Aufgabenbeschreibung
Softwareentwicklungspraktikum

Wintersemester 2008/2009
Universität München
Martin Lange und Ulrich Schöpp

Teil II – Architektur
Version 1

1 Allgemeine Architektur

Im ersten Teil der Aufgabenstellung wurde der Ablauf der Ameisenspiele beschrieben. In diesem Teil wollen wir nun die allgemeine Architektur des zu entwickelnden Programms festlegen.

Die Aufgabe beinhaltet die Entwicklung zweier unabhängiger Komponenten, einem Server und einem Client, die über eine RMI-Netzwerkverbindung miteinander kommunizieren können. Das Kommunikationsprotokoll ist dabei durch die Interfaces in dem Package `de.lmu.SEP0809.Communication` festgelegt, das von der Praktikumshomepage bezogen werden kann.

Dieses Package definiert Interfaces, die in diesem zweiten Teil der Aufgabenstellung näher beschrieben werden. Es werden weiterhin Anforderungen formuliert, welchen eine Implementierung dieser Interfaces durch Server und Client genügen muss.

1.1 Server

Der Server stellt über RMI einen Dienst zur Verfügung, mit dem sich mehrere Mitspieler mit ihren Client-Programmen verbinden können. Er erlaubt den Mitspielern, auf einer Anzahl von Spielbrettern ihre Ameisenstämme einzureichen und diese gegeneinander antreten zu lassen. Wie die Mitspieler das machen können ist durch die Schnittstellen in `de.lmu.SEP0809.Communication` geregelt, die der Server implementieren muss und die im Anschluss erklärt sind.

Der Server muss lediglich diese Schnittstellen implementieren, entsprechend Anforderungen in Abschnitt 4. Er muss keine graphische Benutzeroberfläche haben und kann lokal wie auch entfernt administriert werden.

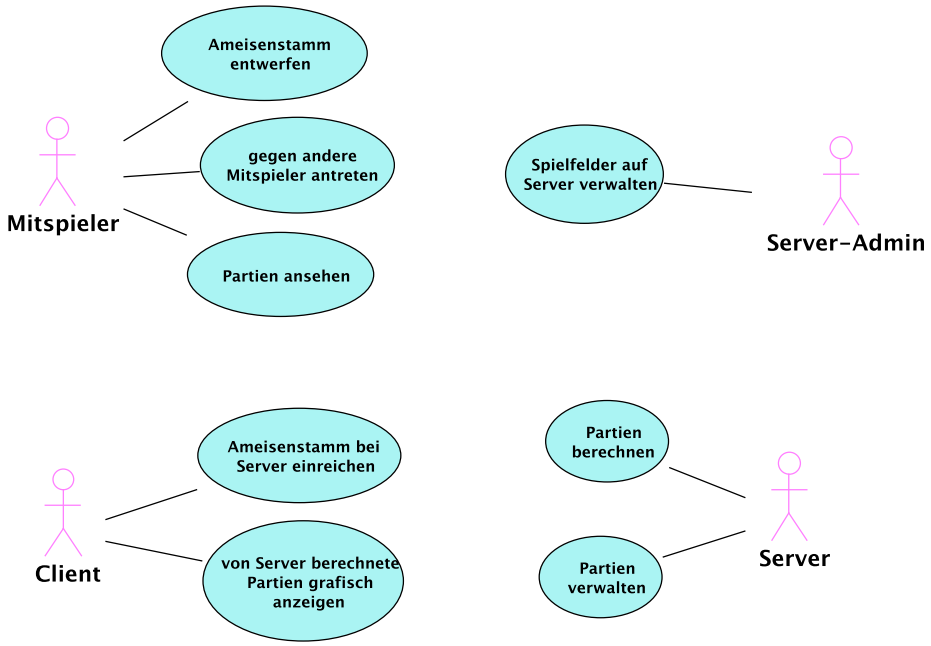

Abbildung 1: Use-Cases

1.2 Client

Die Clientprogramme müssen dazu in der Lage sein, sich mit jedem die Schnittstellen in `de.lmu.SEP0809.Communication` implementierenden Server zu verbinden. Sie müssen dort Ameisenstämme einreichen können und sie müssen in der Lage sein, den Ablauf von Spielpartien anzuzeigen.

Dazu müssen die Clientprogramme eine graphische Benutzeroberfläche haben. Sie sollten eine Benutzerschnittstelle bereitstellen, mit der es leicht möglich ist, Ameisenstämme einzureichen (und vielleicht auch zu editieren und ausprobieren) und sich Spielpartien vom Server geben zu lassen und anzuzeigen.

2 Kommunikationsschnittstellen

Die Kommunikation zwischen Client und Server ist durch die Interfaces in dem Package `de.lmu.SEP0809.Communication` geregelt. Das Klassendiagramm in Abbildung 2 gibt einen Überblick über dieses Package.

Das Hauptinterface des Servers ist `ServerInterface`. Jeder Server muss eine Implementierung dieser Klasse unter dem Namen „ServerInterface“ bei seiner RMI-Registry anmelden. Unter Benutzung dieser Referenz können sich dann eine Reihe von Clients mit dem Server verbinden.

Der Server stellt eine gewisse Anzahl von Spielarenen bereit (als Objekt vom Typ `Arena`). Eine Spielarena verwaltet die Einreichung von Ameisenstämmen auf einem einzigen

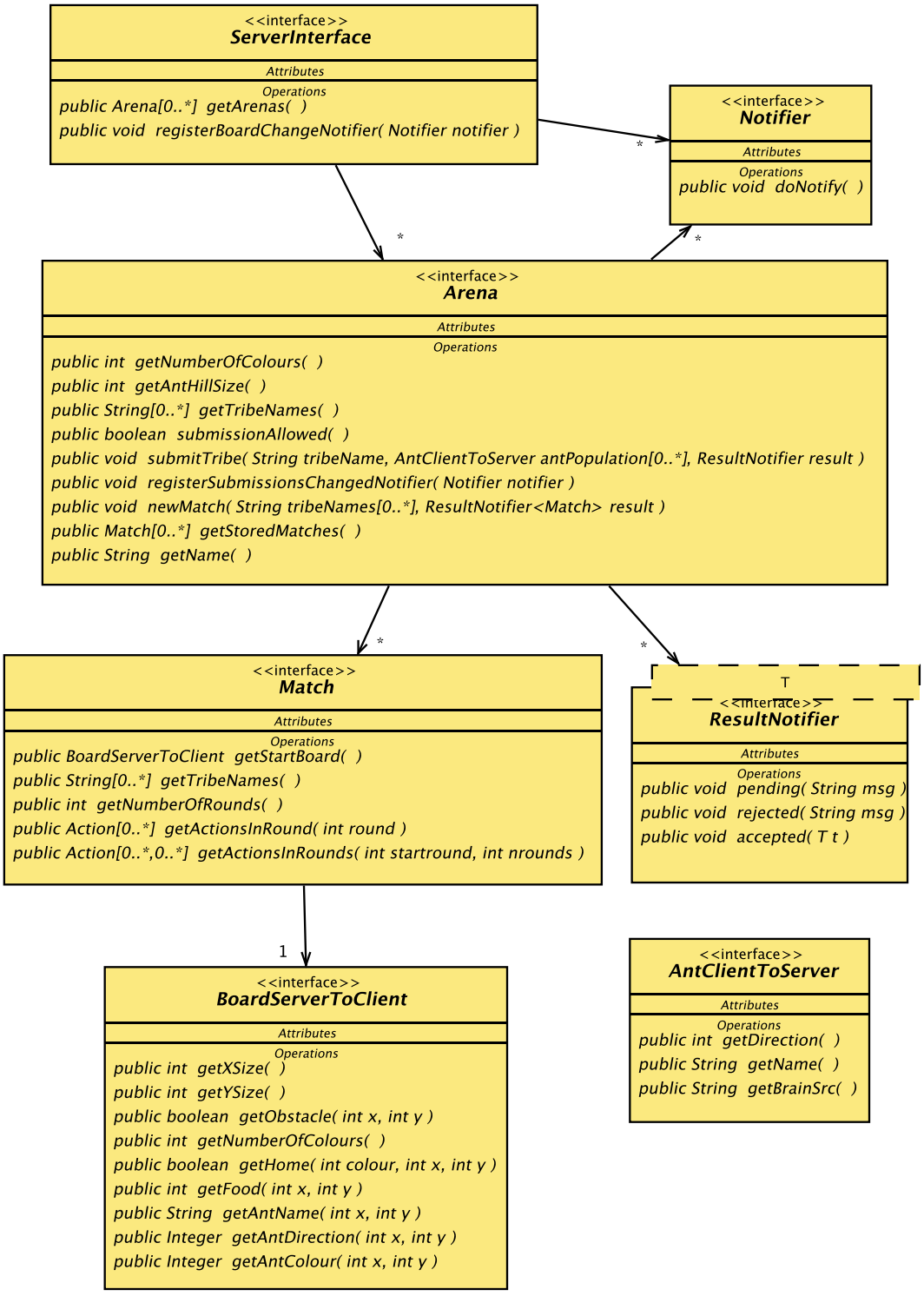

Abbildung 2: Klassendiagramm für das Package `de.lmu.SEP0809.Communication`

Spielfeld und wickelt die Spiele zwischen den verschiedenen Stämmen auf diesem Spielfeld ab. Die Client-Programme haben durch die Funktion `getArenas` vom Hauptinterface des Servers aus Zugriff auf die vom Server gespeicherten Arenen.

Hat ein Client erst einmal ein Objekt von Typ `Arena` ausgewählt, so kann er mittels dessen Funktionen Ameisenstämme einreichen sowie Spiele zwischen Ameisenstämmen ansehen (Typ `Match`). Im nächsten Abschnitt wird beschreiben, wie die Kommunikation zwischen Client und Server mit den gegebenen Interfaces ablaufen soll.

3 Benutzung des Server-Interfaces durch den Client

3.1 Einreichen eines Ameisenstamms

In Abbildung 3 ist der typische Verlauf der Einreichung eines Ameisenstammes dargestellt. Zunächst verbindet sich der Client mit dem Server, indem er sich eine Referenz auf dessen `ServerInterface` von der RMI Registry holt. Dann ruft er `getArenas` auf, um die Liste aller vom Server bereitgestellten Spielarenen zu erhalten. Daraus wählt er dann ein Objekt `arena:Arena` aus. Von diesem Objekt kann er dann Informationen über die Arena erfragen. In der Abbildung erfragt der Client den Namen der Arena und er stellt fest, dass neue Ameisenstämme darauf eingereicht werden können. Danach ruft der Client `submitTribe` auf, um seinen eigenen Ameisenstamm einzureichen. Wie Ameisenstämme kodiert sind, d.h. was das Argument `antPopulation` zu bedeuten hat, ist in der Javadoc-Dokumentation der Methode `submitTribe` erklärt.

Die Ergebnisrückgabe von der Funktion `submitTribe` wird über ein Objekt der Klasse `ResultNotifier` gehandhabt. Der Client erzeugt kurz vor dem Aufruf von `submitTribe` ein Objekt `result` vom Typ `ResultNotifier`. Dieses Objekt hat drei Funktionen `pending`, `rejected` und `accepted`. Der Client möchte, dass der Server `accepted` aufruft, wenn die Einreichung erfolgreich war. Der Server hat aber auch die Möglichkeit, `rejected` aufzurufen und die Einreichung so abzulehnen. Braucht der Server noch Bedenkzeit, so kann er dies dem Client mit einer Funktion `pending` mitteilen.

In dem Beispiel in Abbildung 3 sagt der Server zunächst, dass er noch etwas Bedenkzeit braucht, um die Einreichung dann zu akzeptieren. Nachdem die Einreichung angenommen wurde, hat das Objekt `result` seine Aufgabe erfüllt und wird vom Client zerstört.

In Abbildung 3 leben die beiden Objekte vom Typ `ServerInterface` und `Arena` in der JVM des Servers. Die anderen beiden Objekte befinden sich dagegen auf der JVM des Clients. Somit sind alle Methodenaufrufe in diesem Beispiel entfernte Methodenaufrufe, die durch RMI übertragen werden.

3.2 Ameisenstämme gegeneinander antreten lassen

Wurden in einer Arena bereits mehrere Ameisenstämme eingereicht, so kann ein Client diese gegeneinander antreten lassen und sich ansehen, wie sich die Ameisen dabei verhalten. Der

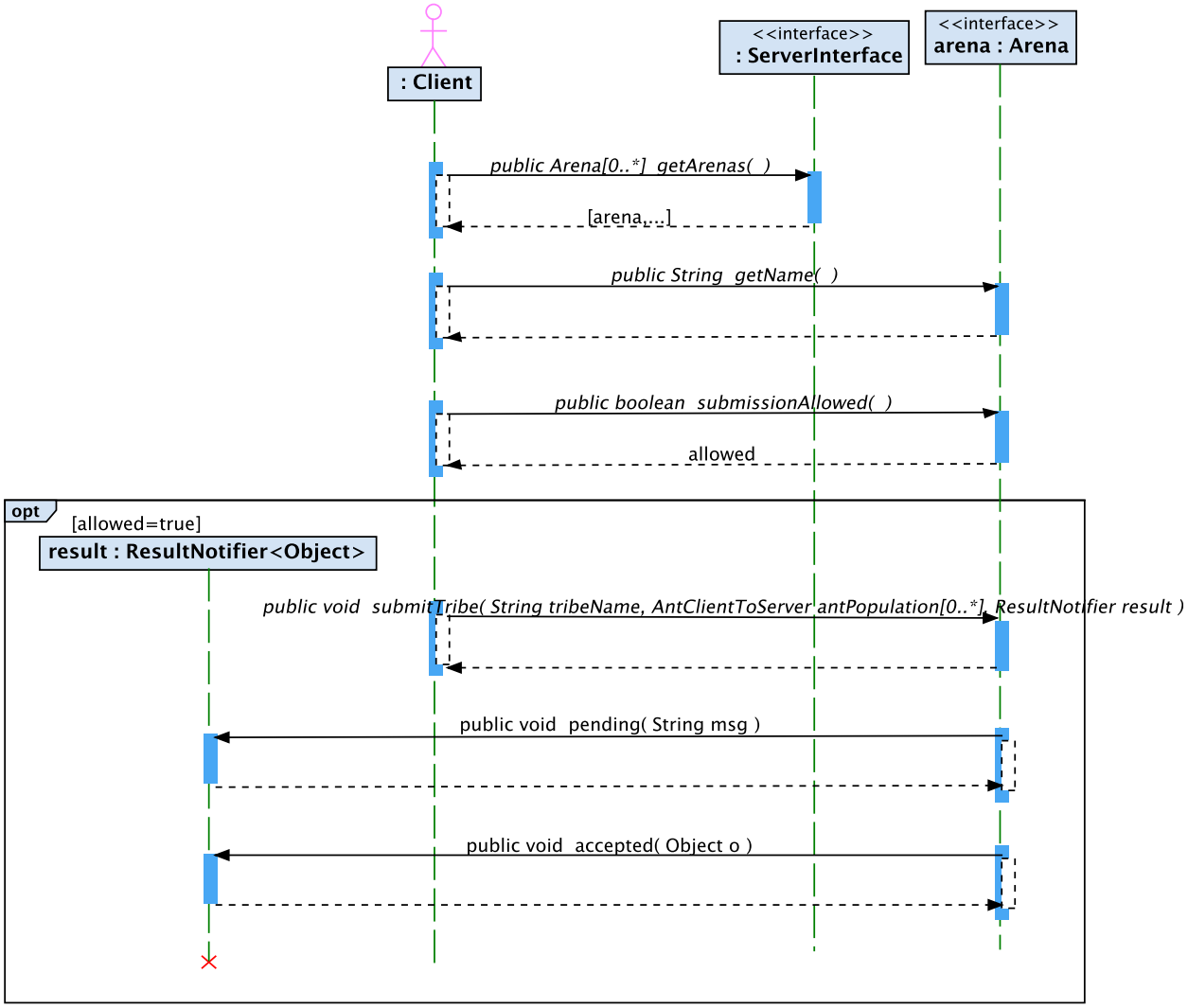

Abbildung 3: Beispielablauf für die Einreichung eines Ameisenstamms

Ablauf, wie ein Client eine solche Partie auf dem Server erzeugt und sie sich ansehen kann, ist in Abbildung 4 dargestellt.

In dieser Abbildung wird angenommen, dass der Client bereits eine Arena `arena` ausgewählt hat. Er erfragt dann zunächst, welche Ameisenstämme überhaupt schon in der Arena eingereicht wurden. Dazu ruft er die Funktion `getTribeNames` auf. Als nächstes erfragt er, wie viele Ameisenhaufen es denn überhaupt auf dem Spielfeld der Arena gibt, d.h. wie viele Ameisenstämme er gegeneinander antreten lassen kann. Dazu benutzt er die Funktion `getNumberOfColours`.

Mit diesen Informationen kann der Client eine Spielpartie von der Arena erfragen. Er schreibt sich die Liste der Namen der Ameisenstämme, die er in dieser Reihenfolge auf die Ameisenhaufen setzen will, auf und fordert mit der Funktion `newMatch` eine Spielpartie zwischen diesen Ameisenstämmen an.

Die Funktion `newMatch` liefert ihren Rückgabewert wieder mit einem `ResultNotifier` zurück. Diesen hat sich der Client vor dem Aufruf von `newMatch` erzeugt. In der Abbildung 4 akzeptiert der Server die Anfrage nach einer Spielpartie sofort. Er erzeugt ein Objekt `m:Match` und gibt es mit der `accept`-Funktion an den Client zurück.

Dieser kann nun das Objekt `m` nutzen, um sich die Spielpartie anzusehen. Zunächst erfragt er mit der Funktion `getStartBoard`, wie das Spielfeld am Anfang der Partie aussieht. Dieses Spielfeld wurde vom Server konstruiert, indem er die vom Client gewünschten Ameisenstämme auf die entsprechenden Ameisenhaufen gesetzt hat. Beachte, dass der Client das Spielfeld nur sehen kann, indem er sich ein `Match` vom Server holt. Da der Server Anfragen nach neuen Matches ablehnen oder mittels `pending` verzögern kann, kann er somit zum Beispiel erzwingen, dass sich niemand erst das Spielfeld ansieht und dann mit diesem Wissen einen Ameisenstamm einreicht.

Nachdem der Client das Startspielbrett erfragt hat, kann er nun erfragen, aus wie vielen Runden die Partie `m` besteht. Dann kann er die Aktionen der Ameisen in den einzelnen Spielrunden nacheinander erfragen und so den Ablauf des Spiels rekonstruieren. Die mögliche Aktionen der Ameisen sind in der Klasse `Match.Action` formuliert. Diese entsprechen den Aktionen im ersten Teil der Aufgabenbeschreibung.

Wie im vorangegangenen Abschnitt befindey sich das Objekt `arena` auf dem Server und die Objekte vom Typ `Client` und `ResultNotifier<Match>` befinden sich beim Client. Für die Position des Objekts `m:Match` gibt es mehrere sinnvolle Möglichkeiten. Dieses Objekt wird auf dem Server erzeugt, also wäre es eine Möglichkeit, dass es auch dort verbleibt. Möglich wäre jedoch auch, dass es vom Server serialisiert wird und dann zum Client übertragen wird.

3.3 Werterückgabe

Die Abläufe in Abbildungen 3 und 4 zeigen, wie Objekte vom Typ `ResultNotifier` benutzt werden, um Ergebniswerte vom Server an Clients zurückzugeben. Diese Objekte sollen von jedem Client so implementiert sein, dass sie beliebig viele `pending`-Meldungen annehmen können, bis letztendlich entweder die Methode `rejected` oder `accepted` aufgerufen wird. Wurde einmal eine dieser beiden Methoden aufgerufen, haben diese Objekte ihre Aufgabe

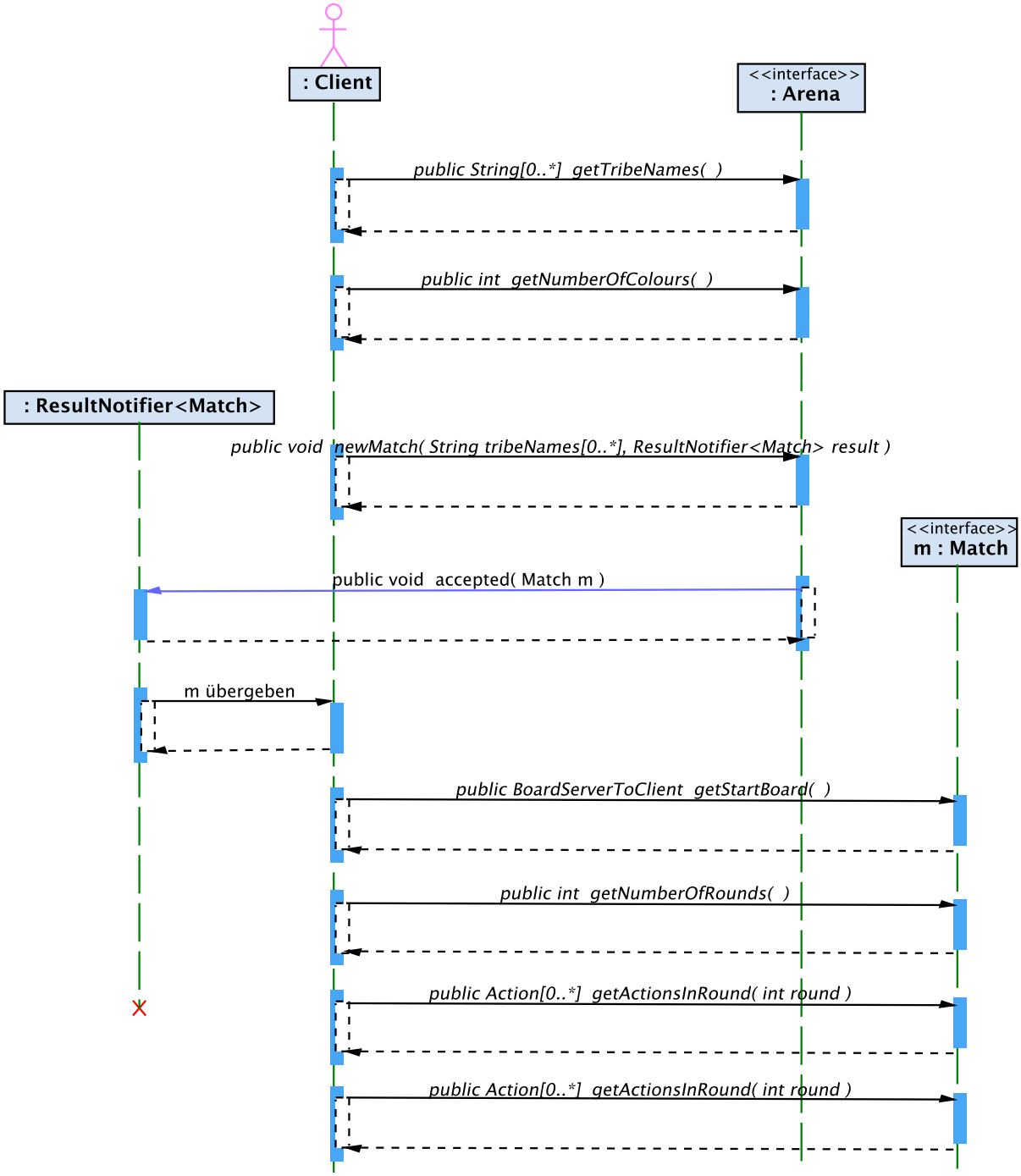

Abbildung 4: Beispielablauf für das Ansehen eines Spiels zwischen Ameisenstämmen

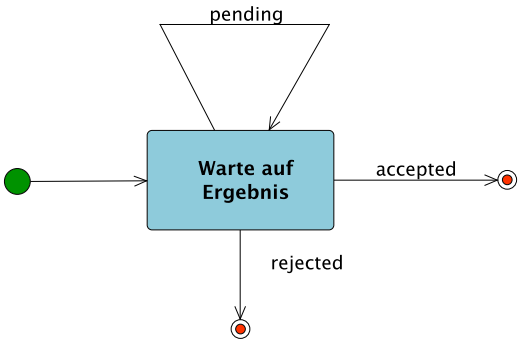

Abbildung 5: Zustandsdiagramm für `ResultNotifier`

erfüllt und können gelöscht werden. Sie müssen dann keine weiteren `pending`-Nachrichten annehmen.

Diese Anforderung ist im Zustandsdiagramm in Abbildung 5 dargestellt.

Die Klasse `ResultNotifier` ist parametrisiert. So kann zum Beispiel bei der Klasse `ResultNotifier<Match>` nach Annahme einer Anfrage ein Wert vom Typ `Match` mitgeliefert werden, siehe Abbildung 4. Dies ist dann gewissermaßen als Rückgabewert zu verstehen, also als Antwort des Servers auf eine Anfrage des Clients, bei der er einen `ResultNotifier` übergeben hat.

Wird `ResultNotifier` ohne Parameter benutzt, so darf ein beliebiger Wert (z.B. `null`) als Rückgabewert übergegeben werden.

3.4 Benachrichtigung über Zustandsänderungen im Server

Um über Änderungen auf dem Server informiert zu werden, können die Client-Programme Notifikationsobjekte beim Server registrieren. Dafür implementieren sie das Interface `Notifier` und übergeben eine Referenz auf ein Objekt von diesem Typ an den Server (mit einer der Funktionen `registerBoardChangeNotifier` und `registerSubmissionsChangedNotifier`). Ändert sich dann der Zustand im Server, so ruft er die Funktion `doNotify` im `Notifier`-Objekt des Clients auf.

4 Anforderungen an den Server

Insgesamt kann man den Server als Programm verstehen, das eine Liste von Arenen speichert über das RMI-Interface `ServerInterface` exportiert.

Der Zustand des Serverprogramms enthält neben der Liste der Arenen auch noch eine Liste von Referenzen auf `Notifier`-Objekte. Die `doNotify`-Methode in all diesen `Notifier`-Objekten soll vom Server immer dann aufgerufen werden, wenn sich die Liste der verfügbaren Arenen ändert.

Wie die Arenen auf den Server kommen und wie Arenen erzeugt und gelöscht werden,

ist nicht spezifiziert. Möglich wäre zum Beispiel, den Server mit einem Kommandozeileninterface oder einer GUI auszustatten. Es wäre aber auch möglich, diese Funktionalität in einen erweiterten Client zu integrieren.

4.1 Eine Arena

Eine Arena besteht aus einem festen Spielfeld und erfasst ein Protokoll zur Einreichung von Ameisenstämmen für dieses Spielfeld und zum Austragen von Spielpartien zwischen diesen.

Am Anfang ihrer Existenz ist eine jede Arena noch leer, d.h. es gibt darin noch keine Ameisenstämme und es gibt dann natürlich auch noch keine Spielpartien. Ameisenstämme werden erst im Laufe der Zeit von Mitspielern mit der Funktion `submitTribe` eingereicht und Spielpartien werden von ihnen mit der Funktion `newMatch` erzeugt.

Da der Wettkampf zwischen mehreren Mitspielern in einer Arena nur dann fair ist, wenn keiner der Mitspieler das Spielfeld gesehen hat, bevor er seinen Ameisenstamm einreicht, kann sich eine Arena in zwei Zuständen befinden. Im Anfangszustand ist es erlaubt, dass neue Ameisenstämme eingereicht werden. Die Funktion `submissionsAllowed` gibt dann `true` zurück. Um sicherzustellen, dass kein Mitspieler, der das Spielfeld schon gesehen hat, einen Ameisenstamm einreicht, kann die Arena in einen Zustand übergehen, in dem keine neuen Einreichungen mehr erlaubt sind. Dann gibt `submissionsAllowed` den Wert `false` zurück. Es ist dann nicht möglich, wieder in einen Zustand übergehen, in dem Einreichungen erlaubt sind.

Unter welchen Bedingungen der Übergang vom ersten zum zweiten Zustand stattfindet, ist nicht genau vorgegeben. Ein sinnvoller Zeitpunkt, um neue Einreichungen zu verbieten, wäre zum Beispiel wenn ein Mitspieler das Spielfeld gesehen hat, z.B. indem er einen Wert vom Typ `Match` für diese Arena erhalten hat. Beachte aber, dass der Server mit dem `ResultNotifier` in `newMatch` selbst festlegen kann, wann das erste Match auf der Arena an einen Mitspieler ausgegeben wird. So kann er zum Beispiel erzwingen, dass es erst eine gewisse Mindestanzahl von Einreichungen geben muss, bevor jemand eine Spielpartie sehen kann, indem er Anfragen nach einer neuen Partie solange mit `pending` beantwortet, bis die erforderliche Anzahl von Mitspielern sich registriert hat.

In einer Arena sind weiterhin noch folgende Daten gespeichert:

1. Eine Liste von erfolgreich mit `submitTribe` eingereichten Ameisenstämmen, wobei ein Ameisenstamm aus einem Paar von einem Namen und einer Liste von Werten des Typs `AntClientToServer` ist. Jeder Name darf nur einmal vorkommen und kann durch eine Neueinreichung nicht überschrieben werden.

2. Eine Menge von eingereichten Ameisenstämmen, über deren Akzeptanz noch nicht entschieden ist. Diese Menge besteht aus den Ameisenstämmen, die mit einem Funktionsaufruf `submitTribe(name,ants,result)` eingereicht wurden, für die aber noch nicht `result.rejected()` oder `result.accepted(...)` aufgerufen wurde.

   Für diese neu eingereichten Ameisenstämme gibt es folgende Forderungen:

- Wenn sich die Arena in dem Zustand befindet, in dem keinen Neueinreichungen mehr erlaubt sind, dann müssen diese neuen Ameisenstämme abgelehnt werden und `result.rejected` mit einer entsprechenden Nachricht aufgerufen werden.
- Wenn sich die Akzeptanz eines eingereichten Ameisenstammes voraussichtlich verzögern wird (z.B. weil erst noch ein Moderator darüber entscheiden muss), dann soll `result.pending` mit einer entsprechenden Nachricht aufgerufen werden.

Ansonsten kann es von der Serverimplementierung abhängen, welche Ameisenstämme angenommen werden. Es ist den Servern somit freigestellt, sinnvolle Richtlinien zu erzwingen, z.B. dass nur eine bestimmte Maximalanzahl von Ameisenstämmen eingereicht werden darf.

3. Eine Liste von gespeicherten Spielpartien. Diese Liste kann von dem Mitspielern mit der Funktion `getMatches` erfragt werden. Sie soll nur Spielpartien enthalten, die vorher durch einen Aufruf an `newMatch` erzeugt wurden. Es ist freigestellt, ob und welche Spielpartien gespeichert werden.

4. Eine Menge von Anfragen über neue Spielpartien, über deren Akzeptanz noch nicht entschieden ist, analog zum Fall für die Einreichung von Ameisenstämmen oben. Nach welchen Kriterien die Akzeptanz entschieden wird, ist nicht festgelegt und kann sich nach sinnvollen Kriterien, wie zum Beispiel dem Speicherverbrauch richten (aber der Server muss schon in der Lage sein, überhaupt Spielpartien zurückzugeben).

5. Schlussendlich speichert eine Arena noch eine Liste von `Notifiern`, deren `doNotify`-Funktionen aufgerufen werden, wenn sich die Liste der erfolgreich eingereichten Ameisenstämme ändert. Die Clients haben die Möglichkeit, Referenzen auf ihre `Notifier`-Objekte mit der Funktion `registerSubmissionsChangedNotifier` zu dieser Liste hinzuzufügen. Wenn der Aufruf zu einer `doNotify`-Funktion fehlschlägt, dann darf das entsprechende Objekt aus der Liste der `Notifier` gelöscht werden.

4.2 Ablauf von Partien auf dem Server

Die Hauptaufgabe des Server ist, den Ablauf von Spielpartien zwischen mehreren Ameisenstämmen zu berechnen. Ein Client kann die Erzeugung einer solchen Partie mit der Funktion `newMatch` erfragen.

Akzeptiert der Server die Anfrage eines Clients nach einer neuen Partie, so muss er diese Partie in Form eines Objekts `m` vom Typ `Match` bereitstellen. Das Spielfeld zu Anfang der Partie (nämlich `m.getStartBoard()`) entsteht dabei, indem die vom Client beim Aufruf von `newMatch` angegebenen Ameisenstämme auf die entsprechenden Ameisenhügel gesetzt werden. Auf welche Zellen in den Hügeln die Ameisen genau gesetzt werden, steht in der Javadoc-Dokumentation von `submitTribe`.

Das Objekt `m:Match` erlaubt nun, die Aktionen der Ameisen von dieser Startposition aus für eine gewisse Anzahl von Runden (nämlich `m.getNumberOfRounds()`) zu verfolgen.

Es ist dabei von der Implementierung des Interfaces `Match` abhängig, wann die Runden berechnet werden. Es wäre zum Beispiel möglich, gleich bei der Konstruktion des Objekts `m` die Aktionen in allen Runden zu berechnen und für die spätere Abfrage zu speichern. Andererseits wäre es aber auch möglich, außer dem Startspielbrett gar nichts zu speichern und erst bei Anfragen nach den Aktionen der Ameisen in einer bestimmten Runde diese dann zu berechnen. Natürlich sind zwischen diesen beiden Extremen auch Zwischenschritte möglich. Zum Beispiel könnte ein Objekt vom Typ `Match` die Spielfelder nach jeweils 1000 Spielrunden speichern; bei einer Anfrage nach den Aktionen in der 1012. Runde könnte es dann mit seiner Berechnung in der 1000. Runde anfangen und müsste nur noch zwölf Runden ausrechnen.

Welche Aktionen die Ameisen in jedem Schritt machen, ist durch die Programme in ihren Gehirnen gegeben. Die genaue Definition dieser Programmiersprache wird in einem weiteren Teil der Aufgabenstellung zu finden sein. Für den Moment kann man sich Ameisen noch beliebig verhalten lassen, zum Beispiel indem man die Ameisen einfach zufällige Aktionen machen lässt. Die Benutzung von zufälligen Aktionen ist gerechtfertigt, da die Programmiersprache ebenfalls auf zufällig erzeugte Zahlen zurückgreifen kann. Das Verhalten einer Ameise kann in jedem Schritt abhängen von ihrem (noch nicht beschriebenen) internen Gehirnzustand, von den zufällig erzeugten Zahlen, sowie dem Aussehen der Spielfeldzelle, auf der sie gerade steht, und dem Aussehen der Spielfeldzelle, in deren Richtung sie schaut. Ein Platzhalter für eine Ameise könnte folgendermaßen aussehen:

```
class Ant {
  private Random r;
  private final String name;

  public Ant(String name, String brainPrg) {
    this.name = name;
  }

  public Action step(Cell here, Cell ahead) {
    // benutze r um eine zufällige Aktion zurückzugeben
  }
  ...
}
```

Der nach außen hin sichtbare Zustand eines Objekts `m` vom Typ `Match` soll sich zu dessen Lebzeiten nicht ändern. Das heißt, die Funktion `getStartBoard` gibt immer ein gleich aussehendes Spielfeld zurück und mehrere Aufrufe zu den anderen Funktionen sollen immer das gleiche Ergebnis haben.

5 Erweiterungen

Es ist möglich, dass Server und Client erweiterte Interfaces implementieren, solange sie die offiziellen Interfaces ebenfalls unterstützen. Zum Beispiel könnte ein Server ein zweites Interface zur Serveradministration exportieren.

Die offiziellen Interfaces können, wie in jedem realistischen Softwareentwicklungsprojekt, ebenfalls noch Änderungen erfahren. Beispielsweise können bei Bedarf noch Erweiterungen zur Authentifizierung von Clients und zum Abfragen von Highscore-Listen aufgenommen werden. Auch können noch Funktionen zur Effizienzoptimierung hinzukommen.