

Refactoring

Was ist Refactoring?

Der Begriff wurde 1990 von Martin Fowler und Kent Beck eingeführt und beschreibt einen Prozess, bei dem es um die Verbesserung des Quellcodes geht, ohne seine Funktionsweise zu verändern.

Natürlich unterschieden sich der Code und der Entwicklungsablauf damals von dem heutigen, jedoch ist Refactoring weiterhin ein fundamentaler Bestandteil der Entwicklung. Denn alle Entwickler, die auf Dauer Code Smells ignorieren, geraten in Gefahr, dass ihr Code unverständlich, schwer lesbar und unwartbar wird.

Leider fehlt meist die Zeit in der Entwicklung, und so entstehen vielerorts Technical Debts.

Der bereits erwähnte Martin Fowler veröffentlichte zu dem Thema Refactoring 1999 ein Buch namens "Refactoring: Improving the Design of Existing Code", welches von vielen als das grundlegende Werk zu diesem Thema wahrgenommen wird.

Zum Glück für unsere junge Generation von Entwicklern gibt es mittlerweile viele IDEs, welche das Refactoring stark vereinfachen.

Wichtige Begriffe

Code Smells sind Stellen im Code, die unübersichtlich, schwer verständlich und schwer zu warten sind. Dies erschwert es, Bugs zu finden, und bei der Implementierung neuer Funktionen im Code können so leichter Fehler auftreten.

Technical Debt bezeichnet die 'Schulden', die Entwickler in Kauf nehmen, wenn sie schnell und unsauber arbeiten, wodurch Code Smells entstehen.

Das Ziel von Refactoring

Das Ziel ist Clean Code oder sauberer Code. Clean Code zeichnet sich dadurch aus, dass er:

- Lesbar,
- Übersichtlich,
- Gut erweiterbar,
- Verständlich und
- Testbar

ist.

Dies führt dazu, dass sich die Performance des Codes verbessern kann und vor allem die Weiterentwicklung des Codes vereinfacht wird.

Bekanntermaßen ist der eigene Code meist verständlich für sich selbst, allerdings heißt das nicht, dass er auch für andere Entwickler verständlich ist oder für ein zukünftiges Ich, welches den Code in ein oder zwei Jahren noch einmal erweitern muss. Daher bietet Refactoring die Möglichkeit, den Code für alle und jeden verständlicher zu machen.

Refactoring Techniken

Im Laufe der Zeit wurden verschiedene Refactoring-Techniken entwickelt, um die verschiedenen Code-Smells effektiv zu entfernen, ohne dabei noch mehr Probleme zu verursachen.

- **Extract Method:** um eine Methode übersichtlicher zu machen und um sie "abzuspecken", wird eine neue Methode erstellt, in welcher der "überschüssige" Code eingefügt und mit einem Call der neuen Methode ersetzt. -> lesbarer Code, weniger doppelter Code, Fehler sind unwahrscheinlicher (isolierter unabhängige Code-Stellen)
- **Inline Method:** Wenn der Inhalt einer Methode offensichtlicher ist als die Methode selbst, wird der Call mit dem Inhalt der Methode ersetzt (sofern die Methode nicht in Subclasses redefined wurde). -> unkomplizierter Code
- **Extract Variable:** bei komplizierten und mehrteiligen Expressions, wird sie aufgeteilt und die einzelnen Teile werden in selbsterklärenden Variablen platziert. -> Code wird besser zu lesen
- **Inline Temp:** Temporäre Variablen, die nur eine simple Expression enthalten, werden entfernt und mit der Expression ersetzt. -> besser Lesbarkeit ABER kann Performance beeinträchtigen, wenn die Variable mehrmals genutzt wurde
- **Replace Temp with Query:** Extract Method bei lokalen Variablen welche das Ergebnis einer Expression enthalten. -> besser Lesbarkeit und (wenn die ersetzte Zeile in mehreren Methoden benutzt wurde) schlanker Code
- **Split Temporary Variable:** Temporäre Variablen, welche mehrmals überschrieben werden, werden aufgeteilt und bekommen eine eigene Variable mit erklärendem Namen für jeden Nutzen. -> Einfachere Wartung, bessere Lesbarkeit, hilfreich bei späterer Nutzung von Extract Method
- **Remove Assignments to Parameters:** Wenn ein Parameter innerhalb einer Methode überschrieben wird, erstellt man stattdessen eine lokale Variable und ordnet ihr den Wert des Parameters zu. -> Einfachere Wartung, hilfreich bei späterer Nutzung von Extract Method
- **Replace Method with Method Object:** Bei einer zu langen Methode, bei der die Werte nicht einfach übergeben werden können wie bei der Extract-Methode, werden die Funktionen in eine andere Klasse ausgelagert und die übergebenen Werte als Attribute gespeichert.-> Macht den Code übersichtlicher.
- **Move Method:** Wenn eine Methode öfters in einer anderen Klasse genutzt wird als in der Eigenen, wird in der anderen Klasse eine neue Methode erstellt und der Code wird darin eingefügt. Danach wird die ursprüngliche Klasse mit einem Aufruf der Neuen ersetzt oder gar gelöscht. -> reduziert Abhängigkeiten zwischen Klassen
- **Replace Conditional with Polymorphism:** Wenn man eine Methode hat, welche eine Menge Aktionen, abhängig von Konditionen, durchführt, wird für jede Kondition eine Unterklasse erstellt mit einer geteilten Methode, in welche der korrespondierende Code bewegt wird. -> entfernt doppelten Code, bessere Wartung

- **Extract Class:** Wenn eine Klasse die Arbeit von 2 erledigt, wird eine neue Klasse erstellt und die relevanten Fields und Methoden werden eingefügt. -> Code wird einfacher zu verstehen, sicherere Wartung
- **Extract Superclass:** Wenn Klassen ähnliche Fields und Methoden haben, wird eine gemeinsame Oberklasse erstellt, in welche alle identischen Fields und Methoden verschoben werden. -> weniger doppelter Code
- **Replace Data Value with Object:** Wenn eine Klasse (oder mehrere) ein Field enthalten, welches ein eigenes Verhalten und zugehörige Daten hat, wird

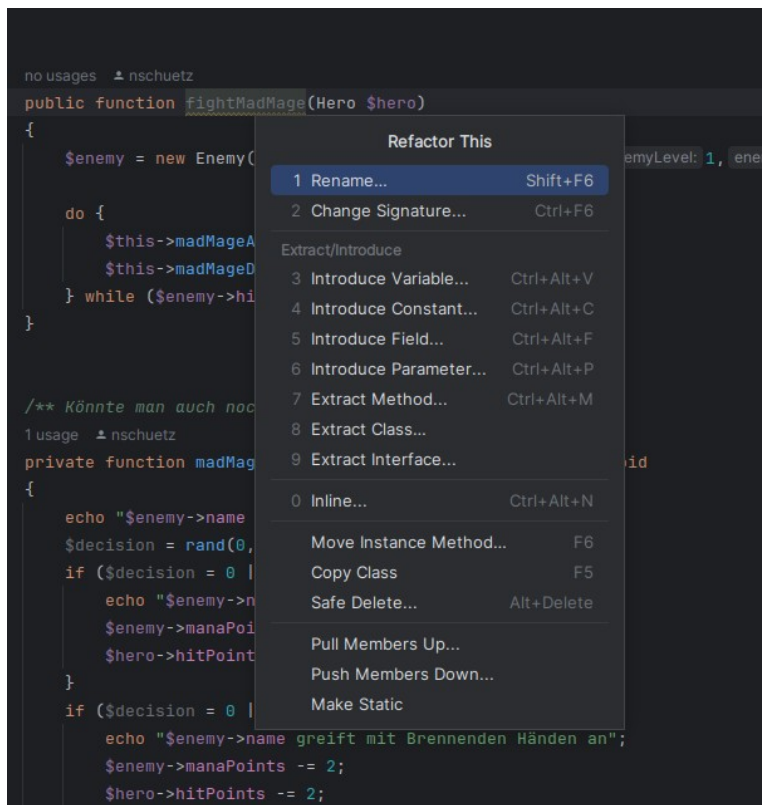
Code bespiele finden Sie auf Github unter
<https://github.com/Skira5/REFACTORING>

IDE

Die IDE PhpStorm von JetBrains bietet eine Vielzahl von Funktionen, um das Refactoring zu vereinfachen.

Refactoring übersicht:

Dies kann man mit Strg + Alt + Shift + T aufrufen. Das Fenster, das sich öffnet, bietet einen Überblick über die verschiedenen Refactoring-Funktionen. Diese können natürlich auch direkt mit Shortcuts aufgerufen werden.



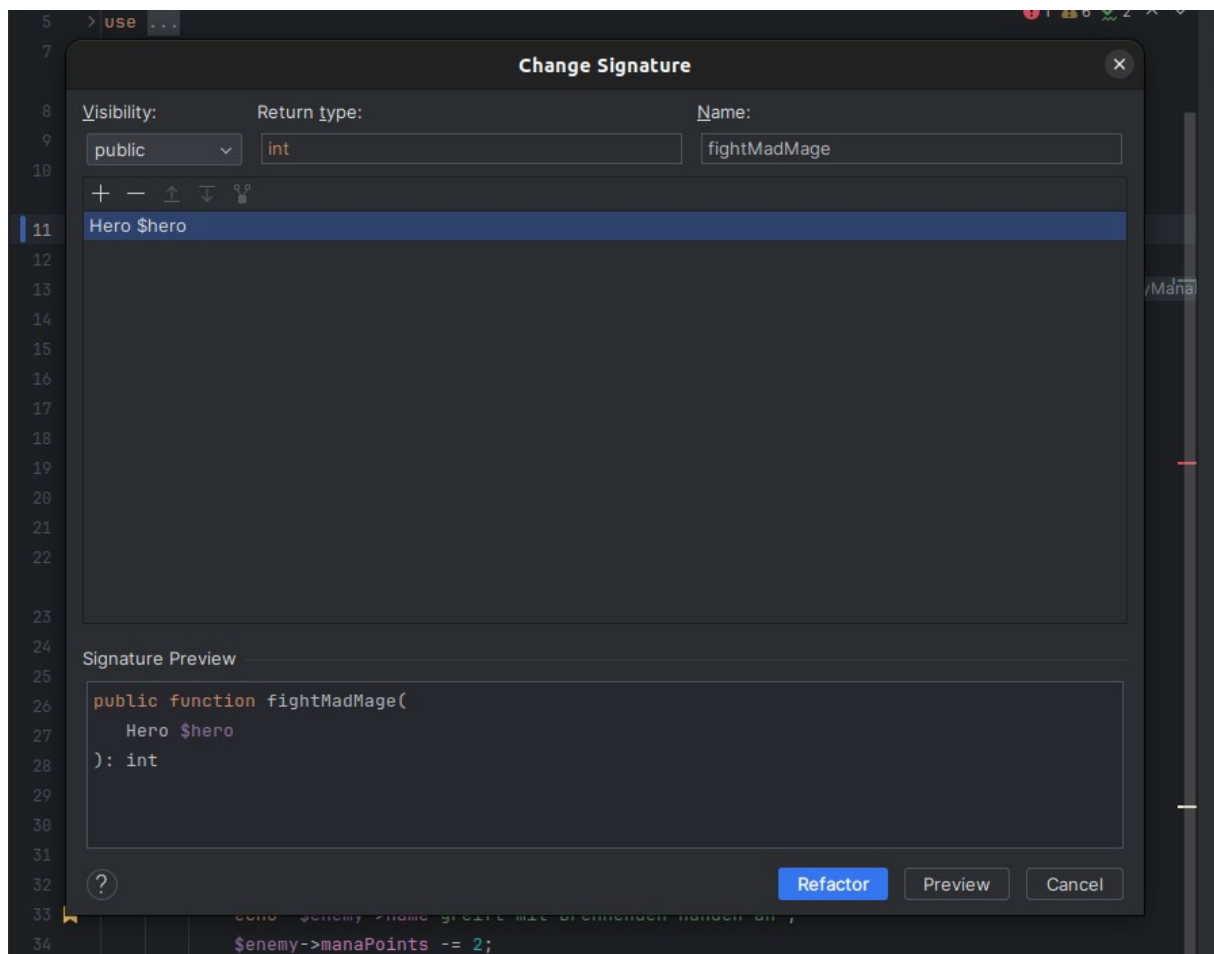
Funktionen:

Change Signature:

Shortcut: Strg F6

Damit ist es möglich, einen guten Überblick über eine Funktion zu erhalten und ihre Werte (z.B. Parameter, Rückgabewerte, Name der Funktion usw.) anzupassen.

Hierbei werden auch die Aufrufe der Funktion refaktoriert.



Extract und Introduce Werkzeuge:

Extract Konstanten:

Shortcut: Strg + Alt + C

Es vereinfacht es, mehrfach auftretende Werte (wie hardcoded Informationen) im Code in Konstanten auszulagern, sodass der Code besser lesbar wird.

Außerdem ist die Information so einfacher an anderen Stellen abrufbar.

```
no usages  nschuetz
public function fightMadMage(Hero $hero):void
{
    $enemy1 = new Enemy( enemyName: "Halaster Blackcloak", enemyLevel: 1, enemyHitPoints: 10, en
    $enemy2 = new Enemy( enemyName: "Halaster Blackcloak", enemyLevel: 1, enemyHitPoints: 10, en
    $enemy3 = new Enemy( enemyName: "Halaster Blackcloak", enemyLevel: 1, enemyHitPoints: 10, en
    $enemy4 = new Enemy( enemyName: "Halaster Blackcloak", enemyLevel: 1, enemyHitPoints: 10, en
    $enemy5 = new Enemy( enemyName: "Halaster Blackcloak", enemyLevel: 1, enemyHitPoints: 10, en

    Multiple occurrences found
    Replace this occurrence only
    Replace all 5 occurrences

5 usages
private const string NAME = "Halaster Blackcloak";

no usages  nschuetz *
public function fightMadMage(Hero $hero):void
{
    $enemy1 = new Enemy( enemyName: self::NAME, enemyLevel: 1, enemyHitPoints: 10, enemyManaPoints: 15);
    $enemy2 = new Enemy( enemyName: self::NAME, enemyLevel: 1, enemyHitPoints: 10, enemyManaPoints: 15);
    $enemy3 = new Enemy( enemyName: self::NAME, enemyLevel: 1, enemyHitPoints: 10, enemyManaPoints: 15);
    $enemy4 = new Enemy( enemyName: self::NAME, enemyLevel: 1, enemyHitPoints: 10, enemyManaPoints: 15);
    $enemy5 = new Enemy( enemyName: self::NAME, enemyLevel: 1, enemyHitPoints: 10, enemyManaPoints: 15);
}
```

Extract Field:

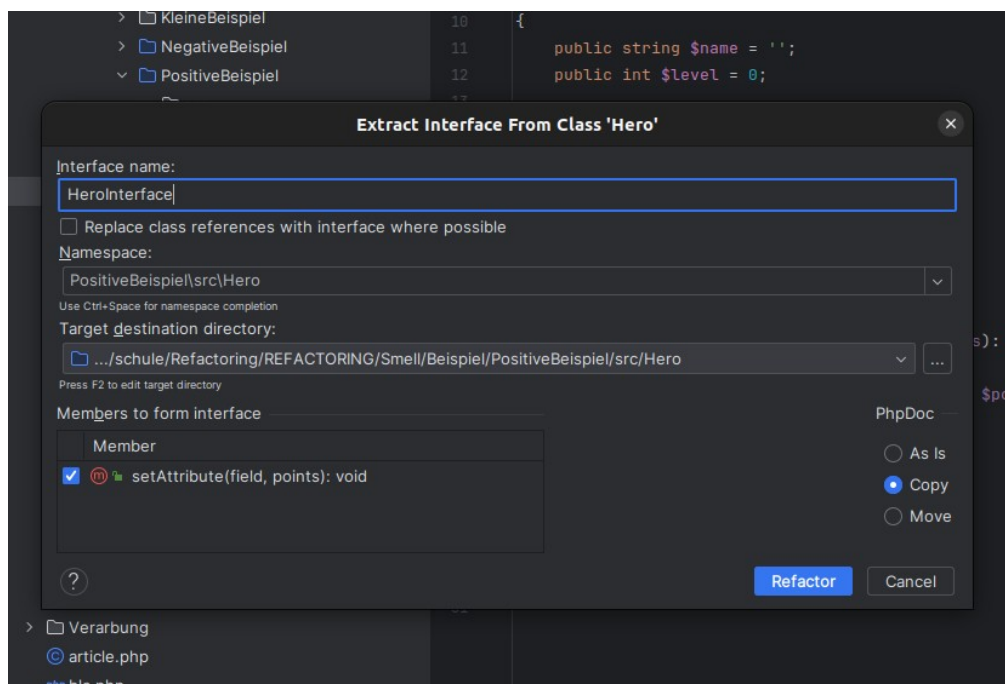
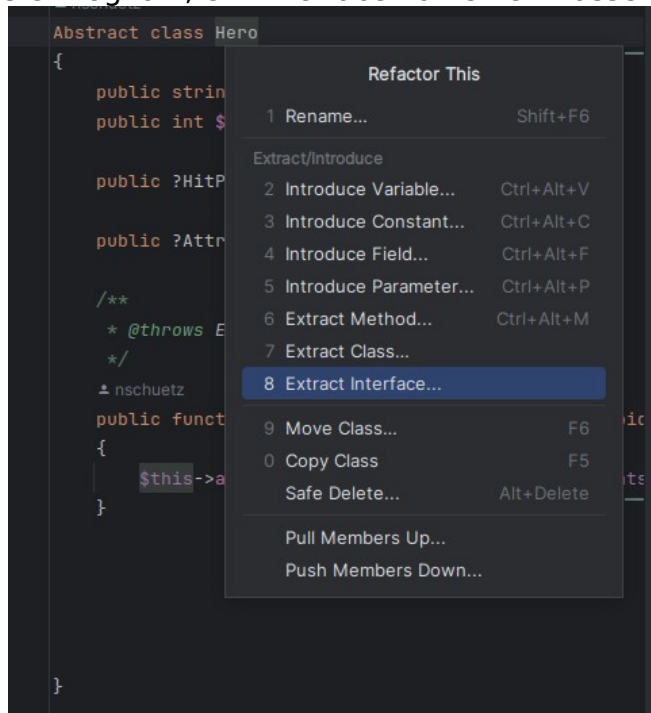
Shortcut: Strg + Alt + F

Macht etwas Ähnliches wie "Extract Konstanten", aber anstatt es in eine Konstante umzuwandeln, wird es in ein Attribut umgewandelt.

Extract Interface:

Dafür gibt es keinen eigenen Shortcut, daher muss man über das Menü gehen.

Es ermöglicht, ein Interface von einer Klasse zu erstellen.



Extract Methode:

Shortcut: Strg + Alt + M

Der ausgewählte Code wird in eine eigene Methode ausgelagert. Dabei beachtet PhpStorm bereits Typisierung und welche Werte übergeben werden müssen.

```
    }
    $attributes = ['Stärke', 'Weisheit', 'Intelligenz', 'Geschicklichkeit', 'Konstitution', 'Charisma'];
    $points = 20;

    echo "Du hast $points Punkte zum Verteilen auf die folgenden Attribute: \n";
    foreach ($attributes as $attribute) {
        echo "$attribute ";
    }
    echo "\n";

    while ($points > 0) {
        // Verschachtelte Schleifen
        foreach ($attributes as $attribute) {
            echo "Verteile Punkte auf $attribute (Verbleibende Punkte: $points): ";
            $input = trim(fgets(STDIN));
            if (is_numeric($input) && $input <= $points && $input >= 0) {
                if ($attribute === 'Stärke') {
                    $hero->setAttribute('Strength', $input);
                } elseif ($attribute === 'Weisheit') {
                    $hero->setAttribute('Wisdom', $input);
                } elseif ($attribute === 'Intelligenz') {
                    $hero->setAttribute('Intelligence', $input);
                } elseif ($attribute === 'Geschicklichkeit') {
                    $hero->setAttribute('Dexterity', $input);
                } elseif ($attribute === 'Konstitution') {
                    $hero->setAttribute('Constitution', $input);
                } elseif ($attribute === 'Charisma') {
                    $hero->setAttribute('Charisma', $input);
                }
                $points -= $input;
            } else {
                echo "Ungültige Eingabe :( . Versuche es erneut.\n";
            }
        }
    }

    echo "Alle Punkte wurden verteilt!\n";
    return $hero;
}
```

```
    }
    return extracted($hero);
}

/**
 * @param HeroMage|HeroWarrior|HeroArcher $hero
 * @return void
 */
public function extracted(HeroMage|HeroWarrior|HeroArcher $hero): void
{
    $attributes = ['Stärke', 'Weisheit', 'Intelligenz', 'Geschicklichkeit', 'Konstitution', 'Charisma'];
    $points = 20;

    echo "Du hast $points Punkte zum Verteilen auf die folgenden Attribute: \n";
    foreach ($attributes as $attribute) {
        echo "$attribute ";
    }
    echo "\n";

    while ($points > 0) {
        // Verschachtelte Schleifen
        foreach ($attributes as $attribute) {
            echo "Verteile Punkte auf $attribute (Verbleibende Punkte: $points): ";
            $input = trim(fgets(STDIN));
            if (is_numeric($input) && $input <= $points && $input >= 0) {
                if ($attribute === 'Stärke') {
                    $hero->setAttribute('Strength', $input);
                } elseif ($attribute === 'Weisheit') {
                    $hero->setAttribute('Wisdom', $input);
                } elseif ($attribute === 'Intelligenz') {
                    $hero->setAttribute('Intelligence', $input);
                } elseif ($attribute === 'Geschicklichkeit') {
                    $hero->setAttribute('Dexterity', $input);
                } elseif ($attribute === 'Konstitution') {
                    $hero->setAttribute('Constitution', $input);
                } elseif ($attribute === 'Charisma') {
                    $hero->setAttribute('Charisma', $input);
                }
            }
        }
    }
}
```

Extract/Introduce Variable:

Shortcut: Strg+Alt+V

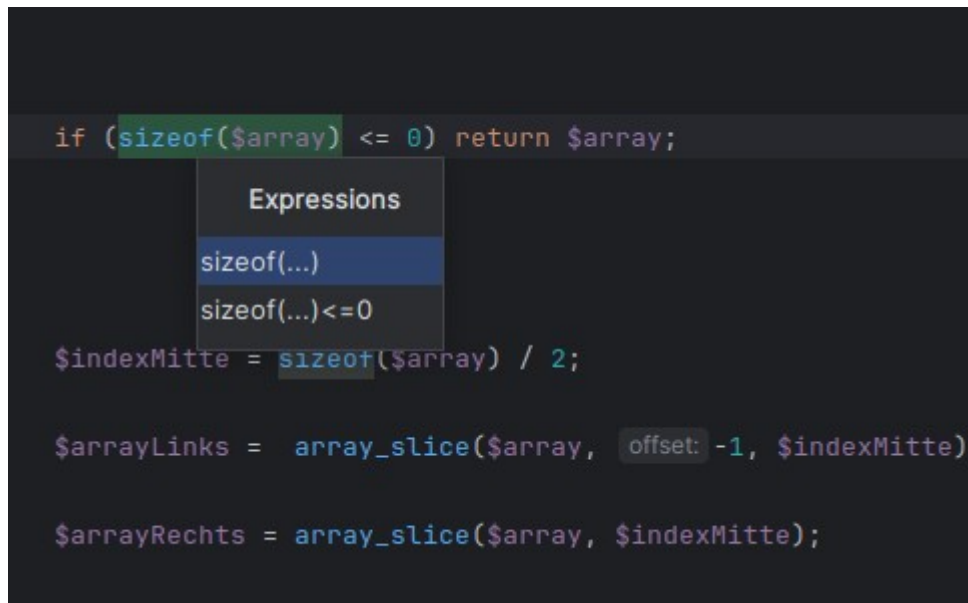
Mit dieser Funktion kann man eine Expression einfach in eine Variable speichern. Es werden alle Stellen ausgetauscht, an denen die Expression genutzt wird.

```
if (sizeof($array) <= 0) return $array;

$indexMitte = sizeof($array) / 2;

$arrayLinks = array_slice($array, offset: -1, $indexMitte)

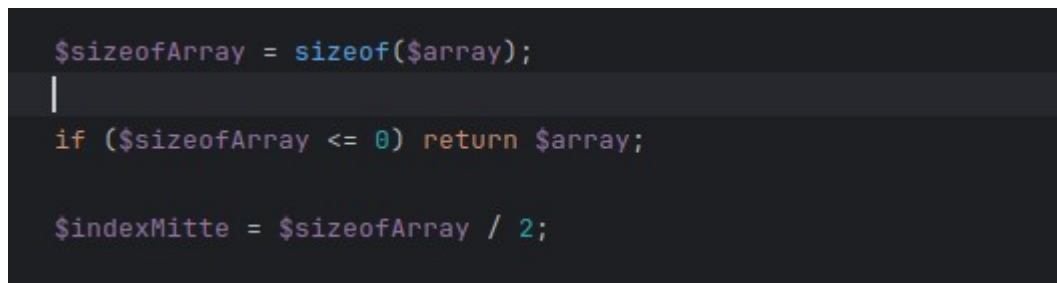
$arrayRechts = array_slice($array, $indexMitte);
```



```
$sizeofArray = sizeof($array);

if ($sizeofArray <= 0) return $array;

$indexMitte = $sizeofArray / 2;
```



Extract Parameter:

Shortcut: Strg+Alt+P

Hiermit ist es möglich, ganz einfach Werte als Parameter auszulagern. Es werden automatisch die vorherigen Werte als Werte bei den Funktionsaufrufen als Parameter übergeben.

```
        $this->madMageAttack($enemy, $hero);
    }

    /** Könnte man auch noch in die Enemy class aus lagern */
    1 usage  nschuetz
    private function madMageAttack(Enemy $enemy, Hero $hero): void
    {
        echo "$enemy->name Greift An";
        $decision = rand(0, 1);
        if ($decision = 0 || $enemy->manaPoints >= 3){
            echo "$enemy->name greift mit Arcane Geschoss an";
            $enemy->manaPoints -= 3;
            $hero->hitPoints -= 4 + $enemy->level;
        }
    }

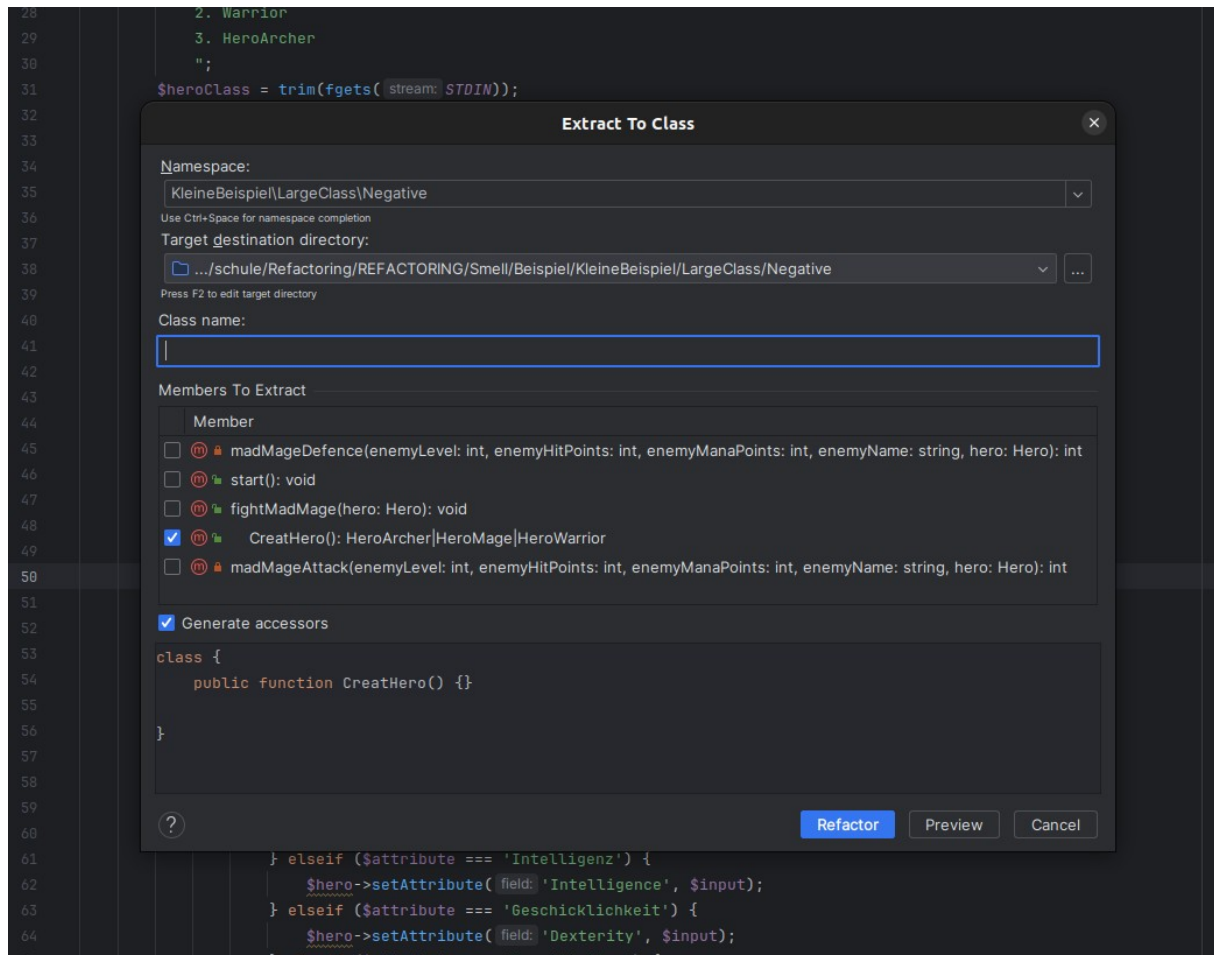
    $this->madMageAttack($enemy, $hero, 4);
}

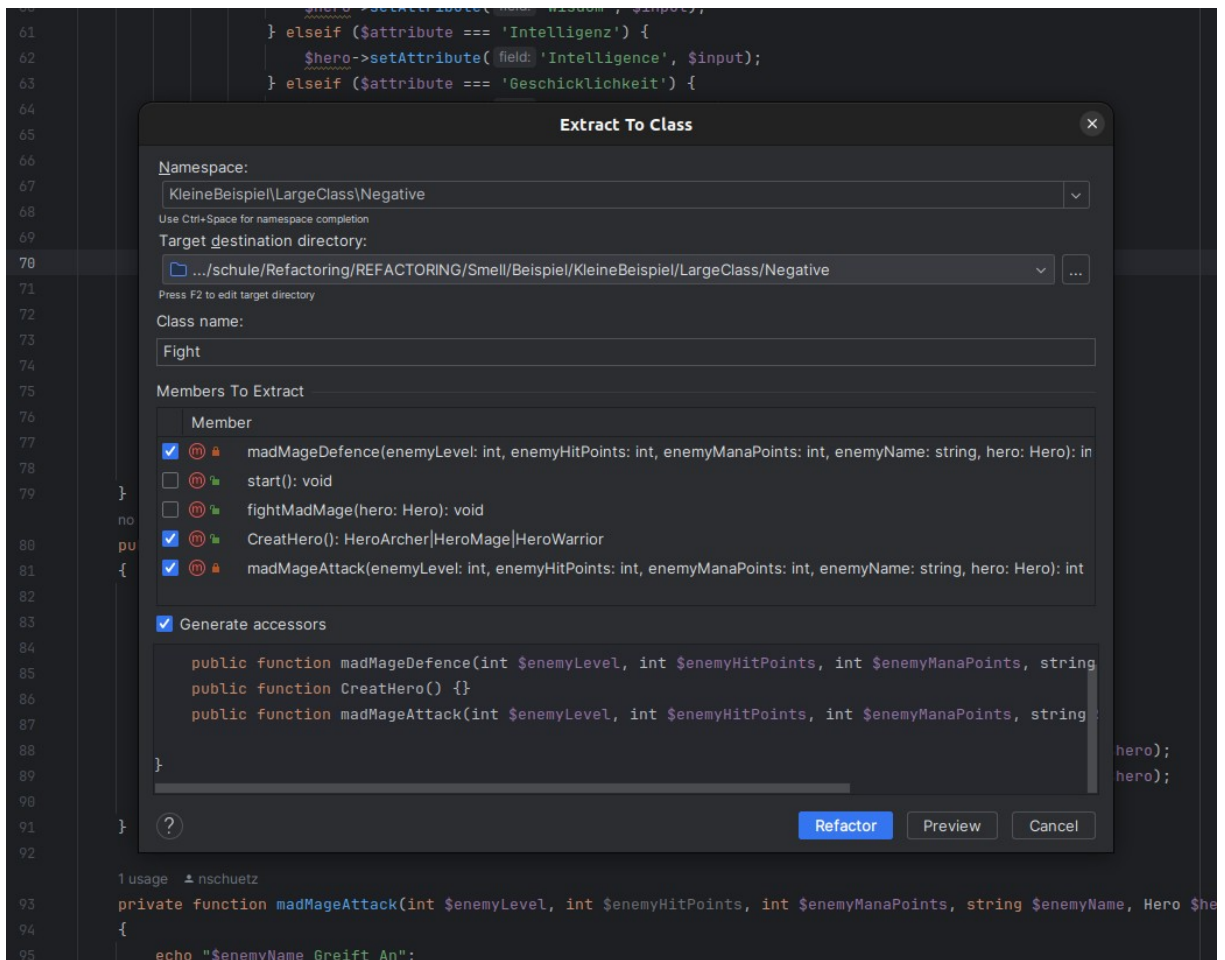
    /** Könnte man auch noch in die Enemy class aus lagern
     * @param Enemy $enemy
     * @param Hero $hero
     * @param $i
     */
    1 usage  nschuetz
    private function madMageAttack(Enemy $enemy, Hero $hero, $i): void
    {
        echo "$enemy->name Greift An";
        $decision = rand(0, 1);
        if ($decision = 0 || $enemy->manaPoints >= 3){
            echo "$enemy->name greift mit Arcane Geschoss an";
            $enemy->manaPoints -= 3;
            $hero->hitPoints -= $i + $enemy->level;
        }
    }
```

Extract Class:

Shortcut: Strg+Alt+Shift+T (Aufrufe über das Menü)

Dies erstellt eine völlig neue Klasse aus zuvor ausgewählten Methoden, welche die alte Klasse ganz automatisch dann aufruft, an den Stellen, wo sie benötigt wird.





```

no usages  nschuetz *
class Game
{
    4 usages
    private Fight $fight;

    no usages new *
    public function __construct()
    {
        $this->fight = new Fight();
    }
}

```

```
2 usages new *
0 class Fight
1 {
2
3     1 usage new *
4     public function madMageDefence(
5         int $enemyLevel,
6         int $enemyHitP
7         int $enemyMana
8         string $enemyN
9         Hero $hero
10    ): int {
11
12    }
```

Unused parameter 'enemyLevel'. The parameter's value is not used anywhere in the function.

Remove all unused parameters Alt+Shift+Enter More actions... Alt+Enter

\$enemyLevel: int

Inline:

Shortcut: Strg+Alt+N

Dies ist ein sehr hilfreiches Werkzeug, welches erlaubt, z.B. Konstanten, die nur an einer Stelle genutzt werden, zu entfernen und dort den Wert direkt reinzuschreiben.

Oder bei Variablen, die vermeidbar sind. Das Schöne daran ist, dass PhpStorm diese Code-Stellen unterstreicht und so das Refaktorisieren stark vereinfacht.

```
1 }
2 1 usage 1 nschuetz *
3 private function madMageDefence(int $enemyLevel, int $enemyHitPoints, int $enemyMana, string $enemyName, Hero $hero): int {
4 {
5     echo "Du Greift $enemyName an";
6
7     $rs = $enemyHitPoints -= rand(1, 10 + $hero->level);
8     return $rs;
9 }
```

```

0         return $enemyManaPoints += 1;
1     }
2     1 usage  1 nschuetz *
3     private function madMageDefence(int $enemyLevel, int $enemyHitPoints, int $e
4     {
5         echo "Du Greift $enemyName an";
6         =
7         return $enemyHitPoints -= rand(1, 10 + $hero->level);
8     }

```

Konstante Inline:

```

1 usage
private const string NAME = "Halaster Blackcloak";

no usages  1 nschuetz *
public function fightMadMage(Hero $hero):void
{
    $enemy = new Enemy(
        enemyName: self::NAME,
        enemyLevel: 1,
        enemyHitPoints: 10,
        enemyManaPoints: 15
    );
    $this->madMageAttack($enemy);
}

```

Inline Constant

Constant NAME

☒ Inline all and remove the constant

☐ Inline all and keep the constant

☐ Inline this reference only and keep the constant

? Refactor Preview Cancel

```

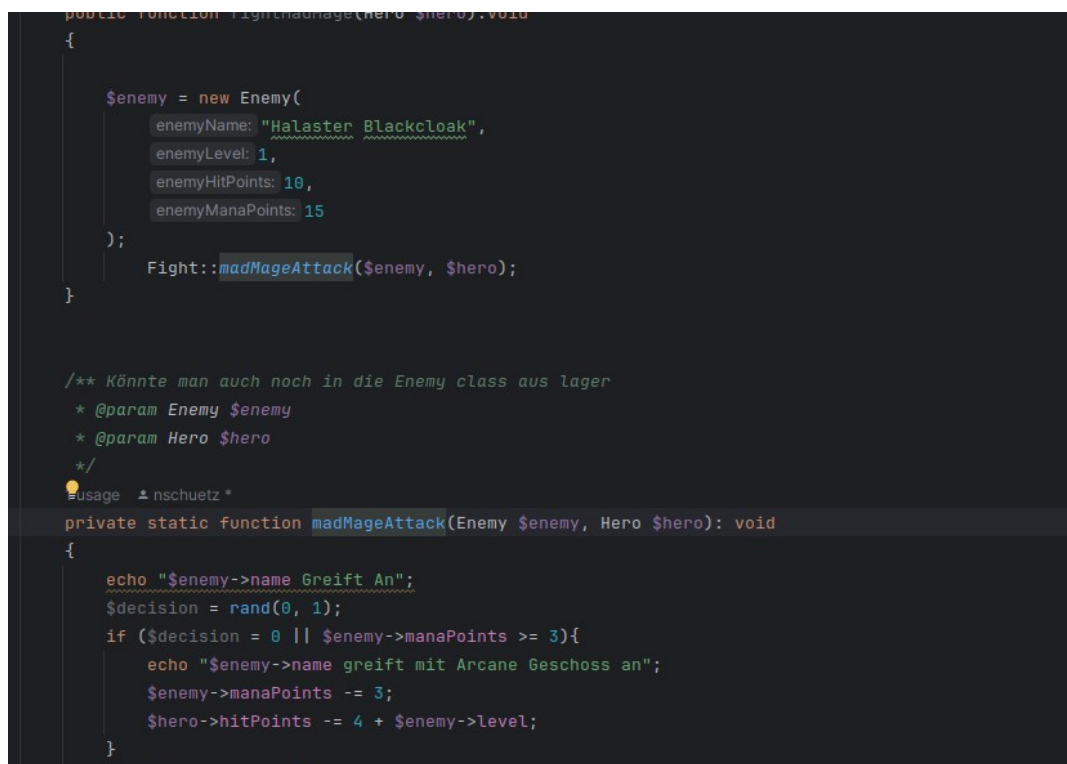
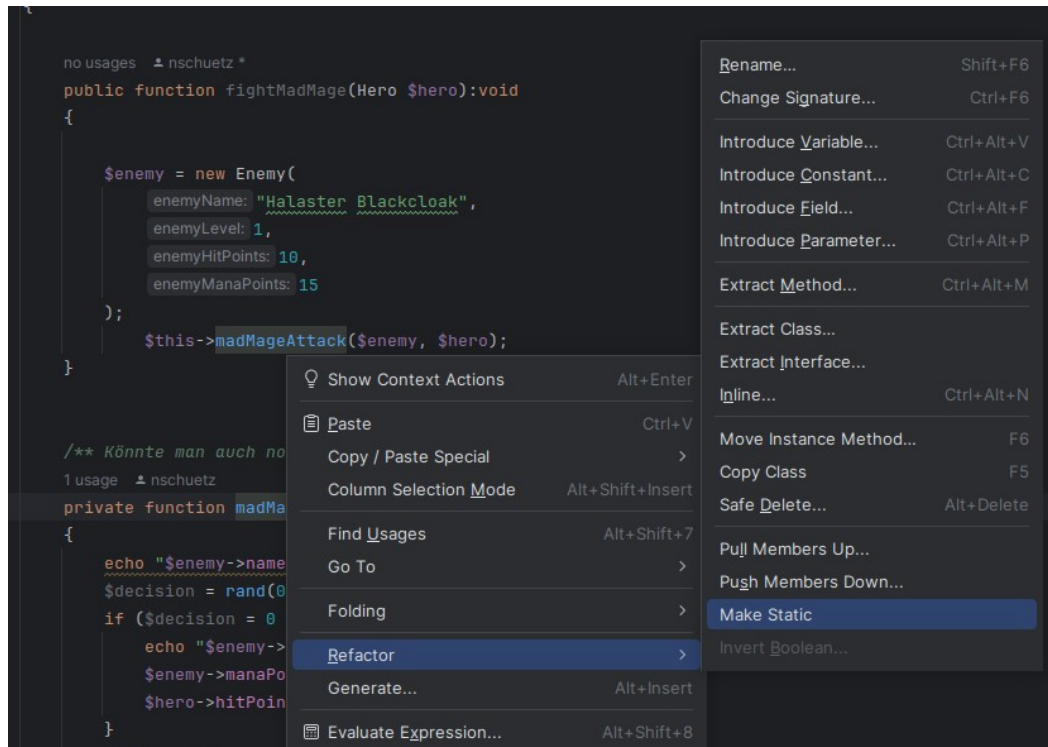
class Fight
{
    1 usage  1 nschuetz *
    public function fightMadMage(Hero $hero):void
    {
        $enemy = new Enemy(
            enemyName: "Halaster Blackcloak",
            enemyLevel: 1,
            enemyHitPoints: 10,
            enemyManaPoints: 15
        );
        $this->madMageAttack($enemy, $hero);
    }
}

```


Make Static:

Macht eine Methode statisch und passt die Aufrufe an.

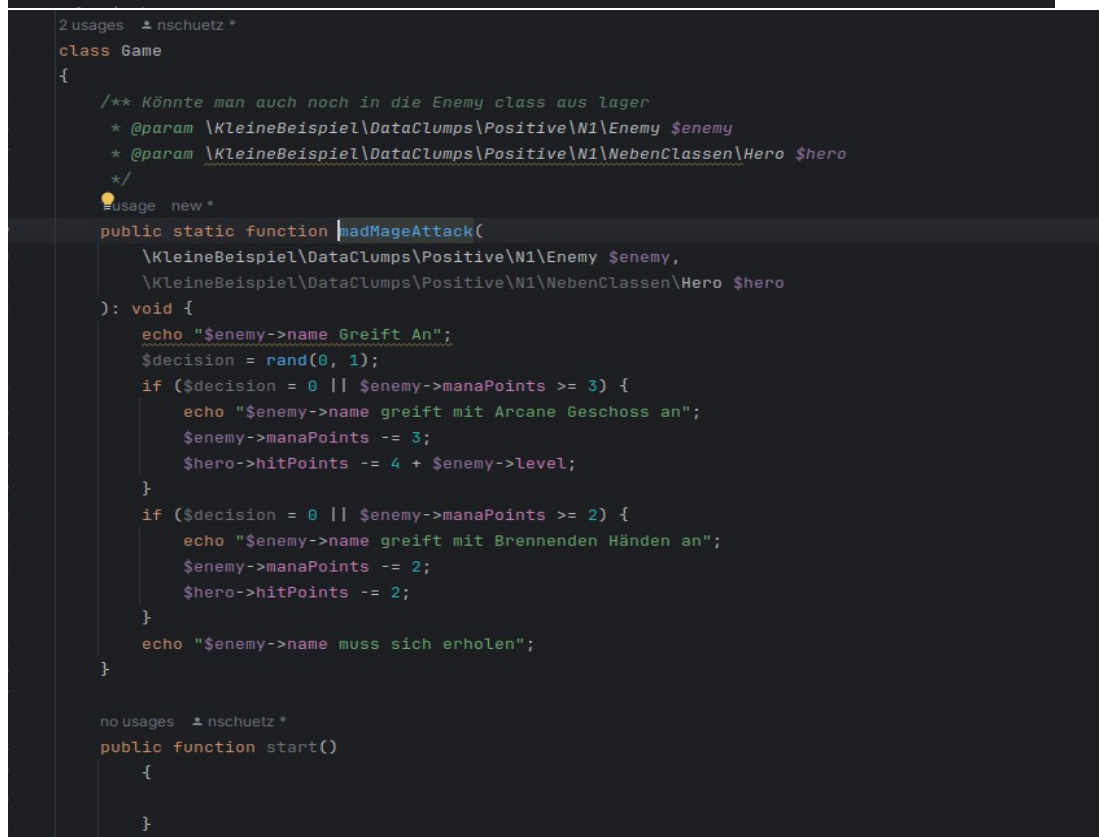
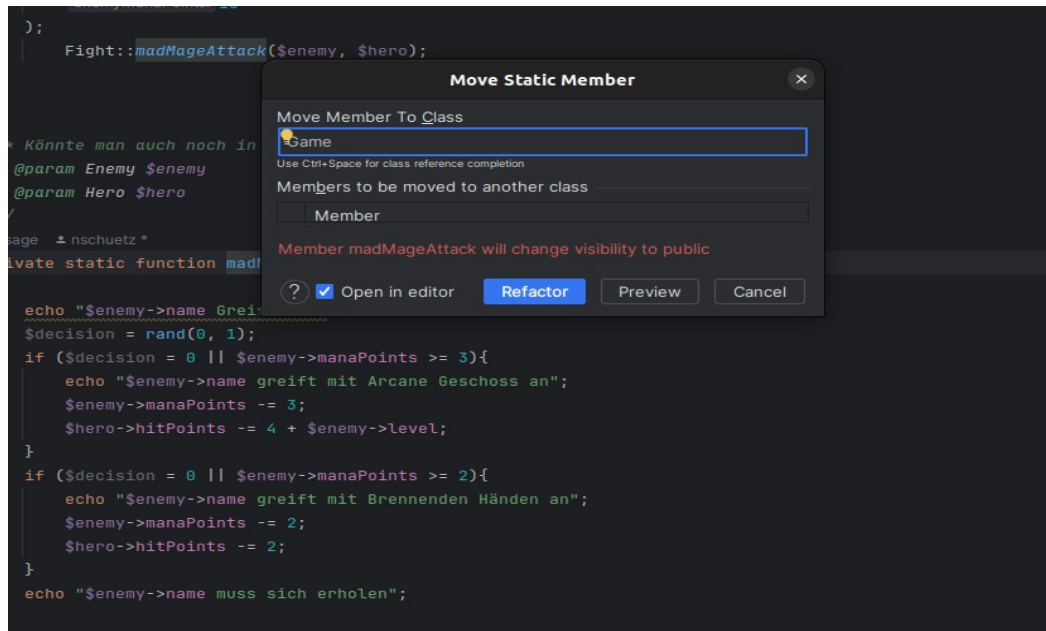
Dazu muss man einen Rechtsklick auf den Methodennamen machen und dann Refactor auswählen und dort Make Static.



Move:

Shortcut: F6

PhpStorm bietet auch Unterstützung, um das Bewegen von Ordnern, Methoden, Konstanten usw. zu erleichtern. Bei der Bewegung werden automatisch die Pfade und Verzeichnisse angepasst.

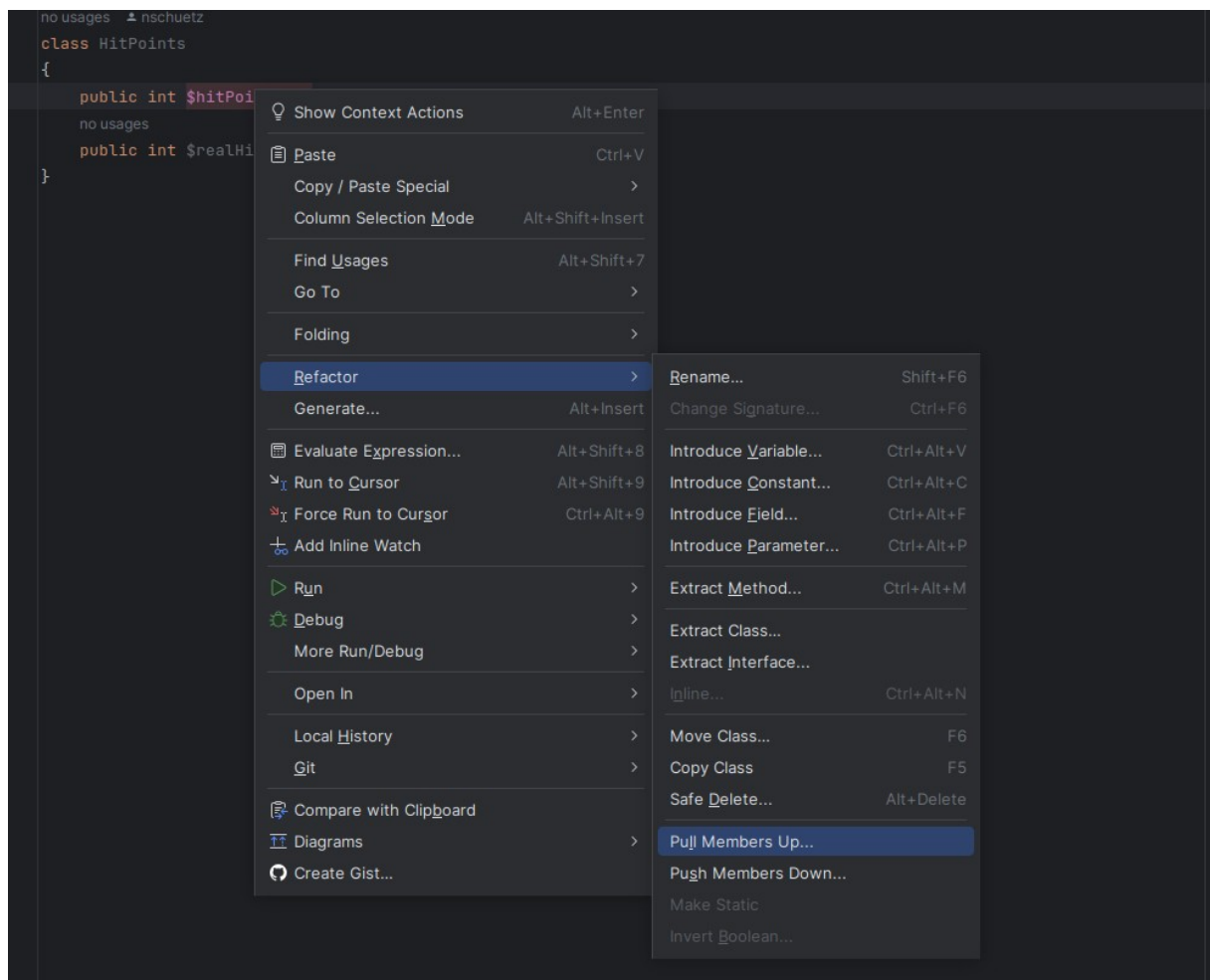


Pull members up/push members down:

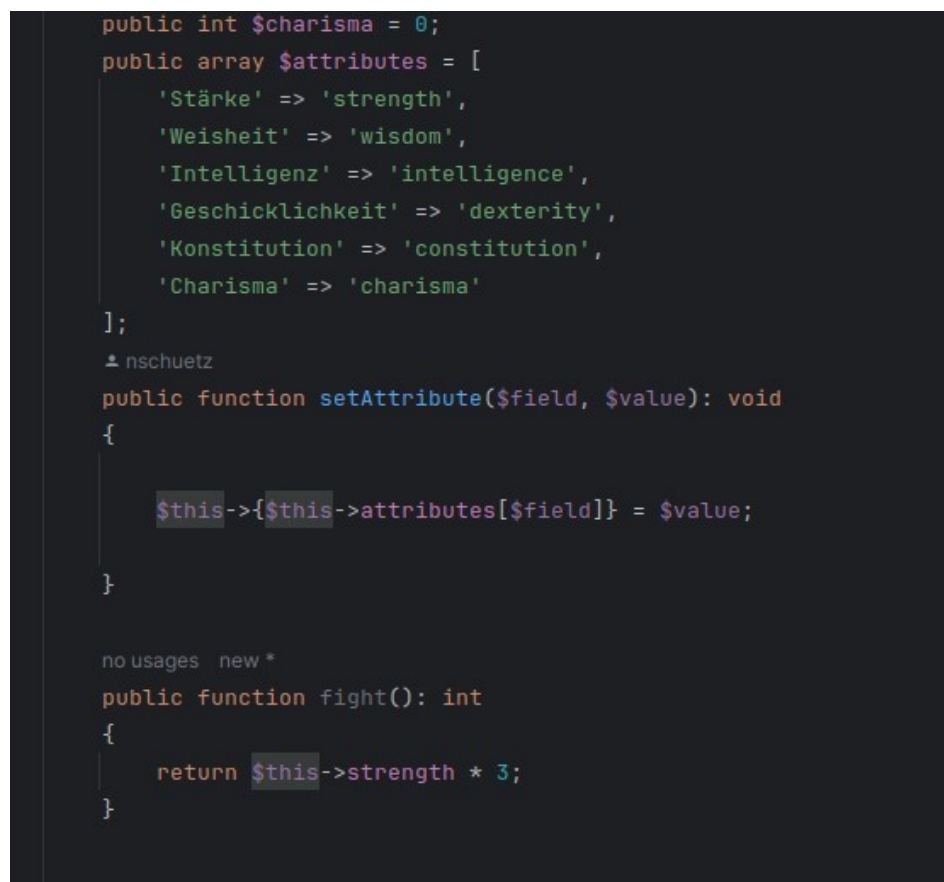
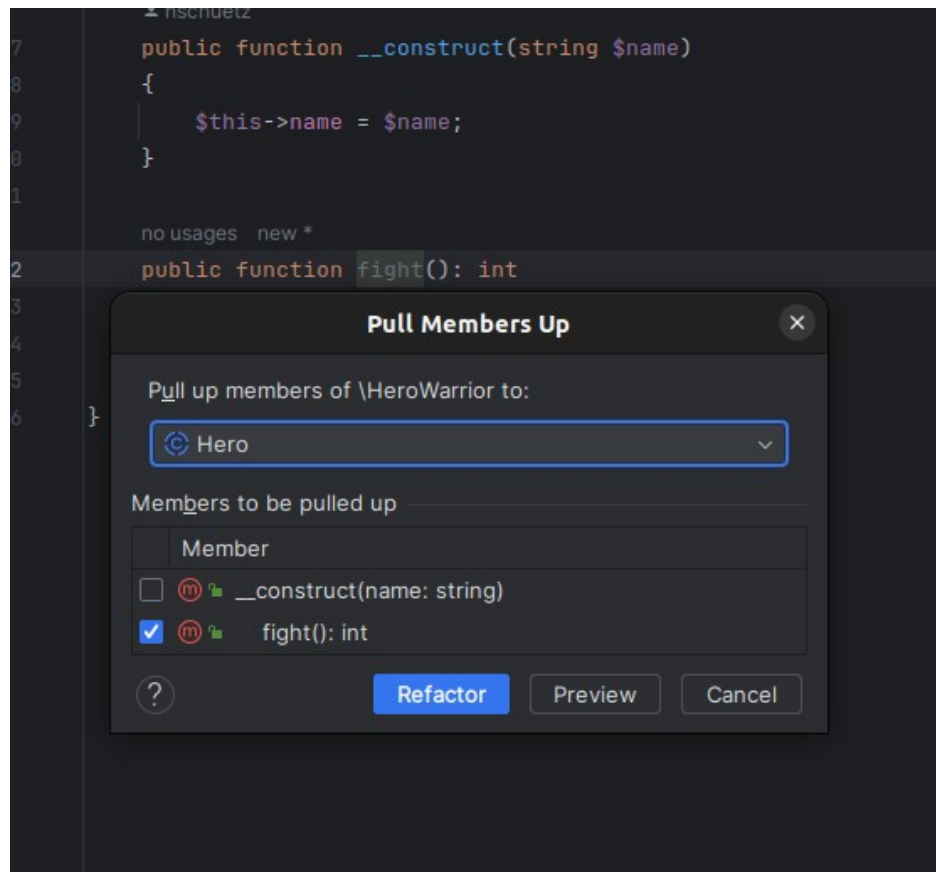
Um eine Funktion zu pullen oder zu pushen, muss man wieder über das allgemeine Menü gehen, welches man mit einem Rechtsklick öffnen kann, und dort auf "Refactor" gehen.

Diese Funktionen können dabei helfen, Klassenhierarchien einfacher zu refaktorisieren, indem sie es ermöglichen, Methoden/Attribute von der Kinderklasse zur Elternklasse zu bewegen (Pull Members Up) oder umgekehrt von der Elternklasse in die Kinderklasse.

Hierbei ist anzumerken, dass alle Kinderklassen die Methode/ Attribute von der Eltern Klasse übertragen bekommen.



Pull Members Up:



Push Members Down:

The image shows a two-part screenshot of an IDE. The top part displays a code editor with the following PHP code:

```
3 3 inheritors nschuetz *
4 Abstract class Hero
5 {
6
7
8     3 overrides
9     public string $name = '';
10    public int $level = 0;
11
12    no usages nschuetz *
13    public function fight(): int
14    {
15        return $this->level * 3;
16    }
17
18 }
19
```

On the right, a 'Push Members Down' dialog box is open. It contains the text 'Push members from Hero down' and a table titled 'Members to Be Pushed Down':

Member	
<input checked="" type="checkbox"/>	fight(): int
<input type="checkbox"/>	level: int = 0
<input type="checkbox"/>	name: string = ''

At the bottom of the dialog are buttons for '?', 'Refactor', 'Preview', and 'Cancel'.

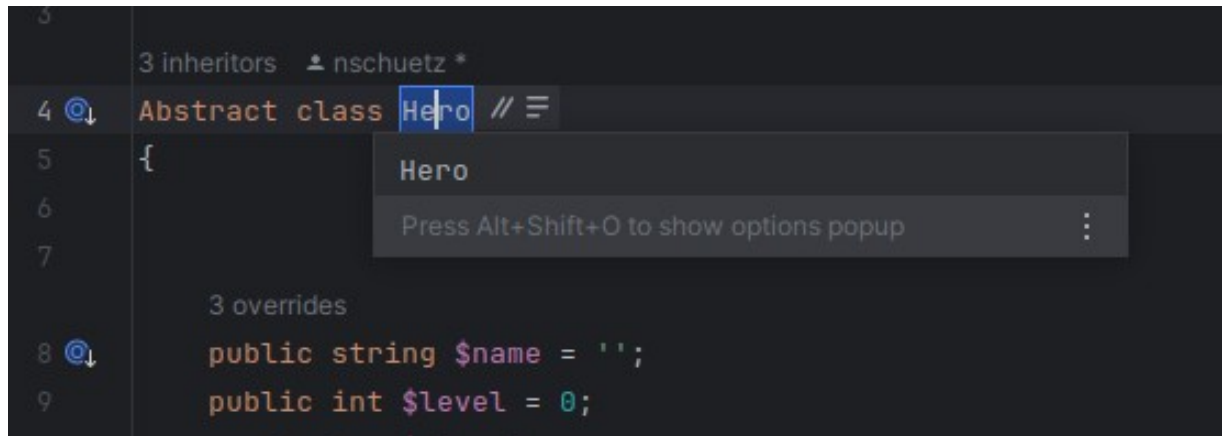
The bottom part of the image shows the code after the refactoring:

```
3
4
5 nschuetz *
6 class HeroWarrior extends Hero
7 {
8     nschuetz
9     public function __construct(string $name)
10     {
11         $this->name = $name;
12     }
13
14     no usages new *
15     public function fight(): int
16     {
17         return $this->level * 3;
18     }
19 }
```

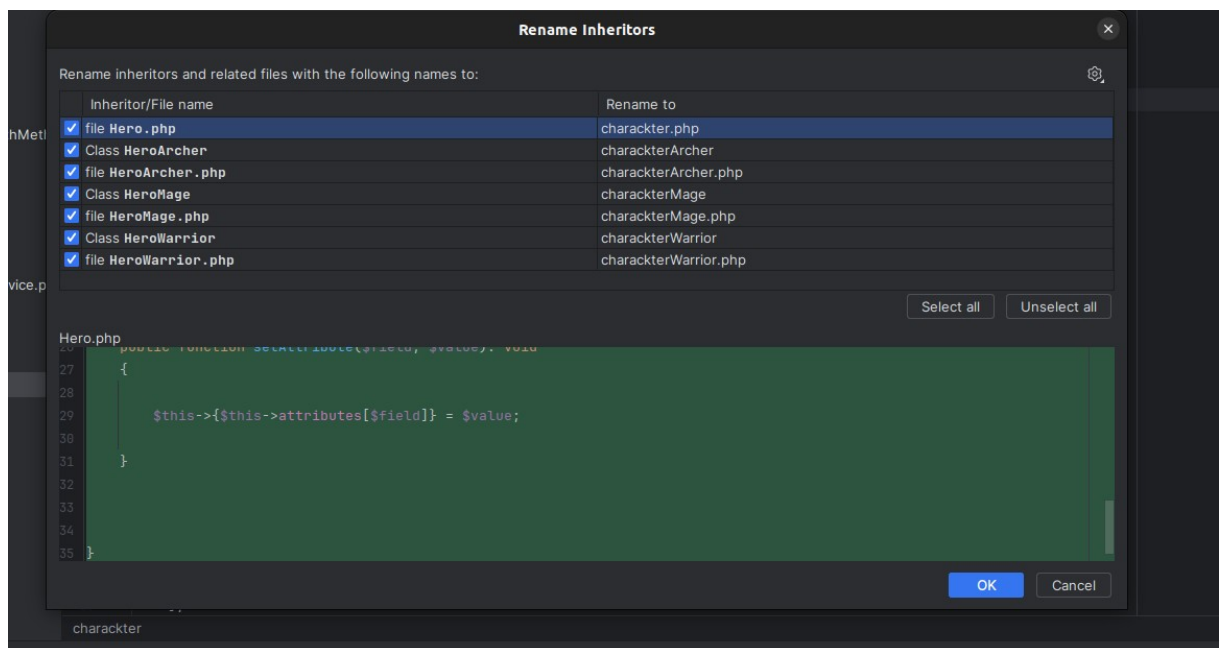
Rename:

Shortcut: Shift F6

Das ist eine der meistgenutzten Funktionen. Es ermöglicht, Dateien, Klassen, Namen und Werte gezielt auszutauschen, ohne dabei jede Stelle herauszusuchen und sie einzeln austauschen zu müssen.



```
3 3 inheritors  nschuetz *
4 4 Abstract class Hero // ≡
5 {
6
7
8 3 overrides
8 public string $name = '';
9 public int $level = 0;
```

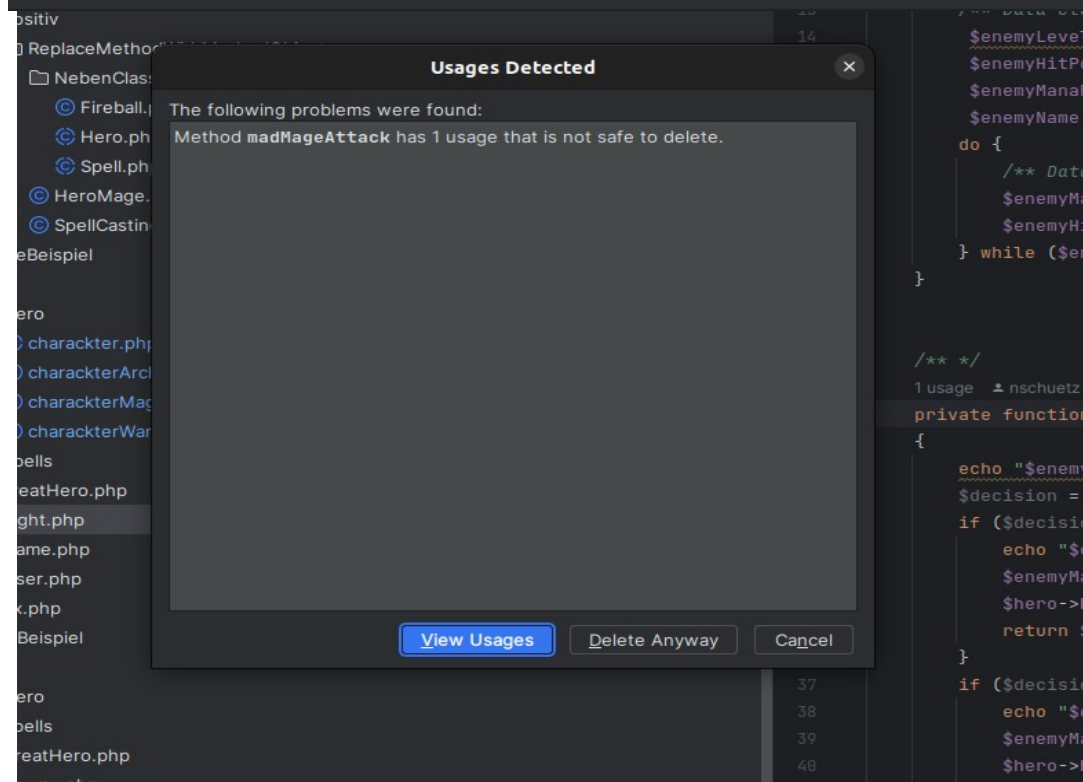
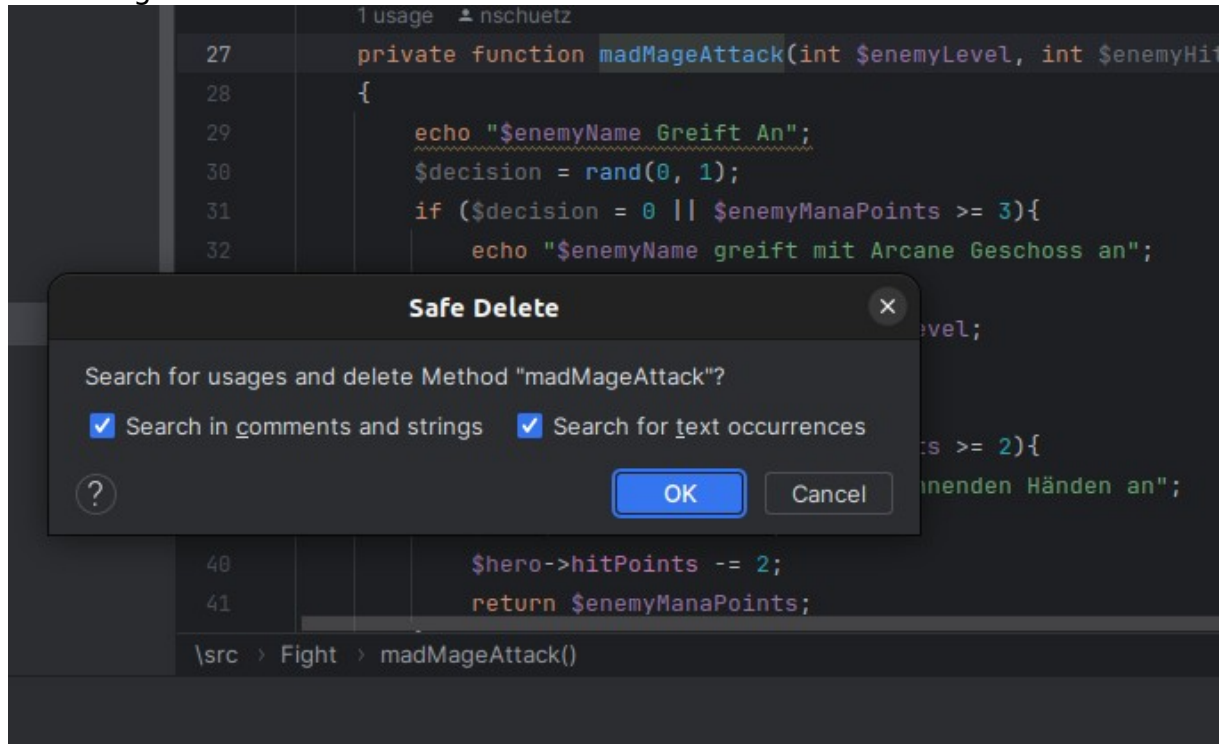


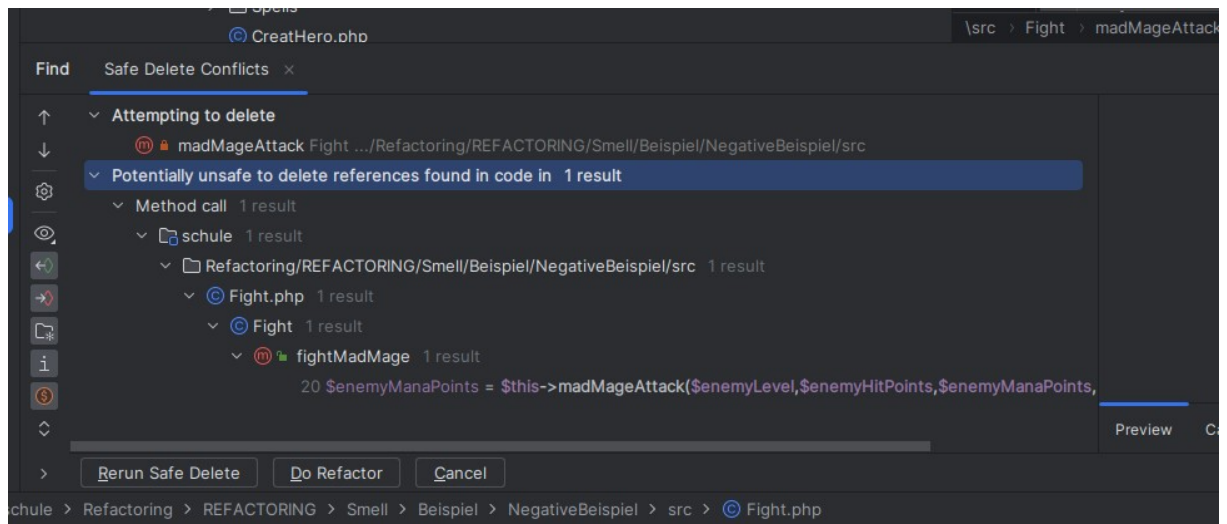
```
4
5
6
no usages  nschuetz *
7 class charackterArcher extends charackter
8 {
    no usages  nschuetz
    9 public function __construct(string $name)
    10 {
    11     $this->name = $name;
    12 }
    13 }
```

Safe delete:

Shortcut: Alt Delete

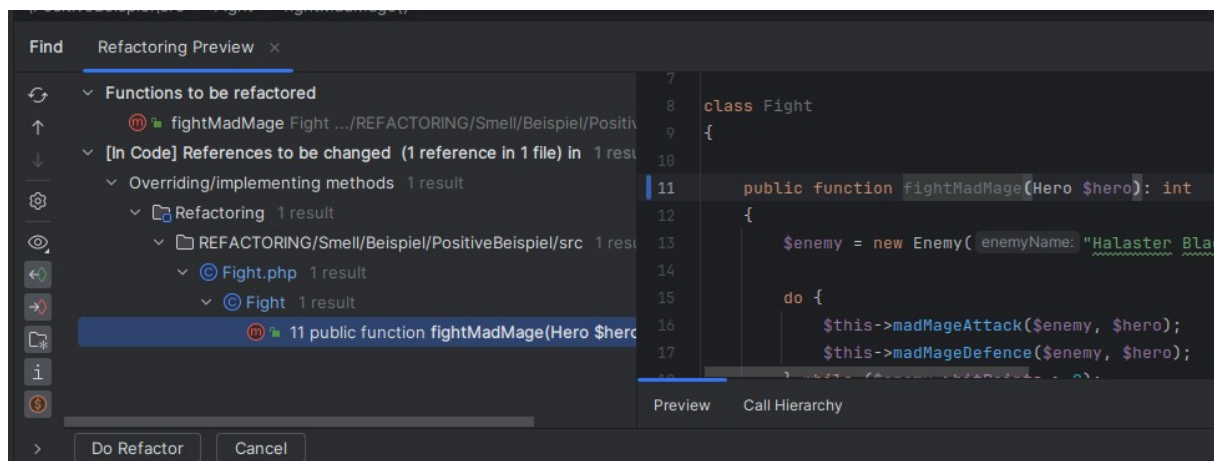
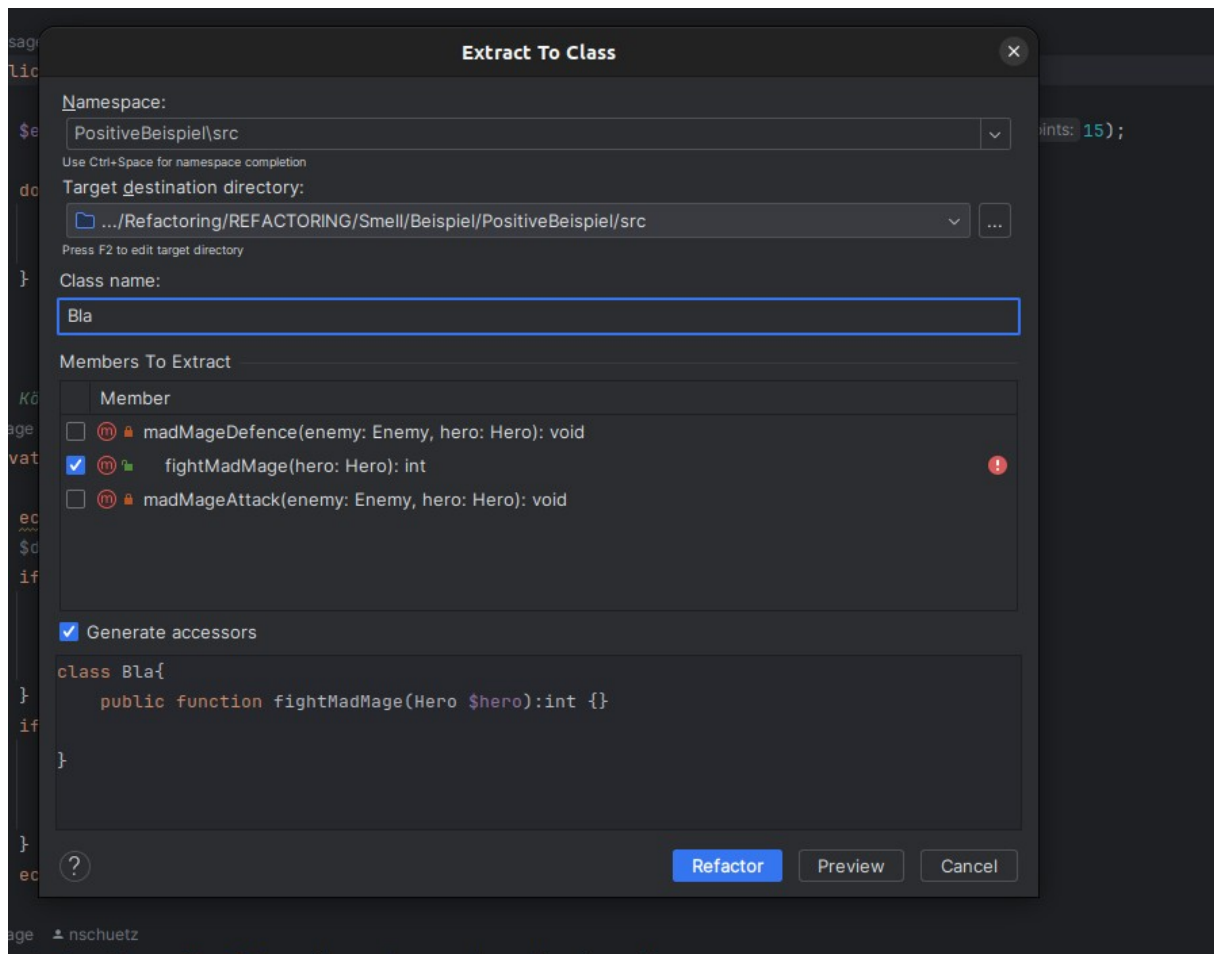
Entfernt die ausgewählte Datei, zeigt dem Benutzer jedoch zuvor, wo Verbindungen zu der zu löschende Datei existieren.





Allgemeiner Hinweis:

Bei größeren Veränderungen bzw. wenn man Preview auswählt, werden die Stellen, die geändert werden, von der IDE noch einmal gezielt angezeigt, sodass man überprüfen kann, ob es das ist, was man wirklich will.



Etwas Ähnliches passiert auch, wenn die IDE merkt, dass es ein Problem beim Refactoring gibt. Hierbei wird ein Konflikt ausgelöst, sodass der Benutzer noch einmal darüber schauen und auf den Konflikt reagieren kann.

Um Änderungen zurückzusetzen, gibt es den Shortcut Strg+Z.

Hier noch ein kleiner Tipp für diejenigen, die es bis hierhin geschafft haben und PHP nutzen:

Es gibt ein Tool namens Rector, mit dem man das Refactoring von Ganzen Projecten stark vereinfachen kann. Damit lassen sich jedenfalls einige Code-Smells ohne großen Aufwand entfernen.

<https://getrector.com/documentation>