Gizmoball Finalest Design Document

Austin Clements, Albert Leung, Dan Ports

May 11, 2004

Contents

1 Requirements **2**
1.1 Overview . . . 2
1.2 Glossary . . . 2
1.3 Revised Specifications . . . 3
1.3.1 User Interface . . . 4
1.3.2 Playing User Interface . . . 4
1.3.3 Editor User Interface . . . 4
1.4 User Manual . . . 4

2 Design **8**
2.1 Overview . . . 8
2.1.1 Design Philosophy . . . 9
2.2 Design Changes . . . 9
2.2.1 Changes For The Amendment . . . 9
2.2.2 Additional Changes . . . 9
2.3 Property System . . . 11
2.4 General User Interface . . . 12
2.4.1 Interaction Modes . . . 13
2.4.2 Editing Interaction Mode . . . 13
2.4.3 Property Table . . . 14
2.5 $\mathbb{R}^2$ Renderer . . . 15
2.6 Game System . . . 16
2.6.1 Architecture . . . 17
2.6.2 Interaction System . . . 17
2.6.3 Game Simulation . . . 18
2.7 Standard Gizmos . . . 19
2.8 Game Serializer . . . 20

3 Testing **21**
3.1 Validation Strategy . . . 21
3.2 Module Testing Strategies . . . 21
3.3 Test Results . . . 22

4 Reflection **22**
4.1 Evaluation . . . 22
4.2 Lessons . . . 23
4.3 Known Bugs and Limitations . . . 23

A Project Plan **23**

B Formats **23**

List of Figures

1 Playing user interface . . . 5
2 Editor user interface . . . 6
3 A frobber menu . . . 7
4 Gizmoball layer model . . . 8
5 Major classes module dependency diagram . . . 10
6 User interface module dependency diagram . . . 12
7 $\mathbb{R}^2$ renderer module dependency diagram . . . 15
8 Game system module dependency diagram . . . 16

1 Requirements

1.1 Overview

Gizmoball is similar to a traditional pinball arcade game with the notable addition that it gives users the ability to construct their own game layouts by placing "gizmos" on the game board. Additionally, Gizmoball can easily be extended with support for new gizmos and new looks and feels, opening up virtually unlimited possibilities.

The Gizmoball user interface is divided into two modes — playing mode and editing mode. Playing mode allows users to play the game, using the keyboard to control the gizmos. Editing mode allows users to edit their game board, as well as to create connections between gizmos that can lead to a variety of interactions, and to assign key triggers to gizmos to allow them to be triggered via the keyboard while playing. Because of the ability to edit games, Gizmoball also includes the ability to save and load game layouts in an XML file format.

1.2 Glossary

Gizmo The fundamental atomic game object. Gizmos are first-class, meaning that everything on the game board, including the ball, is a gizmo, and all gizmos are treated identically by the game system.

Squark The sound made by the collision of two gizmos.

Tick The process of updating all gizmos in a board for some time period.

Zero-knowledge system A component of Gizmoball that uses other components, but has only a weak coupling because it deduces information about the other components at run-time.[1]

Frobber A component of the editor that is displayed around the selected gizmo that allows its properties to be intuitively set. For example, the rotation frobber, when clicked, rotates the current gizmo.

Frobnicate To use the frobber menu of a gizmo.

Frobber menu [2] A menu similar in design to a pie menu, but persistent and containing frobbers. A frobber menu has a fairly small maximum radius to make it possible to select other gizmos while the menu is visible. The frobbers are rearranged if necessary to avoid the edge of the board.

Interaction mode A method of interaction between the user and Gizmoball . Either playing or editing.

Renderer The part of the system responsible for displaying the game board to the user and mediating user interaction with the game board. These are designed to be easily replaceable if a different look and feel is desired.

Drawer A component of a renderer that is responsible for drawing some component (such as a particular gizmo type) of the game board.

1.3 Revised Specifications

A number of minor revisions and clarifications have been made to the original specifications.

- When switching from playing mode to editing mode, all moving gizmos will freeze as they are. This includes flippers, which will stop at partial orientations if necessary. However, all gizmos will retain their state (and the editor will reflect this in the properties). One ramification of this is that the ball will have the velocity it left off with when editor mode was switched to. If no edits are made, the game should resume precisely as it left off when edit mode was switched to.
- The original specifications required gizmos to be laid out on a grid in the editor. However, this has been changed to allow the user to disable the grid if they wish to place gizmos in any valid location.
- When flippers are triggered while they are flippering, they will immediately reverse direction. This is the most like how real pinball games work, therefore it will probably be the most intuitive for users to use.
- Because of first-class gizmos, the ball is no longer a special entity in the system. It is simply another gizmo that operates by the same rules that the other gizmos do. The generality of the interaction system (see Section 2.6.2) makes this possible without additional work.

The following sections cover the user interface specifications in greater detail.

[1] Almost, but not quite, entirely unlike a zero-knowledge cryptosystem

[2] Patent pending

1.3.1 User Interface

The User Interface is divided between two interaction modes, one for building boards, and the other for playing Gizmoball. A switch toggles between the two modes, appropriately adjusting the appearance of the interface and its behavior.

1.3.2 Playing User Interface

In playing interaction mode, the interface will only respond to a limited number of menu items (load, save, mode-switch, exit) and the keys that have been connected to gizmos.

1.3.3 Editor User Interface

The editing interaction mode expands the functions accessible to the user, revealing new areas of the interface that allow for modifications to the board. The load, save, mode-switch, and exit keys are still available, and the board from the play mode is frozen.

The first set of expanded features in the editing mode is a toolbar for creating gizmos. Gizmos can be dragged from the toolbar and placed onto the board.

To facilitate the transformation of gizmos, a *frobber* menu (see Glossary 1.2, and Figure 3) is applied to the gizmo in focus. This method was chosen over the alternative of having tools to perform the actions — rotate, move, delete, set color, connect, etc. — because it improves access time (Fitt's law). Connections are also created using one of the options in the frobber menu interface. The connecting feature supports both click-and-drag and click-and-click to indicate the two gizmos involved. This allows for greater flexibility. One possible problem was the potential confusion between connecting to a second gizmo, and selecting that gizmo, but this problem was resolved using modality of the connection creation method. This modality is indicated by an arrow from the trigger gizmo trailing the mouse pointer.

Ball velocity is also set using a frobber menu option. When the option is selected, an arrow appears, with which the user can indicate direction and magnitude. The option snaps to logical values (0, vertical and horizontal angles, etc).

All gizmo information for the selected gizmo is displayed in a pane at the bottom of the screen. This will contain name, type, color, and the connections to and from the gizmo. The property list will be in a tree form, to allow the information to be better navigable. Properties can also be set through the list.

1.4 User Manual

Loading a game board To load a game board, click "Load..." and navigate to the file you wish to load. In the event of an error, a message will appear explaining the nature of the error.

Saving a game board After editing or while playing, a game board can be saved by clicking "Save...". If the file was one that was loaded or previously saved, you can overwrite the existing file or choose a new filename and location.

Adding gizmos to the game board To add a gizmo, click the gizmo on the gizmo toolbar. Moving the mouse over the gameboard area will reveal a gizmo trailing the pointer. Clicking the mouse again will place the gizmo in the current location. Multiple gizmos of the chosen type can be

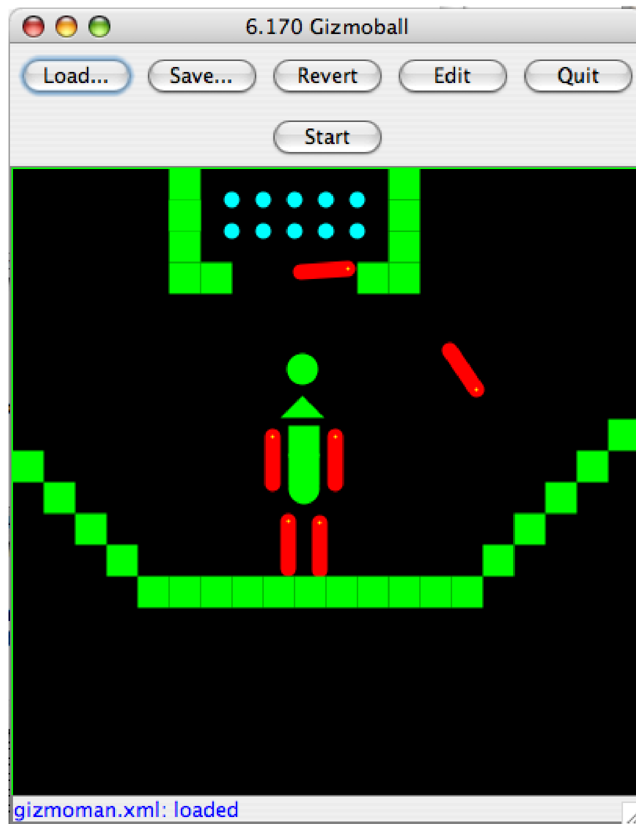

Figure 1: Playing user interface

added at a time. To cancel gizmo buildding, press 'Esc' or click the gizmo in the toolbar. Gizmos may be aligned to a grid by activating the "Snap to grid" option, or they may be freely placed. Balls are also added as gizmos.

Switching between editing and playing To switch between the two modes, click the appropriate button in the menu. This button is labeled "Play" while in Editing mode, and "Edit" while in Playing mode.

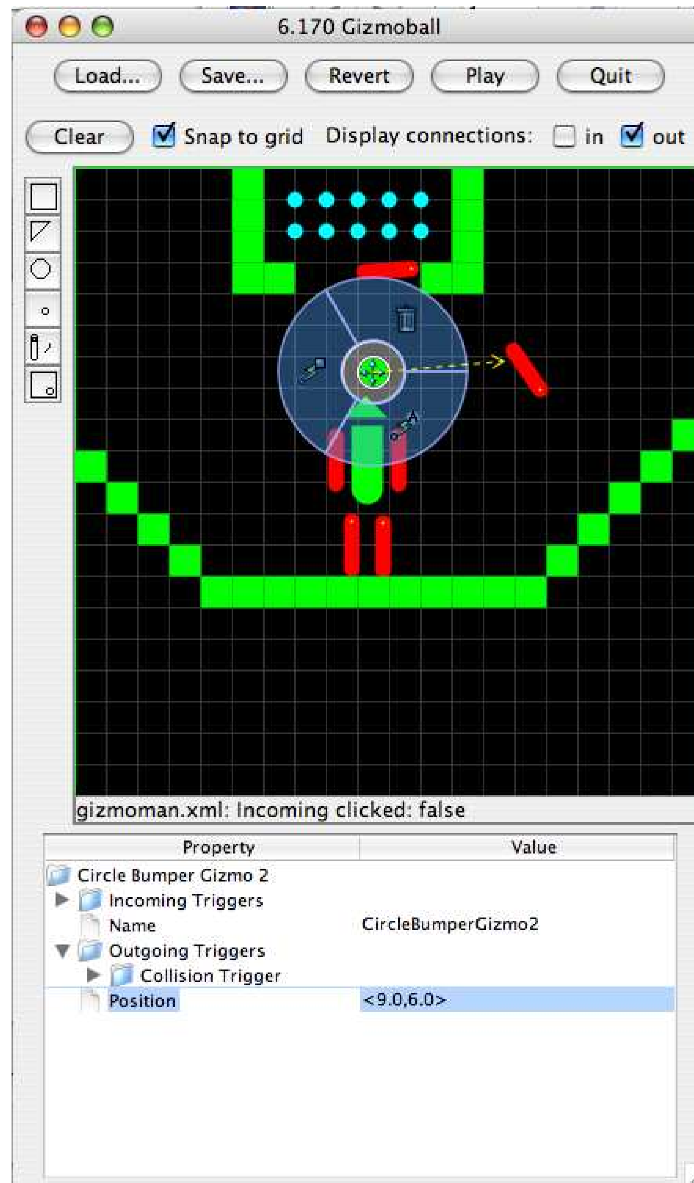

Figure 2: Editor user interface

Clearing the game board To clear the gameboard, switch to Editing mode and click the "Clear" button. You will be prompted to confirm that you want to clear the board.

Reverting the board to a saved state The game board can be set back to its last recorded state. If no files have been loaded or save, reverting the board will clear the board. Otherwise, the board will be set back to the state of the last loaded or saved file.

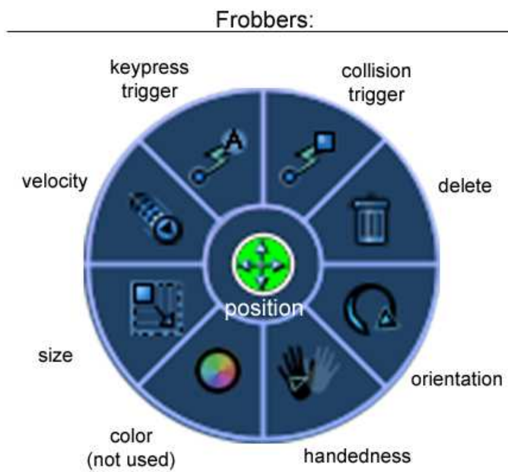

Figure 3: A frobber menu

Transforming/deleting gizmos Clicking on a gizmo on the game board will select it. Once selected, a menu will appear around the gizmo displaying a set of frobbers available to that gizmo. Clicking on the frobber will allow you to make the desired change. For example, clicking the "Position" frobber causes the gizmo to follow the mouse. Clicking again sets the gizmo in its new location. Alternatively, most frobbers support clicking and dragging.

Transforming gizmos and setting properties (advanced) Gizmo properties can be set through the property list. Each property (name, position, orientation, color, triggers) will have a corresponding entry in the property list. Double-clicking the property value will allow the value to be edited. Input the desired value to set the property. This method allows properties not accessible through the frobber menu to be set.

Connecting gizmos Once a gizmo has been selected, one of the displayed frobbers is the "Trigger" frobber. To connect the selected gizmos trigger to another gizmo's action, click and drag the connect option to the gizmo whose action is to be triggered, and then release. An alternative method is to click the connect option on the first gizmo and then click the second gizmo. By either method, an arrow from the trigger gizmo will trail the mouse pointer to indicate that a connection is being created. Once the connection has been created, it will be indicated by an arrow (depending on the view mode) and will also appear in the property lists of the two gizmos.

Connecting gizmos to keys To connect a gizmo to a key-press, use the gizmo's "Keypress" frobber. A dialog will appear explaining the setting procedure: pressing a key will set it as the keypress trigger. Using F1 and F2 allows you to choose whether the trigger activates on the key going up or down or both. After these options have been set, hitting Enter assigns the keypress trigger. Keypress triggers can also be set in the property table.

2 Design

2.1 Overview

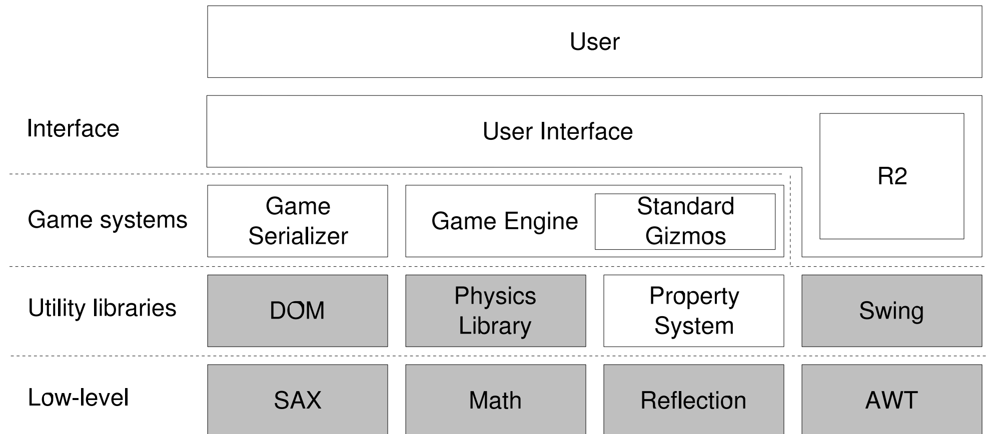

Figure 4: Gizmoball layer model

The design of Gizmoball is organized into three layers:

Utility libraries These libraries provide basic services to the Gizmoball game and interface. The physics library, Swing, and XML DOM are provided libraries. Additionally, this includes the `gizmoball.property` package (see Section 2.3), which encapsulates generic support for object property access. The property system is used throughout Gizmoball , particularly by the game system.

Game systems The game system consists of the `gizmoball.game` and `gizmoball.xml` packages. `gizmoball.game` (see Section 2.6) lies at the core of Gizmoball and captures the game physics. Note that this component of Gizmoball is entirely independent of the user interface. Also, this only defines the abstract gizmo, which is itself quite general. The `gizmoball.xml` (see Section 2.8) package also resides in the game systems, and encapsulated support for loading and saving Gizmoball game board configurations in the standard XML format. Furthermore, this includes the implementations of the standard gizmos, in the `gizmoball.gizmos.standard` package.

Interface The user interface is built on top of the game systems and the Swing library. It is responsible for driving the game system and presenting it visually to the user. The game system relies on the interface for timing the simulation and for information on interactions from the user (namely, key presses that result in triggers). The `gizmoball.ui` (see Section 2.4) package takes care of the overall interface, but abstracts out the mechanism for actually rendering the game board. Currently, this renderer abstraction is implemented by the $\mathbb{R}^2$ renderer in `gizmoball.ui.r2` (see Section 2.5), which displays a basic 2D game board.

2.1.1 Design Philosophy

Our guiding design principle was generality.[3] Our lower layers make as few assumptions as possible about the upper layers, providing generalized API's that the upper layers can leverage to control and direct the functionality of the lower layers.[4] Due to our goal of generality, we have code-named this project Gizmo*all* .

Following generality, we designed for component simplicity. The overall system achieves complex behavior by combining many of these simple components. As such, the design of Gizmoball is a tour de force of abstraction.

We leveraged dynamic polymorphism whenever reasonable in order to achieve drop-in replacability and to enforce our goals of generality. [2] Many components of the system resemble plug-in architectures because anything presenting the appropriate simple interface can be dropped in to change the behavior of various aspects of Gizmoball .

Finally, many of Gizmoball 's components are decoupled via run-time discovery. This goes hand-in-hand with our use of dynamic polymorphism because it allows components to discover the capabilities and structure of other components at run-time instead of relying on static knowledge.

2.2 Design Changes

Since the preliminary release, a number of changes have been made to the design. A few of these were necessitated by the amendment that was released following the preliminary release, though most were made for the platonic joy felt as a result of increasing the elegance of a design. Due to our approach of using many small components, the effects of these changes on other parts of the system remained fairly minimal.

2.2.1 Changes For The Amendment

No changes to the design were necessary to support multiple balls. Because our balls are simply gizmos and our collision system is general, the only necessary changes were in the implementations of the ball gizmo and the absorber gizmo.

The only necessary change to support polygonal gizmos was the addition of collection property support to the property system and the loader. However, both of these are still general (indeed, we had considered supporting collection properties from the beginning, but had decided that it wasn't necessary). The loader assumes that any sub-elements of a gizmo tag should be added to the corresponding collection property, therefore it could be used for any gizmo with collection properties.

2.2.2 Additional Changes

Most of the remaining design changes were with respect to making the game engine and gizmo designs more elegant. The gizmos were moved out of the game engine package and into their own because the game engine itself works entirely with abstract gizmos and because the game package was getting rather cluttered (see Section 2.7). To accompany this stronger division of game engine and gizmos, the collision system was replaced with a brand new "interaction system" (see Section 2.6.2). This system is designed to concretely capture a fully general mechanism through

[3] Our design was handed down to us, inscribed upon stone tablets by the gods of generality.

[4] Thus, we follow the ravioli model instead of the lasagna model, without reaching the extent of the risotto model.[1]

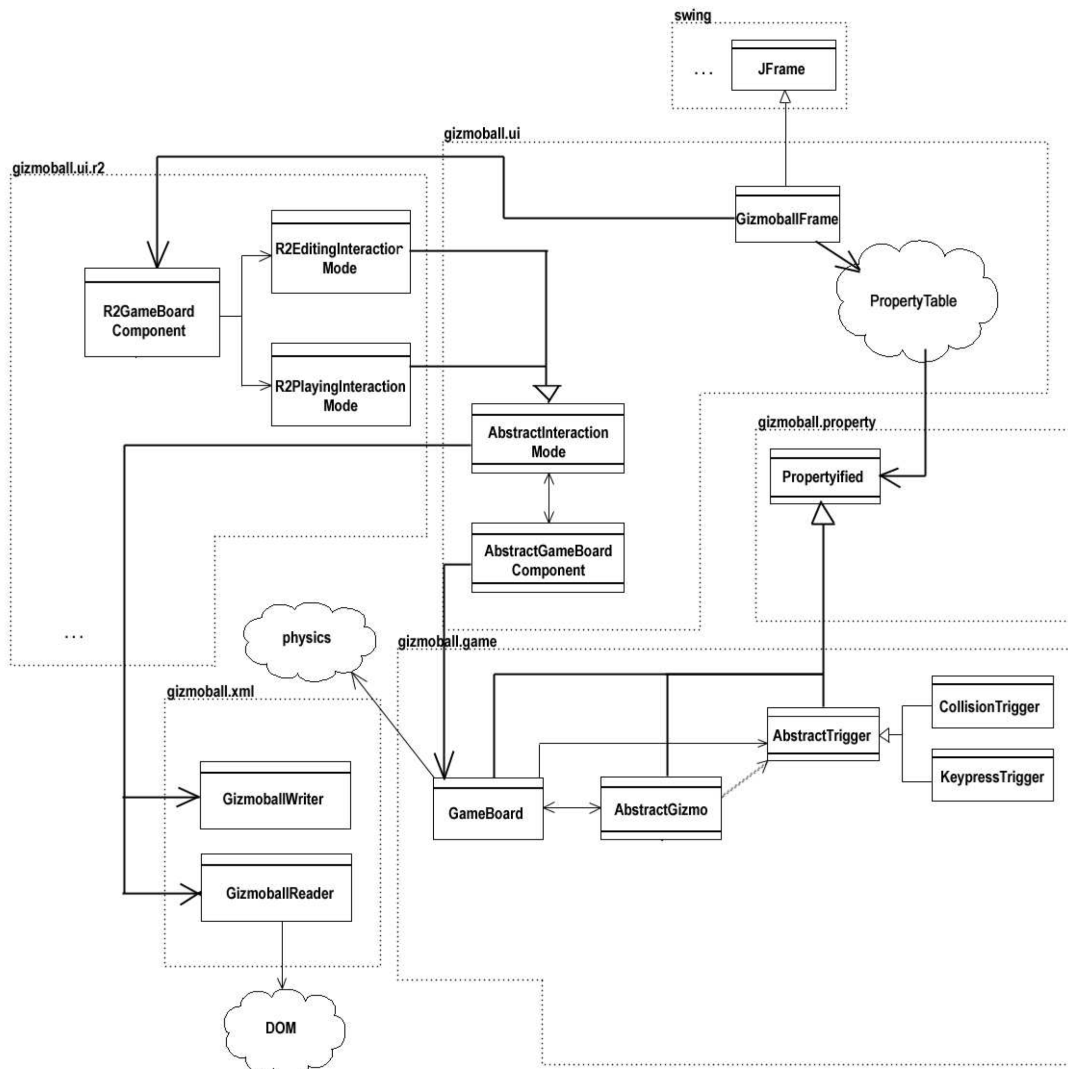

Figure 5: Major classes module dependency diagram

which gizmos can "interact". For the Gizmoball standard gizmos, this is used to model collisions, but the system itself is general.

Along with the new interaction system, what used to be `AbstractGizmoWithPolygonalGeometry` was generalized into a full-blown abstraction around the physics library that allows gizmos to specify their geometric properties and their motion. This abstraction allows any gizmo to collide with any other gizmo that the underlying physics library supports. Thus, this dramatically reduces the coupling between gizmos.

2.3 Property System

The properties system exists to abstract out and unify the process of getting and setting "properties" of objects. It is designed to be a general support library, and as such has no dependency on any of the other Gizmoball components. However, as such, it is used throughout Gizmoball as a general mechanism to decrease coupling between the components.

The properties system leverages an approach similar to that of JavaBeans.[5] Like JavaBeans, it assumes objects will follow a particular design pattern with get and set methods for accessing their properties. It presents unified methods to access the value of and set the value of any presented properties by name. The property system works through reflection, leveraging objects' existing get and set methods, thus allowing it to be combined with virtually any object without any change or penalty to code that does not need the unified access of the property system, while still incurring minimum changes to objects presenting their properties through the system. The run-time nature of the property system makes it more prone to certain types of errors that would normally be caught at compile-time, but it gives it a great deal more flexibility than pure compile-time code has.

There are three things objects are required to do in order to present their properties through this system:

- Extend `gizmoball.property.Propertyified`
- Report what properties the object supports (in the form of a list of property names)
- Call a method that in turns fires property observers when a set method is called

The property system is the base of the "zero-knowledge" systems in Gizmoball because, despite knowing nothing about the properties of objects at compile-time, it can present them for use by other systems. Thus, by leveraging the properties system, other components of Gizmoball can also be designed as zero-knowledge systems. All of the gizmos are written on top of the property system. This allows the game serializer (see Section 2.8) to get and set the properties of gizmos as they are loaded and saved without requiring knowledge of the types of gizmos it is loading or saving. The editor uses it to deduce which frobbers to display for a gizmo, as well as to display the properties table and keep it up to date. Because these systems are completely decoupled, it is trivial to add

[5] With the advent of Sun's JavaBean technology, one can finally answer the immortal question "What does that have to do with the price of Beans in Africa?" Because the JavaBean API is certified 100% Pure Java and Java itself is portable, the price of Beans in Africa is the same as it is nearly anywhere else in the world. However, we decided that the price of Beans was nevertheless too high to warrant their use in Gizmoball, instead choosing to implement our own property framework similar in spirit to JavaBean's property capabilities, but taylored to fit our needs without the overhead incurred by the other capabilities of JavaBeans.

new gizmos and change the properties that gizmos support; the remaining systems simply adapt to these changes at run-time.

The representation of the property system is fairly straightforward due to the fact that the values of the properties are all being stored in the classes that use the property system and not in the system itself. Otherwise, it makes ample use of reflection-based types to capture the methods and classes it uses to access properties.

We also considered using a non-reflection based property system which would have been more efficient. However, it would have required a great deal of additional support from objects implementing properties. Furthermore, it would not have fixed the only reasonable problem with the properties system, that is, the lack of compile-time checking. Because the property system is never used by Gizmoball under real-time circumstances, the decreased efficiency of the reflection based solution is acceptable.

2.4 General User Interface

The user interface for the Gizmoball game is implemented in the `gizmoball.ui` package. It is a generalized design that allows for multiple renderers to draw the game board state, and uses a shared interface framework with *interaction modes* for the game player and editor in order to achieve code reuse.

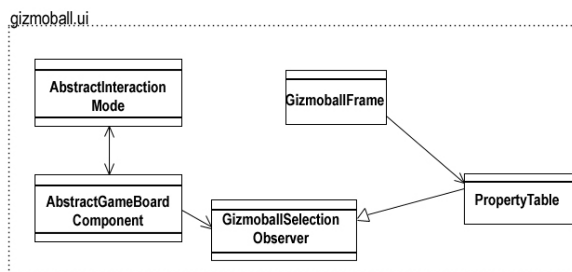

Figure 6: User interface module dependency diagram

The root of the interface is the `gizmoball.ui.GizmoBallFrame` class. This class handles the initial bootstrapping of the system, and displays the main window. It contains two rows of option buttons: a universal set that handles system-level functions such as loading, saving, and switching between editing and playing modes; and a mode-specific set with controls that are only relevant to one mode or the other (e.g. for playing mode, a start/pause button; for editing mode, connection display options). It also contains a `GameBoardComponent` that draws a gameboard on the screen. It also contains a `PropertyTable` that displays information about the selected gizmo, and an editor toolbar that selects gizmos to be placed.

The actual drawing of the gameboard is performed by a *renderer*. We separate the renderer from the UI package in order to make the design more general; by abstracting away the renderer, we can drop in a new renderer that provides different features. Initially, we intend to implement the $\mathbb{R}^2$ renderer (described in detail in Section 2.5), which draws a two-dimensional representation of the game board using the Swing API. If time permits, we may implement an $\mathbb{R}^3$ renderer using the Java3D APIs.

Each renderer has a `GameBoardComponent` class (e.g. `R2GameBoardComponent`) that extends the `gizmoball.ui.AbstractGameBoardComponent` abstract base class. This class handles all things

related to the gameboard and its display on the screen. It contains the `GameBoard` object that represents the game board (see Section 2.6.1). It accesses the game board's state, and displays it in a renderer-specific manner.

Key presses and mouse clicks on the game board are also handled by the `GameBoardComponent`. However, we maintain only a single `GameBoardComponent`, regardless of whether the game is in the editor or player mode. This decision eliminates the need to simultaneously maintain or update state in two different game boards, and facilitates code reuse. Since Gizmoball obviously behaves very differently depending on whether the user is currently editing or playing, we use *interaction modes*, which derive from `gizmoball.ui.AbstractInteractionMode` and encapsulate how the game reacts to user input (see Section 2.4.1). The `GameBoardComponent` selects either the playing or editing interaction mode (for the $\mathbb{R}^2$ renderer, `R2PlayingInteractionMode` and `R2EditingInteractionMode`). The `GameBoardComponent` harnesses the power of dynamic polymorphism by passing click and keypress events to the interaction mode object, which chooses the correct action for that mode and invokes the appropriate method of the `GameBoardComponent`.

2.4.1 Interaction Modes

Interaction modes provide a pluggable system of user control handlers that interface between events in the user interface and input domain and the internal game systems. The decision to adopt such a system stems from a need for different strategies of handling user interactions depending on both the current mode (playing or editing) and the current renderer ($\mathbb{R}^2$ or others). One can easily see how editing would require different functionality than playing: keypresses may be shortcuts rather than key triggers, and the gameboard component would need to handle mouse events to deal with gizmo placement and selection. Similarly, a 2d renderer would not require camera angle user controls that users of a 3d renderer would appreciate. The choice of an interaction mode system is also consistent with the overall emphasis on designing a system that is as general and expandable as possible.

Implementations of interaction modes act as the keyboard and mouse event listeners for the `GameBoardComponent`. As such, the interaction modes can handle user interactions from these input devices correctly. In addition, implementations of interaction modes have various methods that serve as relays from the GUI's components to the representation systems for the gameboard and display. This design allows the interface and the internal systems to be completely isolated; the interface and devices are unaware of underlying systems it controls, and the internal systems are unaware of buttons and input devices (or even a user).

2.4.2 Editing Interaction Mode

The editing interaction mode is designed in accordance with the philosophy of generality guiding the other aspects of the design. The enhanced display required for the editor is achieved by a system of editing artifacts that can be fluidly added to the gameboard component and updated. The editor's functionality is separated into submodes corresponding to different realms of user interaction: building gizmos, selecting gizmos, and modifying gizmo properties. To support the discrete interval placement of gizmos, a pluggable system of snap modes was implemented that could be expanded to handle a wide range of user-input discretizations.

Editing interaction mode is further divided into three submodes. The first mode activates when a user selects a gizmo type from the gizmo palette. Through this mode, the user must be able

to define gizmos and place them on the gameboard. However, gizmos of different types may vary significantly in how they are defined. Simple gizmos like the standard set can just be stamped at positions, but more complex ones like poly gizmos require schemes such as a series of clicks that define a closed polygon. To support a most general set of possible gizmo types, the build mode must be able to poll arbitrary volumes of information from the user, from a single click to an unpredictably long sequence. However, the mode must also be aware of when enough input has been collected that is sufficient for building a gizmo. To meet these requirements, a probe system was created.

Probes are temporary monitors of the user input channels–mouse and keyboard. As input is received, the probes cache the information, checking each time new input arrives whether sufficient input has been collected. Each type of probe is task-specific and will wrap a specific pattern of input sequence, signaling success when the pattern has been achieved and returning an atomic set of data representing the input sequence. For example, a simple probe would have sessions consisting of monitoring a single mouse click, messaging the submode when this has fired. Other probes could monitor for click-drag-release patterns and then notify a build mode for sizable gizmos to operate only after this pattern has been observed.

In conjunction with probes, submodes become adaptible so that a single submode can handle a class of actions. All gizmos can be built through a general submode design that reflects on the gizmo's properties, determines the user-definition scheme, selects the appropriate probe, and builds after the probe has finished its session. Similarly, the other two submodes assume a simple structure: the selection submode probes for mouse click collisions with frobber menus and other gizmos, and the property setting submode reflects on the property being set to figure out the input pattern.

Supporting the submode system is a system of editing artifacts that portray transient images and editing elements like connection arrows, snap grids, and frobbers. Artifacts are, however, not separate components that only exist for display purposes; any object can be used as an artifact, so long as the renderer has support for displaying it. For example, while building positionable gizmos, the transient artifact is an actual gizmo whose position is being refreshed in real time. The advantages of managing editing artifacts in this way are many: there is no redundancy, there are no synchrony issues, and the same mechanism used to draw the gizmo while in the gameboard can be reused to draw the gizmo in this transitional state. These artifacts are used to indicate modality and represent information.

Through this system, any editor-side task is divided into three components. On the user side is the probe that interfaces between the user input and the editing system. On the system side is the editing artifact that uses information gathered by the probe and translates it into visual feedback for the user. Combining these two aspects is the submode, responsible for carrying about the actual edits controlled by the user's input and portrayed by the editing artifacts.

2.4.3 Property Table

A property table is displayed at the bottom of the editor interface. It allows the user to browse through a tree view of the property system, viewing and editing the values of each property. If a gizmo is selected, it appears as the root of the property hierarchy; if no gizmo is selected, the `GameBoard` is the root.

The property table is implemented in the `gizmoball.ui.propertytable` package. The external interface is a `PropertyTable` class that, given a root node, traverses the property hierarchy to

generate and display a tree-table. Changes in the table are translated into changes in game state by parsing the text values and translating them into `setProperty` calls. The `PropertyTable` has `startListening` and `stopListening` calls that allow it to be registered as a `PropertyObserver` to receive notifications when properties change. This can be disabled during gameplay for speed.

We will not describe the internal workings of the `PropertyTable` and its associated package in great detail, as they are neither especially interesting nor easy to understand. No sane person should be subjected to the `gizmoball.ui.propertytable` package source. Part of the reason for the complexity is Swing's lack of a TreeTable class. Thus we were forced to implement our own `TreeTable` by using a `JTree` as a renderer for a `JTable`. The remainder of the code is a data model which builds the property tree, providing implementations of the required accessor and mutator functions for the `JTree` and `JTable`. To accomplish this, layers upon layers of abstractions wrapping other abstractions and being swaddled themselves in still more abstractions were required. We can only quote Thompson and Ritchie's infamous quote from the Unix source: "You are not expected to understand this." [3]. It will be better for your sanity if you don't.

2.5 $\mathbb{R}^2$ Renderer

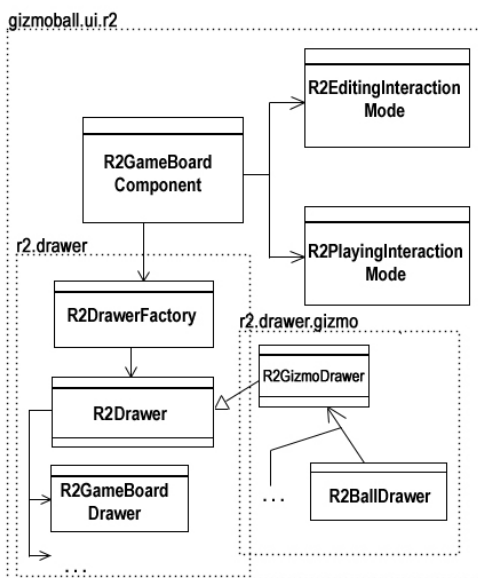

Figure 7: $\mathbb{R}^2$ renderer module dependency diagram

The $\mathbb{R}^2$ renderer[6] is Gizmoball's basic game board renderer that displays 2 dimensional representations of the game board. It provides the default implementation of the `AbstractGameBoardComponent` and `AbstractInteractionMode` abstractions.

The display mechanism works through a system of "drawers". One drawer exists for each type of gizmo and contains the methods necessary to render the respective type of gizmo to the $\mathbb{R}^2$ AWT graphics context. Because all of the drawers implement a common interface, the process of actually

[6]We considered calling this the $\mathbb{N}^2$ renderer, because the Java Virtual Machine is technically an integer machine, incapable of representing true reals, but decided it was best to remain outside of the `double` abstraction provided by Java.

rendering the board is simple a matter of traversing the set of drawers. Furthermore, a factory abstraction around the production of drawers also unifies the process of creating drawers from the game board components. The `R2DrawerFactory` is responsible for taking a gizmo from a live game board and producing an instance of the appropriate `R2GizmoDrawer` subclass. These drawers paint the gizmos onto the graphics context displayed by the gameboard component. Repainting of the gameboard is also controlled by the component and occurs at a rate of once per 50 milliseconds.

$\mathbb{R}^2$ also provides the necessary interaction modes for game playing and editing. Both interaction modes, `R2EditingInteractionMode` and `R2PlayingInteractionMode` sit atop the interaction mode abstraction provided by the general user interface. They mold the behavior of the `R2GameBoardComponent` to the playing and editing interfaces expected by the user, as described in Section 1.3.1.

2.6 Game System

The game system module, implemented in the `gizmoball.game` package, contains the classes necessary to represent a game state and simulate it.

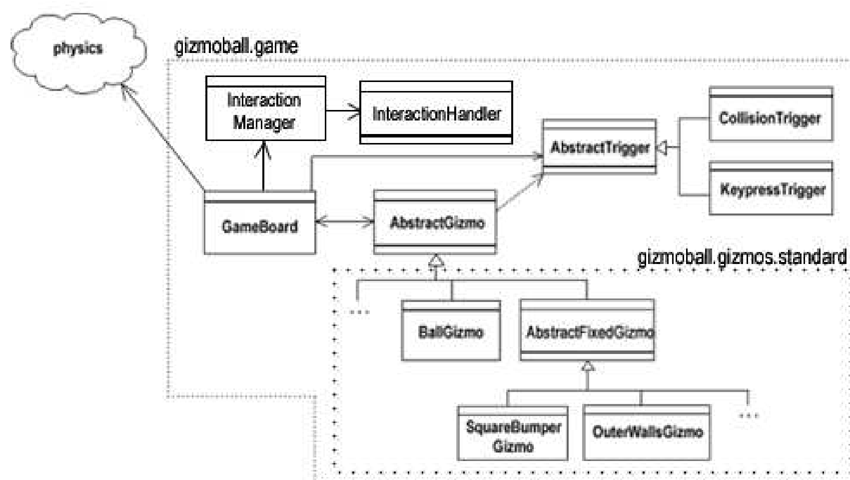

Figure 8: Game system module dependency diagram

In keeping with the Gizmo*all* design philosophy (Section 2.1.1), the design of the game system is extremely general, and can handle not only the specified Gizmoball behavior, but many additions and even entirely different games.

Perhaps the most important form of generality is the decoupling of the game system and the gizmos. The game simulator has no dependencies on the specific types of gizmos, and makes no assumptions specific to the gizmos in Gizmoball . It is equally at home simulating other systems such as Tetris or Traffick. It does not have concepts of motion, friction, or gravity — indeed, it has no concept of physics or geometry at all. These decoupled concepts are left to the gizmos to define, specify, and implement as they see fit. By giving the gizmos such wide latitude in their behavior (see Section 2.7), this frees the system from the limitations of the Gizmoball model.

Moreover, the game system does not rely on the concept of collisions. Instead, it uses the more general notion of *interactions*, which is a totally general model for the ways in which gizmos

interact with each other. The interaction system, described extensively in Section 2.6.2, provides a superset of Gizmoball 's collision model, as well as allowing entirely different forms of gizmo-gizmo interactions.

2.6.1 Architecture

The principal interface to this module is through the `GameBoard` class, which captures the state of a Gizmoball game board. A `GameBoard` can be constructed in an empty state, or it can be generated by a factory function in `gizmoball.xml.GizmoballReader` (Section 2.8).

Intuitively, a game board can be represented as a collection of gizmos, in addition to a few additional pieces of state information associated with the board itself. This is the representation used by the `GameBoard` class, a composite object that contains gizmos. We model the ball as a gizmo, so the ball does not need to be considered separately. Each gizmo maintains its own state information, including its position on the board.

Each type of gizmo is represented by its own class which inherits from the `AbstractGizmo` class. A gizmo class needs only support one type of operation: simulation for a given time duration. The `GameBoard` will call the gizmo's `tick` method with the time duration.

Optionally, it may also be triggerable. The `GameBoard` handles the management and firing of triggers. A gizmo may implement a `fireTrigger` handler that is called when it is triggered. This handler may do anything at all, including nothing at all.

Triggers are represented using a `AbstractTrigger` class that implements the triggering of the target gizmo, and has `CollisionTrigger` and `KeypressTrigger` subclasses. The name *collision trigger* is a misnomer: it should more accurately be known as an *interaction trigger*, since it is fired when an interaction occurs; the `CollisionTrigger` class name is used for historical reasons only. It is also associated with a source gizmo; whenever that gizmo is involved in an interaction, the trigger will be fired and trigger the target gizmo. A `KeypressTrigger` is associated with a key, and can either be a "key-down" trigger that triggers the target gizmo when the key is pressed, or a "key-up" trigger that triggers it when the key is released[7]. The `GameBoard` maintains lists of each type of triggers, and fires each trigger at the appropriate times.

A gizmo class may also define *interaction handlers*[8] handlers and register them with the game system's *interaction manager*. These handlers describe interactions that the gizmo may have with other types of gizmos. For more information on the interaction system, see Section 2.6.2.

The gizmos are also *propertyified*, i.e. they contain properties as used in the properties system (Section 2.3). This allows properties of the gizmos to be accessed and manipulated by the editor and serializer in a standard way. The only properties a gizmo is required to have are a name — a string used to uniquely identify the gizmo in the editor and in saved Gizmoball XML files — and the position at which the gizmo will be drawn on the board.

2.6.2 Interaction System

The interaction system manages the ability of gizmos to "interact". Interactions are completely general and require nothing more than the fact that they occur during the process of simulation and involve two gizmos. The standard Gizmoball gizmos use the interaction system to model collisions,

[7]Note that the terms "key-up" and "key-down" will no longer be correct if the keyboard is used upside down.

[8]Not to be confused with *interaction modes* — the two concepts are entirely unrelated.

though it is not necessary that interactions have anything to do with the geometry or physics of the gizmos (again, the game board has no concept of these).

Two classes sit at the core of the interaction system. The `InteractionManager` is a registry for `InteractionHandlers`. The interaction handler interface provides two methods: one for getting the time until the next interaction between two gizmos, and the other for actually performing an interaction once it occurs. Instances of interaction handlers are registered with the interaction manager by the gizmos, and the manager provides a method for later obtaining the appropriate interaction handler for interactions between two gizmos. Essentially, the interaction system is an object-oriented approach to generic programming. By acting as a registry, the actual handlers can exist in the gizmos and be loaded and registered as needed.

The `GameBoard` maintains a list of *interaction pairs*: pairs of gizmos that can interact. The game board constructs this list as gizmos are added so that the simulator can simply iterate over exactly which interaction pairs need to be considered without necessarily incurring $O(n^2)$ interaction checks during simulation time.

Prior to the adoption of this general interaction system, three alternatives were considered. First, we considered using a *collision manager*, similar to our current one, but containing methods for the collisions of gizmos. However, this would have had the disadvantage of requiring all gizmo types to be known about by this collision manager, making it more difficult to add new gizmos. We also considered using a technique similar to the operator overloading mechanism of the Python language. In this model, collision resolution would first attempt to notify the collider of the collision with the colidee. If this reported that it did not know how to resolve such a collision, a *reverse resolution* would be attempted, in which the collidee would be notified of the collision with the collider. If both of these resolution attempts failed, then it would be assumed that the two gizmos could not collide with each other (such as two fixed bumper gizmos). However, the resolution methods would have been difficult to implement, could have led to bizarre cases of infinite recursion, and may have resulted in code duplication. The third system we considered (and implemented for the preliminary release, but have since discarded) involved a distributed mechanism by which each gizmo could report what other gizmos it knew how to collide with, didn't know how to collide with, or knew it would never collide with. This allowed the game board to deduce the same interaction pairs that it maintains now, but required a great deal more complexity in the gizmos, and made it unclear exactly what could interact with what.

2.6.3 Game Simulation

After the `GameBoard` is constructed by either the serializer or the user interface (in edit mode), it can be simulated. The game simulator is decoupled from the user interface and the gizmos. The user interface module regularly calls the `GameBoard`'s tick method to request that the game state be simulated for a specified time duration. Simulation is performed according to the description and procedure below. Once simulation is complete, the UI module may access the new states of the gizmos.

Game simulation is performed by a *modified dynamic adaptive continuous-time discrete-event simulator*[9]. This divides a given time interval into time segments bounded by *events*. The archetypical simulator event is an interaction between two gizmos. In addition, the simulator makes use of two types of events that do not have an obvious concrete meaning: an event representing the

[9]Buzzword bingo!

end of the current tick, and periodic events to ensure that all interaction times are recalculated regularly. The simulation of each gizmo is performed in continuous time in the intervals between discrete events.

A timer in the UI triggers the `GameBoard`'s `tick` method regularly. This initiates the following procedure:

1. If the specified tick time is greater than the maximum tick length (a constant), it is divided into multiple ticks whose length is the maximum tick length or less. The remainder of the procedure assumes that the tick has been divided up.

2. An *end-of-tick* event is placed in the queue `time` seconds in the queue.

3. The earliest event in the queue is repeatedly processed, until the *end-of-tick* event is reached:

   (a) All gizmos on the board are simulated up to the event. Their `tick` method is called. If it returns true for any gizmo, indicating that the gizmo's velocity changed, that gizmo is scheduled for interaction time recomputation.

      - If the event represents an interaction, the interaction is processed using the interaction handler obtained from the interaction manager. Both gizmos involved in the interaction are scheduled for interaction time recomputation.
      - If the event is a *recalculate-interaction-time* event, both gizmos associated with it are scheduled for interaction time recomputation.
      - If the event is the *end-of-tick* event, the tick ends.

   (b) Interaction time recomputation is performed. First, every event involving any gizmo identified for interaction time recomputation is removed from the queue.

   (c) Next, every pair involving any gizmo identified for interaction time recomputation earlier in this processing algorithm is processed:

      - If the time to next interaction for the pair is less than the maximum tick length, an *interaction* event is placed in the queue at the time the interaction will occur.
      - Otherwise, if the time to interaction for the pair is greater than the maximum tick length, a *recalculate-interaction-time* event is placed in the queue, the maximum tick length in the future.

4. The remaining events in the event queue are stored so that they can be reused for the next event `tick` call. The game state is made available to the UI module and any other clients.

2.7 Standard Gizmos

Gizmoball comes with a standard set of gizmos which implement the abstract gizmo that the game board works with. These gizmos are the: ball, square bumper, triangular bumper, circular bumper, polygonal bumper, flipper, and absorber.

The standard gizmos augment the required gizmo properties with the following:

- the gizmo's name (a string used by the editor to identify a specific gizmo to the user)

- the gizmo's position on the board

- the gizmo's orientation, if appropriate for the gizmo type
- for a ball, its current velocity
- for an absorber, its size
- for a flipper, its handedness (whether it is a left flipper or a right flipper)

All of the gizmos except the ball utilize the `AbstractFixedGizmo` base class. This class allows its subclasses to represent their geometry as a union of circles, polygons, and line segments (the elements supported by the physics library). Additionally, it allows them to set the gizmo's rotation and coefficient of reflection. This is enough to describe all of the gizmos except for the ball (which is special because it is the only translatable gizmo). Furthermore, this precisely reflects[10] the capabilities of the physics library. Thus, `AbstractFixedGizmo` acts as an abstraction and a bridge between the physics library and the gizmo system.

Prior to `AbstractFixedGizmo`, we used an `AbstractGizmoWithPolygonalGeometry` class, which was only capable of representing bumpers with polygonal geometry that did not rotate. However, this was discarded in favor of the `AbstractFixedGizmo`, which has a completely superset of the functionality.

2.8 Game Serializer

The game serializer, contained in the `gizmoball.xml` package, is responsible for loading and saving game board states in the standard Gizmoball XML format. This package itself uses Java's DOM API for reading and writing the necessary XML data.

The loading and saving of game board states is divided into two classes: `GizmoballReader` and `GizmoballWriter`. `GizmoballReader` is a static factory responsible for loading an existing game state. It operates by using the standard GameBoard constructor to construct an empty board, then populating it with gizmos and connections as the XML data is traversed. `GizmoballWriter` performs the reverse, taking in a file (which may or may not exist) and an already populated GameBoard and writing the appropriate XML data to the file. Both of these processes can be initiated by the user through the user interface.

To decrease coupling, the XML reader and writer leverage reflection to find the gizmo classes and the property system to configure created gizmos and discover gizmo configurations, respectively. This way, the reader and writer need to know almost nothing about the individual gizmos, or even about which gizmos exist in the game. All of this information is simply inferred. Without this system, the reader and writer would have needed explicit knowledge of each supported type of gizmo.

Reflection is well suited to this system because the names of the gizmo classes are user-specified in the XML file, making it easy to add new gizmos to the system by simply changing the XML schema to allow them to be specified in the XML data. However, because naming conventions are enforced (as well as XML validation), this still restricts the user to valid inputs. The flippers are one exception to this rule because they are represented by a single class in the game system, but by two distinct tags in the XML format. This is an anacronism from the original specifications and may be fixed later (while retaining backwards compatibility). Currently this is handled by a

[10]Not, no that type. Or that other type.

special case syntax transformer which translates the incoming tags into an internal representation used for the flippers.

3 Testing

3.1 Validation Strategy

Gizmoball is currently and will continue to be tested rigorously, comprehensively, and extensively. Because the implementation of the design is proceeding in parallel, with many different components being implemented simultaneously, a variety of different testing strategies are appropriate. Each module is unit tested as it is implemented, and integration tests are performed as modules are integrated. Automated regression tests are performed regularly as changes are made. Each of the top-down, bottom-up, and inside-out testing methodologies are used, comprising both black-box and glass-box tests.

In particular, we are careful to test all types of inputs, including those that can cause errors. Correct error behavior, including exceptions raised and error messages displayed, is verified.

3.2 Module Testing Strategies

User Interface The user interface is essentially developed top-down, starting with the top-level `GizmoballFrame` and working down to the interface with the game system module and $\mathbb{R}^2$ renderer. Hence, testing is performed in a top-down manner, using simple stub classes as necessary. Very little unit testing can be performed on the user interface, so tests are performed by hand. A test script providing a sequence of actions to perform and the expected results (essentially a unit test, but performed by hand) has been written and used to verify the correct operation of the user interface and the system as a whole.

A great deal of testing was performed through the user interface. This verified that the user interface correctly provided access to all desired behavior. The disadvantage was that it was more difficult to track down the source of errors. However, extensive debug logging using `log4j` and our comprehensive representation invariants greatly simplified the debugging process.

$\mathbb{R}^2$ Renderer The $\mathbb{R}^2$ renderer and drawers were unit tested as well as tested with the rest of the UI. A separate application, `R2DrawerTestApp`, was created; it simply draws several gizmos. This was used to verify that gizmos could be drawn before performing integration tests with the UI.

Game System The game system was tested using stub gizmos. A simple `StubbyGizmo` that simply counted the number of times each of its methods was called was used to test adding and deleting gizmos and triggers. A more complex `ScriptedGizmo` whose responses to various queries could be scripted[11] was used for testing the simulator. This verified that simulation, including collisions, ticks, and recalculations were performed according to the specifications. The individual gizmos were integration tested with the game system framework.

Property System The properties framework was unit tested using a simple `PropertifiedClass` class that implemented properties. Its methods and the observer system were tested thoroughly.

[11] As in the script to the play *Unit Testing: A Half-Second Play In Two Acts, Entirely Divorced of Any Physical Meaning*

Game Serializer The serializer was tested using a number of sample XML files that made use of all the entities that could be used in a Gizmoball XML file. Both valid and invalid XML files were used. In addition, the supplied example XML file was used to verify that the loaded configuration matched what was specified.

Usability Testing Since our design introduces some non-traditional user interface elements such as frobbers, multiple user tests were performed in which different users were prompted to simulate a game, and perform various editing tasks.

3.3 Test Results

All unit tests successfully passed. Integration testing was successful.

Usability testing identified several problems with the frobber menu. The users indicated that they preferred click-and-drag behavior for the frobbers (e.g. move and resize), rather than the modal click-source then click-destination behavior. Also, they preferred the rotation frobber action to be relative to the initial position of the gizmo, not absolute. These changes were made in response to the user input.

4 Reflection

4.1 Evaluation

We now reflect on our project, and our project reflects back.

Our greatest success was our ability to maintain the generality, dynamic polymorphism, and decoupling that we set out to achieve with our design philosophy. The property system proved invaluable in many aspects of the system, whereas the alternative of managing properties at compile-time would have greatly complicated much of our code and led to a great deal of coupling. The interaction system and abstract gizmo proved highly general, able to easily capture the semantics of the Gizmoball standard gizmos (and, we believe, all of the gizmos that would be necessary for many other diverse systems).

However, not all of our components were able to maintain our goal of component simplicity. The game board, for example, succeeds in presenting a simple interface, but its inner workings are very complex. The user interface also grew into a relatively hairy mass once within the abstractions we defined in our overall design (the property table is particularly notable for its hairiness). However, despite this, we still maintained a great deal of isolation between our modules, thus keeping the hairiness of these modules well contained.

Our quest for generality occasionally led us down stray paths. The greatest example of this was our original collision system, which was capable of expressing any collision pair in a way that allowed a gizmo's collision abilities to be deduced without it ever actually being explicitly declared. However, this meant that it was also non-obvious to us what was going on in the collision system. Once this was replaced with the new interaction system, it became quite obvious which handlers took care of which gizmo interactions.

4.2 Lessons

If we were to do this project again, we would probably delve deeper into the design of the user interface from the beginning. As it was, we defined the interfaces between the public components of the user interface packages, but did not specify up-front how these would be designed internally. The rest of the components were simple enough that these public interfaces were sufficient to specify the design of the components; however, this was not the case with the user interface, which proved far more internally intricate than we had expected (partially because Swing lacked some functionality that we had assumed it would have).

4.3 Known Bugs and Limitations

Many of the known limitations arise from limitations of the underlying libraries. Particularly, the physics library only supports collisions between balls and other shapes (ie, it does not support arbitrary pairs of objects to be collided). As such, we can only model a subset of the conceivable Gizmoball physics (though it is sufficient for what we were trying to achieve with the standard gizmos).

There is also some limitation imposed by the use of properties because properties are often not useful unless the system using the properties understands the names of the properties. While the limitations of this are far less than they would have been without the generality of the properties system, it still forces some knowledge on the other systems.

A Project Plan

The project was completed according to the project plan in Table 1.

B Formats

Our XML format is a simple extension of the staff-specified XML format. All of the position-related properties that were specified as integers (including orientation) are stored as floats in our format because our game system is capable of handling arbitrary location and orientation. This still loads the standard format, therefore it is entirely backwards compatible.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:complexType name="ball">
 <xs:attribute name="name" type="xs:string" use="required" />
 <xs:attribute name="x" type="xs:float" use="required" />
 <xs:attribute name="y" type="xs:float" use="required" />
 <xs:attribute name="xVelocity" type="xs:float" use="required" />
 <xs:attribute name="yVelocity" type="xs:float" use="required" />
 <xs:anyAttribute />
</xs:complexType>

<xs:complexType name="squareBumper">
 <xs:attribute name="name" type="xs:string" use="required" />
 <xs:attribute name="x" type="xs:float" use="required" />
 <xs:attribute name="y" type="xs:float" use="required" />
 <xs:attribute name="orientation" type="xs:float" use="optional" />
 <xs:anyAttribute />
</xs:complexType>

<xs:complexType name="circleBumper">
 <xs:attribute name="name" type="xs:string" use="required" />
 <xs:attribute name="x" type="xs:float" use="required" />
 <xs:attribute name="y" type="xs:float" use="required" />
 <xs:anyAttribute />
</xs:complexType>
```

Task	Who	When	
Preliminary design		**4/13**	√
Detailed specs for XML loader	Austin	4/16	√
Detailed specs for game internals	Dan	4/16	√
Detailed specs for UI	Albert	4/16	√
Detailed specs for property framework	Austin	4/16	√
Agree on detailed specs		4/17	√
XML loader	Austin	4/23	√
Basic UI framework	Albert	4/23	√
Game system framework and trigger mechanism	Dan	4/23	√
Property framework	Austin	4/23	√
Player interaction mode	Albert	4/23	√
Standard gizmos [a]	Austin	4/25	√
Game loop and collision mechanism	Dan	4/25	√
$\mathbb{R}^2$ renderer/drawers [b]	Albert	4/25	√
Implement multi-ball collisions	Austin	4/27	√
Preliminary release		**4/27**	√
Plan and (retroactively) schedule amendment		4/28	√
Specs for collection properties	Austin	5/3	√
Collection properties	Austin	5/5	√
Editor collision detection	Dan	5/5	√
XML saver	Austin	5/7	√
Finish flippers	Austin	5/7	√
PolyGizmos and multi-ball absorber support	Austin	5/7	√
Editor display	Albert	5/7	√
Editor interaction mode/frobber menu	Albert	5/7	√
Property table	Dan	5/7	√
Implementation and Critique		**5/11**	√

[a] Modulo flippers collisions and ball-ball interactions
[b] Using just drawing primitives

Table 1: Timeline

```
<xs:complexType name="triangleBumper">
 <xs:attribute name="name" type="xs:string" use="required" />
 <xs:attribute name="x" type="xs:float" use="required" />
 <xs:attribute name="y" type="xs:float" use="required" />
 <xs:attribute name="orientation" type="xs:float" use="optional" />
 <xs:anyAttribute />
</xs:complexType>

<xs:complexType name="leftFlipper">
 <xs:attribute name="name" type="xs:string" use="required" />
 <xs:attribute name="x" type="xs:float" use="required" />
 <xs:attribute name="y" type="xs:float" use="required" />
 <xs:attribute name="orientation" type="xs:float" use="optional" />
 <xs:anyAttribute />
</xs:complexType>

<xs:complexType name="rightFlipper">
 <xs:attribute name="name" type="xs:string" use="required" />
 <xs:attribute name="x" type="xs:float" use="required" />
 <xs:attribute name="y" type="xs:float" use="required" />
 <xs:attribute name="orientation" type="xs:float" use="optional" />
 <xs:anyAttribute />
</xs:complexType>

<xs:complexType name="absorber">
 <xs:attribute name="name" type="xs:string" use="required" />
 <xs:attribute name="x" type="xs:float" use="required" />
 <xs:attribute name="y" type="xs:float" use="required" />
 <xs:attribute name="width" type="xs:float" use="required" />
 <xs:attribute name="height" type="xs:float" use="required" />
 <xs:anyAttribute />
</xs:complexType>


<xs:complexType name="gizmoVertex">
 <xs:attribute name="x" type="xs:float" use="required" />
 <xs:attribute name="y" type="xs:float" use="required" />
 <xs:anyAttribute />
</xs:complexType>


<xs:complexType name="polyGizmo">
 <xs:sequence>
  <xs:element name="gizmoVertex" type="gizmoVertex" minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="name" type="xs:string" use="required" />
 <xs:attribute name="x" type="xs:float" use="required" />
 <xs:attribute name="y" type="xs:float" use="required" />
 <xs:attribute name="width" type="xs:float" use="optional" />
 <xs:attribute name="height" type="xs:float" use="optional" />
 <xs:attribute name="orientation" type="xs:float" use="optional" />
 <xs:anyAttribute />
</xs:complexType>



<xs:simpleType name="keyDirection">
 <xs:restriction base="xs:string">
  <xs:enumeration value="down" />
  <xs:enumeration value="up" />
 </xs:restriction>
</xs:simpleType>


<xs:complexType name="connect">
 <xs:attribute name="sourceGizmo" type="xs:string" use="required" />
 <xs:attribute name="targetGizmo" type="xs:string" use="required" />
 <xs:anyAttribute />
</xs:complexType>

<xs:complexType name="keyConnect">
 <xs:attribute name="key" type="xs:integer" use="required" />
 <xs:attribute name="keyDirection" type="keyDirection" use="required" />
 <xs:attribute name="targetGizmo" type="xs:string" use="required" />
 <xs:anyAttribute />
</xs:complexType>



<xs:group name="allGizmos">
  <xs:choice>
   <xs:element name="squareBumper" type="squareBumper" />
   <xs:element name="circleBumper" type="circleBumper" />
   <xs:element name="triangleBumper" type="triangleBumper" />
   <xs:element name="leftFlipper" type="leftFlipper" />
   <xs:element name="rightFlipper" type="rightFlipper" />
   <xs:element name="absorber" type="absorber" />
```

```
    <xs:element name="polyGizmo" type="polyGizmo" />
   </xs:choice>
</xs:group>

<xs:group name="allConnections">
 <xs:choice>
  <xs:element name="connect" type="connect" minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="keyConnect" type="keyConnect" minOccurs="0" maxOccurs="unbounded" />
 </xs:choice>
</xs:group>

<xs:element name="board">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="ball" type="ball" minOccurs="0" maxOccurs="unbounded" />

   <xs:element name="gizmos">
    <xs:complexType>
     <xs:sequence>
      <xs:group ref="allGizmos" minOccurs="0" maxOccurs="unbounded" />
     </xs:sequence>
     <xs:anyAttribute />
    </xs:complexType>
   </xs:element>

   <xs:element name="connections">
    <xs:complexType>
     <xs:sequence>
      <xs:group ref="allConnections" minOccurs="0" maxOccurs="unbounded" />
     </xs:sequence>
     <xs:anyAttribute />
    </xs:complexType>
   </xs:element>

   <xs:any minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="friction1" type="xs:float" default=".025"/>
  <xs:attribute name="friction2" type="xs:float" default=".025"/>
  <xs:attribute name="gravity"   type="xs:float" default="25.0"/>
 </xs:complexType>
</xs:element>

</xs:schema>
```

References

[1] R. J. Rubey, “The pasta theory of software,” *Crosstalk, Journal of Defense Software Engineering*, 1992, available at http://www.gnu.org/fun/jokes/pasta.code.html.

[2] J. Bergin and R. Winder, “Understanding object oriented programming,” Pace University, New York City, Tech. Rep., 2000, available at http://csis.pace.edu/ bergin/patterns/ppoop.html.

[3] D. M. Ritchie, “You are not expected to understand this,” in Odd Comments and Strange Doings in Unix, available at http://cm.bell-labs.com/cm/cs/who/dmr/odd.html.