C:\Users\cyril\Pictures\Screenpresso\2014-03-27_11h41_55.pngBomberGirl

# Entscheidungen

* PC-Basiert
* Library: QT 5
* IDE: QT Creator
* Designmethode: Unified Process

# Design

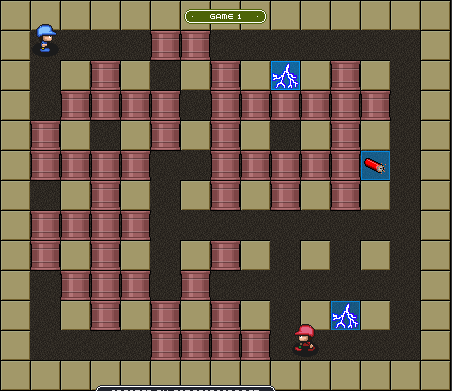
Use Case

Was macht der Anwender mit dem Spiel: „starten“, „spielen“, „high score ansehen“, „Einstellungen machen“, und „beenden“. Anwendungsfall „spielen“ weiter vertiefen.

* **Anwendungsfall aus Sicht des Spielers 🡪 Spielregel**
* Beschreibung der einzelnen Anwendungsfälle in Prosa (mit Dot-List)

Spielregeln Beschreibung

Die genauere Betrachtung vom Anwendungsfall „spielen“ ist eine Spielregelbeschreibung. Wir orientieren uns für diese Spielregeln am Spiel „Bomberman“ welches auf der Website [www.spielen.com](http://www.spielen.com) gratis spielbar ist (<http://www.spielen.com/spiel/fire-play-2>).



**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 darüber Gehen 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 Bombe. 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 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 darüber Gehen aufgenommen werden. Danach verschwindet das Item und der Spieler erhält das entsprechende Feature 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 QT5 programmiert. Ausser den Zeichnungsklassen und den Klassen für den grundlegenden Fensteraufbau werden von QT keine Funktionalitäten oder Klassen verwendet.

Klassendiagramm

Beschriebt alle Klassen.

Paketdiagramm

Bestehende Klassen in Pakete kapseln: **Logik** und **Grafik von QT**.

Sie werden in zwei Pakete gegliedert: Spielbasierte Klassen und Grafikbasierte Klassen (welche von QT zur Verfügung gestellt werden).

(QT-spezifische Klassen werden nur per Klassennamen erwähnt. Und auch nur die, welche gerade Aktivitätsdiagramm oder so gebraucht werden).

Objektdiagramm

Situation:

* Zustand nach Spielstart (keine Bombe, kein Feuer, keine Items)
  + Nur Objekte zuunterst im Vererbungsbaum darstellen, bei mehreren fast identischen Objekten mit hoher Anzahl wird einfach ein einzelnes Objekt erstellt und dann einerseits anders *gefärbt*, in der *Beschreibung* erwähnt, wie viele es davon gibt und schlussendlich im *Inkscape* mehrere Blöcke hintereinander skizziert.
* Ein Spieler hat eine Bombe gelegt
* Eine Bombe ist am explodieren

Sequenzdiagramm

* Ein Spieler legt eine Bombe, sie explodiert und ein Feuer entsteht.
* (Eine Bombe trifft den Gegner)

# Notiz: Datentyp Umsetzung

Field

Die Position (x, y) eines Feldes auf dem Spielfeld ist **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.

const int m\_value;

Example::Example(const int value) : m\_value(value)

{

}

Map

Normales 2D-Array aus Field-Pointern. Jede Position ist entweder eine Wall ode rein Ground-Feld. Die Groundfelder ihrereseits beinhalten dann entweder noch ein Brick oder ein Item und ein Brick. Und wenn darauf eine Bombe abgelegt wurde oder darüber eine Bombe explodiert, enthalten sie noch ein Bomb oder Fire-Objekt.

Dafür wurde das Ground-Objekt mit vier „Speicherplätzen“ 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.

Bomben

Die Spieler merken sich beide über einen normalen Integer, wieviele Bomben sie gerade aktiv „rumliegen“ haben. Detoniert eine Bombe, dekrementiert sie diesen Zähler selbstständig beim Spieler.

# Sichtbarkeits-Layer (Reihenfolge)

Der Spieler wird nicht auf der map gespeichert. Er wird deshalb auch separat gezeichnet.

Die Reihenfolge des Zeichnens legt die überschreibende draw()-Methode des Ground-Feldes vor: sie zeichnet (je nachdem ob überhaupt so ein Objekt besteht -> NULL-Pointer-Überprüfung) zuerst ihr eigenes Ground-Feld (also „Boden“), dann evtl. das Item-Objekt (welches ja bei noch intaktem Brick von diesem optisch verdeckt wird), eine eventuelle Bombe, dann den Brick und schlussendlich noch das Fire.

# Notiz: Ablauf Spielfeldgenerator

1. Leere Map mit Null
2. Walls platzieren
3. Ecken mit Ground füllen
4. Bricks randomly auf verbliebene Null-Felder platzieren (40x) (dabei wird die Map stur von oben nach unten durchlaufen)

# Notiz: Fire

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.

FireSize ist definiert als die anzahl ExtensionFires oder EndFires plus eins (dem CoreFire).

C:\Users\cyril\Pictures\Screenpresso\2014-03-27_11h41_55.png

# VisitorPattern

Für die neu-Positionierung des Spieler und die Interaktion von Feldern mit Bombenexplosionen wird das C++ Design-Pattern *VisitorPatter* 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. Die verschiedenen Reaktionen auf das „Betreten“ des Feldes werden hier beschrieben:

Rückgabewerte von onStep(player &p)

(ist rein virtuel auf Stufe Field - es macht keinen Sinn, ein Field Objekt zu erzeugen)

* NOENTRY ab Stufe HardField, belässt den Player unverändert
* ENTRY ab Stufe SoftField (ist es überhaupt nötig, dort einen rückgabewert zu geben? 🡪 könnte dort ebenfalls als rein virtual definiert werden)
  + ENTRY auf Stufe Ground
  + UPGRADE ab Stufe ItemField (ist es überhaupt nötig, dort einen rückgabewert zu geben? 🡪 könnte dort ebenfalls als rein virtual definiert werden)
    - UPGRADEFLASH auf Stufe FlashItem, inkrementiert fireSize
    - UPGRADEBOMBS auf Stufe BombItem, inkrementiert maxBombs
  + DIE ab Stufe Fire, führt Funktion die() des spielers aus oder zieht einfach ein Leben ab

### 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()

(ist rein virtuel auf Stufe Field – es macht keinen Sinn, ein Field Objekt zu erzeugen)

* NOACTION ab Stufe SoftField
* HARDBLOCK ab Stufe HardField
  + BLOCK ab Stufe Destroyable
    - BURN ab Stufe Brick, Funktion onDestroy() aufrufen? Oder das zerstören dort einfach in die onBurn() Overwrite Methode packen?
  + TRIGGER ab Stufe Bomb, Funktion onExplode() der Bombe aufrufen? Oder das triggern dort einfach in die onBurn() Overwrite Methode packen?

# ObserverPattern

Für das Timing von Bomben und Fire wird das Observer Design-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 Timer ein, welche ihm dann jede 100ms einen Tick schickt. Wenn das Objekt stirbt, gibt es als return-Wert der tick()-Methode -1 zurück. Darauf ruft die Timer-Klasse den Destruktor des Observers (also der Bombe 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.

Das Subject (der Timer) hat eine STL-Liste mit den Elapsing-Objekten. FRAGE: KANN IN DIESER LISTE JETZT AUCH EIN ELEMENT VOR ODER NACH DEM AKTUELLEN ERASED WERDEN??? Dies ist nämlich notwendig, wenn eine Bombe eine andere triggert: während dem masterTick() wird sich die andere Bombe ausklinken wollen. 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.