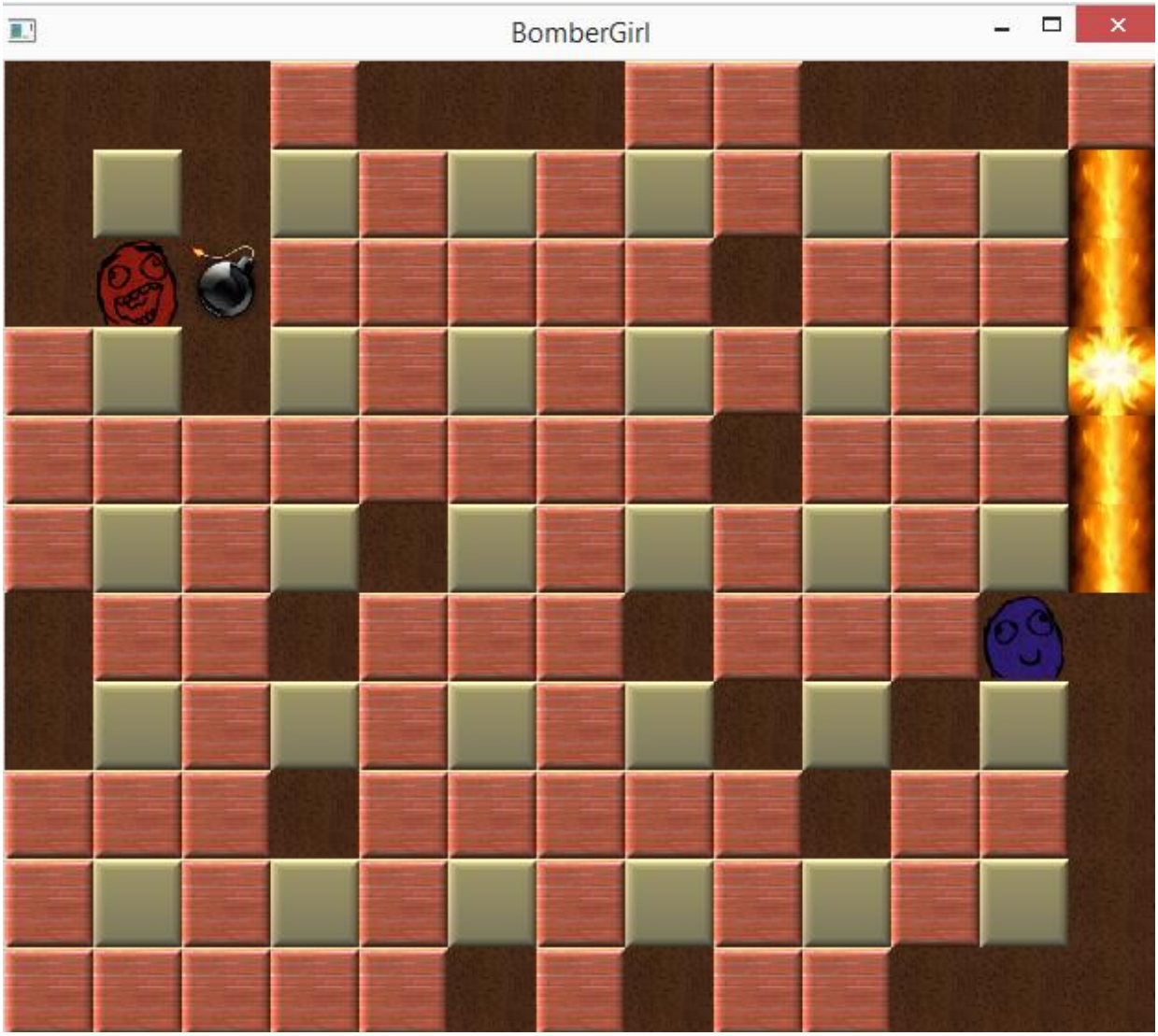
C++ Projekt: BomberGirl



Einleitung

Dieses Dokument beinhaltet die Idee, die grundsätzlichen Entscheidungen und das Design des Projektes *BomberGirl*.

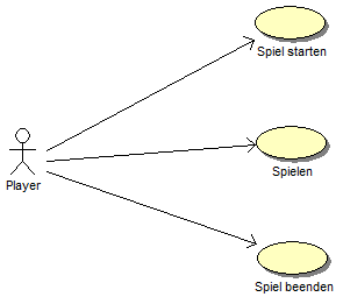
Das Ziel war es, ein kurzes Softwareprojekt für das Unterrichtsmodul *C++ in Embedded Systems* durchzuziehen. Dabei soll das Konzept der objektorientierten Programmierung mit der Programmiersprache C++ und als Software-Projektentwicklungsmethode der *Unified Process* genutzt werden.

Wir haben uns entschieden, das Spiel *BomberMan* zu klonen. Die Zielplattform ist ein PC. Mit der Tastatur können beide Spieler gesteuert werden. Als Grafik-Backend setzen wird das Framework *Qt 5* ein und arbeiten entsprechend auch gerade mit der mitgelieferten IDE *Qt Creater*.

Inhalt

Anwendungsfälle

Als Anwendungsfälle in unserem Spiel haben wir und auf die gröbsten unterschiedlichen Benutzeraktionen beschränkt: Starten, spielen und beenden.



Anwendungsfall 1: „Spiel starten“

Zusammenfassung: Der Benutzer startet die ausführbare Datei und landet schlussendlich im Spiel.
Vorbedingung: Keine.
Ablauf: Das Fenster wird erstellt, das Spielfeld wird initialisiert.

Anwendungsfall 2: „Spielen“

Zusammenfassung: Der Benutzer startet interagiert als Spieler mit dem Spielfeld und dem zweiten Spieler.

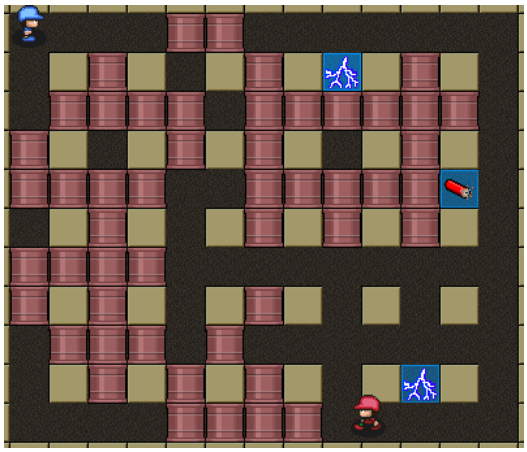
Vorbedingung: Das Spiel wurde gestartet und initialisiert.
Ablauf: Selbstverständlich ist der Anwendungsfall *Spielen* im Hintergrund extrem komplex, aber es macht keinen Sinn, die verschiedenen möglichen Benutzeraktionen noch weiter auszuführen, da es fast unendliche viele unterschiedliche Zustände im Spiel gibt. Deshalb haben wir uns entschieden, den Anwendungsfall *Spielen* im Rahmen der Spielregelbeschreibung genauer zu betrachten.

Anwendungsfall 2: „Spiel beenden“

Zusammenfassung: Der Benutzer beendet das Spiel durch Schliessen der laufenden Anwendung (entweder über das Kreuz oben rechts oder die Tastenkombination [Alt]+[F4]).
Vorbedingung: Das Spiel wurde gestartet und initialisiert.
Ablauf: Die Ressourcen werden wieder freigegeben, das Fenster wird geschlossen und der Prozess beendet.

Spielregeln Beschreibung

Wir orientieren uns für diese Spielregeln am Spiel „Bomberman“ welches auf der Website www.spielen.com gratis spielbar ist (http://www.spielen.com/spiel/fire-play-2). Hier ein Printscreen davon:



Umgebung

Das Spielfeld ist in quadratische Felder aufgeteilt. Es ist 13 Felder breit und 11 Felder hoch. Es gibt verschiedene Arten von Feldern.

Felder

- *Ground*. Auf diesem Boden kann man sich fortbewegen.
- *Wall*. Durch diesen kann man nicht hindurchgehen und man kann ihn auch nicht zerstören.
- *Brick*. Diesen kann man zerstören. Er verschwindet danach. Eventuell ist darunter dann noch ein *Item*.
- *Item*. Dieses kann man durch darübergehen aufnehmen. Es verschwindet danach. Es gibt zwei unterschiedliche *Items*:
    - *BombItem*: Gibt dem Spieler die Möglichkeit, mehr Bomben zur gleichen Zeit zu legen.
    - *FlashItem*: Vergrössert die Ausdehnung der Bombenexplosion um ein Feld in jede Richtung.
- *Bomb*. Diese wird vom Spieler platziert und explodiert nach einer gewissen Zeit.

- *Fire*. Hat eine gewisse Ausdehnung und resultiert aus der Explosion der *Bomb*. Bleibt eine gewisse Zeit stehen und verschwindet danach. Wenn der Spieler das *Fire* berührt, stirbt er.

Feldanordnung

Die *Walls* (30 Stück) werden wie in der Abbildung oben angeordnet (jedoch *ohne* die äusserste Begrenzung). Auf vier Feldern befindet sich in der Regel eine *Wall*. Diese Anordnung ist fix. Daneben werden beim Spielstart auf den verbliebenen leeren Feldern 73 *Bricks* zufällig verteilt. Diese bestehen aus 30 (ca. 40%) *Bricks* mit *Item* darunter und 43 (ca. 60%) *Bricks* ohne *Item* darunter. Die *Items* bestehen zufälligerweise aus *BombItems* (ca. 50%) und *FlashItems* (ca. 50%).

Spielablauf und Ziel

Zwei Spieler spielen gegeneinander. Sie können ihre Spielfigur gleichzeitig kontrollieren. Das Ziel des jeweiligen Spielers ist, den anderen Spieler durch eine Bombenexplosion zu töten.

Aktionen

Die beiden Spieler können ihre Spielfigur im Raster des Spielfeldes auf den *Ground*-Feldern horizontal und vertikal bewegen.

Sie können auf jedem Ground-Feld eine Bombe platzieren welche nach einer gewissen Zeit detoniert. Das Feuer der Explosion hat eine gewisse Ausdehnung (kann durch das *FlashItem* vergrössert werden) in horizontaler und vertikaler Richtung und wird durch *Walls* oder *Bricks* eingedämmt und verschwindet nach einer gewissen Zeit. Trifft das Feuer auf einen *Brick*, wird dieser nach verschwinden des Feuers ausgelöscht. Wenn es ein *Brick* mit *Item* darunter war (optisch nicht vom leeren *Brick* zu unterscheiden), wird danach auf dem gleichen Feld ein zufälliges *Item* platziert.

Dieses kann von der Spielfigur durch darübergehen aufgenommen werden. Danach verschwindet das *Item* und der Spieler erhält die entsprechende Fähigkeitsverbesserung gutgeschrieben (*BombItem* oder *FlashItem*).

Jede Spielfigur kann zu Beginn nur eine Bombe gleichzeitig legen. Mit dem *BombItem* kann diese Anzahl erhöht werden.

Steuerung

Die Spielfiguren können mit der Tastatur gesteuert werden. Jeder Spieler hat fünf Tasten zur Verfügung: Vier für die Laufrichtung und eine für das Platzieren der Bomben:

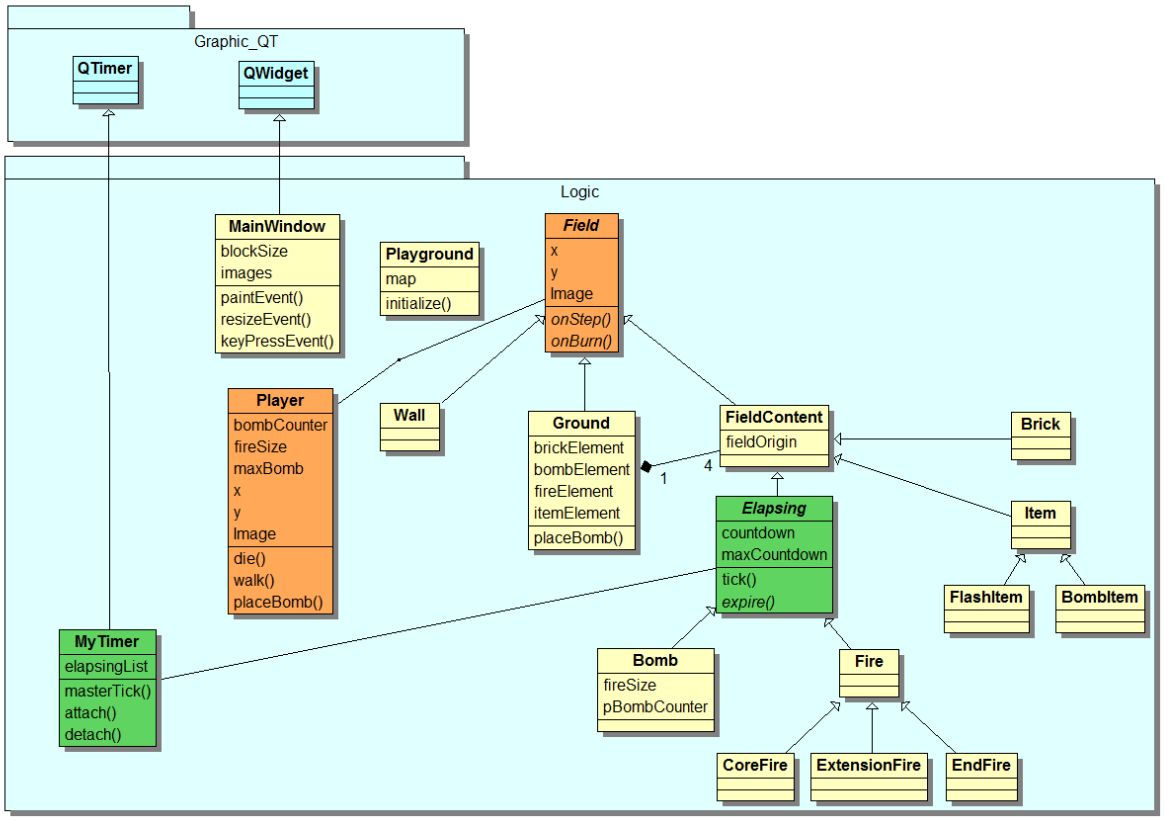
Steuerung	Spieler 1	Spieler 2
Up	*W*	*Up Arrow*
Down	*S*	*Down Arrow*
Left	*A*	*Left Arrow*
Right	*D*	*Right Arrow*
Drop Bomb	*Tab*	*Dot (.)*

Grafik

Das Spiel wird in einem Fenster implementiert, welches ausschliesslich das Spielfeld beinhaltet. Die grafische Darstellung wird mit dem Framework *Qt 5* programmiert. Ausser den Zeichnungsklassen und den Klassen für den grundlegenden Fensteraufbau werden von Qt keine Funktionalitäten oder Klassen verwendet.

Klassendiagramm

Hier ist eine grobübersicht über alle Klassen und ihre Vererbungen:



Es wurde in diesem Diagramm auch bereits eine erste Komposition zwischen *Ground* und *FieldContent* eingezeichnet, da diese sehr wichtig für das Verständnis der Klassenhierarchie ist.

Grundsätzlich lassen sich die Klassen in zwei Pakete aufteilen: Spielbasierte Klassen und grafikbasierte Klassen. Die grafikbasierten Klassen werden von Qt zur Verfügung gestellt. Sie werden nur per Klassennamen erwähnt und dabei auch nur die, welche in anderen Diagrammen vorkommen oder wichtig für das Verständnis sind.

Erklärung zur Datenstrukturierung rund um Ground und FieldContent

Bei den spielbasierten Klassen, die wir selbst geschrieben haben, ist vor allem der Baum von *Field* und dessen Vererbungen wichtig: Ein *Field* ist ein einzelnes abstraktes quadratisches Feld, aus welchem sich das Spielfeld zusammensetzt. Davon gibt es dann diverse Variationen. Alle diese Felder sind in einer 2D-Variable *map* einsehbar.

Die *map* ist ein normales 2D-Array aus *Field*-Pointern. Jede Position ist entweder eine *Wall* oder ein *Ground*-Feld. Die *Ground*-Felder ihrerseits beinhalten dann entweder noch ein *Brick*, oder ein *Item* und darüber ein *Brick*. Wenn auf einem *Ground*-Feld eine Bombe abgelegt wurde oder darüber eine Bombe explodiert, enthält es noch ein *Bomb* oder *Fire*-Objekt.

Dafür wurde das *Ground*-Objekt mit vier „Speicherplätzen“ aus *FieldContent*-Vererbten Objekten ausgestattet. Diese vier sind: Ein Platz für ein *Brick*-Objekt, ein Platz für ein *Item*-Objekt, ein Platz für ein *Bomb*-Objekt und ein Platz für ein *Fire*-Objekt. Beim *Fire*- und beim *Item*-Objekt gibt es dann wiederum unterschiedliche Varianten welche wir noch einmal durch Vererbung kategorisiert haben.

Diese sehr tiefe und strukturierte Klassenhierarchie der *Field*-Vererbung haben wir gewählt, damit man darauf schlussendlich in Zusammenhand mit dem Spielen das Design-Pattern *Visitor-Pattern* anwenden kann.

VisitorPattern

Für die Neupositionierung des Spieler und die Interaktion von Feldern mit Bombenexplosionen wird das C++ Design-Pattern *Visitor-Pattern* angewendet. Die Klasse Field besitzt zwei Methoden: onStep() und onBurn().

onStep()

Will einer der Spieler seine Position auf ein neues Feld setzen, führt er die Methode onStep() auf dem gewünschten Feld aus. Die Methode onStep verlangt als Argument eine Referenz auf den Spieler um Variablen wie fireSize und maxBombs inkrementieren zu können. Die verschiedenen Reaktionen auf das „Betreten“ des Feldes werden hier beschrieben:

Rückgabewerte von onStep(Player &p)

Diese Methode ist rein virtuell auf Stufe *Field* - es macht keinen Sinn, ein *Field*-Objekt zu erzeugen.

- ➔ NOENTRY bei Wall, belässt den Player unverändert
- ➔ ENTRY bei einem Ground-Field ohne etwas darauf.
- ➔ UPGRADE bei einem Ground-Field mit einem Item-Objekt darauf.
  - UPGRADEFLASH wenn das Item-Objekt ein FlashItem ist. Inkrementiert fireSize des Spielers.
  - UPGRADEBOMBS wenn das Item-Objekt ein BombItem ist. Inkrementiert maxBombs des Spielers.
- ➔ DIE bei einem Ground-Field mit einem Fire-Objekt daraud. Führt die Funktion die() des Spielers aus.

onBurn()

Breitet sich die Explosion einer Bombe auf dem Spielfeld aus, wird hier ebenfalls für jedes „verbrannte“ Feld eine Methode ausgeführt: onBurn. Diese Methode besitzt kein Argument, sie arbeitet alleine mit Rückgabewerten. Eventuell könnte man die spezifische Funktionalität der untergeordneten Klassen (z.B. onExplode() für eine Bombe oder onDestroy() für einen Brick) dort noch in separate Methoden kapseln, welche dann von der endgültigen onBurn()-Implementation aufgerufen werden. Aber eigentlich kann das auch gerade die onBurn()-Implementaton handlen.

Die verschiedenen Reaktionen auf das „Verbrennen“ des Feldes werden hier aufgeführt:

Rückgabewerte von onBurn()

Diese Methode ist rein virtuell auf Stufe Field – es macht keinen Sinn, ein Field Objekt zu erzeugen.

- ➔ HARDBLOCK bei Wall
- ➔ NOACTION bei einem Ground-Field ohne etwas darauf.
- ➔ BLOCK bei einem Ground-Field mit einem Brick-Objekt darauf.
- ➔ TRIGGER bei einem Ground-Field mit einem Bomb-Objekt darauf. Hier wird dann direkt die expire()-Methode der Bombe aufgerufen (sofortige Explosion).

Da das onBurn() und onStep() des *Ground*-Feldes dort eigentlich „stehen bleibt“ (weil keine weitere Vererbung nach unten vorhanden ist), wird das weitergeben dieses Visitor-Befehls dort manuell gemacht. Je nachdem welche der vier FieldContent Speicherplätze besetzt sind, haben diese unterschiedliche Priorität und dementsprechend wird nur der Rückgabewert des höchstpriorisierten Objekts zurückgegeben.

ObserverPattern

Für das Timing von *Bomb* und *Fire* wird das Design-Pattern *Observer-Pattern* eingesetzt. Der Observer ist ein Objekt, welches von *Elapsing* erbt (also entweder *Bomb* oder *Fire*). Er hängt sich in eine Liste der Subject-Klasse *MyTimer* ein, welche ihm dann in Abständen von 100ms einen Tick schickt (also es führt die Methode tick() des Observers auf). Wenn das Objekt stirbt, gibt es als return-Wert der tick()-Methode -1 zurück. Darauf ruft die *MyTimer*-Klasse den Destruktor des Observers (also der *Bomb* oder des *Fires*) auf und entfernt es anschliessend aus seiner Liste mit zu benachrichtigenden Objekten.

Die tick()-Methode zählt einen countdown runter und ruft bei Erreichen von 0 die Methode expire() auf. Diese muss dann die eigentliche Aktion bei Ablaufen des Timers vornehmen, deshalb ist sie in der Elapsing-Klasse noch rein virtuell deklariert.

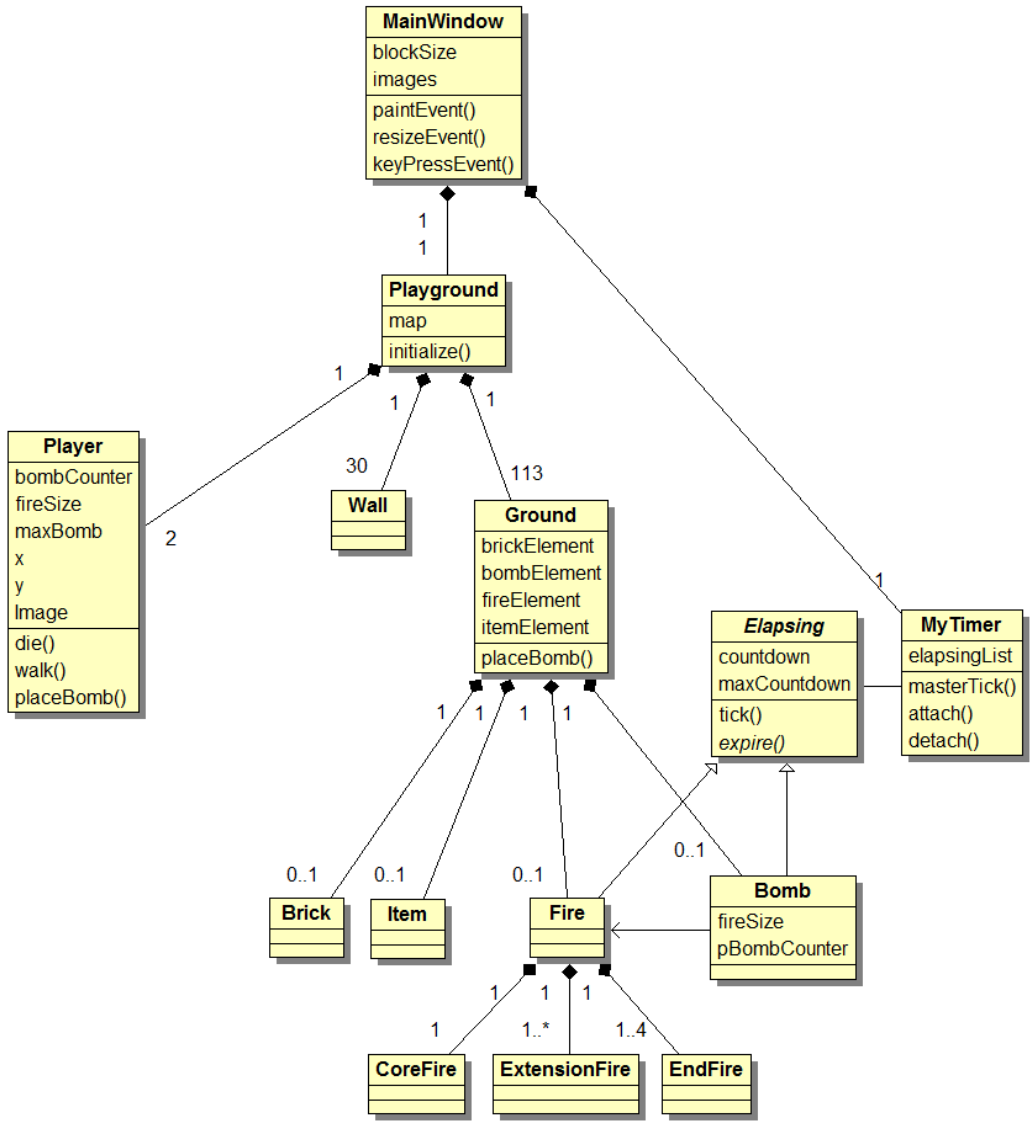
Das Subject (die Klasse *MyTimer*, von welcher nur ein Objekt instanziiert wird) hat eine STL-Liste mit den *Elapsing*-Objekten.

Spezialfall wenn eine Bombe eine andere triggert: Während dem masterTick() wird sich die andere Bombe ausklinken wollen. Das heisst es wird nötig sein, ein Element aus dieser Liste vor oder nach dem aktuellen Element zu löschen (welches dann auch gerade gelöscht wird). Ein Workaround dafür wäre, dass ein „elapsed“-Rückgabewert der tick()-Methode das Objekt (oder am besten einen Zeiger auf das Listenelement) in der danach ausgeführten detach()-Methode erst in eine Liste von zu löschenden Listenobjekten eingetragen wird. Diese Liste wird dann jedes Mal nach der For-Schleife der MasterTick()-Methode abgearbeitet und die entsprechenden Objekte gelöscht.

Der Countdown der *Bomb* startet bei 20 (also 2 Sekunden) und der Countdown des *Fires* bei 10 (also 1 Sekunde)

Assoziationen der Klassen

Die Anzahl Felder, welche von jeder Sorte auf dem Spielfeld sind, wurde schon vorher erwähnt, ebenso die Beziehungen unter den Klassen. Auf dem folgenden Klassendiagramm mit eingezeichneten Assoziationen werden diese noch einmal etwas strukturierter dargestellt:



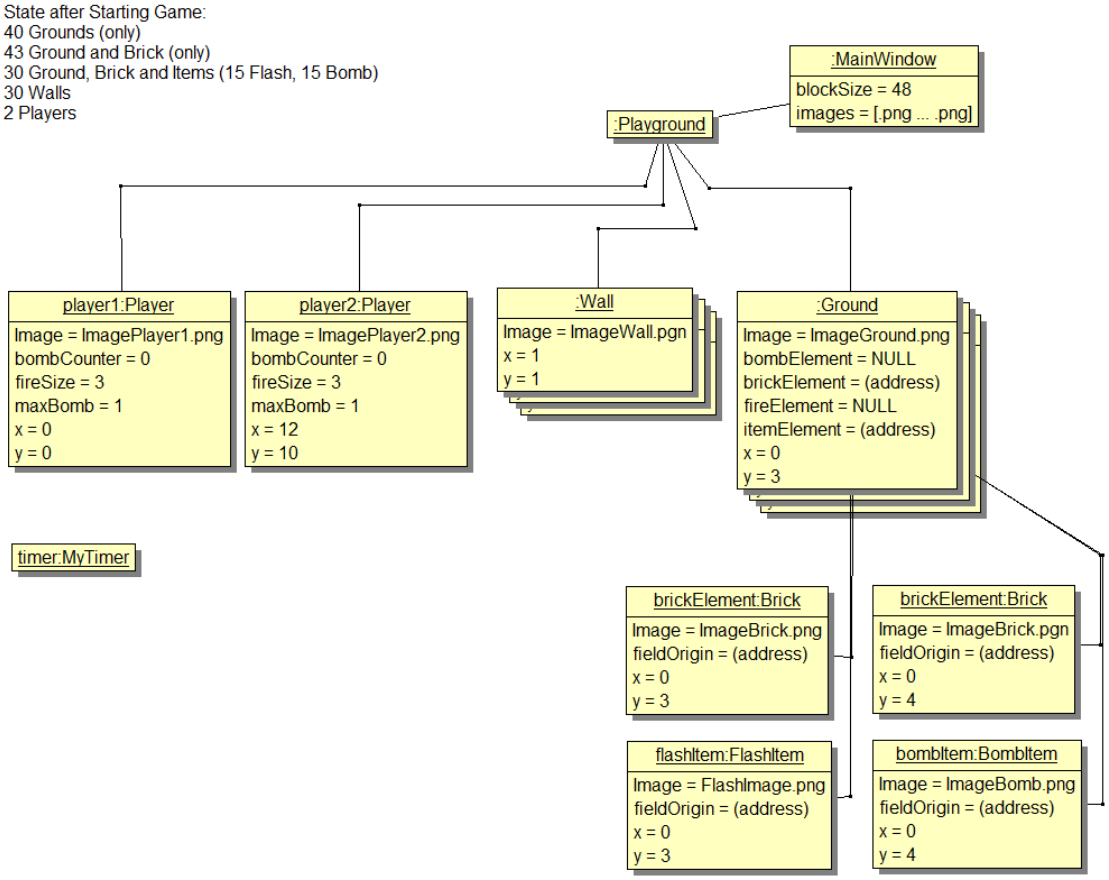
Die oberste Klasse ist das *MainWindow*, welches auf das grafische Fenster initialisiert und kontrolliert. Dieses besitzt dann ein *Playground* Objekt in dem alle anderen Objekte vertreten sind.

Objektdiagramm

Wir haben mit dem Objektdiagramm zwei unterschiedliche Situationen dargestellt:

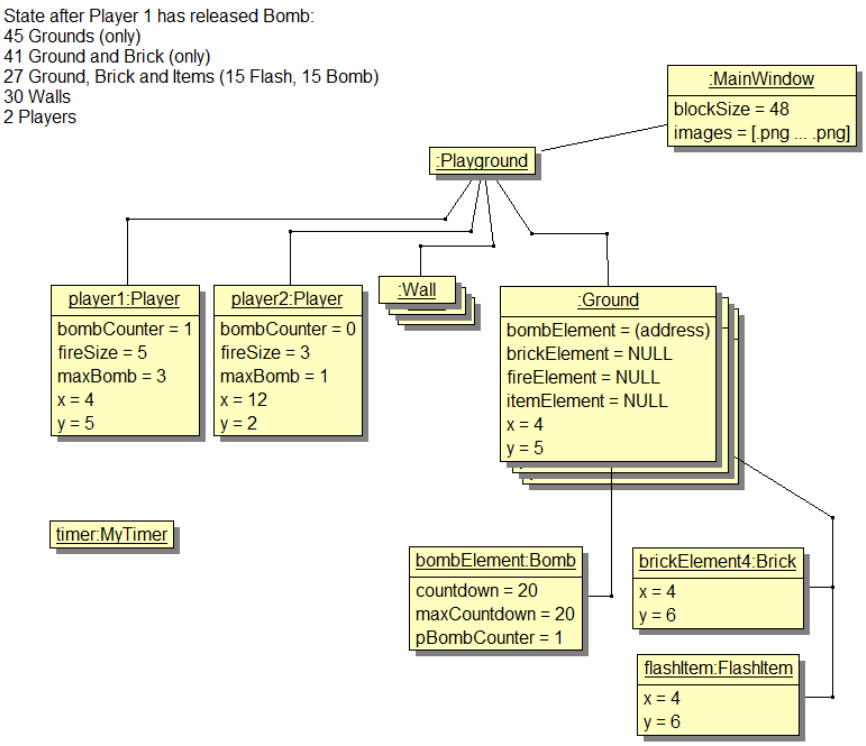
- Der Zustand direkt nach dem Spielstart und
- Der Zustand, nachdem ein Spieler eine Bombe gelegt hat.

Zustand bei Spielstart



Beim Spielstart wird zuerst ein Spielfeld mit einer standardgrösse gezeichnet, worauf sich nur Bricks, Walls, Grounds und die zwei Spieler befinden. Die Items sowie die Bricks liegen jeweils über einem Groundfield. Die beiden Spielfiguren werden diagonal übers Spielfeld voneinander weg platziert (Position [0, 0] und [12, 10]).

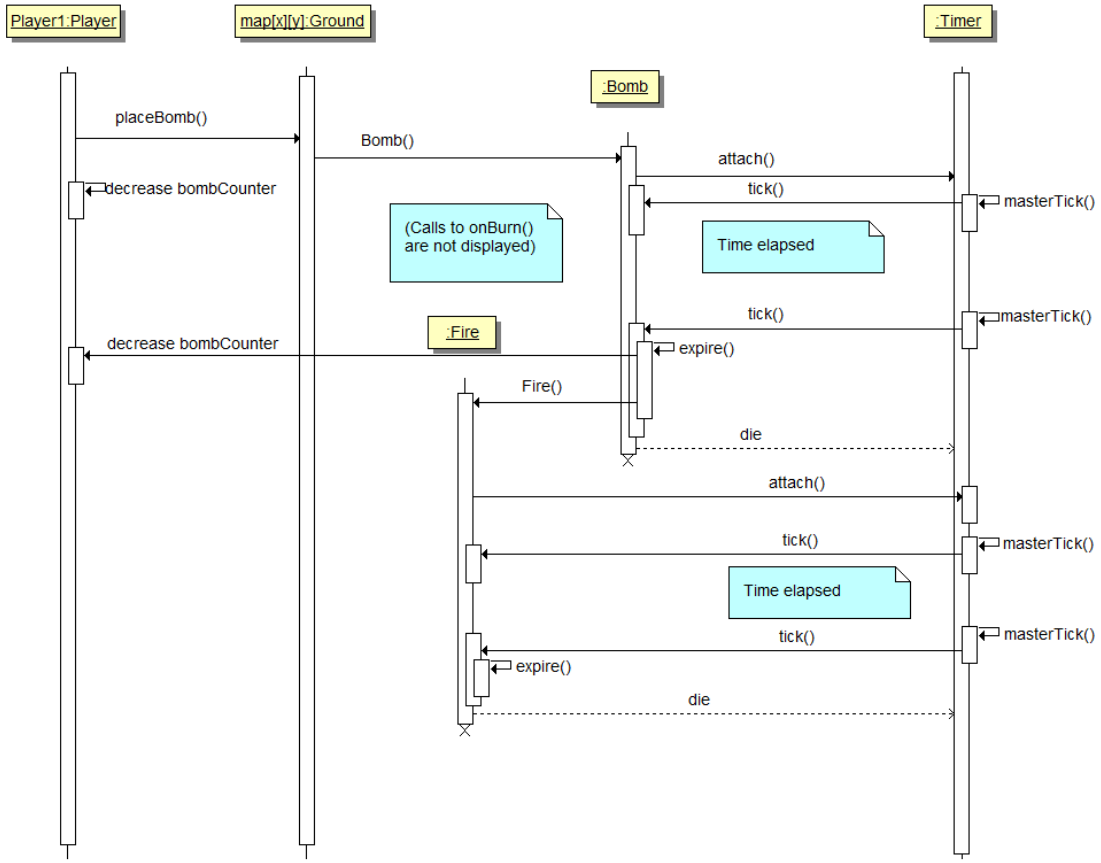
Zustand nach Bombe gelegt



Es gibt beinahe beliebig viele Möglichkeiten wie die Objekte zu einem beliebigen Zeitpunkt existieren. Hier wird eine Momentaufnahme eines Spiels gezeigt, wo gerade eine Bombe gelegt wurde. Man erkennt, dass im Ground-Field ein BombElement vorhanden ist, welches an der Position [4, 5] liegt. Der Player1 befindet sich in diesem Moment auch noch auf diesem Feld, da er dort die Bombe gelegt hat. Der Countdown der Bombe ist noch auf dem Initialwert von 20.

Sequenzdiagramm

Die komplexeste Aktion, nämlich das Platzieren einer Bombe und die darauffolgenden Ereignissen werden hier noch in einem Sequenzdiagramm dargestellt:



Die Bombe wird durch den Spieler platziert und geschieht in der Methode placeBomb(). Diese erstellt auf dem aktuellen Ground-Field eine neue Bombe im reservierten Speicherfeld des Ground-Objekts und erhöht den Zähler der gelegten Bomben des Speilers. Die Bombe selbst, hängt sich beim Erzeugen in einer Liste des Timers ein, damit sie nach einer gewissen Zeit explodiert. Wenn die Bombe explodiert wird der Zähler der Bomben des Spielers wieder heruntergezählt. Danach werden die onBurn-Methoden aller Felder im Radius der Bombsize ausgeführt und das Feuer entsprechend gesetzt. Die Feuer werden auch wieder in die Timer-Liste eingetragen und verschwinden nach einer gewissen Zeit. Läuft ein Spieler in ein Feuer, stirbt er.

Klassenbeschreibungen

Im Folgenden wird jede Klasse kurz erwähnt. Bei manchen Klassen werden zusätzlich noch Erklärungen zu Funktionsweisen oder Datenstrukturen angegeben.

MainWindow

Diese Klasse ist das Hauptfenster, welches alle Spielelemente miteinander verbindet und sie mit Hilfe der Qt Klassen auf einem Fenster graphisch darstellt.

Playground

Hier wird die *map* erzeugt. Diese Klasse ist für das Aufbauen des Spielfeldes zuständig und wird in allen Klassen benutzt, welche einen direkten Einfluss auf die Spieloberfläche haben.

Die Initialisierung des Spielfeldes bei Programmstart geht folgendermassen vor sich:

1. Leere *map* mit Null-Pointern.
2. *Wall*-Objekte platzieren.
3. Ecken der beiden Spieler (links oben und rechts unten) im rechten Winkel mit *Ground*-Objekten füllen.
4. *Brick*-Objekte zufällig auf verbliebene Null-Pointer-Felder platzieren (40x) (dabei wird die Map stur von oben nach unten durchlaufen).

Field

Diese Klasse ist zuständig für die Eigenschaften jedes einzelnen Feldes. Mit der onStep()- und onBurn()-Methode wird das Visitor-Pattern implementiert (oben erwähnt).

Die Position (x, y) eines Feldes auf dem Spielfeld ist dabei **const**. Dem Konstruktor wird die initiale Feldposition mitgegeben. Alle von Feldern abgeleitete oder erstellte Objekte besitzen die *gleiche Feldposition* wie das „Mutterobjekt“, welche nicht verändert werden kann.

Player

Hier wird ein Spieler erzeugt und seine Eigenschaften (Anzahl erlaubte Bomben, maximale Ausbreitung des Feuers, aktuelle Position) gespeichert. Der Spieler ist grundsätzlich während der ganzen Spieldauer lebendig. Sobald seine Methode „die()“ ausgeführt wird, ist das Spiel beendet.

Wall

Diese Klasse stellt fixe “Mauern” dar. Diese bleiben während des ganzen Spieles statisch auf dem zugewiesenen Feld.

Ground

Diese Klasse beschreibt den “Boden”, also die Oberfläche, auf der sich der Spieler bewegen kann und Bomben (Elemente) platzieren darf. Auf dem Ground können sich auch Items, Bricks, Fire oder Bomben befinden. Siehe auch *Erklärung zur Datenstrukturierung rund um Ground und FieldContent*.

FieldContent

Siehe *Erklärung zur Datenstrukturierung rund um Ground und FieldContent*.

Elapsing

Diese abstrakte Klasse ist Teil des Observer-Patterns und stellt das Framework zur Verfügung, um den Zeittackt der Bomben- und Feuer-Elemente zu synchronisieren und die Lebensdauer dieser Elementes zu Verwalten.

Bomb

Diese Klasse stellt die Bombe (Element) dar und speichert ihren aktueller Zustand (zeitlicher Ablauf).

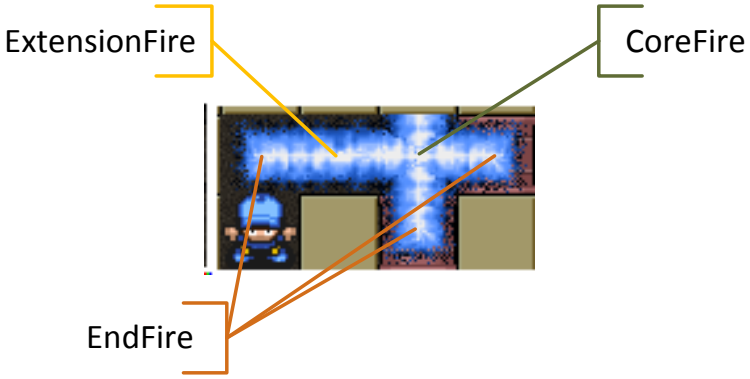
Die Spieler merken sich beide über einen normalen Integer, wie viele Bomben sie gerade aktiv „rumliegen“ haben. Detoniert eine Bombe, dekrementiert *sie* diesen Zähler selbstständig beim Spieler. Dafür erhält sie im Konstruktor einen Pointer auf diesen Integer des Spielers.

Der initialisierte Countdown (Variable wird von Elapsing geerbt) ist 20, also eine Sekunde.

Fire

Mit dieser Klasse wird das Feuer dargestellt, welches nach der Bombenexplosion entsteht. CoreFire, ExtensionFire und EndFire erben von dieser Klasse.

Das *Fire* besteht im Feld der ursprünglichen Bombe aus zumindest einmal aus einem *CoreFire* Objekt. Trifft das *Fire* irgendwo auf eine *Wall*, wird es begrenzt, besteht aber bis zu diesem Feld weiterhin aus *ExtensionFire* Objekten. Wenn es jedoch von einem *Brick* begrenzt wird geht es ein Feld über diese Begrenzung hinaus (sofern das *Fire* so „lang“ sein darf) und besteht dort in der Überlappung mit dem *Brick* Feld aus einem *EndFire* Objekt. Zur Veranschaulichung dient dieser Screenshot:



FireSize ist definiert als die Anzahl *ExtensionFires* oder *EndFires* plus eins (dem CoreFire). Die FireSize ist beim Spieler zu Beginn des Spiels auf 3 festgelegt. Der initialisierte Countdown (Variable wird von Elapsing geerbt) ist 10, also eine Sekunde.

CoreFire

Diese Klasse ist der Ursprungspunkt des Feuers und gilt als unser 1. Feld welches im „FireSize“ Attribut des Spielers angezeigt wird. Pro Bombenexplosion existiert mindestens ein solches Objekt.

ExtensionFire

Die ExtensionFires sind die weitere Ausbreitungen des Feuers. Je nach Umfeld der Bombenexplosion können hier mehrere Objekte existieren.

EndFire

Diese Klasse stellt den letzten Block der Feuerausbreitung dar. Es existieren bis zu 4 EndFire pro Bombenexplosion.

Item

Die Klasse Item ist für die Extra-Features zuständig (Flash- und Bomben-Items).

FlashItem

Mit diesem Item kann der Spieler die Reichweite seiner Bomben um eine Einheit in jede Richtung erhöhen.

BombItem

Mit diesem Item erhöht der Spieler die Anzahl gleichzeitig möglicher aktiver Bomben.

Brick

Diese Klasse stellt einen „Ziegel“ dar. Dieser Ziegel ist von Bomben zerstörbar und versteckt zum Teil Items unter sich.

Code-Versioning und Online Verfügbarkeit

Der gesamte Code und alle notwendigen Ressourcen wurden mit dem Software-Versionsverwaltungstool Git verwaltet. Ausserdem haben wir GitHub als online Repository-Host genutzt. Unser gesamtes Projekt kann auf https://github.com/baertschi/BomberGirl eingesehen werden.

Tests

Es wurde ein Blackboxtest durchgeführt, welcher das Spiel auf die Funktionalität testet. Der Test war nicht 100% erfolgreich. Besonders das Erkennen ob ein Spieler gewonnen funktioniert nicht. Das gesamte Testprotokoll ist im Dokument „Testprotokoll_v1.0“ ersichtlich.

Ausblick

Durch die Anpassung des Designs, dass ein Ground-Field die darübergelegten Objekte Enthält, konnte das Visitor-Pattern nicht sauber umgesetzt werden. Stattdessen musste für den Zugriff auf die Bomben, Bricks, Items und die Fires immer das Field der map zu einem Ground-Field gecastet werden. Das Design konnte aber auch so umgesetzt werden. Die Implementierung ist jedoch noch nicht abgeschlossen. Folgende Punkte müssen noch gemacht werden:

- Codieren
  - Prüfen ob Spieler getroffen wird, wenn sich das Feuer ausbreitet
  - Gameover-Handling: Was geschieht wenn ein Spieler gewinnt bzw verliert?
  - Sofortiges Auslösen der Bomben, welche vom Feuer getroffen werden
- Validierung
  - Testen der neu hinzugefügten Codeteile

Schlussbetrachtung

Das Projekt war sehr hilfreich um die gelernten Vorgehensweisen des RUP und der UML anzuwenden und zu verinnerlichen. Auch die Umsetzung im Code war sehr lehrreich. Jedoch war es auch sehr zeitintensiv, da es C++ dem Programmierer nicht gerade einfach macht, Fehler schnell zu finden. Das grösste Problem bei der Implementierung bereiteten zyklische Includes und Vorwärtsdeklarationen.