Space Clean Up

Spieleprogrammierung und Softwarearchitektur in Squeak

Kai Fabian, Dominik Moritz, Matthias Springer, Malte Swart

Hasso-Plattner-Institut

23. Januar 2012

Überblick

1 Einführung

2 Demo

3 Architekturübersicht

4 Designentscheidungen

5 Abschließendes

Spielprinzip

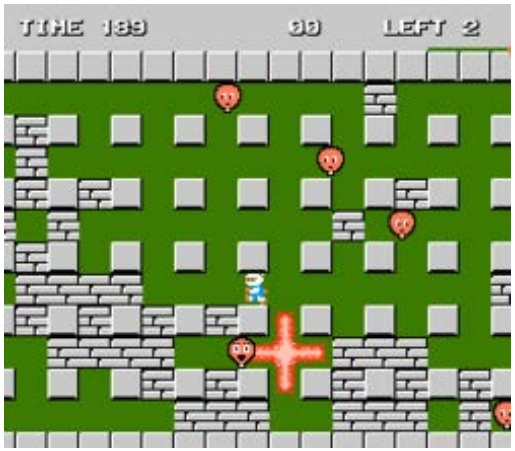
- Der Spieler kontrolliert einen Roboter, der seine Umgebung von Schleim und Schleimmonstern befreien soll.
- Zu diesem Zweck besitzt er eine Menge von Wassereimern, welche nach kurzer Zeit Wasser in alle Richtungen verteilen.

Spielprinzip (ausführlich)

- Das Spielprinzip basiert hauptsächlich auf Hudson Softs *Bomberman*.
- Der Spieler kontrolliert einen kleinen Roboter, dessen Aufgabe es ist, ein Raumschiff zu reinigen.
- Die Wege des Raumschiffs sind mit Schleim verschmutzt; dieser Schleim macht ein Betreten unmöglich.
- Der Schleim wird von – ansonsten friedlichen – grünen Schleimmonstern produziert, welche sich innerhalb des Raumschiffs bewegen.
- Um den Schleim zu entfernen, kann der Roboter Wassereimer platzieren. Diese werden nach kurzer Zeit auslaufen und die entstehende Wasserfront beseitigt den Schleim und auch eventuelle Schleimmonster.
- Sobald der gesamte Schleim, und dadurch bedingt auch alle Schleimmonster, beseitigt sind, ist das Level erfolgreich abgeschlossen.

Originale

- *Bomberman* von 1983 Hudson Soft (später weitergeführt als *Dynablaster* durch Ubisoft) ist ein zweidimensionales Actionspiel.
- Weiterhin basiert *Space Clean-Up* in Teilen auf *Pacman* (Bewegungslogik der Schleimmonster) und *Portals* (Portale)



Original NES-Gameplay

Did you know, that...?

- Hudson Soft entwickelt eine Bomberman-Auflage für den Nintendo 3DS.
- Es existieren 43 offizielle Auflagen von Bomberman, hauptsächlich für Nintendo-Systeme

Originale (ausführlich)

Bomberman

- *Bomberman* von 1983 Hudson Soft (später weitergeführt als *Dynablaster* durch Ubisoft) ist ein zweidimensionales Actionspiel.
- Der vom Spieler gesteuerte Protagonist ist *Bomberman*. Er hat die Fähigkeit, jederzeit mit seinen Händen Bombem zu erschaffen.
- Aufgabe des Spielers ist es, durch geschicktes Platzieren von Bomben Hindernisse zu zerstören und Gegner auszuschalten.
- Es existieren eine Reihe von Power-Ups, welche zum Beispiel die Anzahl der gleichzeitig platzierbaren Bomben erhöht, oder ihre Reichweite.

Weitere Inspirationsquellen

- *Pacman* (Bewegungslogik der Schleimmonster), *Portals* (Portale)

Überblick

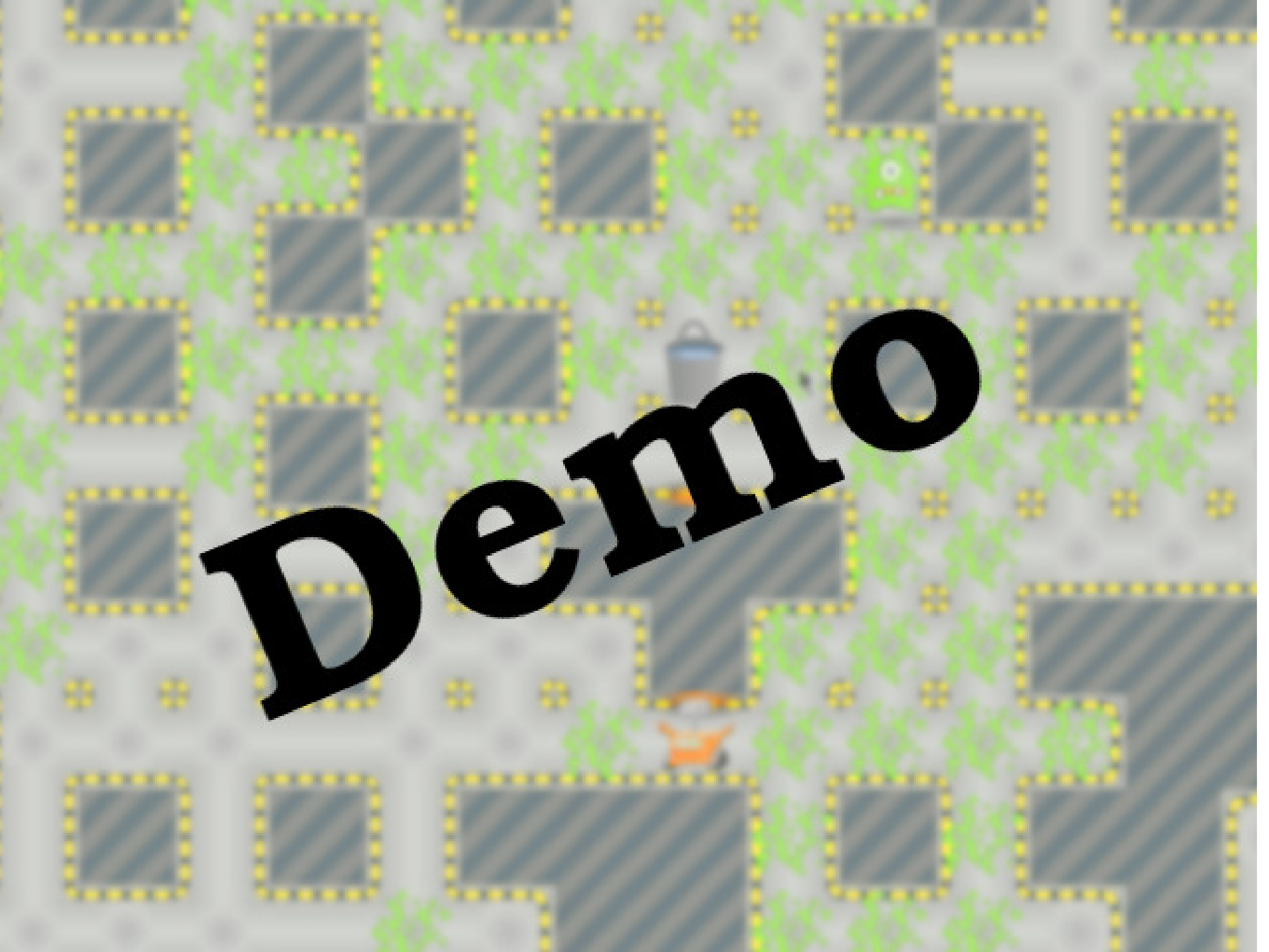
1 Einführung

2 Demo

3 Architekturübersicht

4 Designentscheidungen

5 Abschließendes


Demo

Überblick

1 Einführung

2 Demo

3 Architekturübersicht
- Zentrale Grundsätze
- Übersicht
- Who is Who?
- Items

4 Designentscheidungen

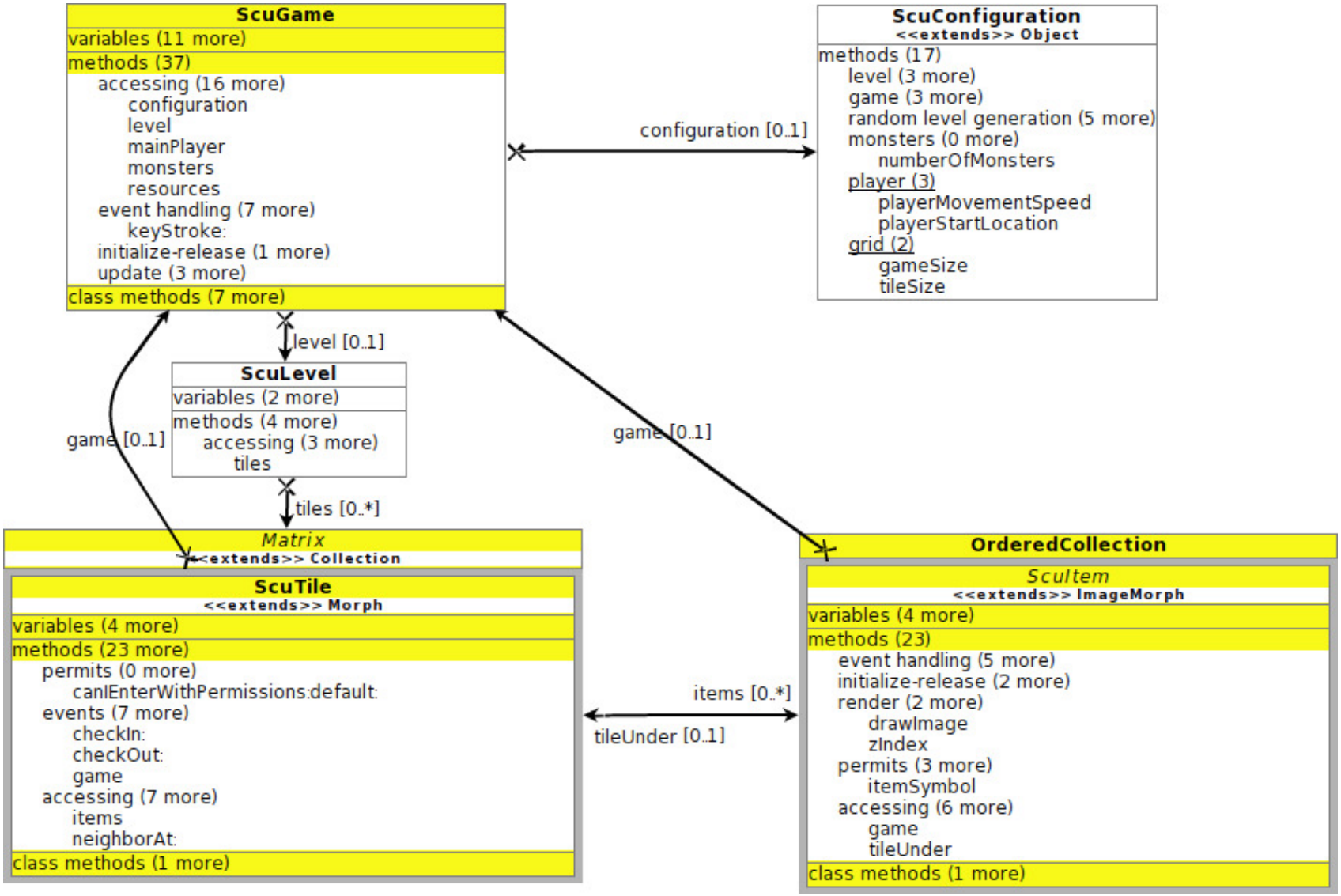
5 Abschließendes

Zentrale Grundsätze

Ziel: Autonomie

- Jedes Objekt handelt autonom
- keine Game-/Event-Loop
- Veränderung (einzeln) über step-Protokoll
- Domäneobjekte als Vorgabe

Übersicht über die Architektur



Übersicht über die Architektur (ausführlich)

- `ScuGame`:
  - Zentrale Spielkomponente. Fast jedes Objekt hält eine Referenz darauf.
  - Referenzen auf `ScuConfiguration`, alle Monster, den Spieler
  - Verarbeitung der Tastatureingaben
- `ScuLevel`: Gruppierung aller Tiles in einer Matrix. Nur beim Aufbau des Spielfeldes wichtig, danach Navigation über `ScuTile>>neighbors` (Graph).
- `ScuTile`: Ein Spielfeld.
  - `items`: Referenzen auf alle Items auf dem Feld
  - `canIEnterWithPermissions`: Enthält Zutrittslogik (wer darf das Feld betreten)
  - `checkIn`, `checkOut`: Hinzufügen, Entfernen von Items
- `ScuItem`: Alle Objekte, die sich auf Feldern befinden können, z.B. Schleim, Wände, Monster, Spieler, Pickup Items.

Who is Who?

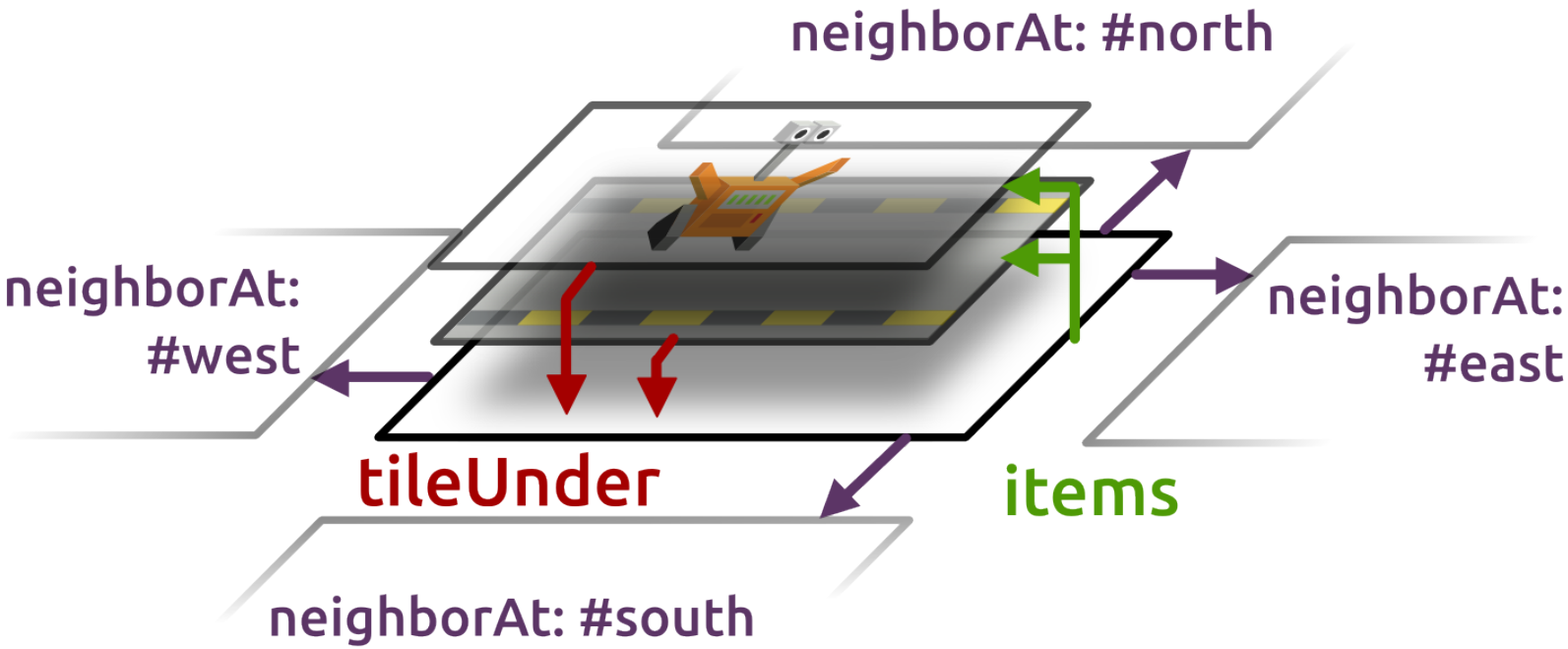


Abbildung: Beziehungen zwischen Tiles und Items

Who is Who (ausführlich)

- Item ist ein **Decorator** von Tile
- Item fügt dem Tile dynamisch zusätzliche Funktionalität und Aussehen hinzu (Objektkomposition hier besser als (Mehrfach-)Vererbung)
- Funktionalitäten können leicht hinzugefügt und entfernt werden

Interaktion zwischen Items

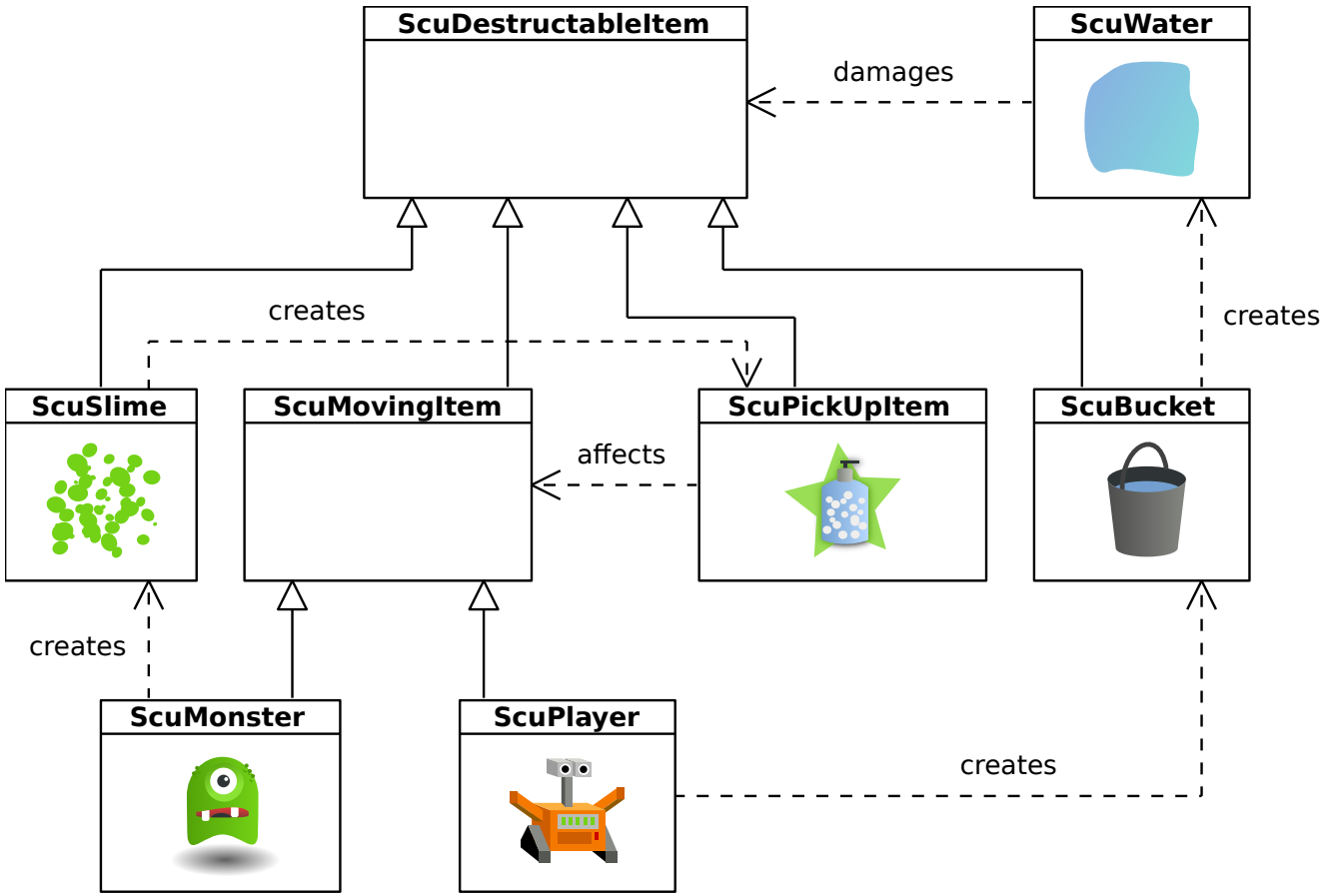
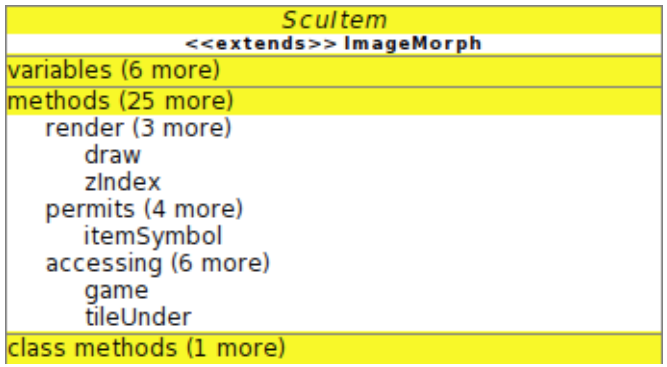


Abbildung: Interaktion zwischen Items

Interaktion zwischen Items (ausführlich)

- `ScuBucket` erzeugt `ScuWater` in alle Richtungen
- `ScuMonster` erzeugt `ScuSlime`
- `ScuPickUpItem` führt eine Aktion auf `ScuMovingItem` aus (kann sowohl Monster als auch Spieler sein)
- `ScuPlayer` erzeugt `ScuBucket`
- `ScuSlime` erzeugt bei Wegwaschen manchmal ein `ScuPickUpItem`
- `ScuWater` fügt allen `ScuDestructableItem` Schaden zu

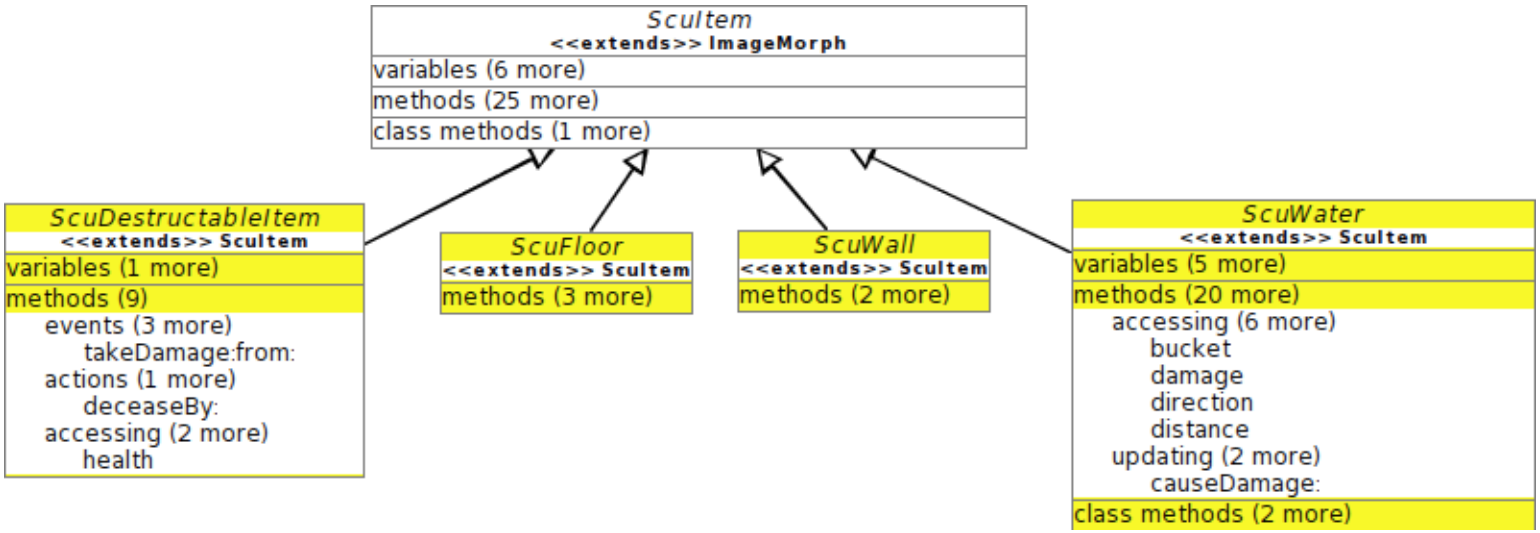
Items


ScuItem
<<extends>> ImageMorph
variables (6 more)
methods (25 more)
render (3 more)
draw
zIndex
permits (4 more)
itemSymbol
accessing (6 more)
game
tileUnder
class methods (1 more)

Items (ausführlich)

- Alle Items leiten von `ScuItem` ab
- `itemSymbol`: zur Ermittlung des Typs des Items, z.B. `#floor` für `ScuFloor` (weil Metaprogrammierung vermieden werden soll)
- `zIndex`: legt fest, in welcher Reihenfolge Items auf einem Tile liegen
- `game`: Referenz auf `ScuGame`
- `tileUnder`: Referenz auf das Tile, auf dem das Item liegt

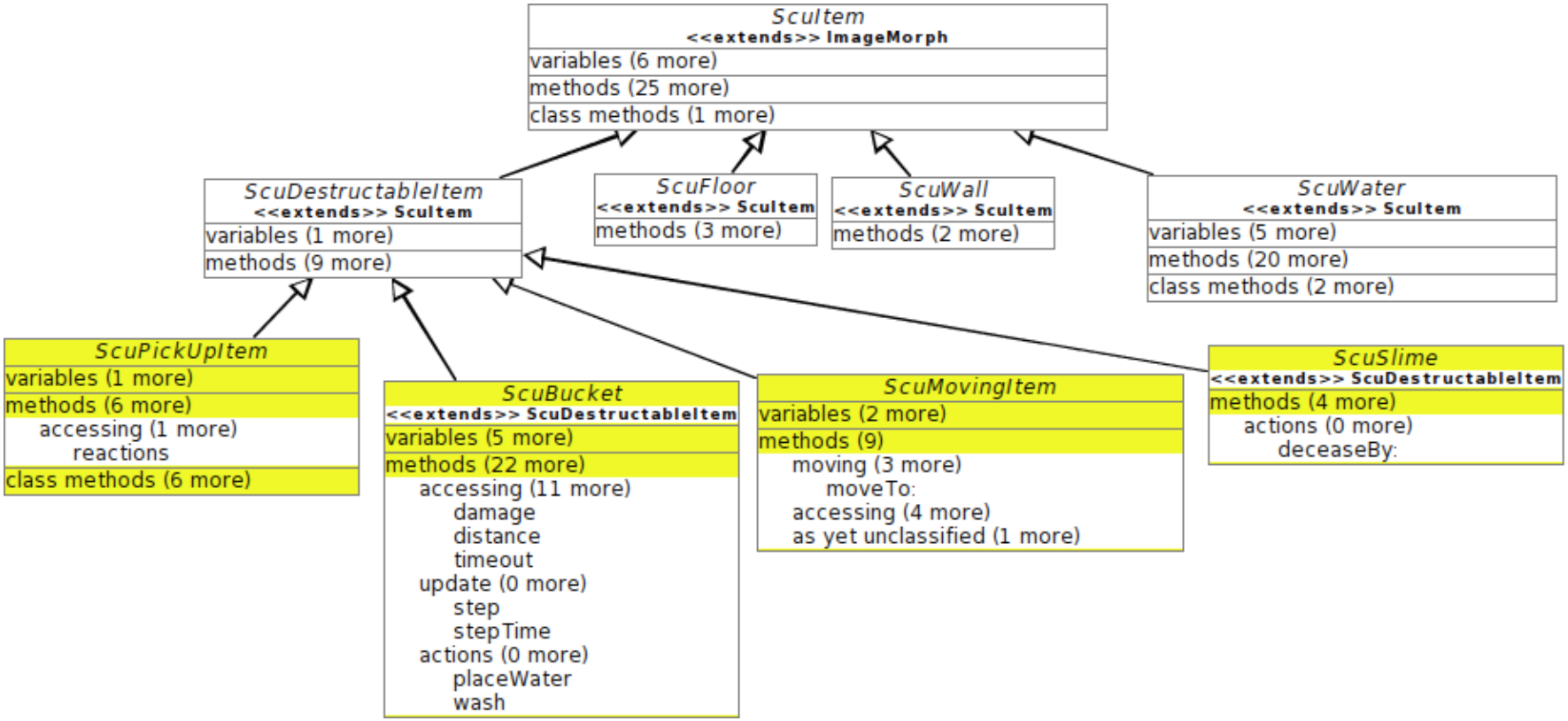
Items


ScuItem
<<extends>> ImageMorph
variables (6 more)
methods (25 more)
class methods (1 more)
ScuDestructableItem
<<extends>> ScuItem
variables (1 more)
methods (9)
events (3 more)
takeDamage:from:
actions (1 more)
deceaseBy:
accessing (2 more)
health
ScuFloor
<<extends>> ScuItem
methods (3 more)
ScuWall
<<extends>> ScuItem
methods (2 more)
ScuWater
<<extends>> ScuItem
variables (5 more)
methods (20 more)
accessing (6 more)
bucket
damage
direction
distance
updating (2 more)
causeDamage:
class methods (2 more)

Items (ausführlich)

- `ScuDestructableItems` können Schaden nehmen (`takeDamage`), *sterben* (`deceaseBy`) und haben einen Gesundheitswert (`health`)
- `ScuFloor` ist der Fußboden und befindet sich u.a. unter `ScuSlime` (Schleim) oder `ScuPickUpItem`
- `ScuWall` ist eine Wand und kann nicht betreten werden
- `ScuWater` ist Wasser, welches sich in eine Richtung weiter ausbreiten kann und Schaden an anderen Items auf dem Tile auslöst (Interaktion geht nur von `ScuWater` aus)

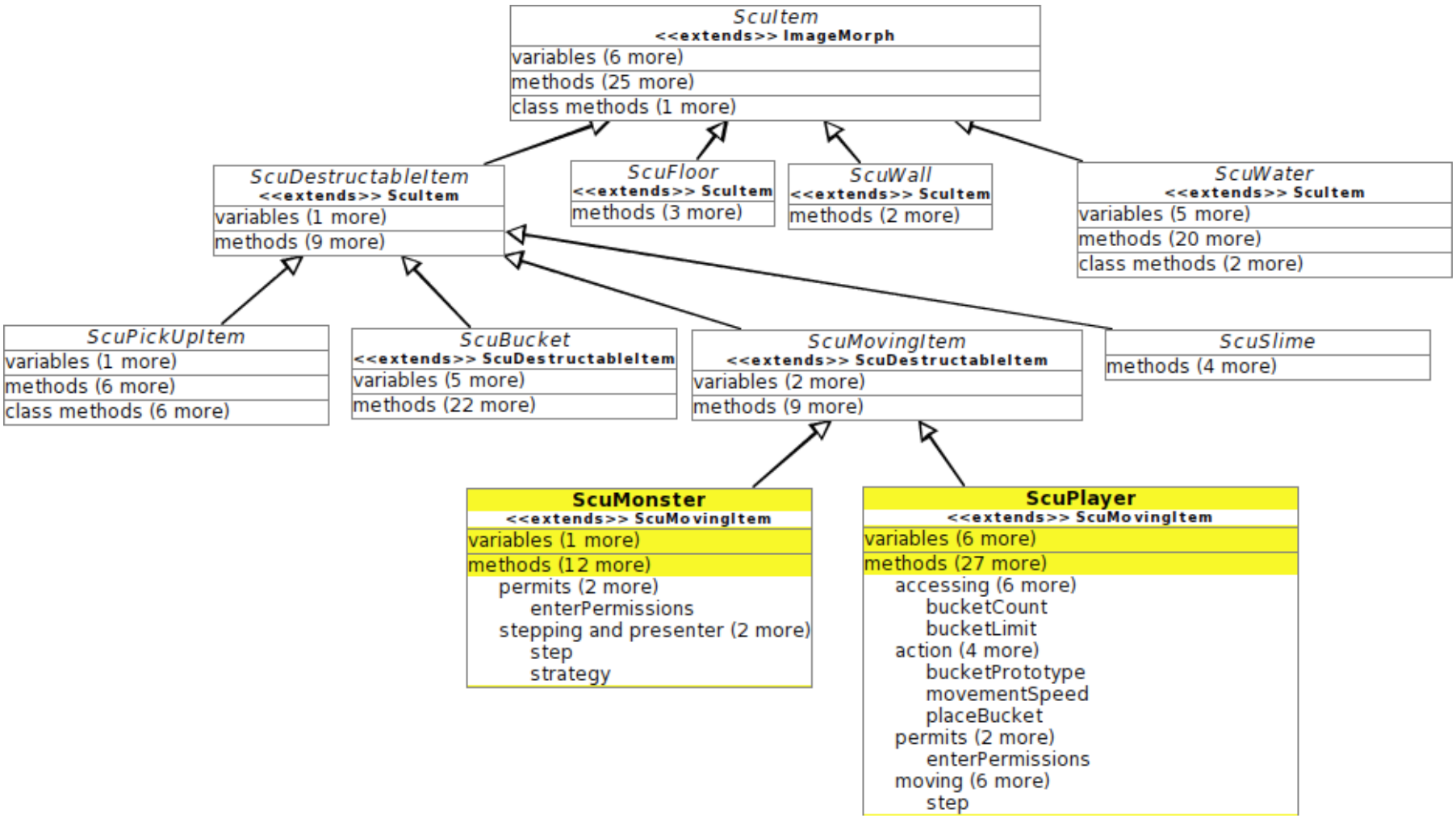
Items


ScuItem
<<extends>> ImageMorph
variables (6 more)
methods (25 more)
class methods (1 more)
ScuDestructableItem
<<extends>> ScuItem
variables (1 more)
methods (9 more)
ScuFloor
<<extends>> ScuItem
methods (3 more)
ScuWall
<<extends>> ScuItem
methods (2 more)
ScuWater
<<extends>> ScuItem
variables (5 more)
methods (20 more)
class methods (2 more)
ScuPickUpItem
variables (1 more)
methods (6 more)
accessing (1 more)
reactions
class methods (6 more)
ScuBucket
<<extends>> ScuDestructableItem
variables (5 more)
methods (22 more)
accessing (11 more)
damage
distance
timeout
update (0 more)
step
stepTime
actions (0 more)
placeWater
wash
ScuMovingItem
variables (2 more)
methods (9)
moving (3 more)
moveTo:
accessing (4 more)
as yet unclassified (1 more)
ScuSlime
<<extends>> ScuDestructableItem
methods (4 more)
actions (0 more)
deceaseBy:

Destructable Items (ausführlich)

- `ScuPickUpItem` können vom Spieler (oder Monstern) aufgenommen werden und führen dort eine Aktion aus (z.B. Bucket-Anzahl erhöhen, Bucket-Schaden erhöhen)
  - Aktion wird im `ScuPickUpItem` in Form einer Closure (`reactions`) gespeichert (Command-Pattern?)
- `ScuBucket` erzeugt nach dem `timeout` `ScuWater` in alle Richtungen
- `ScuMovingItem` implementiert grundlegende Bewegungslogik und Animationslogik
- `ScuSlime` kann mit Wasser weggewaschen werden und hinterlässt manchmal ein `ScuPickUpItem`

Items


ScuItem
<<extends>> ImageMorph
variables (6 more)
methods (25 more)
class methods (1 more)
ScuDestructableItem
<<extends>> ScuItem
variables (1 more)
methods (9 more)
ScuFloor
<<extends>> ScuItem
methods (3 more)
ScuWall
<<extends>> ScuItem
methods (2 more)
ScuWater
<<extends>> ScuItem
variables (5 more)
methods (20 more)
class methods (2 more)
ScuPickUpItem
variables (1 more)
methods (6 more)
class methods (6 more)
ScuBucket
<<extends>> ScuDestructableItem
variables (5 more)
methods (22 more)
ScuMovingItem
<<extends>> ScuDestructableItem
variables (2 more)
methods (9 more)
ScuSlime
methods (4 more)
ScuMonster
<<extends>> ScuMovingItem
variables (1 more)
methods (12 more)
permits (2 more)
enterPermissions
stepping and presenter (2 more)
step
strategy
ScuPlayer
<<extends>> ScuMovingItem
variables (6 more)
methods (27 more)
accessing (6 more)
bucketCount
bucketLimit
action (4 more)
bucketPrototype
movementSpeed
placeBucket
permits (2 more)
enterPermissions
moving (6 more)
step

Moving Items (ausführlich)

- `ScuMonster`: ein Monster
  - Erzeugt `ScuSlime` auf allen betretenen Tiles
  - Bewegungsstrategy (`strategy`) legt Bewegungsrichtung fest und wird bei Erzeugung des Monsters (momentan) zufällig festgelegt
  - `MonsterRandomStrategy` zur zufälligen Bewegung
  - `MonsterToPlayerStrategy` läuft immer auf den Spieler zu (einfache Breitensuche dank Graphstruktur der Tiles) und macht das Spiel wesentlich schwieriger
- `ScuPlayer`: ein Spieler
  - Kann Buckets platzieren
  - `bucketPrototype` ist Prototyp eines Buckets, kopiert wird (Prototype-Pattern)
  - `bucketLimit`: maximale Anzahl der Bucket, ...

Überblick

1 Einführung

2 Demo

3 Architekturübersicht

4 Designentscheidungen
- Zutrittskontrolle
- Item-Interaktion
- SVG
- Animationen

5 Abschließendes

Zutrittskontrolle

Problem: Zutrittskontrolle

Es gibt wenige `MovingItem` (2) und viele nichtbewegliche Items (zur Zeit 7).

- Zum Beispiel: *Darf ein Spieler ein Feld mit einer Wand betreten?*
- Festlegung über Zulassung sollte in `ScuMovingItem` definiert sein
- Die Prüflogik hingegen im `ScuTile`, da dieser Vermittler zwischen Tiles und Items ist.

1. Idee: Visitor Pattern

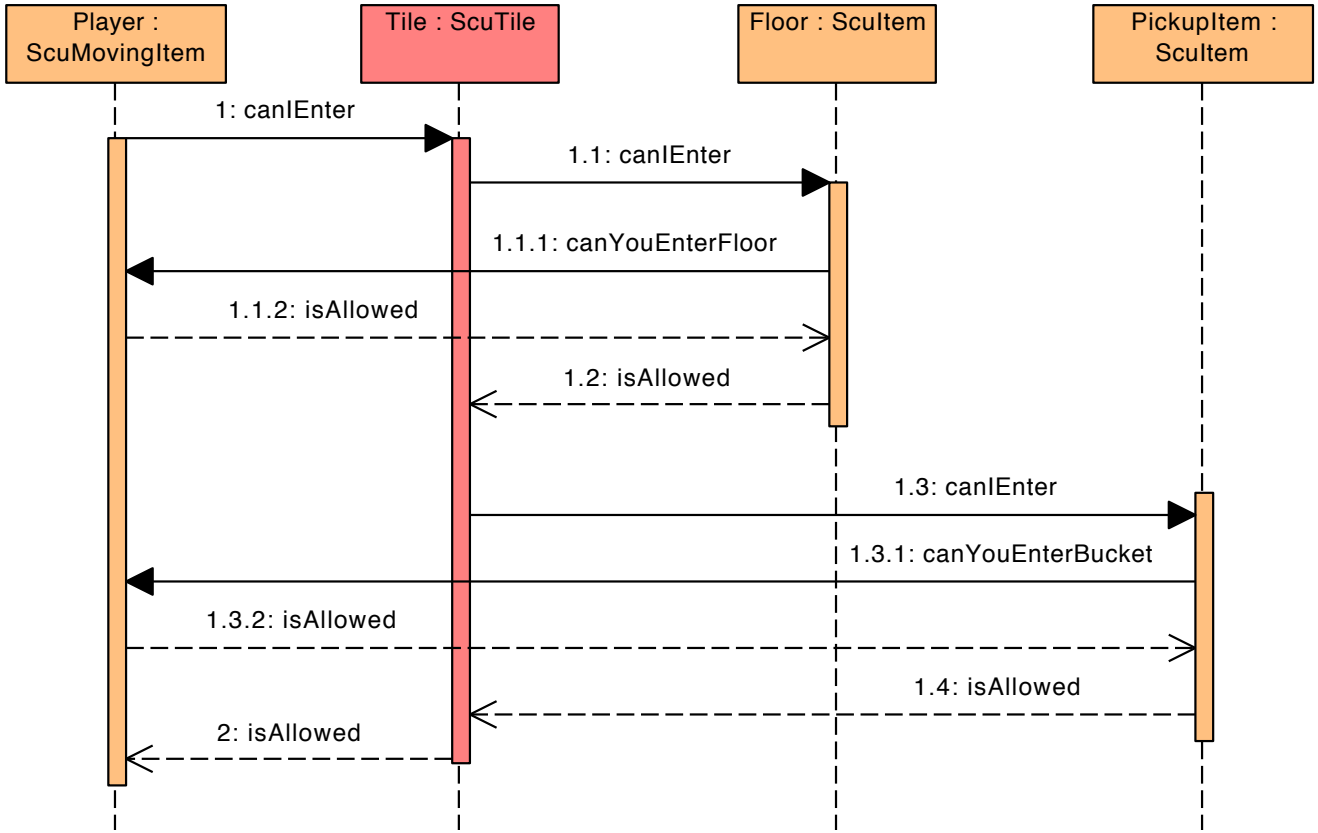


Abbildung: Double Dispatch als Implementierungsstrategie des Visitor Pattern für die Bestimmung der Zulassung

1. Idee: Auswertung

Vorteile

- Zulassung geschieht in Abhängigkeit von dem konkreten Visitor (z.B. Player) und dem konkreten Node (Tile mit Dekoratoren)
- Neue `MovingItems` lassen sich leicht ergänzen, was aber selten vorkommt

Nachteile

- Aufwändige Kommunikation
- Indirektion schwer verständlich
- Den `MovingItems` (Monster, Player) müssen alle nichtbewegliche Items (Floor, Bucket...) bekannt sein

2. Idee: Permissions

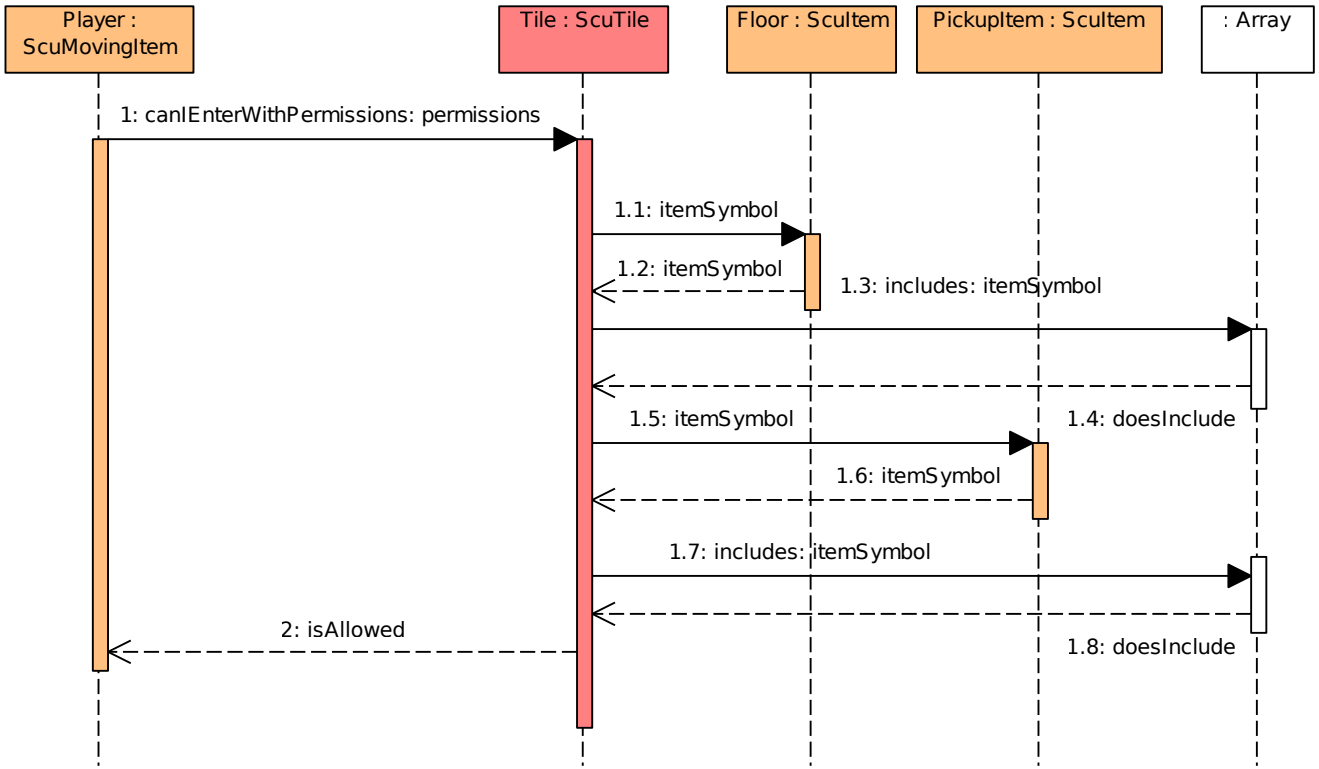


Abbildung: Permissions für die Bestimmung der Zulassung

2. Idee: Implementierung

```
1 Tile>>canIEnterWithPermissions: permissions
2   "example: canIEnterWithPermissions: #(floor
      pickUpItem)"
3   ^self items allSatisfy: [ :anItem |
4     permissions includes: anItem itemSymbol ].
```

Code 1: Implementierung von canIEnterWithPermissions

2. Idee: Auswertung

Vorteile

- Allgemeine Definitionen möglich
- Feststehende Items lassen sich leicht ergänzen, `MovingItems` ebenso
- Logik in `Tile`, Erlaubnis in `MovingItem`

Nachteile

- Verwendung von Symbolen zur Identifikation (werden auch für Darstellung genutzt)

Item-Interaktion

Problem: Item-Interaktion

`Items` können sich gegenseitig beeinflussen.

- Zum Beispiel: *`ScuWater` fügt `ScuPlayer` Schaden in Höhe von 2 zu.*
- Problem: Das `ScuWater` kennt den `ScuPlayer` nicht.

1. Idee: Direkte Beeinflussung

- `ScuPlayer` stellt Funktion `takeDamage: anAmount` bereit.
- `ScuWater` kann Referenz auf `ScuPlayer` über Feldreferenz (`aTile items`) erhalten.

Nachteile

- Starke Kopplung der Items untereinander und mit der restlichen Spiellogik (z.B. Statistiken)
- Alle Item müssen Funktion `takeDamage` bereitstellen, auch wenn sie keinen Schaden nehmen (z.B. `ScuFloor`)

2. Idee: Event-Pattern

- `ScuItems` registrieren sich bei einem `ScuEventDispatcher` für Events auf einem bestimmten Feld.

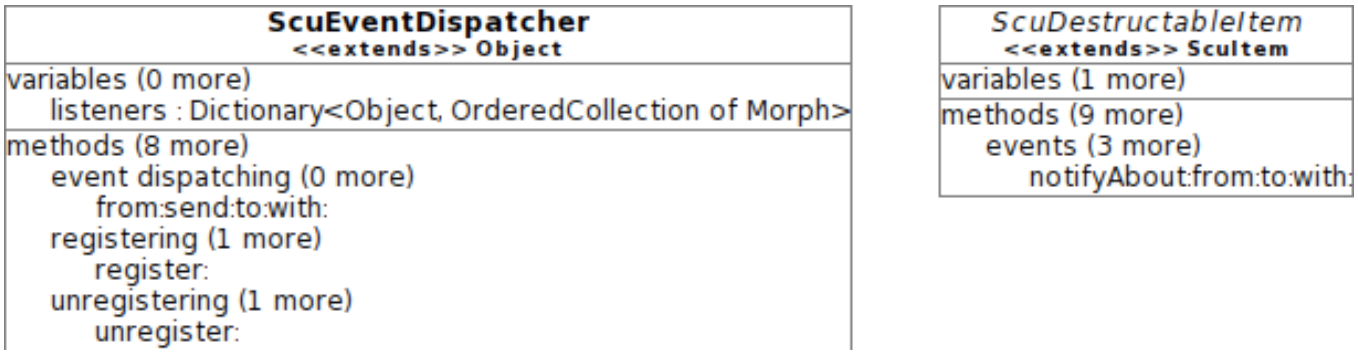


Abbildung: Event Dispatching bei zerstörbaren Items

2. Idee: Event-Pattern

Vorteile

- Lose Kopplung – Absender muss Empfänger (und ihr Verhalten) nicht kennen
- Empfänger können Verhalten/Reaktion auf Ereignisse selbst bestimmen
- Jederzeit ohne großen Aufwand und eventuelle Nebeneffekte um weitere Events erweiterbar

Nachteile

- Mehr Vorwissen für Verständnis notwendig
- Durch zusätzlichen Zwischenaufruf marginal langsamer

Graphikart

Problem: Darstellung

Wie kommen die Bilder der Items zustande?

Unser Ansatz: SVG Image im Quelltext

Vorteile

- Auflösungen flexible anpassbar
- Keine manuelle Aktionen der Designer notwendig

Nachteile

- Externe Bibliothek (SVGMorph) notwendig
- SVGMorph stellt nicht alle Aspekte korrekt dar
- Darstellung langsamer

Graphikart: Umsetzung

- SVG String wird zu Form gerendert

```
1  | doc svg form |
2  doc := XMLDOMParser parseDocumentFrom: aStream.
3  svg := SVGMorph new createFromSVGDocument: doc.
4  form := Form extent:  self imageSize depth: 32.
5  svg fullDrawOn: form getCanvas.
6  ^ form
```

Code 2: SVGMorph erstellen

- Bilder als String im Quelltext gespeichert

Animationen

Problem: Fließende Bewegung der `ScuMovingItem`

Muss asynchron sein, damit die UI nicht blockiert.

```
1 AnimPropertyAnimation new
2     duration: 500;
3     target: myMorph;
4     property: #position;
5     startValue: 10@10;
6     endValue: 100@100;
7     start;
```

Code 3: Animations Framework

Überblick

1 Einführung

2 Demo

3 Architekturübersicht

4 Designentscheidungen

5 Abschließendes
- Weitere Architekturdetails
- Ausblick
- Fazit

Weitere Architekturdetails

- Aufbau des Spiels mit `ScuRandomLevelBuilder` und `ScuLoaderLevelBuilder`
- Funktionsweise der `ScuPickUpItems` (**Command** Pattern)
- Die verschiedenen Spiel-Strategien der Monster
- Das Platzieren der Wassereimer (**Prototype** Pattern)
- Caching und Rendern der Bilder (**Flyweight** Pattern)
- Umsetzung der Portale
- Resourcemanager (**Singleton** Pattern)

Ausblick

1. **Spielrahmen** mit Punkteanzeige, Einstellungen, etc.
2. **Schwierigkeitsstufen**
   - Anzahl der Monster
   - Wahl und Parameterisierung der Monster Strategies
   - evtl. Weiterentwicklung der Strategies, sodass Monster von Buckets weglaufen
3. **Verschiedene Level** (original Bomberman Level)

We liked

- Die bereitgestellten **Beispiele** und **Bibliotheken** (z.B. Animations)
- Die meisten Pattern erst nach **Refactoring** angewandt (z.B. Prototype Pattern bei `ScuBucket`) - **Lerneffekt**
- Umsetzung eines **Spiels**

We wish

- Die **Pattern** etwas **früher** in der Vorlesung vorstellen
- Monticello ist deutlich mächtiger als vorgestellt

Weitere Designentscheidungen

Die folgenden Folien sind kein Teil unserer Präsentation und werden nur gezeigt, wenn Fragen aus dem Publikum kommen.

Funktionsweise der Pickup Items

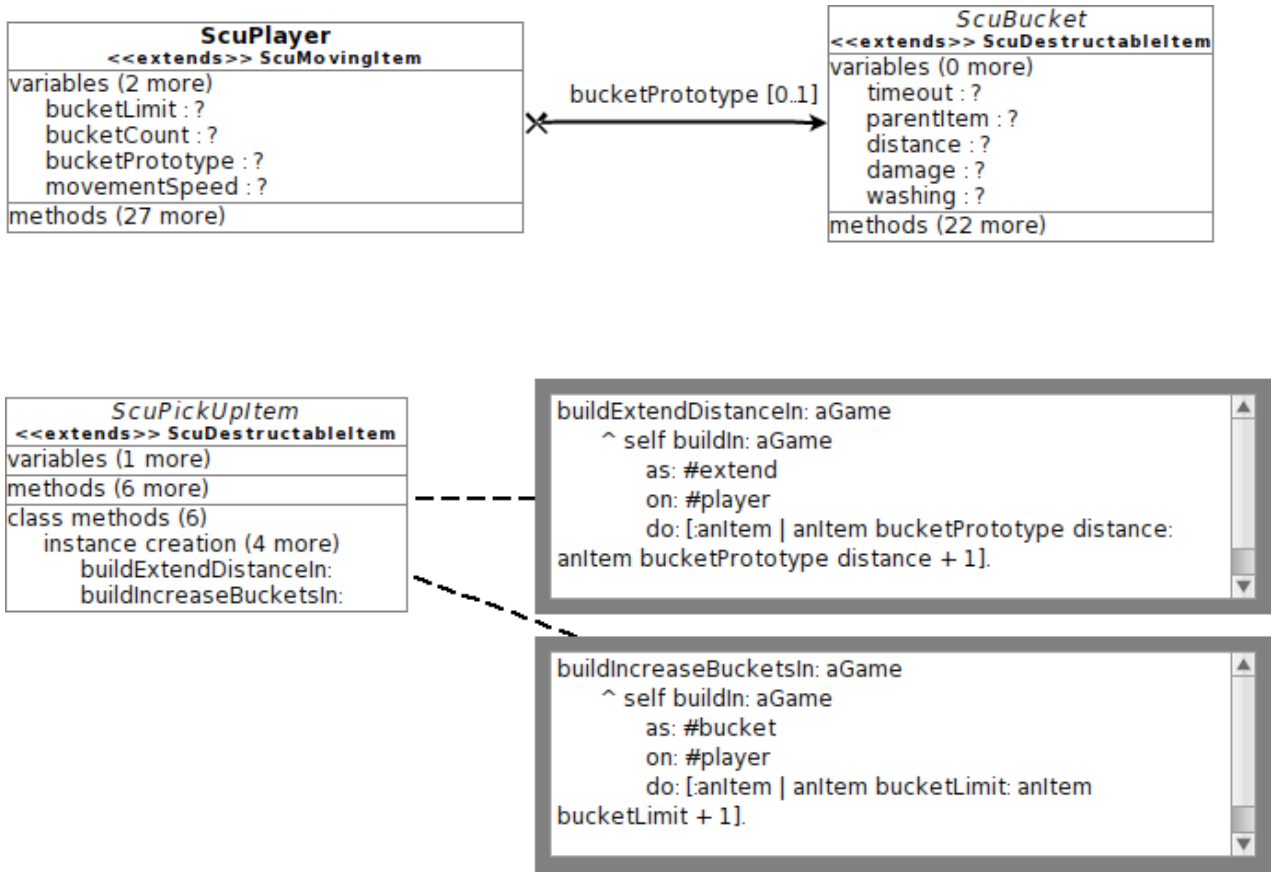
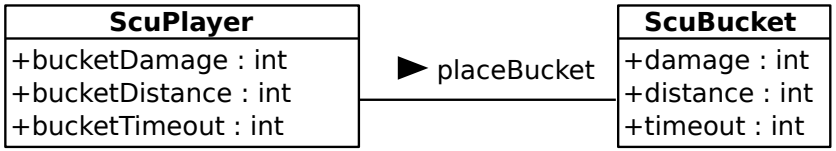


Abbildung: Funktionsweise der Pickup Items

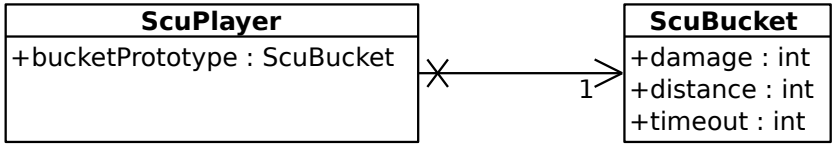
Pickup Items (ausführlich)

- Pickup Items betreffen bestimmte Akteure (z.B. Spieler, Monster)
- Bei Erzeugung wird festgelegt, was ein Item macht
- Pickup Items werden nach dem Entfernen von Schleim erzeugt

Bucket Prototyp



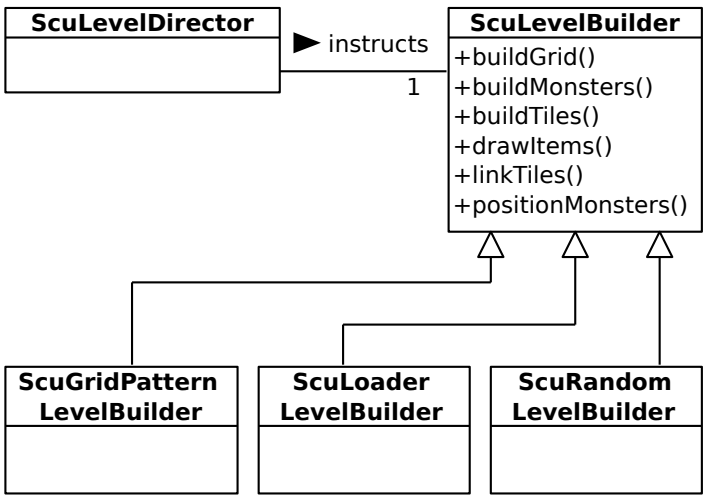
(a) Vorher: `ScuPlayer` speichert Informationen über `ScuBuckets`

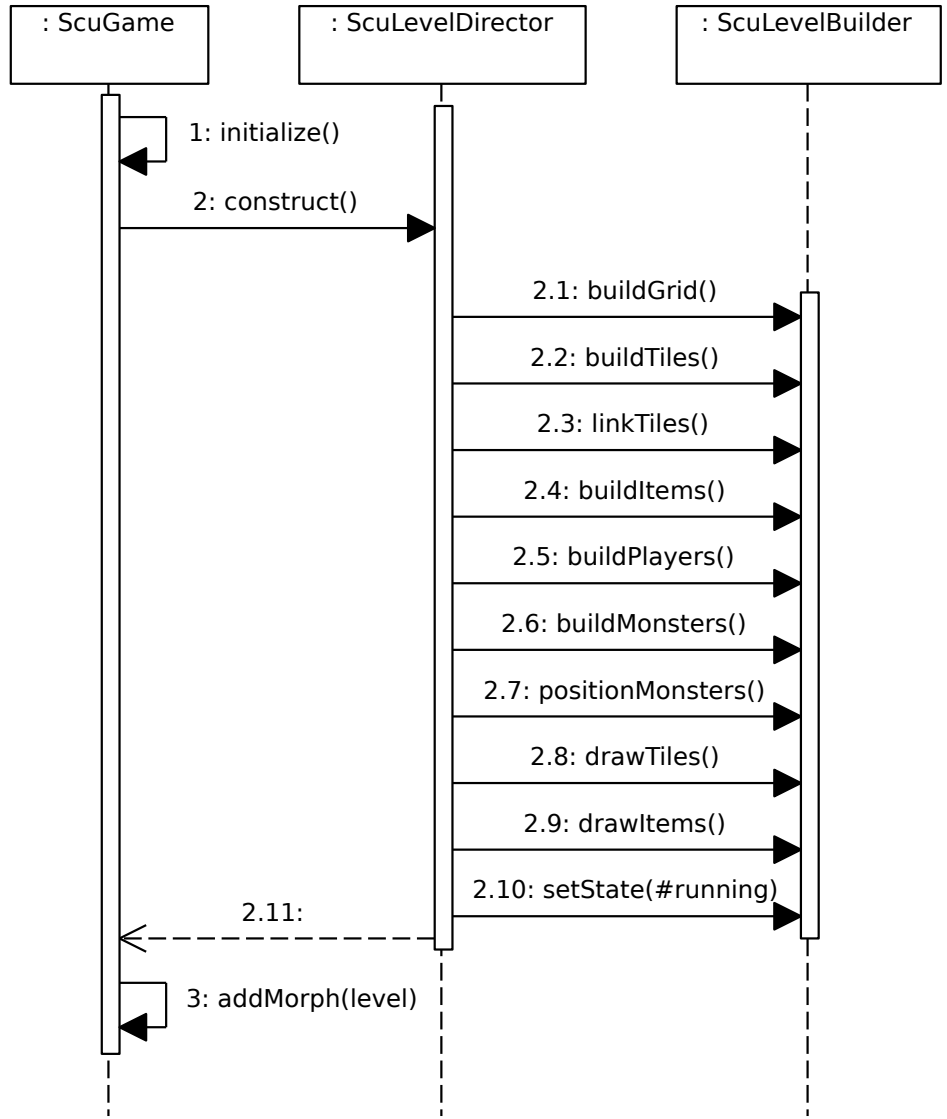


(b) Nachher: `ScuPlayer` hält nur Referenz auf Prototyp

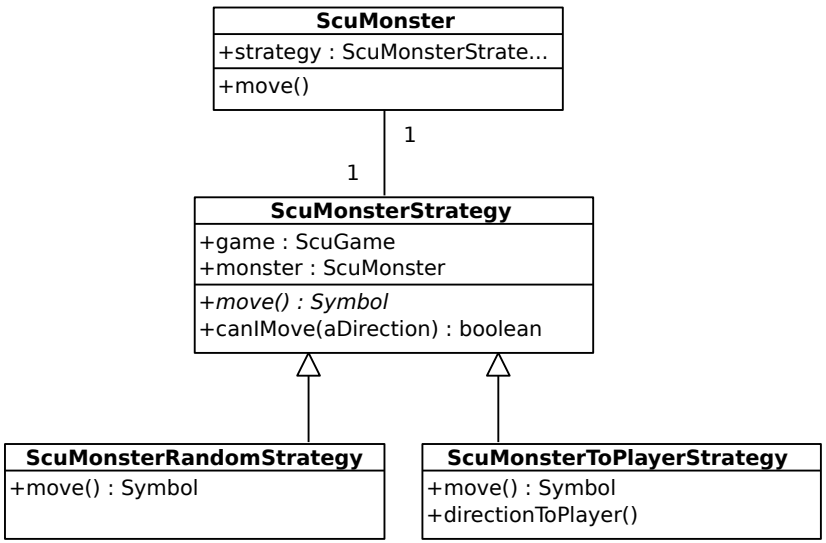
- `bucketPrototype` ist Prototyp für neue Buckets
- Kopieren des Prototypen mit `self bucketPrototype copy`
- **Vorteile:** Saubere Trennung von `ScuPlayer` und `ScuBucket`, `ScuPlayer` ist übersichtlicher

Levelbuilder

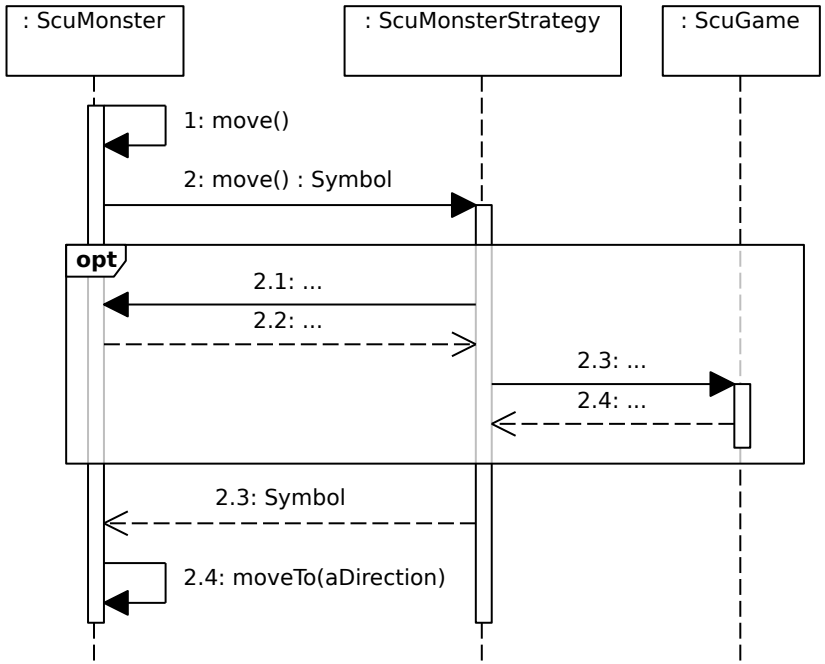




Monsterverhalten



(e) Monster Strategy Klassen



(f) Monster Strategy Interaction

- Verschiedene Monster unterscheiden sich nur im Verhalten
- Algorithmen zur Berechnung der Bewegungsrichtung sind in `ScuMonsterStrategy` Subclasses ausgelagert