

Bericht

C++ Tetris

Autoren Nicola Käser
Simon Plattner
Dozent Ivo Oesch
Datum 12. Juni 2014
Ort Burgdorf
Version 1.0.0

Im Rahmen des C++ Unterrichts wird an der Berner Fachhochschule jeweils in Gruppen ein kleines Projekt geplant und durchgeführt. Diese Dokumentation beschreibt das Vorgehen sowie das Ergebnis eines solchen Projektes.

Abstract

Im C++ Unterricht der Berner Fachhochschule wird im Rahmen eines Projektes ein „Klon“ des bekannten Spiels *TETRIS*¹ (welches dieses Jahr das 30-jährige Jubiläum feiert) implementiert. Das Projekt ist weitestgehend objektorientiert aufgebaut und umgesetzt. Die Entwicklung fand dabei hauptsächlich im Bereich der Spiellogik statt. Die Darstellung wird durch nicht zum Projekt gehörende Komponenten erledigt.

Das Projekt dient dazu, das gelernte Wissen aus dem zugehörigen Vorlesungsmodul anzuwenden und zu verinnerlichen. Die Dokumentation dient dazu, die gewonnenen Kenntnisse im Bereich UML zu demonstrieren.

Aufgrund anderer, parallel verlaufener Projekte mit höherer Priorität, konnte das Spiel nicht vollständig fertig gestellt werden. Da das Design bereits für das gesamte Spiel vorliegt, musste jedoch ausschliesslich auf *die Implementierung* einiger Elemente verzichtet werden.

Eine Weiterentwicklung an diesem Projekt scheint wenig Sinnvoll. Es ist von Anfang an nur dazu gedacht, das erworbene Wissen in der Praxis anzuwenden und ist daher eher als C++ Sandbox der Autoren aufzufassen und sollte nicht weiterverwendet werden.

¹Wir haben keinerlei Rechte weder am ursprünglichen Spiel noch am Namen desselben. Da das Wort *TETRIS* jedoch bereits im normalen Sprachgebrauch üblich ist, verwenden wir in diesem Dokument die Bezeichnung *TETRIS* für die Spielidee, nicht jedoch als Produktname.

Inhaltsverzeichnis

1	Analyse	1
1.1	Pflichtenheft	1
1.2	Use-Cases	1
1.3	Hardware	4
2	Design	5
2.1	Klassendiagramm	5
2.2	Beschreibung der Tetromino-Klassen	7
2.3	Sequenzdiagramme	9
3	Fertigstellungsgrad	12
4	Fortsetzung	13
4.1	Bekannte Bugs	13
A	Code	16

1 Analyse

Zuerst soll eine Analyse des Projektes durchgeführt werden. Dazu werden verschiedene UML Diagramme erzeugt. Als erstes kommt das Use-Case-Diagramm zum Einsatz. Abbildung 1.1 zeigt einige Anwendungsfälle auf:

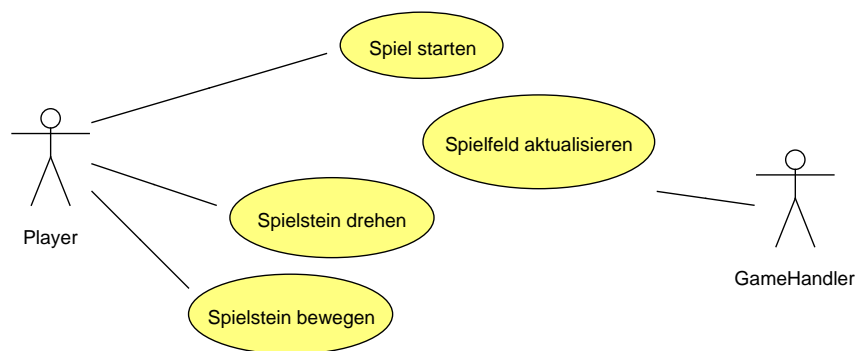


Abbildung 1.1: Anwendungsfall-Diagramm

1.1 Pflichtenheft

Es soll ein TETRIS-Spiel in C++ implementiert werden. Das Spielprinzip wird dabei dem klassischen TETRIS-Spiel nachempfunden. Der Funktionsumfang unserer Kopie wird aus Zeitgründen etwas eingeschränkt. Namentlich verzichten wir auf eine Menüführung und ein Punktesystem sowie auf die Vorschau auf den nächsten erzeugten Block.

1.2 Use-Cases

Im Nachfolgenden sollen einige Anwendungsfälle betrachtet werden. Bei einigen Fällen wurden die Details reduziert abgebildet.

1.2.1 Spiel starten

Tabelle 1.1 zeigt den Anwendungsfall, bei dem ein neues Spiel gestartet wird. Dieser Fall kommt genau einmal beim Start der Applikation vor. Das Spiel kann ohne Programmneustart nicht neu gestartet werden. Ein erneutes Auftreten des Anwendungsfalls ist damit wegbedingt.

Fall Nr.	1	Spiel starten
Beschreibung:	Der Spieler startet ein neues Spiel. Das Spielfeld wird geleert, der GameHandler wird gestartet.	
Vorbedingung:	Das System ist eingeschaltet. Es läuft noch kein Spiel.	

Tabelle 1.1: Anwendungsfall *Spiel starten*

1.2.2 Spielfeld aktualisieren

Tabelle 1.2 zeigt den Anwendungsfall, bei dem das Spielfeld durch den GameHandler aktualisiert wird. Dieser Fall tritt periodisch auf. Für den Aufruf entsprechender Methoden sorgt ein Softwaretimer.

Fall Nr.	2	Spielfeld aktualisieren
Beschreibung:	Der GameHandler reagiert auf Benutzereingaben und aktualisiert das Spielfeld. Wenn eine Reihe komplettiert wurde, wird diese entfernt. Wenn das Spielfeld voll ist, wird das Spiel abgebrochen.	
Vorbedingung:	Das Spiel läuft. Der Player macht eine Bewegung oder eine Drehung des Spielsteins oder der GameHandler aktualisiert das Spiel.	

Tabelle 1.2: Anwendungsfall *Spielfeld aktualisieren*

1.2.3 Spielstein drehen

Tabelle 1.3 zeigt den Anwendungsfall, bei dem ein Spielstein (ein sogenanntes *Tetromino*¹) gedreht wird. Das passiert unabhängig von den Zeitpunkten, zu denen der GameHandler das Spielfeld aktualisieren lässt. Siehe hierzu auch Abbildung 2.4 in Abschnitt 2.3.1.

Fall Nr.	3	Spielstein drehen
Beschreibung:	Ein Tetromino wird nach einer Benutzereingabe rotiert. Um unzulässige Drehungen zu vermeiden, prüft das Spiel die Eingabe auf Zulässigkeit.	
Vorbedingung:	Das Spiel läuft.	
Ablauf:		
	E1)	Der Spieler drückt die Taste F1 oder F2
	A1)	Das Spiel rotiert falls möglich das aktive Tetromino
Auswirkungen:	Das Tetromino hat eine neue Ausrichtung.	
Weitere Informationen:	Drehen ist nur möglich, falls dabei eine zulässige Bewegung statt findet (beispielsweise darf das Tetromino nicht in ein anderes Tetromino „hineingedreht“ werden).	
Diagramme:	Abbildung 2.4	

Tabelle 1.3: Anwendungsfall *Spielstein drehen*

¹Die Bezeichnung *Tetromino* steht im englischen für eine Geometrische Figur, die aus vier Quadraten aufgebaut ist (und einigen weiteren Anforderungen genügt). Die klassischen Spielsteine aus *TETRIS* sind Tetrominos.

1.2.4 Spielstein bewegen

Tabelle 1.4 zeigt den Anwendungsfall, bei dem ein *Tetromino* bewegt wird. Auch das passiert unabhängig von den Zeitpunkten, zu denen der GameHandler das Spielfeld aktualisieren lässt.




Fall Nr.	4	Spielstein bewegen
Beschreibung:	Ein Tetromino wird nach einer Benutzereingabe verschoben. Um unzulässige Positionen zu vermeiden, prüft das Spiel die Eingabe auf Zulässigkeit.	
Vorbedingung:	Das Spiel läuft.	
Ablauf:		
	E1)	Der Spieler drückt eine der Richtungstasten  ,  , oder 
	A1)	Das Spiel verschiebt falls möglich das aktive Tetromino
Auswirkungen:	Das Tetromino hat eine neue Position.	
Weitere Informationen:	Bewegen ist nur möglich, falls an der Zielposition alle vier Blöcke frei sind.	
Diagramme:	Abbildung 2.4	

Tabelle 1.4: Anwendungsfall *Spielstein bewegen*

1.3 Hardware

Die fertige Software soll am PC (unter Windows) ausgeführt werden können. Dazu soll auf eine bestehende Grafik-Bibliothek² zurückgegriffen werden.

Die genannte Bibliothek stellt einige weitere Funktionalität zur Verfügung. So kann sie beispielsweise eingesetzt werden, um Tastatureingaben einzulesen.

²Ivo's *Worm-Library* :)

2 Design

Durch die Aufgabenstellung ist eine Implementierung in der objektorientierten Sprache C++ vorgeschrieben. Dieses Kapitel zeigt die Überlegungen zum Software-Design.

2.1 Klassendiagramm

Das Projekt ist gemäss den Klassen in Abbildung 2.3 aufgebaut. Die Darstellung der Klassen entspricht dabei der Konvention von Bild 2.1.

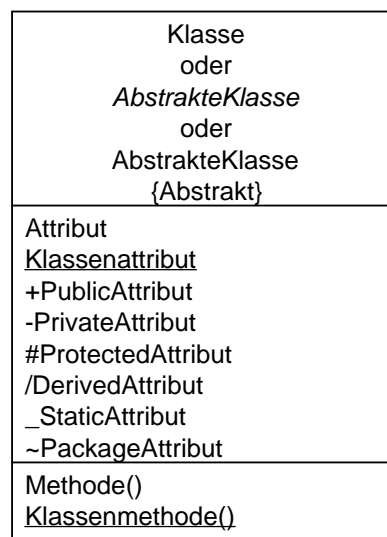


Abbildung 2.1: Konvention zum Klassendiagramm

Die cyanfarbenen Klassen sind Singletons, existieren also je genau einmal. Die weiss dargestellten Klassen werden zur Laufzeit erzeugt und zerstört. Von ihnen können zu einem gegebenen Zeitpunkt mehrere Instanzen existieren¹.

¹Ausnahme abstrakte Klassen; diese sind auch weiss dargestellt, können jedoch nicht instanziiert werden.

Die Tetrominos kommen in drei Grössen vor. Das kleinste mögliche Tetromino (mit der zugrunde liegenden Klasse *Tetromino4*) besteht aus vier Blocks. Jeder dieser Blocks ist einerseits ein Square, andererseits besitzt er eine Farbe. Der Aufruf der Methode `Rotate()` hat für diese Tetromino-Art keinen Einfluss.

Die grösseren Tetrominos (mit den zugrunde liegenden Klassen *Tetromino9* und *Tetromino16*) verfügen zusätzlich zu den sichtbaren Quadraten (= Blocks) auch noch eine Menge an unsichtbaren Elementen. Abbildung 2.2 stellt diesen Sachverhalt dar. Die schwarzen Bereiche bestehen aus Spaces. Spaces sind Quadrate ohne Farbe. Es sind reine Platzhalter. Da jedes Tetromino genau vier Blocks (farbige Quadrate) enthält, ist auch die Anzahl Spaces (unsichtbare Quadrate) für jedes Tetromino über seine Lebenszeit konstant.

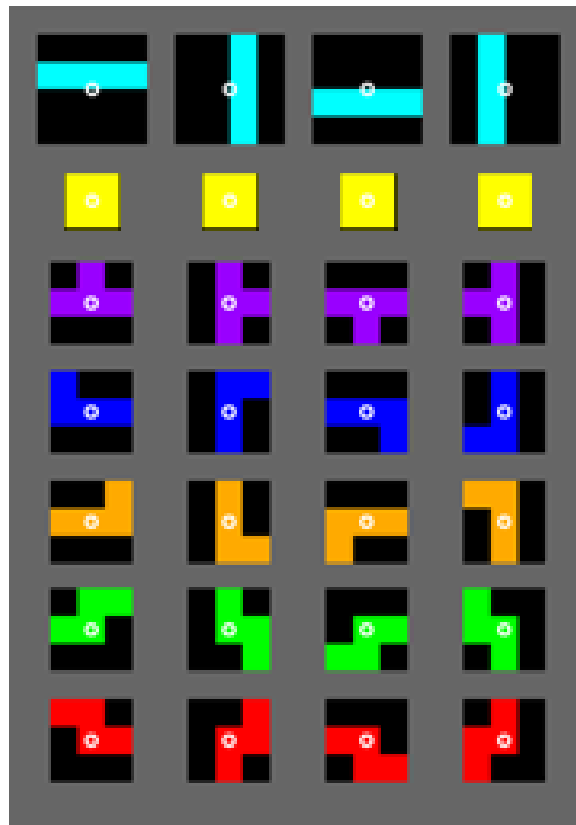


Abbildung 2.2: Ausrichtung der Elemente (Reihenfolge: **I, O, T, J, L, S, Z**)
(Quelle: <http://tetris.wikia.com/wiki/SRS>, von uns modifiziert)

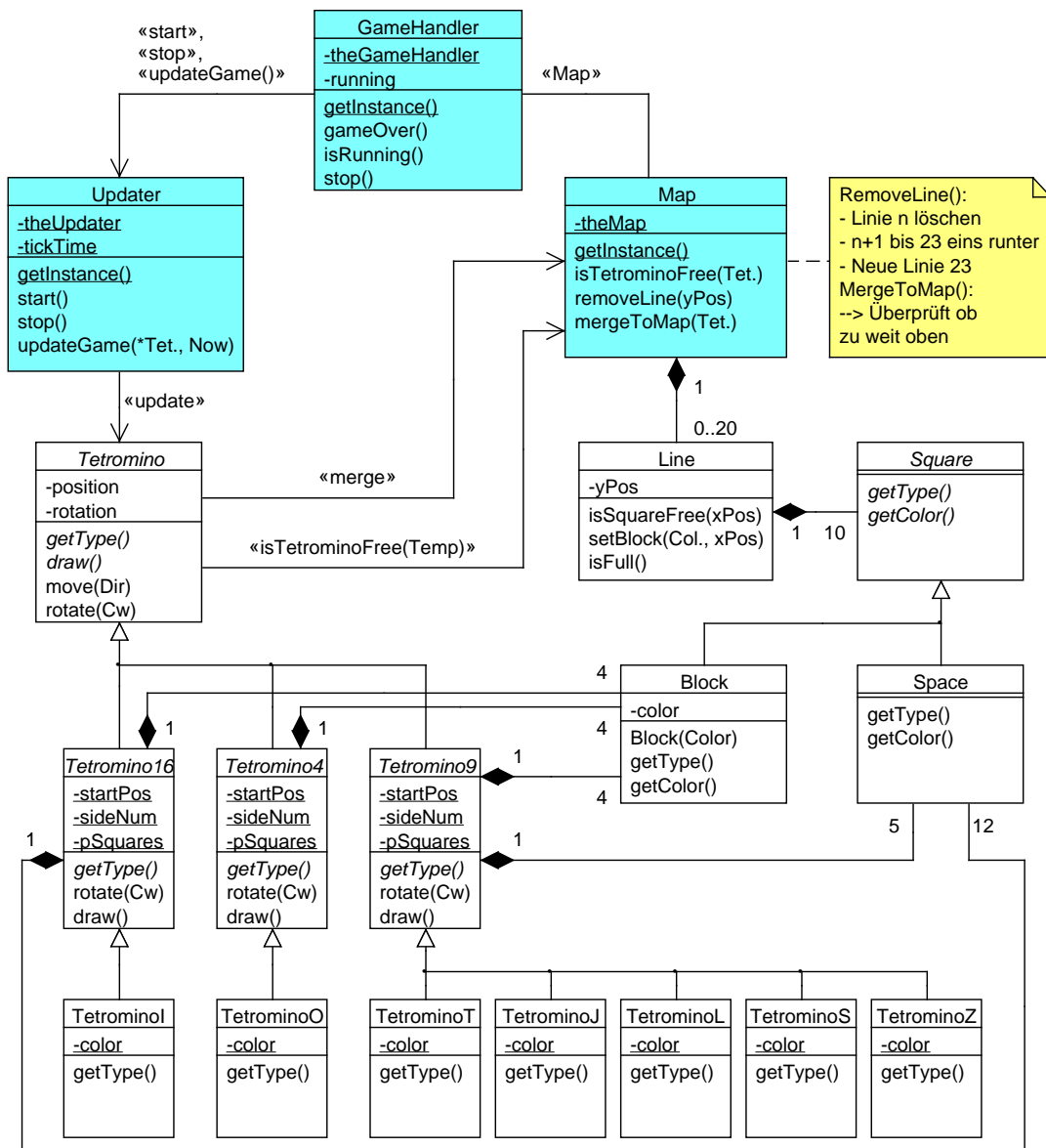


Abbildung 2.3: Klassendiagramm

2.2 Beschreibung der Tetromino-Klassen

In diesem Abschnitt werden die verschiedenen Tetromino-Klassen beschrieben.

2.2.1 Tetromino

Dies ist die oberste der Tetromino-Klassen, sie enthält allgemeine Informationen über die Position (**position**) und Ausrichtung (**rotation**) des Tetrominos. Zum Aktualisieren dieser Informationen sind zwei Methoden vorhanden:

move(Point2D Dir) ändert die Positionsinformation um soviel in x-/ y-Richtung wie mit **Dir** angegeben wird.

rotate(bool Cw) ändert die Rotationsinformation auf den nächsten Wert, je nach dem welche Richtung mit **Cw** angegeben wird.

Die Klasse besitzt zusätzlich zwei *rein virtuelle* Methoden **getType()** und **draw()**, und ist somit eine *abstrakte* Klasse.

2.2.2 Tetromino16, Tetromino4 und Tetromino9

Dies sind die direkten Unterklassen der **Tetromino**-Klasse, sie enthalten Informationen die abhängig von der Tetromino-Grösse sind. Da es genau drei unterschiedliche Tetromino-Grössen im Spiel gibt, sind diese Informationen in drei Klassen zusammengefasst. Die vorhandenen Informationen sind die Seitenlänge in Anzahl Quadraten (**sideNum**), die Startposition (**startPos**) sowie die zweidimensionale Liste der Quadraten (**pSquares**).

Zwei Methoden behandeln diese Informationen:

rotate(Cw) ist auch in diesen Klassen wieder vorhanden. Diese Methode ruft die gleichnamige Methode der Überklasse auf und ordnet zusätzlich die Quadrate in der Liste neu an.

draw() zeichnet das Tetromino an der richtigen Stelle auf den Bildschirm. Die *virtuelle Methode* aus der Überklasse wird also hier implementiert.

Auch diese Klassen besitzen zusätzlich je eine *rein virtuelle* Methode **getType()**, und sind somit auch *abstrakte* Klassen.

2.2.3 TetrominoI, TetrominoO, TetrominoT, TetrominoJ, TetrominoL, TetrominoS, TetrominoZ

Dies sind die untersten Tetromino-Klassen, sie enthalten Informationen über Form (indirekt) und Farbe (**color**) des Tetrominos. Die *virtuelle Methode* **getType()** der Oberklassen wird erst hier implementiert und liefert die Information über den Typ (Form) des Tetrominos.

2.3 Sequenzdiagramme

2.3.1 Tetromino drehen

Das Sequenzdiagramm in Abbildung 2.4 zeigt den Ablauf zum Rotieren eines Tetrominos. Der zugehörige Akteur ist in jedem Fall der Spieler. Er gibt eine gewünschte Rotation über das Keyboard vor.

Zum Zwecke der Kollisionsverhinderung wird ein neues Tetromino erzeugt, welches die gewünschte Ausrichtung bereits besitzt. Anschliessend wird dieses Tetromino dem Map-Objekt übergeben (mit *IsTetrominoFree(Tetromino)*), um überprüfen zu lassen, ob die gewünschte Ausrichtung ohne Kollision erreicht werden kann.

Das Map-Objekt zerlegt das Tetromino in seine Blöcke und prüft für jeden dieser Blöcke, ob er mit einem existierenden Block (im Line-Container) kollidiert. Nach dieser Überprüfung wird die Information durch Rückgabewerte an den Aufrufer zurückgeliefert, sodass der Aufrufer weiss, ob die Bewegung zulässig ist oder nicht.

Falls die gewünschte Bewegung zulässig ist, wird das gültige Tetromino rotiert. Das neu erzeugte Tetromino wird in jedem Fall anschliessend gelöscht.

Falls die Rotation nicht zulässig ist, hat die Eingabe keinerlei weiterer Auswirkungen auf das Spiel.

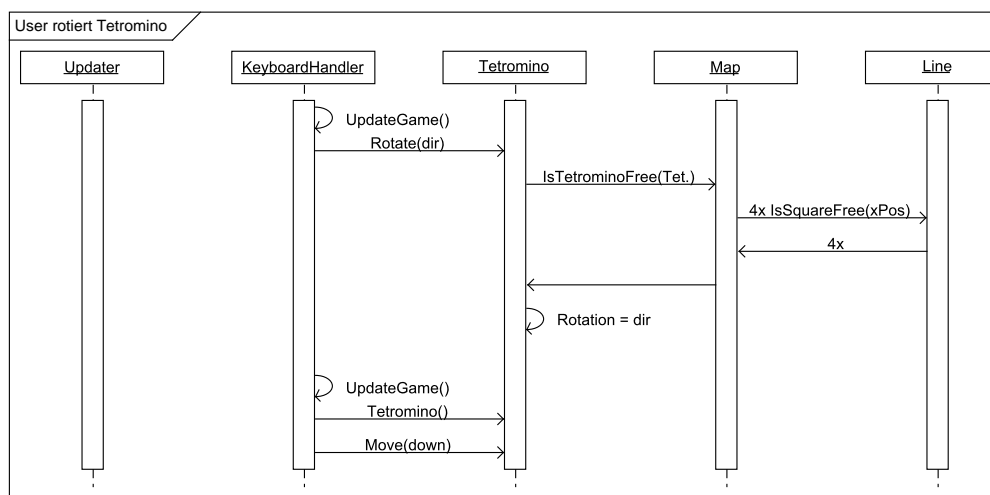


Abbildung 2.4: Sequenzdiagramm: Tetromino rotieren

2.3.2 Tetromino füllt eine Linie

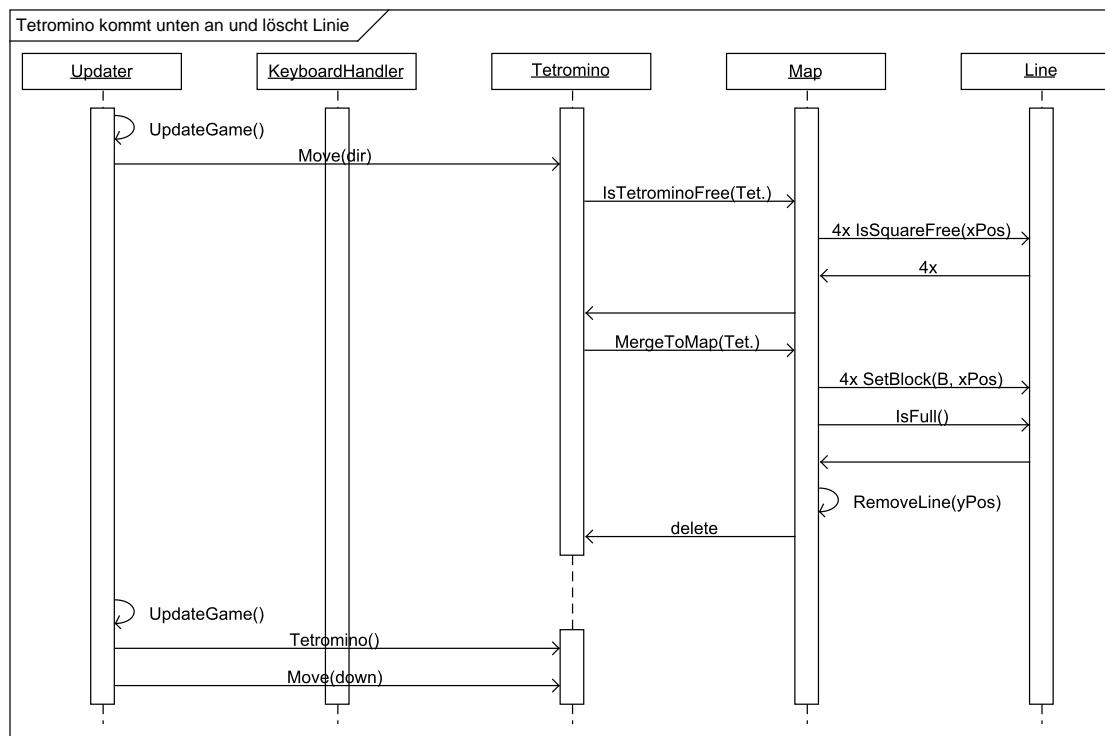


Abbildung 2.5: Sequenzdiagramm: Linie löschen

Abbildung 2.5 zeigt einen etwas komplizierteren Fall. Es zeigt, wie mit einem Tetromino verfahren wird, das eine Linie vervollständigt.

Im Gegensatz zum vorhergehenden Beispiel ist nicht der Spieler sondern der Updater (ein Timer gesteuertes Objekt) der Auslöser der Sequenz. Auch in diesem Fall wird ein neues Tetromino erzeugt. Diesesmal wird es jedoch nicht rotiert sondern verschoben. Auch in diesem Fall wird das neue Tetromino anschliessend dem Map-Objekt übergeben, um mit Hilfe der Line-Container auf Kollisionen untersucht zu werden.

Falls keine Kollision statt findet, das Tetromino also bewegt werden darf, ist der Ablauf sehr ähnlich zu dem der Rotation (Abbildung 2.4).

Falls jedoch eine Kollision statt findet, muss das Tetromino an seiner alten Position in

die Map *merged*² werden. Zu diesem Zweck wird für jedes betroffene Line-Objekt die Methode *SetBlock(...)* ein- bis viermal aufgerufen (je nach Anzahl Blöcke, die in diese Linie gemerged werden sollen). Für jede Betroffene Line wird anschliessend mit der Methode *IsFull()* überprüft, ob sich die Linie als vollständig betrachtet.

Im Fall, dass sich eine Linie als voll betrachtet, wird die betroffene Linie komplett gelöscht. Alle Linien, die sich oberhalb der zu entfernenden Linie befinden, werden jeweils um eine Linienhöhe nach unten verschoben, so dass die entstandene Lücke ausgefüllt wird. Nach diesem Verschieben wird zuoberst in der Map (ausserhalb des Sichtbaren Bereiches) eine neue, leere Linie erzeugt.

²von *to merge*, englisch für *zusammenfügen*, *verschmelzen*

3 Fertigstellungsgrad

Das Projekt ist zum Zeitpunkt der Abgabe nicht vollständig implementiert. Das liegt daran, dass aufgrund parallel laufender anderer Projekte die Zeit für weitere Arbeiten an diesem Projekt fehlte. Folgende Arbeiten sind nicht oder nur teilweise fertig gestellt:

- **„Map“-Klasse:** Diese Klasse ist zum jetzigen Zeitpunkt nur als „Gerüst“ implementiert. Das bedeutet, die Klasse inklusive Attribute und Methoden ist vorhanden, die Methoden haben jedoch noch keine Funktion.


4 Fortsetzung

Eine Weiterführung des Projektes ist nicht vorgesehen und auch nicht empfehlenswert. Trotzdem lassen sich Use-Cases kreieren, in denen eine Weiterführung des Projekts wünschenswert wäre. Als mögliche Erweiterung könnten beispielsweise folgende Funktionen umgesetzt werden:

- **Spiel pausieren:** Eine Funktion zum vorübergehenden Unterbrechen eines laufenden Spiels könnte umgesetzt werden
- **Punktestand festhalten:** Für das laufende Spiel könnten Punkte gesammelt werden. Diese Funktion wäre insbesondere in Kombination mit einer Bestenliste vorstellbar
- **Mehrspielermodus:** Mit grossem Aufwand könnte das Spiel für einen Mehrspielerbetrieb umgerüstet werden

4.1 Bekannte Bugs

Die nachfolgend genannten Bugs sind bekannt:

- Das Tetromino kann sich aus der Map bewegen und bleibt nicht unten liegen. Dieser Bug ist auf die unvollständige Implementation der Klasse „Map“ zurückzuführen. (Siehe Kapitel 3)
- Mit der Richtungstaste  wird das Tetromino entfernt und ein neues am Startpunkt erstellt. Dieses Verhalten wurde absichtlich implementiert, damit die einzelnen Tetrominoklassen zum jetzigen Softwarestand getestet werden können.

Abbildungsverzeichnis

1.1	Anwendungsfall-Diagramm	1
2.1	Konvention zum Klassendiagramm	5
2.2	Ausrichtung der Elemente (Reihenfolge: I, O, T, J, L, S, Z) (Quelle: http://tetris.wikia.com/wiki/SRS , von uns modifiziert)	6
2.3	Klassendiagramm	7
2.4	Sequenzdiagramm: Tetromino rotieren	9
2.5	Sequenzdiagramm: Linie löschen	10

Tabellenverzeichnis

1.1	Anwendungsfall <i>Spiel starten</i>	2
1.2	Anwendungsfall <i>Spielfeld aktualisieren</i>	2
1.3	Anwendungsfall <i>Spielstein drehen</i>	3
1.4	Anwendungsfall <i>Spielstein bewegen</i>	4

A Code

Der Code ist zu umfangreich um sinnvoll im Anhang verpackt werden zu können. Er wird daher in geeigneter digitaler Form der Dokumentation beigelegt.