

LUDII Programming Guide

Version 1.0

Cameron Browne Eric Piette

Department of Data Science and Knowledge Engineering (DKE)
Maastricht University

14 janvier 2019



European Research Council

Established by the European Commission

Abstract

This document describes the basic operation of the LUDII general game system and its game description grammar, and provides a number of guidelines for using the system effectively and for correctly adding code to the LUDII code base.

Please contact the authors with any comments or corrections at:
`{cameron.browne,eric.piette}@maastrichtuniversity.nl`

Contents

1	Overview	5
1.1	Basic Principles	5
1.2	Scope	6
1.3	Game Database	6
1.4	Ludemes	7
1.4.1	Mathematical Profile	9
1.5	Architecture	9
2	Common Module	10
2.1	Annotations	10
3	Library Module	11
3.1	API	11
3.2	Game State	12
3.2.1	The <code>State</code> Class	14
3.2.2	The <code>ItemStateContainer</code> Class	14
3.2.3	Supported Game Types	16
3.3	Graph Class	17
3.4	Ludeme Class Hierarchy	17
4	Grammar Module	19
4.1	Class Grammar	19
4.1.1	Syntax	19
4.1.2	Generation	20
4.1.3	Symbols	21
4.1.4	Symbol Return Types	21
4.1.5	<code>Function</code> and <code>Constant</code> Classes	22
4.2	Algorithm	23
4.3	Game Descriptions	24
4.3.1	Instantiation	24
4.3.2	Formatting Guidelines	25
4.3.3	Optional Parameters	25
4.3.4	Explicit Parameter Names	25
4.3.5	Default Values	26
4.3.6	Library Structure	26
4.3.7	Abstract Classes	26

4.3.8	Inner Classes	27
4.3.9	Collections	27
5	AI Module	28
5.1	Default AI Agents	28
5.1.1	Lightweight Local Features	28
6	Player Module	30
6.1	Graphical User Interface (GUI)	30
6.2	Command Line Interface (CLI)	33
7	Environment	34
7.1	Compatibility	34
7.2	Repository	34
7.3	Version Control	35
8	Coding Style	36
8.1	Philosophy	36
8.2	Coding Standard	37
8.3	Optimisations	44
9	Conclusion	45

1

Overview

The LUDII general game system is a software system for playing, evaluating, comparing, designing and optimising a wide range of games. LUDII is being developed as part of the ERC-funded *Digital Ludeme Project* and will be used to model the full range of traditional strategy games throughout recorded human history. It must therefore support a wide range of games and mechanisms, including nondeterministic elements and imperfect information (i.e. luck and hidden information) and geometries of varied style and complexity.

This document is intended for anyone who will be adding code to the LUDII code base. It outlines the basic principles behind LUDII, some design guidelines and constraints that should be kept in mind, and some notes on the preferred coding style to maintain consistency in the code base.

1.1 Basic Principles

LUDII is designed with the following principles in mind, in order of priority:

1. *Generality*: The system must support the full range of games required for the *Digital Ludeme Project*. These will be detailed shortly in Section 1.2. The system should be as *extensible* as possible with the capacity to easily add further functionality as required.
2. *Clarity*: Games should be described in the simplest, clearest way possible, such that the core game concepts and mechanisms are *encapsulated* in discrete *chunks* that can be easily manipulated – either automatically or by human designers – to optimise rule sets and even design new games. This encapsulation is achieved using a *ludemic* approach (see Section 1.4).
3. *Performance*: The system must be able to play out random simulations for any given game with sufficient speed to allow meaningful Monte Carlo analysis, in particular *Monte Carlo tree search* (MCTS) methods [10] which will be used for implementing default AI agents for playing the games. Performance is measured in terms of complete random

playouts of each on a single thread of a standard consumer machine (e.g. single *i7* core). As a rule of thumb, it is desirable to achieve playout rates of at least:

- 200,000/s for simple games,
 - 100,000/s for games of moderate complexity, and
 - 10,000 for games of any complexity.
4. Compactness: The system should be implemented to produce as small a memory footprint as possible. Third party libraries should be avoided; all generated bytecode should be derived from code contained within the LUDII project itself (with the exception of standard Java libraries). It is intended to port the LUDII system to mobile devices.

Some compromises have therefore been made in the design and implementation. For example, some standard software engineering principles have been violated in order to maximise the clarity of the game descriptions (described in Section 4), and some standard performance tweaks/hacks have been neglected for the sake of generality.

1.2 Scope

The *Digital Ludeme Project* deals with *traditional games of strategy*, i.e. games with no proprietary owner [2, p.5] that exist in the public domain,¹ and in which players succeed through mental rather than physical acumen.

This category includes most board games, some card games, some dice games, some tile games, etc., and may involve non-deterministic elements of chance or hidden information as long as strategic play is rewarded over random play. It excludes dexterity games, social games, sports, video games, etc.

It is difficult to even estimate the number of known traditional strategy games. For example, of the thousands of known Chess variants,² hundreds could fall under the umbrella of “traditional”. There also exist over 800 known variants of Mancala, let alone the undocumented ones [3].

1.3 Game Database

The aim is to model a representative sample of 1,000 of the world’s traditional strategy games in the LUDII Game Database, which will include the most influential examples throughout history. For each representative game, several variant rule sets may be stored. Chess and Mancala, for example, might be represented in the Game Database by say a dozen entries for the key variants, each of which might contain several variant rule sets themselves.

Since one of the aims of the *Digital Ludeme Project* is to optimise known rule sets and improve reconstructions where possible, then it might be necessary to test hundreds of rule variants per entry in order to find improved versions. The Game Database will therefore eventually contain hundreds of thousands of entries consisting of:

- the 1,000 representative games,
- their known variants, and

¹The more precise distinction between traditional games and those invented by known individuals and distributed by games companies [4] can lead to ambiguous cases [5].

²[http:// www.chessvariants.com](http://www.chessvariants.com)

- automatically generated reconstructions.

Further, it is anticipated that the LUDII system could be of particular interest to game designers, as a tool for quickly prototyping and testing game ideas with unprecedented convenience and rigour. I therefore envisage a secondary database of modern or non-traditional games that will act as a sandbox for game designers, which may eventually contain hundreds or thousands of entries itself (plus automatically generated variants).

Each database (see Figure 1.1) entry should include the following information:

1. *Description*: Plain text description of the game, including its equipment and rules, in the LUDII class grammar (explained in Section 4) in the form of LISP-like *s-expressions* or symbolic expressions. Each game description should include a **metadata** section with details such as the game’s designer and date of creation (if known), help text for playing the game, “About” text describing the game and its context, GUI “hints” for displaying the game optimally, etc.
2. *Executable*: Java bytecode for playing the game according to the LUDII API, compiled directly from the above game description.
3. *AI Hints*: Plain text description of features relevant to the game for biasing MCTS playouts (format to be decided), plus any other “hints” to help the AI achieve the desired level of play. See Section 5 for details.
4. *Cultural Profile*: Plain text description of historical and cultural information relevant to the game, such as where and when it was known (or thought) to have been played, demographics of players, known precedents and antecedents, etc.

1.4 Ludemes

Games are modelled as structures of *ludemes*, i.e. game memes or conceptual units of game-related information [6]. These constitute a game’s underlying building blocks, and are the high-level conceptual terms that human designers use to understand and describe games.

For example, Tic-Tac-Toe might be described in *ludemic form* as follows:

```

❁ Ludeme 1
(game "Tic-Tac-Toe"
 (play {(player "P1") (player "P2")} Alternating)
 (equipment (board "Board" (square 3)))
 (rules
  (moves (to (indexOf Mover) (empty)))
  (end (line Mover Any 3) (result Mover Win)))
 )
 )

```

An important benefit of the ludemic approach is that it encapsulates key game concepts and gives them meaningful labels. This allows the automatic description of game rule sets, allows comparisons between games, and potentially allows the automated explanation of learnt strategies in human-comprehensible terms. **Each ludeme corresponds to a Java class in the Ludeme Library.**

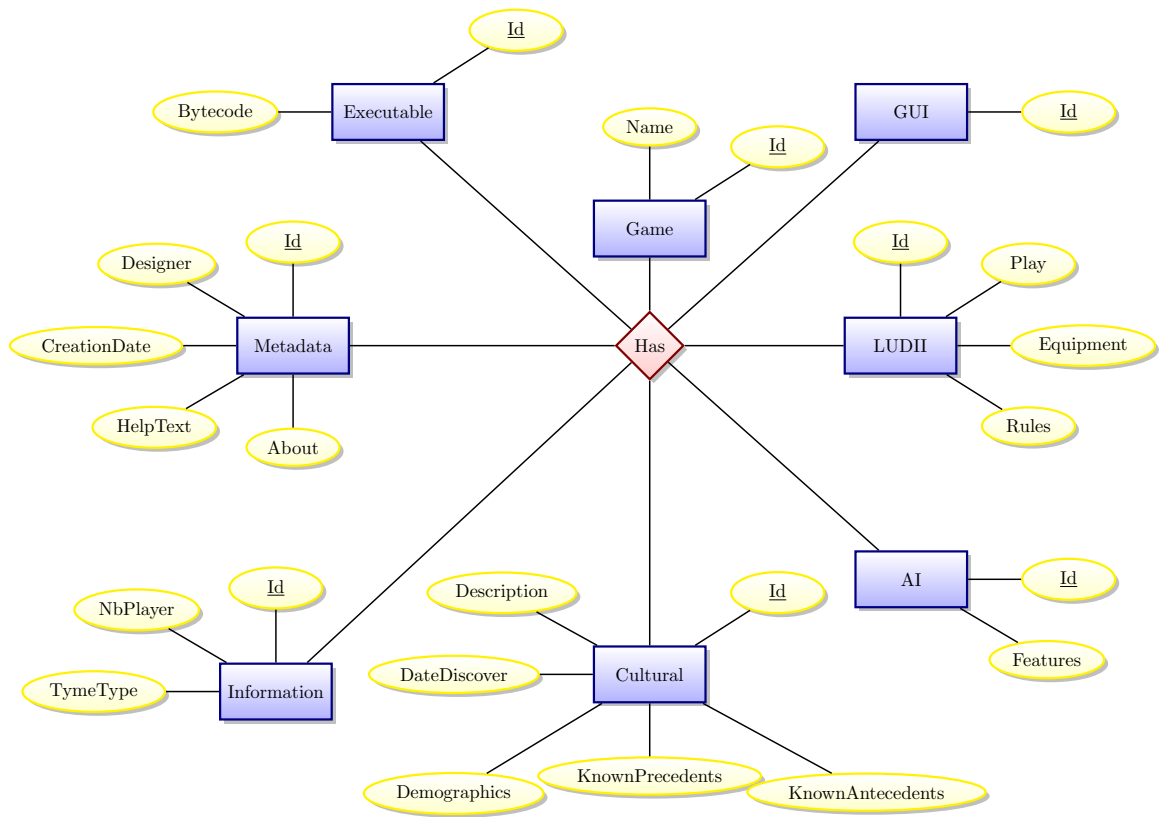


Figure 1.1: The Entity-Relationship diagram.

1.4.1 Mathematical Profile

Each ludeme will also be tagged with keywords indicated the basic mathematical principles that it embodies. A *mathematical profile* can then be automatically derived for each game based on its component ludemes. The taxonomy of mathematical principles and associated keywords, and the exact tagging mechanism, are yet to be decided.

1.5 Architecture

The LUDII project is implemented in Java in the following modules (i.e. sub-projects):

- *Common*: Repository for common project-wide variables, constants, annotations and methods shared by all modules.
 - *Library*: The core Ludeme Library that contains all ludemes in a structured class hierarchy. Games in the Game Database are defined in terms of these ludemes.
 - *Grammar*: The mechanism for deriving the LUDII class grammar directly from the Ludeme Library.
 - *AI*: Collection of default AI agents for playing the games in the Game Database.
 - *Player*: Main controller for playing games, including graphical (GUI) and command line (CLI) interfaces.
-

2

Common Module

The Common module contains common project-wide variables, constants, annotations and methods shared by all modules. The `Global` class contains global constants such as `MaxPlayers`, `MaxDimensions`, `MaxPieceCount`, etc. The `BitTwiddling` class provides a number of methods for low-level bitwise operations that can be useful for optimising playouts.

Each player within a given game is assigned a unique consecutive index starting at 1, corresponding to the constants `P1`, `P2`, `P3`, etc. up to N players. These constants are used to identify players and do not necessarily define the play order within a game. For example, in a match of paired games, `P1` might start the first game of the match while `P2` starts the second game of the match.

2.1 Annotations

The following Java annotations are implemented for LUDII:

- **Anon**: Used to anonymise (i.e. hide the name of) selected class constructor parameters in the generated LUDII class grammar.
- **Name**: Used to force the name of selected class constructor parameters to be shown in the generated LUDII class grammar. This is the opposite of the **Anon** annotation.
- **Opt**: Used to specify that selected class constructor parameters are [optional] in the generated LUDII class grammar.

Their use will be described in more detail in Section 4.

3

Library Module

The core Ludeme Library that contains all ludemes in a structured class hierarchy. Additional support classes, such as API, may also be included.

3.1 API

The root `Game` class implements the following minimal API. All compiled games must support this API:



Java 1

```
public void create ();  
public void start (Context context);  
public List<Action> actions (Context context);  
public void apply (Context context, Action action);  
public Status playout (Context context, List<AI> ais);  
public Controller getController (int resolution);  
public View getView (int resolution);
```

Where:

- Each `Context` object contains a reference to the relevant `Game` object and its static final data members (equipment, players, rules, etc.) and a reference to the current `Trial`.
- Each `Trial` object is a record of a complete game played from start to end, including the moves made and hash values of intermediate states if needed.
- Each `Action` object describes one or more atomic actions to be applied to the game state to effect a move. Actions typically include removing components from containers, adding components to containers, changing component counts or states within containers, deciding which player moves next, etc.

- Each AI object describes the AI implementation chosen for each player, including computational budget/time limits, hints such as features for biasing payouts, etc.
- The `Controller` object provides the mechanism for updating the game state based on user input such as mouse clicks.
- The `View` object provides the mechanism for showing the current game state on the screen.

The API class (see Figure 3.1) is a support class and not a ludeme. It is included at this level in the class hierarchy (i.e. below the root `Game` class) as **additional domain types beyond games may be added to the system in future**. The user defines games in the LUDII class grammar but executes them through the API. This API decouples the grammar from its implementation from the user's perspective.

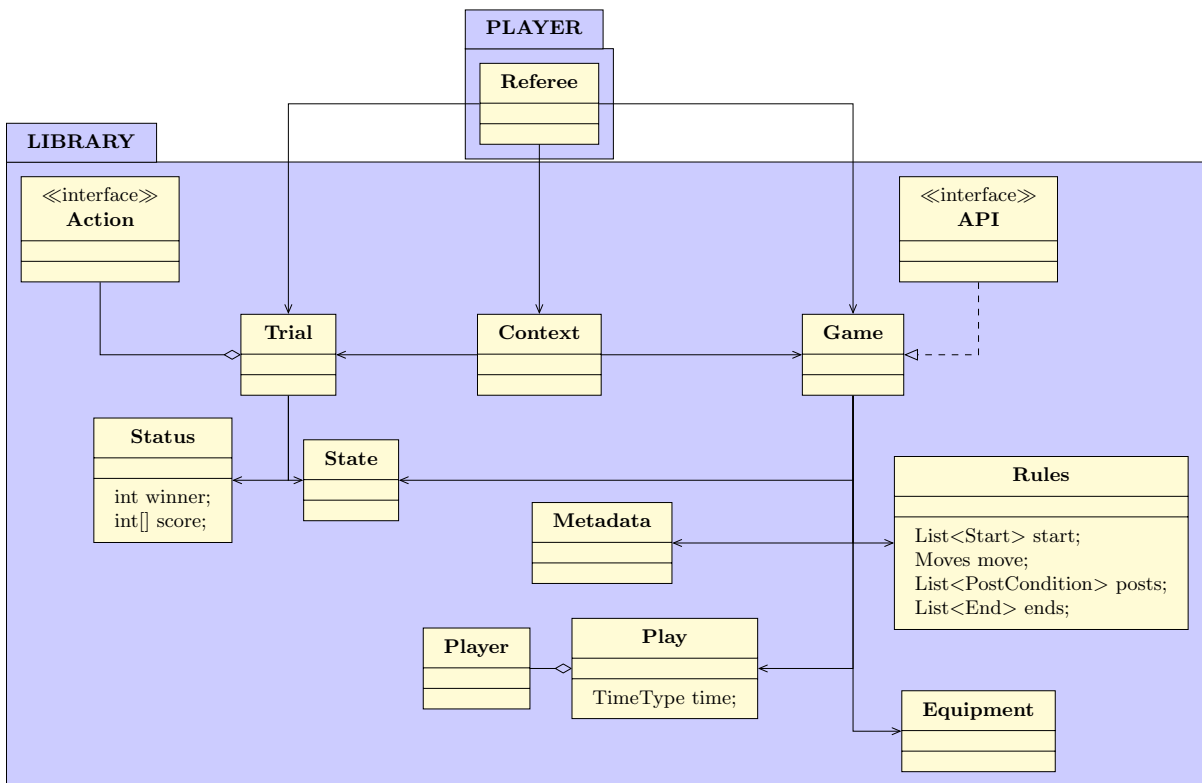


Figure 3.1: The general diagram of the API.

3.2 Game State

Games (see Figure 3.1) are described in terms of:

- *Play*: Number of players and type of game (alternating, discrete, realtime, etc.).
- *Equipment*: Equipment used for playing the game.

- *Rules*: Rules for playing the game, including rules for: start, moves and end.

The equipment (see Figure 3.2) is defined according to a *component/container* model, where:

- A **Component** is an atomic item of equipment, such as a playing piece, dice, tile, card, etc. Components may have specified constant properties (owner, size, value, etc.) in addition to dynamic properties that may vary throughout each trial (state, direction, etc.).
- A **Container** holds one or more components, such as a board, rack, hand, cup, bowl, pool, etc.¹ Each **Container** has an associated **Graph** object that defines playable **Sites** at which **Components** can be placed, and adjacencies between them (see Section 3.3).

All equipment implements the **Drawable** interface, which means that each item of equipment must be able to draw a default bitmap image for itself at a given resolution, for displaying the board state on the screen. **Containers** must be able to draw their current **Components** at the appropriate positions, orientations, states, etc.

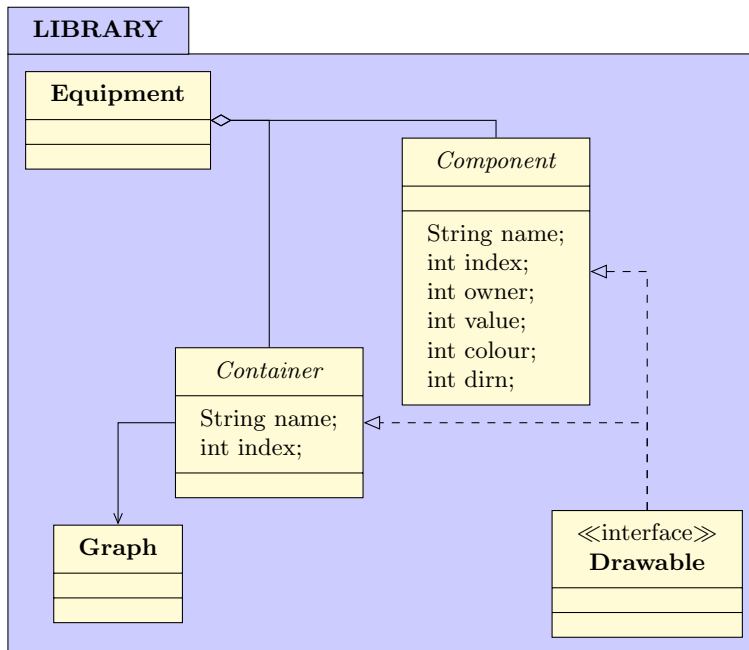



Figure 3.2: The general diagram of the equipment.

The rules of a game are divided in four types:

- **Start** rules: applied at the beginning of a game.
- **Moves** rules: defined all the legal moves in the current state.
- **PostCondition** rules: applied after each move played.
- **End** rules: used to know if the state is terminal or not.


¹Containers may also recursively hold sub-containers with independent state information (sub-state, sub-direction, etc.) although it is not clear if this degree of functionality is required for traditional games. The only game I know that would benefit from such a mechanism is the non-traditional game [Pentago](#).

All these rules are applied in the following method in the `Game` class:

```
 Java 2  
public void apply(final Context context, final Action action);
```

3.2.1 The State Class

The game state is defined by the following class:


```
 Java 3  
public final class State  
{  
    private int info;  
    private final ItemState [] itemState;  
}
```

Where:

- The variable `info` encodes general state information including the player indices of the current, next and previous movers, and which players are still active (useful for multi-player games).
- The list of `ItemState` objects describes the state for each piece of equipment. For the moment, components are stateless entities; only containers contain further state information.

3.2.2 The ItemStateContainer Class

Each container has an associated state defined as follows:

```
 Java 4  
public abstract class ItemStateContainer extends ItemState  
{  
    protected final ChunkSet what;  
    protected final ChunkSet who;  
    protected final ChunkSet count;  
    protected final ChunkSet state;  
    private final Region empty;  
}
```

Where:


- `ChunkSet` is a custom bitset class extended from the standard `java.util.BitSet` for added functionality. The member variable:
 - `ChunkSet what` specifies the item index of the component at each site (0 if empty).
 - `ChunkSet who` specifies which player owns each site (0 if none).
 - `ChunkSet count` specifies how many identical components are on at each site (null if not needed).

- `ChunkSet state` specifies the state of the component at each site (null if not needed). Component states might include direction, side, promotion status, etc.


Note that constants used in the Ludeme Library should generally be positive, to avoid sign issues when storing their values as chunks in `ChunkSet` objects.

- `Sites` specifies a set of playable sites within a container. The member variable `empty` maintains the list of currently empty sites in the this container, for fast empty cell checking.²


The `ChunkSet` class extends the standard `java.util.BitSet` by subdividing its bits into regularly sized *chunks* that encode positive integer values in compact form. An additional constructor is provided to define the chunk size and number of chunks (typically the number of playable sites in the container):

```
 Java 5  
public ChunkSet(final int chunkSize, final int numChunks) {...}
```

Note that `chunkSize` should be a power of 2 between 1 and 64, in order to avoid chunks straddling long boundaries and thus reducing performance. Chunks are accessed and set as follows:

```
 Java 6  
public int getChunk(final int chunk) {...}  
public void setChunk(final int chunk, final int value) {...}
```

The main reason for using the custom `ChunkSet` class is for its fast feature-matching capabilities. Lightweight local features (described in Section 5.1.1 can be used to bias MCTS playouts with negligible performance impact using the following function:

```
 Java 7  
public boolean matches(final ChunkSet mask, final ChunkSet pattern)  
{  
    if (wordsInUse < mask.wordsInUse)  
        return false;  
    for (int n = 0; n < wordsInUse; n++)  
        if ((words[n] & mask.words[n]) != pattern.words[n])  
            return false;  
    return true;  
}
```

`ChunkSet` also supports left and right shifts, which the standard `java.util.BitSet` class does not.

²While a `BitSet` could be used to indicate empty cells within a container, the `Sites` class was implemented to allow element removal. However, `BitSet` might prove a better choice when other operations such as `union` are added.

3.2.3 Supported Game Types

The `ItemStateContainer` class is intended to support a wide range of games, including games for which the following information is required per site:

- Owner of site (e.g. Tic-Tac-Toe).
- Index of component (e.g. Chess).
- Owner of site and index of component (e.g. Conhex, in which pieces are placed on the board to claim regions).
- Component count (e.g. Mancala).
- Component state (e.g. Stratego).

That's why the LUDII system provides some different classes for game type extends from the `ItemStateContainer`:

- `ItemStateContainerCount`: Game with one component by player but more than one component by site.
- `ItemStateContainerIndex`: Game with more than one component by player.
- `ItemStateContainerIndexCount`: Game with more than one component by player and more than one component by site.
- `ItemStateContainerIndexCountState`: Game with more than one component by player, more than one component by site and many local states for each component.
- `ItemStateContainerIndexState`: Game with more than one component by player and many local states for each component.
- `ItemStateContainerPlayer`: game with one component by player.
- `ItemStateContainerState`: game with one component by player but many local states for each component.

The LUDII system provides three others special game states:

- Stacking games are a special case because a complete layer of sites for each stacked level, as required. That's why a specific `ItemStateContainerStacking` class is implemented, for handling stack-related operations such as querying who owns the stack, moving the stack as a single entity, etc.
- Boardless games whose containers grow iteratively as components are placed, such as Domino games, are handled by adding `Sites` to the container as needed and updating the associated `Graph` object with necessary adjacency information. To do that two more `ChunkSet` are used in `ItemStateContainerBoardless` abstract class:
 - `ChunkSet playable` denotes the sites playable in the current state
 - `ChunkSet occupied` specifies the sites occupied in the current state
- Games with hidden information requires to define if a site is visible or hidden to each player in any state. That's why a `ChunkSet hidden` is defined in the `ItemStateContainerHidden`.

Each container chooses and instantiates the appropriate `ItemState` type at run-time, when the `Game` object is created. This is done in the Constructor of the `State` class:



Java 8

```
public State(final Game game);
```

This class uses some others methods define in the Game class:

- `boolean requiresItemIndices()`: True if more than one component is owned by a player.
- `boolean requiresStack()`: True if one ludeme needs a stack.
- `boolean requiresCount()`: True if one ludeme needs a counter.
- `boolean requiresState()`: True if a component has more than one local state.
- `boolean requiresHiddenInformation()`: True if one ludeme needs some hidden information
- `boolean isBoardless()`: True if one of the container is boardless.

3.3 Graph Class

Each container in a game definition has an associated **Graph** class, which defines:

- A list of **Cell** objects that define playable *sites* at which components (and possibly nested containers) can be placed. Sites are labelled 0, 1, 2, 3, . . . within each container. A **Loc** is a record of a *location* defined by container index and site index (and optional level number). The average cell size is used to initialise the graphics for associated components.
- A *dual* of the graph is maintained, where each **Vertex** defines a cell centre and each **Edge** defines adjacency between two cells. Adjacencies are stored both by: 1) cell index, and 2) cardinal direction from each cell (N, E, W, S, NE, SE, NW, SW, U (up), D (down), UN, UE, US, UW, etc.).

For example, Figure 3.3 shows a game with a single container (the board, blue) and its dual (grey) revealing cell adjacencies.

Note that the game graph may be modified for some games, such as “graph games” in which player moves involve operations on the graph (e.g. adding or cutting edges or vertices), or for boardless games such as Dominoes in which an implied board incrementally grows as pieces are placed. For such games, it may be necessary to include a local copy of the graph state in the container’s game state object, to be modified as appropriate.

Some different information are pre-generating and storing in the **Graph** class (corners of the board, exterior vertices, top of the board, ect.) and in the **Vertex** class (adjacent vertices, neighbours in each possible direction, the turtle positions, ect.)

3.4 Ludeme Class Hierarchy

The following directory tree shows the key Java packages (i.e. folders) in the Ludeme Library:

```
game
├── metadata
├── play
│   └── model
```

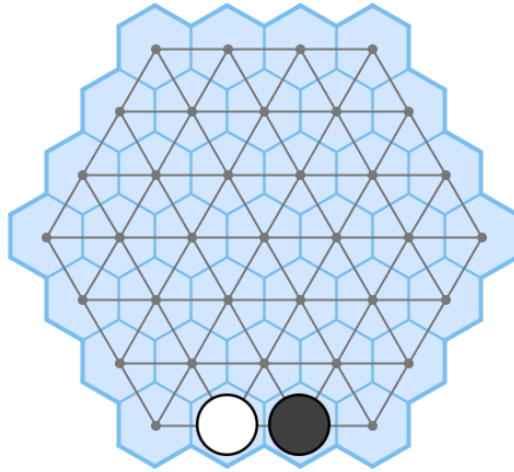
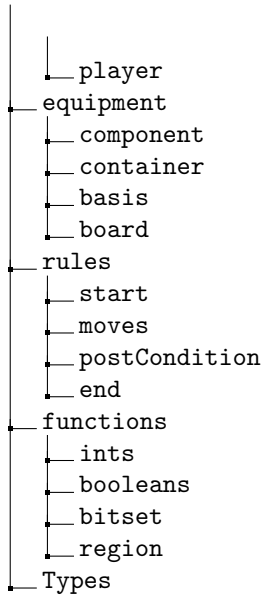


Figure 3.3: A board container and the dual of its graph (which is itself a graph).



The actual Java class for each ludeme should be located in the relevant package in this hierarchy. Appendix A shows the grammar generated from this class hierarchy.

4

Grammar Module

The Grammar module provides the mechanism for deriving the LUDII class grammar directly from the Ludeme Library. The mechanism is an improvement of the version described in an earlier paper [7] and involves two basic steps:


1. *Forward Step*: Generate the grammar from the classes Ludeme Library.
2. *Backward Step*: Compile game descriptions into executable bytecode by instantiating the relevant classes in the Ludeme Library.

4.1 Class Grammar

The class grammar is set of *production rules* in which sequences of *symbols* on the RHS are assigned to a *nonterminal symbol* on the LHS, very much like an Extended Backus-Naur Form (EBNF) grammar. It is intrinsically bound to the underlying code library, but is a *context-free grammar* that is self-contained and can be used without knowledge of the underlying code.

4.1.1 Syntax

The basic syntax of the class grammar is as follows:

```
 Grammar 1  
<class> ::= { (class [{{<arg>}}]) | <subClass> | terminal }
```

where:

- <class> denotes a LHS symbol that maps to a class in the code library (i.e. ludeme).
- (class [{{<arg>}}]) denotes a `class` constructor and its arguments.
- <subClass> denotes a subclass derived from `class`.
- terminal denotes a terminal symbol (fundamental data type or `enum`).

[...]	denotes an optional item.
{...}	denotes a collection of one or more items.
	denotes a choice between options in the RHS sequence.


Class names typically start with an uppercase character, but are converted to *lowerCamel-Case* in the grammar for readability, convenience, and in keeping with the traditional form of EBNF style grammars. Appendix A shows the grammar generated from the Ludeme Library summarised in the previous section.

4.1.2 Generation

The forward step of converting source code to grammar involves recursively parsing the code library from a specified root class (`Game` in this case) downwards, storing a new symbol for each new class encountered. A chain of dependency is then created from the root class, linking the arguments of each visited constructor by data type, until terminal symbols are reached. Fundamental data types and `enums` constitute terminals, while all other user-defined classes constitute *non-terminals*.

The grammar is generated with each class name forming the LHS symbol of a production rule, whose RHS is a sequence of constructors that instantiate that class (or subclasses derived from it) and their parameters. For example, the following abstract base class:


```

 Java 9
public abstract class Start {...};

```


and its two derived subclasses:

```

 Java 10
public class Place extends Start
{
    public Place(final String what, final int where)
}

```


```

 Java 11
public class Store extends Start
{
    public Store(final int who, final String what, final int count)
}

```

generate the following production rules:

```

 Grammar 2
<start> ::= <place> | <store>
<place> ::= (place (what String) (where int))
<store> ::= (store (who int) (what String) (count int))

```

The resulting grammar is a summary of the class hierarchy, based on constructors and parameters, that offers full functionality while hiding the implementation details. The program-

ming language (Java) effectively becomes the game description language; it should theoretically be possible to add to the system almost any functionality that can be defined in Java.

4.1.3 Symbols


Each symbol in the grammar is one of the following `SymbolTypes`:

- **Primitive:** Primitive Java data type (int, boolean, etc.)
- **Predefined:** Predefined data types from standard Java libraries (`java.lang.String`, `java.util.Bitset`, etc.) and custom data types used by the Ludeme Library (`State`, `Action`, etc.).
- **Constant:** Values of `enum` types defined within ludeme classes.
- **Class:** User-defined ludeme classes.


Each symbol object has a `boolean isList` member variables indicating whether that symbol actually refers to a list of symbol objects of that type, and an `int nesting` member variable indicating whether the symbol actually refers to an array of symbols and the number of array dimensions (default value `nesting = 0` indicates no array). We can therefore distinguish between the symbols `<action>` and `<list<action>>` even though they both refer to the same base object type.

4.1.4 Symbol Return Types

Each class returns an object of its own type through its constructor by default. However, it is desirable to allow symbols to specify a return type apart from themselves, in order to allow ludemes to be chained into complex structures in a clear way. For example, it is convenient to specify that the `Or` class takes two `boolean` arguments and returns a `boolean` result, as follows:

```
 Grammar 3  
<boolean> ::= boolean | <or>  
<or>      ::= (or <boolean> <boolean>)
```

Symbol return types can be overridden by defining an optional `eval(final Context context)` method for that class. For example, the `Or` class described above implements the `BooleanFunction` interface which defines `eval(final Context context)` as having a `boolean` return type:

```
 Java 12  
public interface BooleanFunction  
{  
    public boolean eval(final Context context);  
}
```

Note that **every non-equipment ludeme must define an `eval(final Context context)` method**. This is because the base `Game` class executes the various game-related functions defined in the API by calling `eval()` on the appropriate ludeme classes. For example, the following method in `Game` generates the set of legal actions for the current state according to the game's movement rules:



Java 13

```
public List<Action> actions(final Context context)
{
    return rules.moves().eval(context);
}
```

Ludemes describing game rules (as opposed to equipment) do not need to override their return types, so must implement the following interface to ensure that an `eval(final Context context)` method exists but has a `void` return type:



Java 14

```
public interface Rule
{
    public void eval(final Context context);
}
```

4.1.5 Function and Constant Classes

It is desirable for certain constructor arguments to allow both primitive data types and functions that return those types. For example, any integer function that returns an `int` can be used for any `<int>` argument:



Grammar 4

```
<int> ::= int | <add>
<add> ::= (add <int> <int>)
<indexOf> ::= (indexOf <String>) | (indexOf <roleType>)
```

This allows more complex and interesting mechanisms to be defined in the grammar by chaining functions together. For example, the following rule describes a winning condition if the current mover completes a line of length 3:



Ludeme 2

```
(line Mover Any (length 3))
```

whereas the following rule describes a winning condition if the current mover completes a line whose length is their player index plus 2:



Ludeme 3

```
(line Mover Any (length (add (indexOf Mover) 2)))
```

This interchangeable mixing of primitive data types and functions with the appropriate return type is achieved by defining custom `Function` and `Constant` class types for each such return type. For example, the following `IntFunction` and `IntConstant` classes allow the interchangeable `int` / `<int>` mechanism described above:



Java 15

```
public interface IntFunction
{
    public int eval(final Context context);
}

public final class IntConstant implements IntFunction
{
    protected final int a;
    public IntConstant(@Anon final int a)
    {
        this.a = a ;
    }

    @Override
    public int eval(final Context context)
    {
        return a;
    }
}
```

There is of course some overhead cost in having to instantiate constant values through the `eval()` method rather than passing those values directly, especially for primitive data types, but this cost appears to be minimal and is perhaps largely circumvented by optimisations performed by the HotSpot compiler.

TODO : CBB: But if any of you guys can devise a better way of achieving this, please let me know!

Such Function / Constant class pairs are currently defined for the following return types:

- `int`
- `boolean`
- `BitSet`
- `Region`

4.2 Algorithm

The grammar is constructed using the following steps:

1. `createSymbols(rootPath)`: Create a `Symbol` object for each class, inner class and enum value from the `rootPath` (i.e. `game` package) downwards.
2. `createRules()`: Create a potential `Rule` object for each symbol with the symbol as LHS, except for `Constant` symbols.
3. `addReturnTypeClauses()`: For each symbol whose return type differs from its class type, by implementing an `eval(final Context context)` method with a non-void return value, add that symbol as a new `Clause` on the RHS of the rule with that return symbol as LHS. This allows rule clauses to be gathered by return type rather than class type, which is much clearer and more convenient when defining games in the grammar.
4. `crossReferenceSubclasses()`: For each symbol that represents a derived class, add that symbol as a new `Clause` on the RHS of the rule with its superclass as LHS.

5. `replaceListFunctionArgs()`: For each symbol that is a `ListFunction` (explained in Section 4.1.5), replace that symbol with the list’s base symbol type and set its `isList` value as `true`. For example, all occurrences of `actionListFunction` will be replaced by `<list<action>>`, for greater flexibility in the grammar.
6. `linkToPackages()`: Traverse the rules generated for the root package (i.e. `game`) and recursively visit the rule associated with each type of argument of each clause (if not already visited), marking each such rule as “visited”. Only those rules marked as “visited” are shown in the grammar. Generally, any rule whose LHS does not occur in the RHS of another rule is not shown in the grammar (apart from the root `<game>` rule).
7. `setDisplayOrder()`: Prioritise the order in which packages are displayed so that the grammar follows the basic structure of the Ludeme Library but lists the functions and types at the end, and prioritise the order in which rules are displayed per package so that rules derived from base classes are listed first.
8. `removeRedundantFunctionNames()`: Tidy up the grammar by removing unwanted leftovers from step 5.

The constructor for a given class is added as a clause to the RHS of a rule if the LHS of that rule is:

1. The return type specified by the class’s `eval()` function (higher priority), or
2. The class itself (a constructor returns its own type).

In other words, classes (represented by symbols) are gathered by return type when determining rules in the grammar.

4.3 Game Descriptions

Each individual game is described as a *symbolic expression* (s-expression) compatible with the LUDII class grammar. For example, Tic-Tac-Toe may be described as follows:

```

❁ Ludeme 4
(game "Tic-Tac-Toe"
  (play {(player "P1") (player "P2")} Alternating)
  (equipment (board "Board" (square 3)))
  (rules
    (moves (to (indexOf Mover) (empty)))
    (end (line Mover Any 3) (result Mover Win))
  )
)

```

4.3.1 Instantiation

Game descriptions are parsed in a *top-down* manner [8, p. 225], with each `(class ...)` instance matched with its generating constructor, and parameters recursively instantiated as required. The calling app can then use the `JavaCompiler` and associated classes from the `javax.tools` library to compile the assembled code and produce an executable version of the game.

To maximise extensibility, it was initially envisaged that the game author may be allowed to append their own custom Java code to the end of the game description file, and call its constructors from within the description as per any other constructor defined in the grammar. However, security concerns make this an undesirable option.


TODO : CBB: the instantiation step has not been implemented yet (20/9/2018).

4.3.2 Formatting Guidelines

While the class grammar is conceptually decoupled from its generating code, the programmer can make the grammar cleaner and clearer by following some basic formatting guidelines. **Clarity in the grammar is paramount!** Please follow these guidelines when adding code to the LUDII code base.

4.3.3 Optional Parameters


Constructor arguments can be *explicitly* specified as [optional] items in the grammar using the custom annotation @Opt. For example, the following code:

```
 Java 16  
public Board(final Basis basis , @Opt final Modify [] modify)
```

will generate the following rule with an optional parameter:

```
 Grammar 5  
<board> ::= (board <basis> [{<modify>}])
```

Parameters can also be *implicitly* made [optional] by providing multiple constructors for a class, such that parameters that occur in one constructor but not another are interpreted as optional. For example, the following pair of constructors would produce the same rule shown above:

```
 Java 17  
public Board(final Basis basis )  
public Board(final Basis basis , final Modify [] modify)
```


The explicit @Opt approach is recommended as it is simpler and less error prone. The implicit approach, although more conceptually elegant, requires care to avoid ambiguous cases, encourages duplication of code, and complicates the initialisation of default values.

4.3.4 Explicit Parameter Names

Constructor parameters that are simple (terminal) data types are explicitly labelled in the grammar by their parameter name, prepended to each argument type as follows. This makes the grammar self-documenting to some extent, easier to interpret and reduces ambiguity and confusion:

```
(className (argName1 <argType1>) (argName2 <argType2>) ...)
```

For example, this:

```
 Grammar 6  
<what> ::= (what (who int) (where int))
```

is more meaningful to the user than this:

```
 Grammar 7  
<what> ::= (what int int)
```

It is sometimes desirable to *anonymise* named parameters, where this simplifies the grammar and does not create ambiguity; for example, the two parameters in `(add int int)` do not need naming. Such parameters can be explicitly denoted using the custom annotation `@Anon` to override the default behaviour.

Conversely, parameters representing complex (non-terminal) data types are not named in the grammar by default, as the data type itself usually gives enough information to infer the parameter's purpose. However, this behaviour can also be overridden to explicitly name such parameters using the custom annotation `@Name`. Note that parameter naming requires the use of Java version 8 for the relevant `reflection` call, but warrants the move to this version.

4.3.5 Default Values

It is useful to set default values for member variables of all classes described in the grammar, in case their corresponding constructor parameters are made optional. However, this is complicated by the fact that we also want to declare them as `final` and make the instantiated objects *immutable* if possible, as per good object oriented design practice [9, pp. 73–80].

Java only allows `final` member variables to be initialised once in the class's execution flow. This is handled in the class grammar by passing parameter values up the `super(...)` constructor chain as appropriate, and instantiating missing values due to optional parameters with their default values in the appropriate constructors. Care must be taken to instantiate the same default values across all constructors for each class, for consistency.

4.3.6 Library Structure

The LUDII code library is organised to reflect the underlying class structure, with each Java *package* containing the base class of the same name and immediate subclasses that will create items in the RHS sequence for the corresponding grammar rule. This makes it easier to navigate and maintain the code library using the class grammar as a reference. It also makes it easier to locate the actual classes associated with ambiguous symbols with the same name but different return types, such as the following:

- (or <boolean> <boolean>) => boolean
- (or <list<action>> <list<action>>) => <list<action>>

4.3.7 Abstract Classes

The programmer can influence the format of the generated grammar through judicious use of abstract classes. Constructors for abstract classes are not shown in the grammar as they

cannot be instantiated by the user. However, **interfaces** should be used rather than **abstract** classes where possible.

4.3.8 Inner Classes

The programmer is free to use inner classes. These will appear in the grammar.

4.3.9 Collections

Use **Lists** rather arrays to define collections where possible.

5

AI Module

The AI module contains the collection of default AI agents for playing the games in the Game Database. External users may use the default AIs, provide their own, or run tournaments between AI agents.

5.1 Default AI Agents

AI move planning will be performed using MCTS with playouts biased by strategies learnt through self-play. MCTS has become the preferred approach for general game playing over recent years, due to its ability to devise plausible actions in the absence of any tactical or strategic knowledge about the given task. Although it can prove weaker for some games than others, it provides a good baseline level of AI play for most games.

The combination of deep learning with MCTS has recently had spectacular success with Go [11]. However, this level of superhuman performance is not required for this project, where a more modest level of play pitched just beyond average human level is preferable, in order to estimate the potential of games to interest human players. Superhuman AI that plays differently to humans could actually bias evaluations; instead, we want an AI that makes moves that human players would plausibly make. We call such skill-adjusted AI agents *plausible AI* [1]. This means that it may be necessary to actually impede the AI in some cases, in order to reduce it to the desired level of play.

5.1.1 Lightweight Local Features

To elevate MCTS to a sufficient level of play for all games, playouts will be biased with domain-dependent information in the form of lightweight features that capture geometric piece patterns, learnt through self-play. For example, the pattern shown in Fig. 5.1, which completes a threatened connection in connection games played on the hexagonal grid, improves MCTS playing strength when incorporated into the playouts of such games [12].

Such patterns represent local strategies that human players typically learn to apply. They

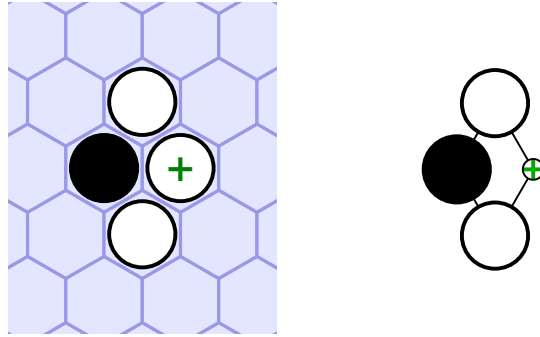


Figure 5.1: A strong pattern for connection games on the hexagonal grid.

will not capture more complex global strategies, but should serve to improve MCTS to plausible levels of play, and – importantly – could give an indication of a game’s strategic potential.

Lightweight local features are described in a simple and compact plain text language (exact format to be finalised) that describes geometric relationships between the actors within each feature independently of any underlying board topology (beyond adjacency of sites). This means that features may be transferred easily and meaningfully between games of differing topology.

One advantage of using the custom `ChunkSet` data type for describing game state information is that such lightweight local features can be used to bias MCTS playouts efficiently and with negligible impact on performance (per feature) as explained in Section 3.2.

6

Player Module

The `Player` module (see Figure 6.1) includes a `Referee` object for managing a currently selected game, and a `MainView` object consisting of a set of custom panes framed within a `javax.swing.JPanel` for showing the current board state and allowing user interaction. **Note that any use of `javax.swing` should be replaced by its `javafx` equivalent as future `java.swing` support is not guaranteed.**

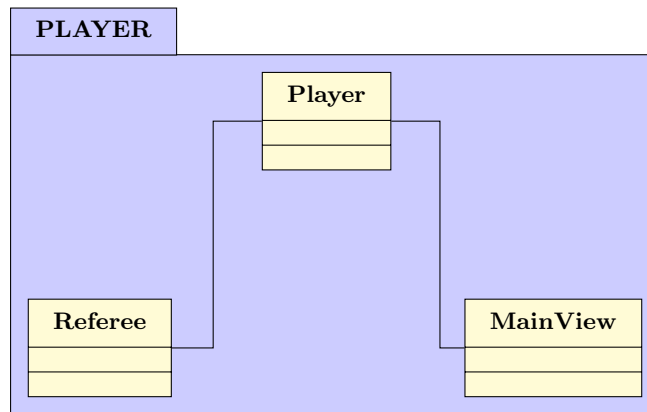


Figure 6.1: The general diagram of the `Player` module.

6.1 Graphical User Interface (GUI)

Figure 6.2 shows the standard LUDII GUI for a simple default game. The interface is simple and clean, with an emphasis on displaying the game and its current state clearly and without

ambiguity, and undecorated by theme. The main class `PlayerApp` initializes the GUI.

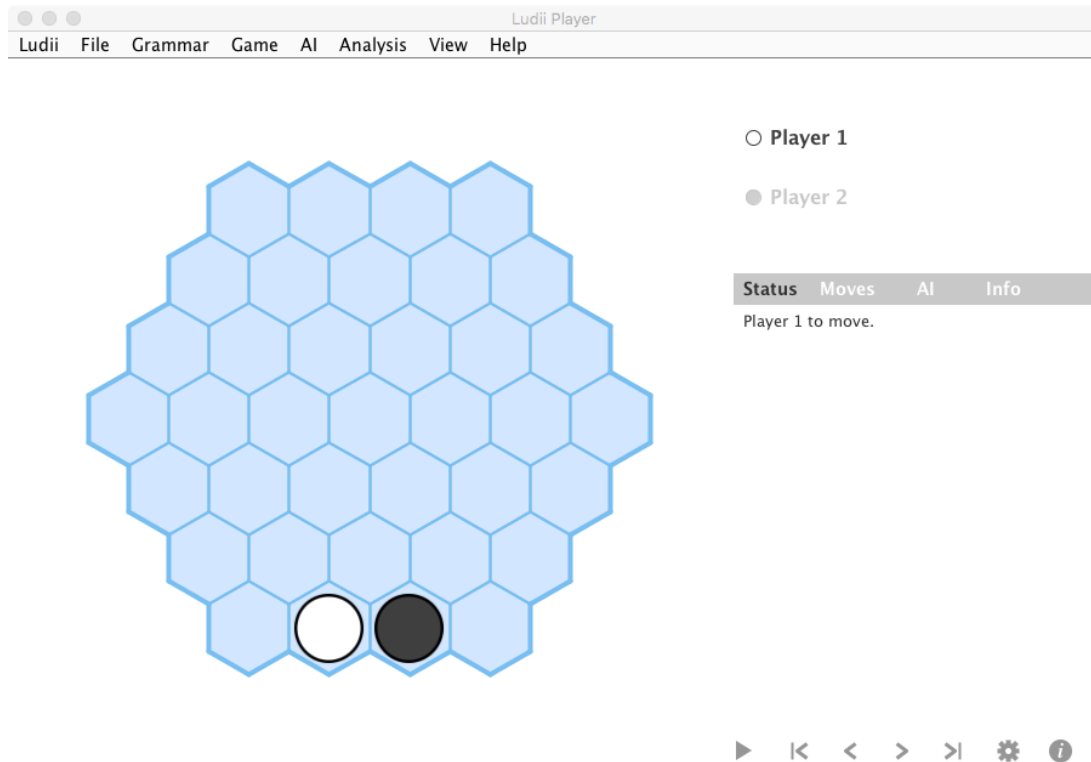


Figure 6.2: The default LUDII GUI.

Each piece of equipment defined for a given game implements the `Drawable` interface and can draw itself, scaled appropriately for the current window size. Each container should be able to draw its components at the appropriate scale at the appropriate positions.

Alternative view types, such as 3D perspective views – especially useful for stacking games, as shown in Figure 6.3 – may be added at a later date.

Functionality may be added in future to allow users to provide custom graphics for equipment to be used instead of the self-drawn default graphics. However, this would hugely inflate the memory footprint of the resulting system due to the additional image resources, so might be more appropriate for deriving single-game applications with custom graphics from the Game Database.

Pending the implementation of all the database in the system, a temporary Loading System is used to run the different games. The `GameList` class defined all the games with the appropriate ludemes for each of them. The Figure 6.4 shows a part of the current loading system filled by the `DBView` class and modeled by the `DBTableModel` class. This is possible to add some others buttons in the `JTable` shows by the loading system thanks to the `ManagePane` Class.

TODO : ERIC: Update Button, Delete Button, ect.

All the classes defined in the `dbView.renderer` package are useful to manage the GUI of the loading system and they have to be adapted when the database will be ready to call the Commons module.

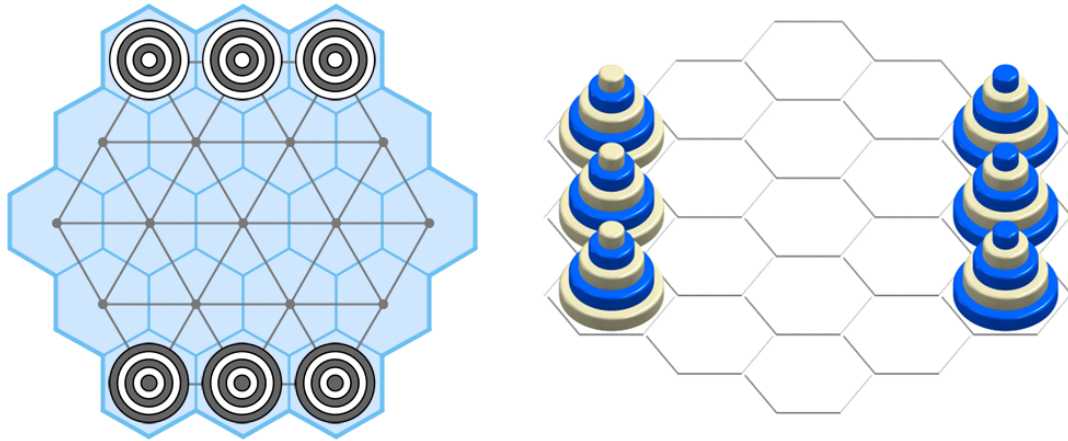


Figure 6.3: A prototype stacking game and suggested 3D perspective view.

Name	Category	Family	Time Type	#Players	Designer	Author	CreationDate	Size	Subtle	Public	Ready	Load
Amazons	Abstract Strategy	Combinatorial	Discrete	2	Walter Zemanek	Walter Zemanek	1982	18x10	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Andarino	Abstract Strategy	Pattern	Discrete	2	David L. Smith	David L. Smith	1995	10x10	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Asaf Ri	Abstract Strategy	Tap	Discrete	2	Unknown	Unknown	Unknown	7x7	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Ashbasa	Board game	Roll game	Discrete	2	Unknown	Unknown	200	8x8	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Battle 32 (without multi-capture)	Card game	Card game	Discrete	2	Unknown	Unknown	Unknown	2x1	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Battle 32 (without multi-capture)	Card game	Card game	Discrete	2	Unknown	Unknown	Unknown	2x1	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Brazilian Draughts	Abstract Strategy	Checkers	Discrete	2	Unknown	Unknown	1930	8x8	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Breaththrough	Abstract Strategy	Combinatorial	Discrete	2	Unknown	William Daniel Truitt	2001	8x8	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Canadian Draughts	Abstract Strategy	Checkers	Discrete	2	Franz Harmsen, Beate Pfister	Franz Harmsen, Beate Pfister	1990	12x12	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Chess	Abstract Strategy	Chess	Discrete	2	Unknown	Unknown	1475	8x8	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Connect 4	Abstract Strategy	Non-s-row	Discrete	2	Ned Strang, Howard Shuler	Unknown	1974	6x7	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Connect 6	Abstract Strategy	Non-s-row	Discrete	2	Professor Chen Wu	Professor Chen Wu	2003	10x10	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Dara	Abstract Strategy	Non-s-row	Discrete	2	Unknown	Unknown	1800	5x5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Diamond Breaththrough	Abstract Strategy	Combinatorial	Discrete	2	Unknown	Unknown	Unknown	8x8	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
English Draughts	Abstract Strategy	Checkers	Discrete	2	Franz Harmsen, Beate Pfister	Franz Harmsen, Beate Pfister	1910	8x8	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Fanorona	Abstract Strategy	Alquerque	Discrete	2	Néstor Rumeral Andrés	Néstor Rumeral Andrés	1880	5x5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Fanoron-Dimy	Abstract Strategy	Alquerque	Discrete	2	Néstor Rumeral Andrés	Néstor Rumeral Andrés	1880	5x5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Fanoron-Telo	Abstract Strategy	Alquerque	Discrete	2	Néstor Rumeral Andrés	Néstor Rumeral Andrés	1880	3x3	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Fit & Pic 10x10	Abstract Strategy	Puzzle	Discrete	1	Unknown	Unknown	Unknown	10x10	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Five men's morris	Abstract Strategy	Non-s-row	Discrete	3	Unknown	Unknown	1480	5x5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Four Field Kono	Abstract Strategy	Combinatorial	Discrete	2	Unknown	Unknown	2005	4x4	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Futoshiki 4x4	Abstract Strategy	Puzzle	Discrete	1	Unknown	Unknown	2006	4x4	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Futoshiki 5x5	Abstract Strategy	Puzzle	Discrete	1	Unknown	Unknown	2006	5x5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Futoshiki 6x6	Abstract Strategy	Puzzle	Discrete	1	Unknown	Unknown	2006	6x6	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Gomoku	Abstract Strategy	Connect 5 game	Discrete	2	Unknown	Unknown	700	10x10	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load
Halma 8	Abstract Strategy	Chinese Checkers	Discrete	2	George Howard Stone	George Howard Stone	1884	8x8	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> A Load

Figure 6.4: A part of the current loading system.

Some shortcuts are available:

- CTRL + A: To show the axes
- CTRL + B: To show the board
- CTRL + C: To run a time move count
- CTRL + D: To show the dual
- CTRL + G: To generate the grammar
- CTRL + I: To show the cell indices
- CTRL + L: To open the Loading system
- CTRL + M: To make a move with the AI chosen (Random by default)
- CTRL + N: To create a new instance of the game
- CTRL + P: To run a complete playout
- CTRL + R: To restart the game
- CTRL + T: To run a time random playouts

6.2 Command Line Interface (CLI)

The *command line interface* (CLI) is defined in the `PlayerCli` class. This is used primarily for running experiments. This is the main command lines:

- `-h`: to print the help
 - `-l`: to print all the games available
 - `-xp all` to run random playouts on all games with 30 seconds of warming up time and 120 seconds by game.
 - `-xp string:gameName int:nbRow int:nbCol int:timeWarmingUpOnSeconds int:timeLimitOnSeconds:` to run a specific game with `nbRow` rows, `nbCol` columns with a specific warming up time and a time limit for the experiment.
-

7

Environment

The LUDII system is implemented in Java, to take advantage of Java's `reflection` library, flexible compilation options, and easy deployment across multiple platforms. Java version 8 is used as this is the first version that allows parameter names to be extracted from constructor arguments.

Developers should use the Eclipse environment where possible.

TODO : CBB: To summarise recommended compiler warning and error settings.

Avoid the use of third party software libraries where possible.

7.1 Compatibility

Known cross-platform compatibility issues include:

- Inconsistent font scaling between different environments. For example, the same code run on a Mac system to produce the GUI (Figure 6.2) produces unreadably small text on Windows systems.

7.2 Repository

The LUDII project and its associated sub-projects are housed on the GitHub repository <https://github.com/cambolbro/Ludii>. Contact the author (cambolbro@gmail.com) for access.

This repository is currently private as it is being developed for initial release, but may revert to a public repository in future, in keeping with the ERC's commitment to Open Access research source materials.

7.3 Version Control

Regression testing is important to guarantee that future additions to the Ludeme Library do not unduly affect existing content. This may be achieved by maintaining a record of deterministic playouts for each entry in the Game Database, generated by seeding the RNG with a hash code based on the game's (unique) name, and storing the moves thus generated. Any change to the library that makes any known game diverge from its stored playout record will be flagged for investigation.

Eventually, an appropriate JUnit test should be automatically generated for each new game entered into the Game Database.

8

Coding Style


All code added to the Ludeme Library – and to LUDII in general – should follow the basic philosophy outlined below and conform to the specified coding standard and optimisation guidelines.

8.1 Philosophy


1. *Generality* comes first. In any design choice, generality is the top priority.
2. *Simplicity* comes second. For any ludeme mechanism added to the Library, simplicity in the generated grammar is paramount, unless it reduces generality. *Note that this refers to the simplicity of the generated grammar rather than the simplicity of the underlying code!*
3. *Efficiency* comes third. In any design choice, efficiency is strongly encouraged unless it reduces generality or simplicity.
4. After adding any ludemes to the Library, check the corresponding rules in the generated grammar. If the resulting grammar is ambiguous, confusing or limits generality, then restructure those ludemes.
5. Ludemes should be as general as possible, to maximise reuse. Ludemic descriptions are used to measure the distance between games, so the more explicit each ludemic description, and the more reused ludemes are by multiple games, the better.
6. Do not over-specialise ludemes. For example, do not specify a custom ludeme for a particular piece such as Chess knight; instead build up the knight behaviour from simpler ludemes. Otherwise, the specialised piece will act as a self-contained unit and sub-ludemes within it cannot be overridden to define variant behaviour. Also, it would not then be possible to measure the distance between this piece's behaviour and another piece's behaviour in any meaningful way by their ludemes.

8.2 Coding Standard


1. Each opening bracket should be on a separate line at same indentation as the corresponding closing bracket:

```
 Java 18  
for (int i = 0; i < n; i++)  
{  
    // do something  
}
```


not:

```
 Java 19  
for (int i = 0; i < n; i++) {  
    // do something  
}
```

2. This includes array initialisations split over multiple lines, e.g.

```
 Java 20  
final int [] fibonacci =  
{  
    1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 144, 233, 377,  
    610, 987, 1597, 2584, 4181, 6765  
};
```

3. Even better, format terms to line up in each column for easier reading and error-checking, e.g.

```
 Java 21  
final int [] fibonacci =  
{  
    1,    1,    2,    3,    5,    8,    13,    21,    34,    55,  
    89,   144,   144,   233,   377,   610,   987,   1597,  2584,  4181,  
    6765  
};
```

4. Make logically grouped lines of code line up where possible, to make inconsistencies stand out, e.g.:



Java 22

```
int row    = 0;
int column = 0;
int level  = 0;
```

or:



Java 23

```
switch (day)
{
case 0: str += "January"; break;
case 1: str += "February"; break;
case 2: str += "March"; break;
case 3: str += "April"; break;
case 4: str += "May"; break;
case 5: str += "June"; break;
case 6: str += "July"; break;
case 7: str += "August"; break;
case 8: str += "September"; break;
case 9: str += "October"; break;
case 10: str += "November"; break;
case 11: str += "December"; break;
default: str += "?"; break;
}
```

5. Leave a space after each keyword and between operators, i.e. `if (a < b)` not `if(a<b)`.
6. However, is is allowed to compress space around operators to save space if necessary, especially if they are being used in calculations for parameters or array indexing, e.g.



Java 24

```
int a = array [ cell / cols ] [ cell % cols ];
```

or:



Java 25

```
final boolean okay = onBoard ( cell / cols , cell % cols );
```

7. If compressing space around operators, do so according to operator precedence, e.g.:



Java 26

```
return first + 2*second + 3*third / fourth - fifth * sixth;
```

8. Format ternary conditionals as follows. One line:



Java 27

```
int n = (a < b) ? a : b;
```

Two lines:



Java 28

```
final int longWindedName = verboseBooleanTest(a, b)
                             ? generate(a) : generate(b);
```

Three lines:



Java 29

```
final int longWindedName = verboseBooleanTest(a, b)
                             ? generateSomethingElse(a)
                             : generateSomethingElse(b);
```

That is, line up the operands where possible.

9. Align if/else blocks as follows:



Java 30

```
if (test)
{
    ...
}
else
{
    ...
}
```

10. Do not leave a space before semicolons, i.e. do this:



Java 31

```
for (int i = 0; i < n; i++)
```

not this:



Java 32

```
for (int i = 0 ; i < n ; i++)
```

11. Lines should not exceed 80 characters long. Split long lines as needed, e.g.



Java 33

```
public void drawLine3D
(
    final Graphics2D g2d,
    final double x0, final double y0, final double z0,
    final double x1, final double y1, final double z1
)
    ...
```



Java 34

```
if
(
    (length > 100 || height > 100)
    &&
    (age > 50 || weight > 100)
    &&
    (hairLength > 50 || shoeSizeheight > 10)
)
    ...
```

12. Scope consistently. If any clause in a if-else test requires brackets, due to having multiple lines, then scope every clause in that statement with brackets, even single-line clauses, e.g. do this:



Java 35

```
if (a < b)
{
    return -1;
}
else if (a == b)
{
    same = true;
    return 0;
}
else
{
    return 1;
}
```

not this:



Java 36

```
if (a < b)
    return -1;
else if (a == b)
{
    same = true;
    return 0;
}
else
    return 1;
```

13. Otherwise, do not scope single line clauses in if-else statements with brackets (unless any of its clauses involve multiple lines). That is, do this:



Java 37

```
if (test)
    return a;
```

and this:



Java 38

```
if (test)
    return a;
else
    return b;
```

not this:



Java 39

```
if (test)
{
    return a;
}
```


or this:




Java 40


```
if (test)
{
    return a;
}
else
{
    return b;
}
```

14. Include a header comment for every file in Javadoc format, briefly describing the purpose of that file's class and listing the initial author and subsequent authors, e.g.:

```
 Java 41  
/**  
 * Container for holding game equipment.  
 * @author cambolbro  
 */
```


15. Include a header comment for every constructor and method in every class in Javadoc format, e.g.:

```
 Java 42  
/**  
 * Constructor.  
 */  
public Basis(final int dim)  
{  
    ...  
}
```

```
 Java 43  
/**  
 * Draw the item at the specified point.  
 */  
public void draw  
(  
    final Graphics2D g2d, final Item item, final Point pt  
)  
{  
    ...  
}
```

It is not necessary to specify `@param` comments if the parameters are well named and self-explanatory.

16. Always describe the return type and any assumptions, side-effects or postconditions in methods that return a value, e.g.:

```
 Java 44  
/**  
 * @return String description of item, else null if item is null.  
 */  
public String extractName(final Item item)  
{  
    ...  
}
```

```
}
```

No other comment is necessary if the `@return` comment gives a sufficiently complete description of the function's operation.

17. Avoid “magic numbers” in code, e.g. do the following:



Java 45

```
for (int i = startAngle; i < endAngle; i++)
{
    // do something
}
```

not:



Java 46

```
for (int i = 2; i < 5; i++)
{
    // do something
}
```

18. Comment any block of code that might be confusing, or which might hide a non-obvious assumption, side-effect, precondition or postcondition that might confuse other team members... or yourself if you need to work on that code again after a couple of years.
19. Use lowerCamelCase for member variables, parameters and local variables.
20. Use UpperCamelCase for class names, constants (declared `final`) and enum constants.
21. Do *not* use underscores in variable names, e.g. do not do: `_tmp`, `out_` or `var_a`.
22. Separate groups of methods that make up logical sections of code with a dividing line, up to the 80 character mark:



Java 47

```
public void draw ()
{
    // ...
}

//-----

public void read ()
{
    // ...
}
```

8.3 Optimisations

1. Avoid using integer modulus `%` as it generates a surprising number of bytecode instructions. Use alternatives where possible, e.g. `(n & 1) == 0` to check odd/even parity.
 2. Make all classes, member variables, local variables and parameters `final` where possible. This increases the likelihood of their being inlined by the HotSpot compiler.
 3. Keep speed-critical methods small, i.e. generating 325 bytecode instructions or fewer,¹ so they can be inlined by the HotSpot compiler. In general, keep any methods to reasonable sizes, e.g. 30–40 lines in length, to fit on a single screen for reading.
-

¹<http://normanmaurer.me/blog/2014/05/15/Inline-all-the-Things/>

9

Conclusion

This document provides an overview of the internal workings of the LUDII general game system, and provides some tips for correctly and effectively adding code to extend the Ludeme Library. This documented will be revised as the LUDII system matures.

Acknowledgement

This work is part of the *Digital Ludeme Project*, funded by €2m European Research Council (ERC) Consolidator Grant 771292, being conducted at Maastricht University over 2018–23.

Bibliography

- [1] C. Browne, “Modern Techniques for Ancient Games”, *IEEE CIG 2018*, Maastricht, IEEE Press, 2018, pp. 490–497.
 - [2] D. Parlett, *The Oxford History of Board Games*, Oxford, Oxford Univ. Press, 1999.
 - [3] A. de Voogt, “Distribution of Mancala board games: A methodological inquiry”, *Board Game Studies*, vol. 2, 1999, pp. 104–114.
 - [4] F. Horn and A. de Voogt, “The development and dispersal of l’Attaque games”, *Board Game Studies*, 2008, pp. 43–52.
 - [5] A. de Voogt, “Moving into Micronesia: Checkers and Sorry!”, lecture, Board Game Studies Colloquium XX!, Athens, 2018.
 - [6] D. Parlett, “What’s a Ludeme?”, *Game & Puzzle Design*, vol. 2, no. 2, 2016, pp. 83–86.
 - [7] C. Browne, “A Class Grammar for General Games”, *Computers and Games (CG 2016)*, Leiden, Springer, LNCS 10068, 2016, pp. 169–184.
 - [8] Ghosh, D.: DLSS in Action. Manning, Stamford (2011)
 - [9] J. Bloch, *Effective Java*, second edition, Addison-Wesley, Boston, 2008.
 - [10] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis and S. Colton, “A Survey of Monte Carlo Tree Search Methods”, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, 2012, pp. 1–43.
 - [11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis, “Mastering the Game of Go with Deep Neural Networks and Tree Search”, *Nature*, vol. 529, no. 7587, 2016, pp. 484–489.
 - [12] T. Raiko and J. Peltonen, “Application of UCT search to the connection games of Hex, Y, *Star, and Renkula!”, *Proc. Finn. Artif. Intell. Conf.*, Espoo, Finland, 2008, pp. 89–93.
-

Appendix A: LUDII Class Grammar Example

The following listing shows the LUDII class grammar generated from an early version of the Ludeme Library. Rules are grouped by package.

```
 Grammar 8

//-----
// game

<game>      ::= (game <string> [{<metadata>}] <play> <equipment> <rules>)

//-----
// game.metadata

<metadata>  ::= (metadata (keyword <string>) (text <string>))

//-----
// game.play

<play>      ::= (play [(players <list<player>>)] [(time <timeType>)])
<timeType> ::= Alternating | Discrete | Real

//-----
// game.play.player

<player>    ::= (player [(index int)] (name <string>))

//-----
// game.equipment

<equipment> ::= (equipment <list<item>>)
<get>       ::= (get <string>)
<item>      ::= <component> | <container>

//-----
// game.equipment.component
```



```

<component> ::= <card> | <die> | <letter> | <number> | <tile> | <piece>
<card>      ::= (card (label <string>) (colour int) (owner int))
<die>      ::= (die (label <string>) (colour int) (owner int))
<letter>   ::= (letter (label <string>) (colour int) (owner int) (value <string>
>))
<number>   ::= (number (label <string>) (colour int) (owner int) (value int))
<tile>     ::= (tile (label <string>) (colour int) (owner int))

//-----
// game.equipment.component.piece

<piece>    ::= <ball> | <cross> | <disc> | <chess>
<ball>     ::= (ball <string> (colour int) (owner int))
<cross>    ::= (cross (label <string>) (colour int) (owner int))
<disc>     ::= (disc <string> (colour int) (owner int))

//-----
// game.equipment.component.piece.chess

<chess>    ::= <bishop> | <king> | <knight> | <pawn> | <queen> | <rook>
<bishop>   ::= (bishop (label <string>) (colour int) (owner int))
<king>     ::= (king (label <string>) (colour int) (owner int))
<knight>   ::= (knight (label <string>) (colour int) (owner int))
<pawn>     ::= (pawn (label <string>) (colour int) (owner int))
<queen>    ::= (queen (label <string>) (colour int) (owner int))
<rook>     ::= (rook (label <string>) (colour int) (owner int))

//-----
// game.equipment.container

<container> ::= <hand> | <board>
<hand>      ::= (hand <string> (owner int) (num int))

//-----
// game.equipment.container.board

<board>     ::= (board <string> <basis> [{<modify>}])

//-----
// game.equipment.container.basis

<basis>     ::= <hexHex> | <rect> | <wheel>
<hexHex>   ::= (hexHex (dim int))
<rect>     ::= (rect (rows int) (cols int)) | <square>
<square>   ::= (square (dim int))
<wheel>    ::= (wheel (spokes int)) | (wheel (spokes {int}) (aligned boolean))

//-----
// game.equipment.container.board.modify

```

```

<modify>      ::= <cut> | <join> | <remove>
<cut>         ::= (cut (cellA int) (cellB int))
<join>        ::= (join (cellA int) (cellB int))
<remove>      ::= (remove (cell int))

//-----
// game.rules

<rules>       ::= (rules [<list<start>>] <moves> <list<end>>))

//-----
// game.rules.start

<start>       ::= <place>
<place>       ::= (place [(who <roleType>)] [(item <string>)] [(count int)] (target
    <string>) [(posn int)])

//-----
// game.rules.moves

<moves>       ::= (moves <list<action>>)
<list<action>> ::= <fromTo> | <moves> | <or> | <to>
<fromTo>      ::= (fromTo [(container <int>)] (sitesFrom <sites>) (sitesTo <sites>)
    )
<or>          ::= (or <list<action>> <list<action>>)
<to>         ::= (to [(container <int>)] (component <int>) (sites <sites>))
<action>      ::= Action

//-----
// game.rules.end

<end>         ::= (end <boolean> <result>)
<result>      ::= (result <int> <resultType>)
<resultType> ::= Win | Loss | Draw | Tie | Abort

//-----
// game.functions.ints

<int>         ::= int | <get> | <add> | <indexOf>
<add>         ::= (add <int> <int>)
<indexOf>     ::= (indexOf <string>) | (indexOf <roleType>)

//-----
// game.functions.booleans

<boolean>    ::= boolean | <and> | <line> | <not> | <occupied> | <or>
<and>        ::= (and <boolean> <boolean>)
<line>       ::= (line [(cont <int>)] [(role <roleType>)] [(dirn <dirnType>)] (
    length int))

```

```

<not>      ::= (not <boolean>)
<occupied> ::= (occupied [(cont <int>)] (site <int>))
<or>      ::= (or <boolean> <boolean>)

//-----
// game.types

<string>   ::= String
<dirnType> ::= None | Any | All | In | Out | Along | Around | CW | CCW |
            Vert | Horz | Orth | Diag | Over | Under | N | E | S | W |
            NE | SE | NW | SW | NNE | ENE | SSE | ESE | NNW | WNW | SSW |
            WSW | U | D | UN | UE | US | UW | DN | DE | DS | DW | UNE |
            USE | UNW | USW | DNE | DSE | DNW | DSW
<roleType> ::= None | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | Any | All |
            Mover | NonMover | Opposite | Next | Prev | Odd | Even |
            Empty | Own | Enemy | Ally | NonAlly | Partner | NonPartner

```