügung stehen.

2.6 Simulation des negativen Stimmgewichts

2.7 Wahlvergleich

Das Paket Wahlvergleich besteht nach der Implementierungsphase aus fünf Klassen. Die Hauptklasse ist der **Wahlvergleich** der zwei übergebene Wahlen vergleicht. Wie im Entwurf festgelegt enthält die Klasse die Methode `vergleiche()`, die für den Vergleich von zwei **Bundestagswahlen** sorgt. Jedoch wurden die Parameter, die übergeben werden musste, lieber als Klassenattribute festgehalt, was die Übergabe der gesamten Klasse an das schon in der GUI beschriebene **VergleichsFenster** erleichtern sollte.
Die im Entwurf erwähnte **Parteidifferenz**-Klasse wurde komplett überarbeitet. In dieser werden nur noch die Partei und die Differenz der Sitze von **Bundestagswahl**1 zu **Bundestagswahl**2 gesichert. Die hat den Grund, dass diese Klasse der **DiffDiagramm**-Klasse übergeben wird, welche die Sitzdifferenz veranschaulicht und ein weiterer Bestandteil des Vergleichspaketes ist.
Die Daten des die von der **Wahlvergleich**s Klasse ermittelt werden werden in der Klasse **VergleichDaten** gespeichert und dann in Form einer Tabelle durch die **WahlvergleichTableMo** Klasse dargestellt.

2.8 Steuerung

Die **Steuerung**sklasse bildet die Fassade des gesamten Projektes. In der Implementierungsphase wurden nur die drei folgenden Methoden verändert:

- `importiere()`
  In der Methode die den Import einleitet werden, anders als im Entwurf, mehrere .csv-Dateien benötigt, da wir jetzt nicht nur die Werte für eine Bundestagswahl auswerten, sondern auch die Namen und Listenplätze von Parteimitgliedern. Aus diesem Grund ist eine zweite .csv-Datei notwendig.
  Zukünftig können dadurch auch Dateien, die andere Daten enthalten auch in das Programm importiert werden.

- `zufaelligeWahlgenerierung()`
  Der Methode `zufaelligeWahlgenerierung()` werden nach der Implementie-

rung nicht nur ein **`Stimmenanteile`**-Objekt übergeben, sondern ein ganzer Vektor dieser, da man durch die GUI dazu in der Lage ist mehreren Parteien Stimmenanteile zu gewähren. Außerdem wird eine Ausgangs**`bundestagswahl`** übergeben, von der die neue generierte Wahl Rohdaten, wie Bundesländer, Parteien, Kandidaten, etc., übernimmt. Auch eine Benennung der neuen Wahl ist möglich, weshalb auch ein String übergeben werden muss.

- *`negStimmgewichtGenerierung()`*
  Im Entwurf wurde festgelegt, dass diese Methode ein **`Stimmenanteile`**-Objekt erhält, in der Implementierung haben wir uns entschieden ein **`Bundestagswahl`**-Objekt zu übergeben. Dieses wird dann darauf überprüft, ob eine verwandte Wahl erstellt werden kann, die das Phänomen des negativen Stimmgewichtes aufweisen kann.

2.9 Chronik

Die Chronik ist der Grund, weshalb wir beschlossen haben, dass Erst- und Zweitstimmen nur noch im Wahlkreis veränderbar sind. Hier hat sich das Problem in den Weg gestellt, z.B. die gesetzten Zweitstimmen in Deutschland können nicht ordnungsgemäß auf die Bundesländer und Wahlkreise verteilt werden, falls die neue Zweitstimmenanzahl die Anzahl der Wahlberechtigten übersteigt. In diesem Falle kann keine Stimme rückgängig gemacht werden.
Unser erster Gedanke war es, eine ganze Bundestagswahl statt einer Stimme zu speichern. Dies ist jedoch ebenfalls verfallen, da die Chronik für die Deep-Copy Funktion der Bundestagswahl serialisierbar sein muss. In diesem Falle, würde auch die ganze Chronik gespeichert werden, was unvorhersehbare Folgen haben könnte.

2.10 Sonstiges

2.10.1 Konfiguration

Dieses Modul ist im Entwurf überhaupt nicht enthalten, und wurde erst in der Implementierung hinzugefügt. Es hat die Aufgabe, Einstellungen/Informationen, ohne festes Coden, im Programm zu benutzen. Ein Anwendungsbeispiel ist die Einwohnerzahl der Bundesländer, die essentiell für das Berechnen der Mandatsverteilung. Außerdem beispielsweise die Farben der Parteien im RGB.
Das Modul speichert die Konfiguration in einem Tabulator-getrennten Format. Dies ermöglicht die Datei sehr leicht und lesbar in einem Texteditor oder Excel zu bearbeiten. Die Konfiguration wurde mit weiteren Funktionen beliefert. So sind die Werte auch programmatisch veränderbar. In der GUI ist die Konfiguration zur Zeit nicht änderbar, jedoch ist dies sehr einfach umsetzbar. In unseren Anforderungen war dies nicht enthalten, und wurde daher weggelassen.

2.10.2 Handbuch/About/Lizenz

Das Handbuch wurde in Form einer Webview umgesetzt. Dieser beinhaltet zur Zeit ein Tutorial mit Bildern, welches dem Benutzer den Einstieg in das Programm erleichtern soll.
Auch ein Lizenz- und About-Fenster sind in der Menüleiste Hilfe enthalten.

2.10.3 Meldung

Die **`Meldung`**-Klasse sollte ursprünglich ausschließlich Fehlermeldungen beinhalten. Jedoch wurde schon Anfang der Implementierung beschlossen, diese Klasse für allerlei Ausgaben an der GUI zu verwenden. Dadurch können später Sprachen sehr leicht ausgetauscht verwenden, falls nötig.
In der letzten Implementierung wurde dies leider nicht ausgiebig verwendet, dies soll sich jedoch in der zukünftigen Entwicklung ändern.

2.10.4 Debug

Der Debugger war ebenfalls ungeplant, und wurde als sehr sinnvoll empfunden. Da unser Programm viele Berechnungen möglichst genau berechnen soll, ist beispielsweise eine strenge Überprüfung von Berechnungen essentiell.
Die Debugger ist eine statische Klasse und kann mit einem Wert `Debug.setActive(true)` sehr leicht aktiviert werden.

3 Fazit

Die Implementierungsphase verlief im Vergleich zu den bisherigen Phasen sehr belebt und um einiges koordinierter. Wir haben die einzelnen Unterphasen, die wir vorher festgelegt hatten, mehr oder weniger eingehalten. Gegen Ende haben sich jedoch Mängel angesammelt, weshalb wir in Zeitdruck gerieten. Durch gute Teamarbeit und Durchhaltevermögen unserer Teammitglieder ist es uns jedoch gelungen unser Zeitplan einzuhalten.
Alles in allem, hat uns die Implementierungsphase um einiges mehr Spaß gemacht, als die Entwurfsphase. Unter anderem, weil wir einen Plan hatten und wussten wie wir vorgehen sollten. Jedoch ist uns auch aufgefallen, dass das Wasserfallprojekt nicht immer ganz optimal ist. Unserer Meinung nach, sind Veränderungen am Entwurf unumgänglich.

4 Vorschau auf die nächste Phase

Die nächste Phase ist die Qualitätssicherung. In dieser Phase werden wir unser Programm ordentlichen Tests unterziehen und einige Performance-Probleme beheben.

4.1 Ideen und Ziele

Zu unseren Zielen gehört es an erster Stelle, JUnit-Tests durchzuführen, um unser Programm stabiler zu machen und übersehene Fehler zu beheben. Unser zweites großes Ziel ist es, Code-Coverage Tests durchzuführen.

4.2 Zeitplan

4.2.1 Woche 1

4.2.2 Woche 2