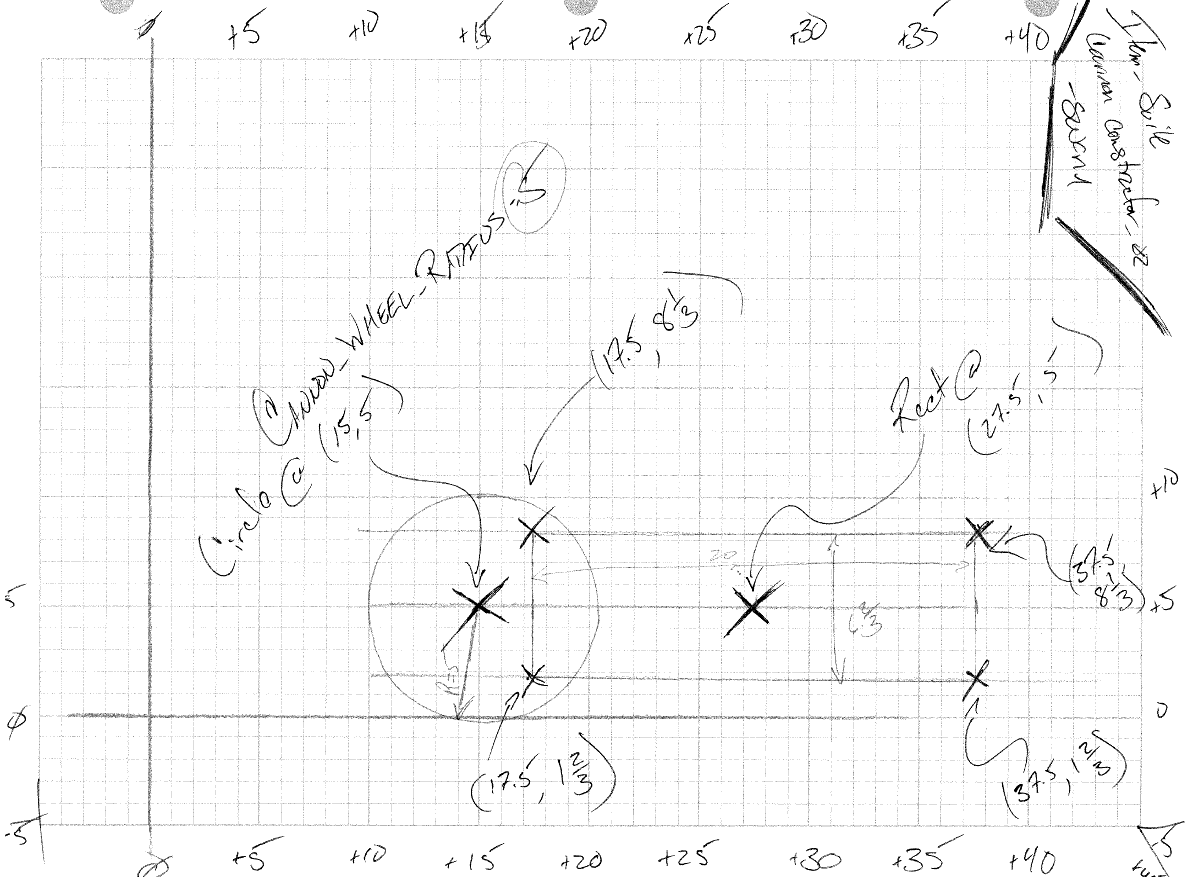
|  |
| --- |
|  |
| TEST |
|  |
| DESCRIPTION |

module :: Item/Cannon.java

test/design :: test\_cannon\_10\_02\_00\_constructor\_equivalence

description :: cannon class design and constructor verification



test notes :: cannon size values changed from initial sketch; sketch not re-drawn to reflect amendments to test

// Cannon constructor equivalence { part 2 }

@Test

**public** **void** test\_cannon\_10\_02\_00\_constructor\_equivalence() {

// Cannon object

Cannon the\_new\_cannon = **new** Cannon(15, *FIVE*, *ZERO*);

// Composite Shape

Shape the\_shape = (Shape) the\_new\_cannon.get\_shape();

// Composite List

ArrayList<Shape> the\_list = the\_shape.get\_composite\_list();

// Admin arrays

**float**[] tracking\_one, tracking\_two;

// Internal shape iterator

Shape inner\_shape = the\_list.get(*ZERO*);

// Non-assert testing --> same effect different angle of attack

**if** ( ! (inner\_shape **instanceof** Circle) ) {

*fail*(" test\_cannon\_10\_02\_00\_constructor\_equivalence --> First shape should be a circle");

}

tracking\_one = inner\_shape.get\_location();

**if** ( ! ( tracking\_one[*ZERO*] == 15 ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> Circle X-coord");

}

**if** ( ! ( tracking\_one[*ONE*] == 5 ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> Circle Y-coord");

}

**if** ( ! ( tracking\_one[*TWO*] == *ZERO* ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> Circle Z-coord");

}

inner\_shape = the\_list.get(*ONE*);

**if** ( ! (inner\_shape **instanceof** Rectangle) ) {

*fail*(" test\_cannon\_10\_02\_00\_constructor\_equivalence --> Second shape should be a rectangle");

}

Node head = inner\_shape.get\_head\_point();

tracking\_two = head.get\_location();

*assertEquals*(tracking\_two[*ZERO*], 127.5, *A\_THOUSANDTH*);

*assertEquals*(tracking\_two[*ONE*], 21.667, *A\_THOUSANDTH*);

*assertEquals*(tracking\_two[*TWO*], *ZERO*, *A\_THOUSANDTH*);

head = head.get\_next();

tracking\_two = head.get\_location();

*assertEquals*(tracking\_two[*ZERO*], 127.5, *A\_THOUSANDTH*);

*assertEquals*(tracking\_two[*ONE*], -11.667, *A\_THOUSANDTH*);

*assertEquals*(tracking\_two[*TWO*], *ZERO*, *A\_THOUSANDTH*);

head = head.get\_next();

tracking\_two = head.get\_location();

*assertEquals*(tracking\_two[*ZERO*], 27.5, *A\_THOUSANDTH*);

*assertEquals*(tracking\_two[*ONE*], -11.667, *A\_THOUSANDTH*);

*assertEquals*(tracking\_two[*TWO*], *ZERO*, *A\_THOUSANDTH*);

head = head.get\_next();

tracking\_two = head.get\_location();

*assertEquals*(tracking\_two[*ZERO*], 27.5, *A\_THOUSANDTH*);

*assertEquals*(tracking\_two[*ONE*], 21.667, *A\_THOUSANDTH*);

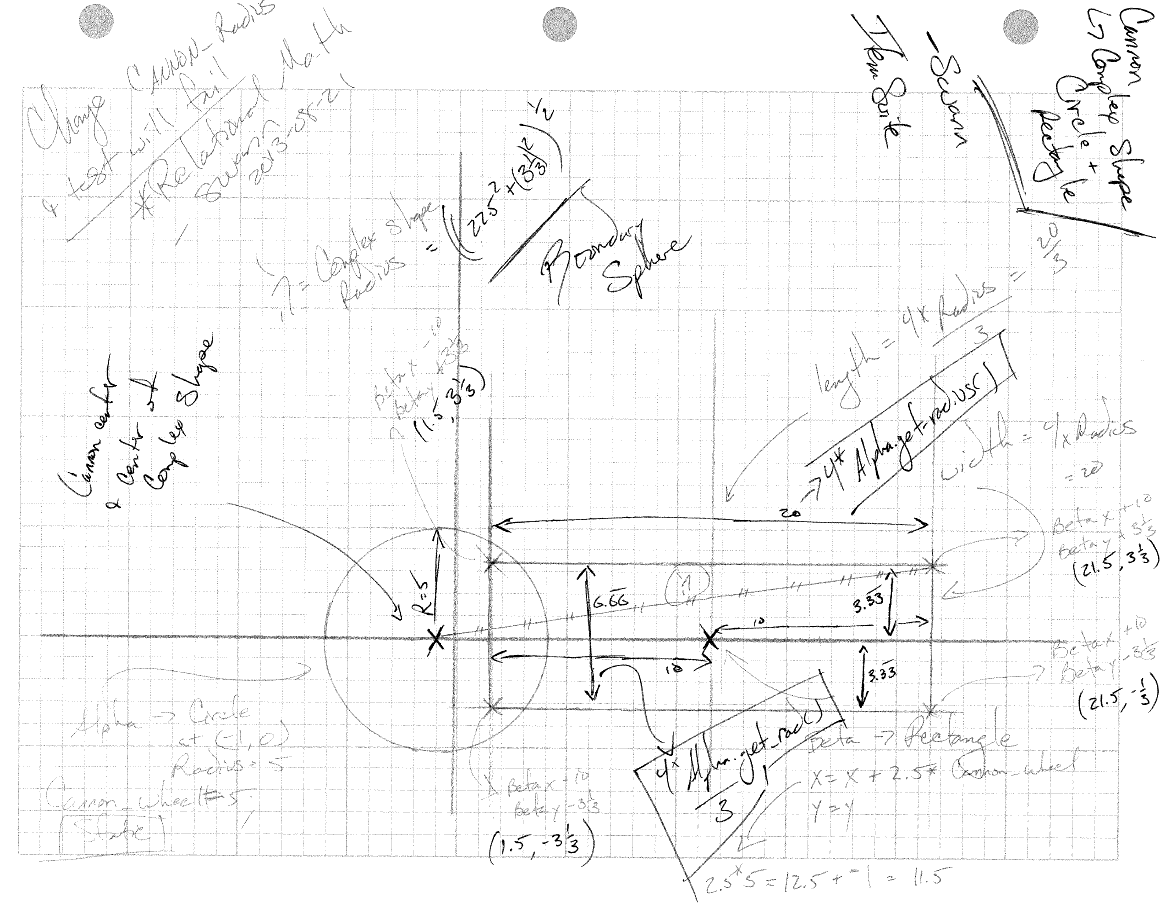
*assertEquals*(tracking\_two[*TWO*], *ZERO*, *A\_THOUSANDTH*);

}

module :: Item/Cannon.java

test/design :: test\_cannon\_10\_01\_00\_constructor\_equivalence

description :: cannon class design and relational math equations; text flexibility installation



test notes :: verification of relative mathematics on cannon aspect proportions

// Cannon constructor equivalence { part 1 }

@Test

**public** **void** test\_cannon\_10\_01\_00\_constructor\_equivalence() {

// Cannon object

Cannon the\_new\_cannon = **new** Cannon(*NEGATIVE\_ONE*, *ZERO*, *ZERO*);

// Composite Shape

Shape the\_shape = (Shape) the\_new\_cannon.get\_shape();

// Composite List

ArrayList<Shape> the\_list = the\_shape.get\_composite\_list();

// Admin arrays

**float**[] tracking\_one, tracking\_two;

// Internal shape iterator

Shape inner\_shape = the\_list.get(*ZERO*);

// Non-assert testing --> same effect different angle of attack

**if** ( ! (inner\_shape **instanceof** Circle) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> First shape should be a circle");

}

tracking\_one = inner\_shape.get\_location();

**if** ( ! ( tracking\_one[*ZERO*] == *NEGATIVE\_ONE* ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> Circle X-coord");

}

**if** ( ! ( tracking\_one[*ONE*] == *ZERO* ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> Circle Y-coord");

}

**if** ( ! ( tracking\_one[*TWO*] == *ZERO* ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> Circle Z-coord");

}

**float** radius = inner\_shape.get\_radius();

inner\_shape = the\_list.get(*ONE*);

**if** ( ! (inner\_shape **instanceof** Rectangle) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> Second shape should be a rectangle");

}

Node head = inner\_shape.get\_head\_point();

tracking\_two = head.get\_location();

**if** ( ! ( tracking\_two[*ZERO*] == tracking\_one[*ZERO*]+4.5\*radius ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point one X-coord");

}

**if** ( Math.*abs*(tracking\_two[*ONE*] - tracking\_one[*ONE*]+2/3\*radius) < *A\_HUNDREDTH* ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point one Y-coord");

}

**if** ( ! ( tracking\_two[*TWO*] == *ZERO* ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point one Z-coord");

}

head = head.get\_next();

tracking\_two = head.get\_location();

**if** ( ! ( tracking\_two[*ZERO*] == tracking\_one[*ZERO*]+4.5\*radius ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point two X-coord");

}

**if** ( Math.*abs*(tracking\_two[*ONE*] - tracking\_one[*ONE*]-2/3\*radius) < *A\_HUNDREDTH* ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point two Y-coord");

}

**if** ( ! ( tracking\_two[*TWO*] == *ZERO* ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point two Z-coord");

}

head = head.get\_next();

tracking\_two = head.get\_location();

**if** ( ! ( tracking\_two[*ZERO*] == tracking\_one[*ZERO*]+.5\*radius ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point three X-coord");

}

**if** ( Math.*abs*(tracking\_two[*ONE*] - tracking\_one[*ONE*]-2/3\*radius) < *A\_HUNDREDTH* ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point three Y-coord");

}

**if** ( ! ( tracking\_two[*TWO*] == *ZERO* ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point three Z-coord");

}

head = head.get\_next();

tracking\_two = head.get\_location();

**if** ( ! ( tracking\_two[*ZERO*] == tracking\_one[*ZERO*]+.5\*radius ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point four X-coord");

}

**if** ( Math.*abs*(tracking\_two[*ONE*] - tracking\_one[*ONE*]+2/3\*radius) < *A\_HUNDREDTH* ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point four Y-coord");

}

**if** ( ! ( tracking\_two[*TWO*] == *ZERO* ) ) {

*fail*(" test\_cannon\_10\_01\_00\_constructor\_equivalence --> point four Z-coord");

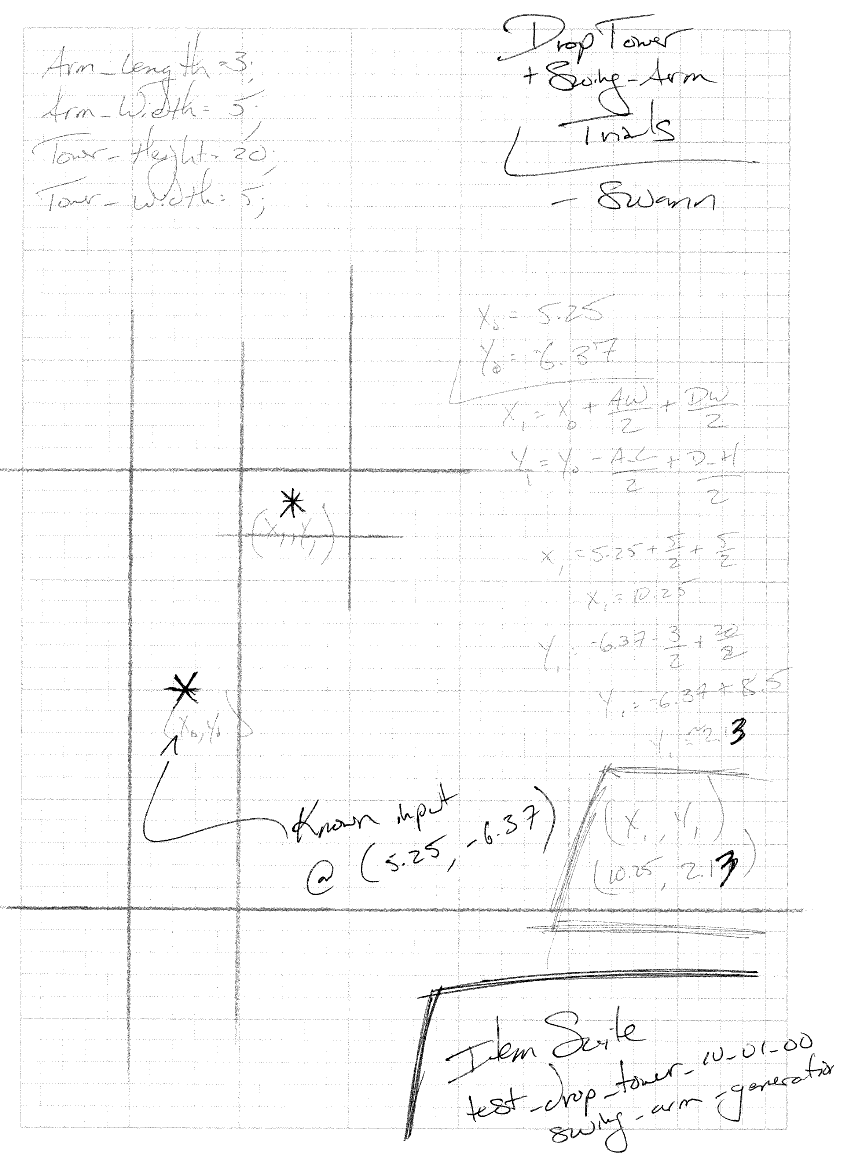
}

}

module :: Item/Drop\_Tower.java

test/design :: test\_drop\_tower\_10\_01\_00\_swing\_arm\_generation

description :: drop tower class design and constructor verification with generation of attached swing arm



test notes :: verification of non-relative mathematics on drop tower and related swing arm constructors

// DropTower.swing\_arm\_generation()

@Test

**public** **void** test\_drop\_tower\_10\_01\_00\_swing\_arm\_generation() {

// Battery One

DropTower tower = **new** DropTower(*TEN*, 15, *ZERO*);

SwingArm swing\_arm = tower.swing\_arm\_generation();

**float**[] location = swing\_arm.get\_location();

*assertEquals*(location[*ZERO*], (**float**) 15, *A\_THOUSANDTH*);

*assertEquals*(location[*ONE*], (**float**) 23.5, *A\_THOUSANDTH*);

*assertEquals*(location[*TWO*], (**float**) *ZERO*, *A\_THOUSANDTH*);

// Battery Two

tower = **new** DropTower((**float**) 5.25, (**float**) -6.37, *ZERO*);

swing\_arm = tower.swing\_arm\_generation();

location = swing\_arm.get\_location();

*assertEquals*(location[*ZERO*], (**float**) 10.25, *A\_THOUSANDTH*);

*assertEquals*(location[*ONE*], (**float**) 2.13, *A\_THOUSANDTH*);

*assertEquals*(location[*TWO*], (**float**) *ZERO*, *A\_THOUSANDTH*);

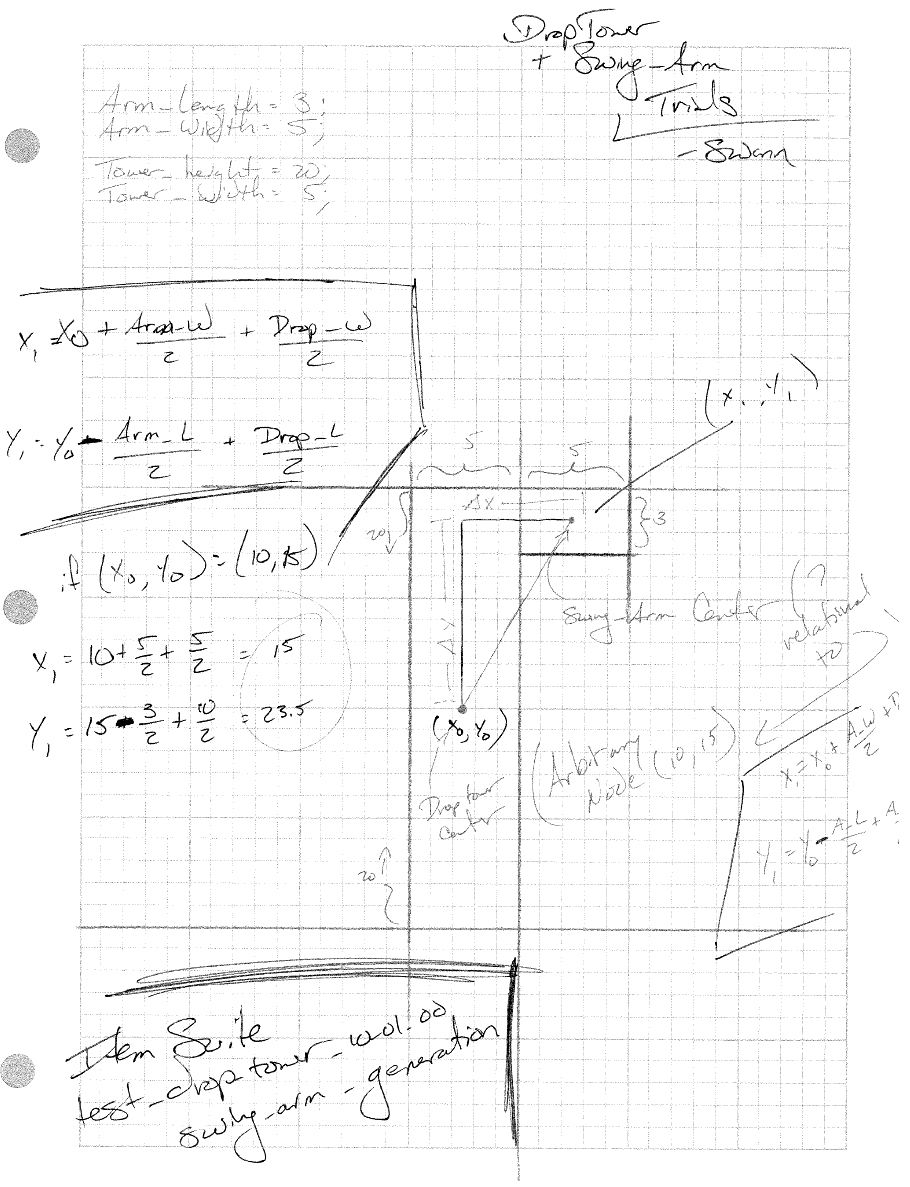
}

module :: Item/Drop\_Tower.java

test/design :: test\_drop\_tower\_10\_01\_00\_swing\_arm\_generation

function :: Drop\_Tower(x\_coord, y, coord, z\_coord) && Drop\_Tower.swing\_arm\_generation()

description :: relational math diagram of generated swing arm



function notes :: implementation of relative mathematics on drop tower and related swing arm

/\*\*

\* Basic constructor for the DropTower Class taking parameters for the x, y, and z

\* coordinates.

\*

\* Current 'z' parameter currently unused.

\* ~swann 2013-11-06

\*

\* **@param** x\_coordinate : Position on 'x' axis

\* **@param** y\_coordinate : Position on 'y' axis

\* **@param** z\_coordinate : Position on 'z' axis

\*/

**public** DropTower( **float** x\_coordinate, **float** y\_coordinate, **float** z\_coordinate ) {

**this**.set\_interactive( **false** );

**this**.shape = **new** Rectangle( x\_coordinate, y\_coordinate, z\_coordinate,

*TOWER\_HEIGHT*, *TOWER\_WIDTH*);

} // end DropTower()

/\*\*

\* Generates and returns a SwingArm object at a relative location to the current

\* DropTower.

\*

\* **@return** { SwingArm } : Object generated in relation to the space occupied by the

\* DropTower itself.

\*/

**public** SwingArm swing\_arm\_generation( ) {

**float**[] location = **this**.get\_location();

**float** x\_prime = (**float**) ( location[*ZERO*] +

( SwingArm.*ARM\_WIDTH*/(**float**)*TWO*) +

( DropTower.*TOWER\_WIDTH*/(**float**)*TWO*));

**float** y\_prime = (**float**) ( location[*ONE*] -

( SwingArm.*ARM\_LENGTH*/(**float**)*TWO*) +

( DropTower.*TOWER\_HEIGHT*/(**float**)*TWO*));

**float** z\_prime = *ZERO*;

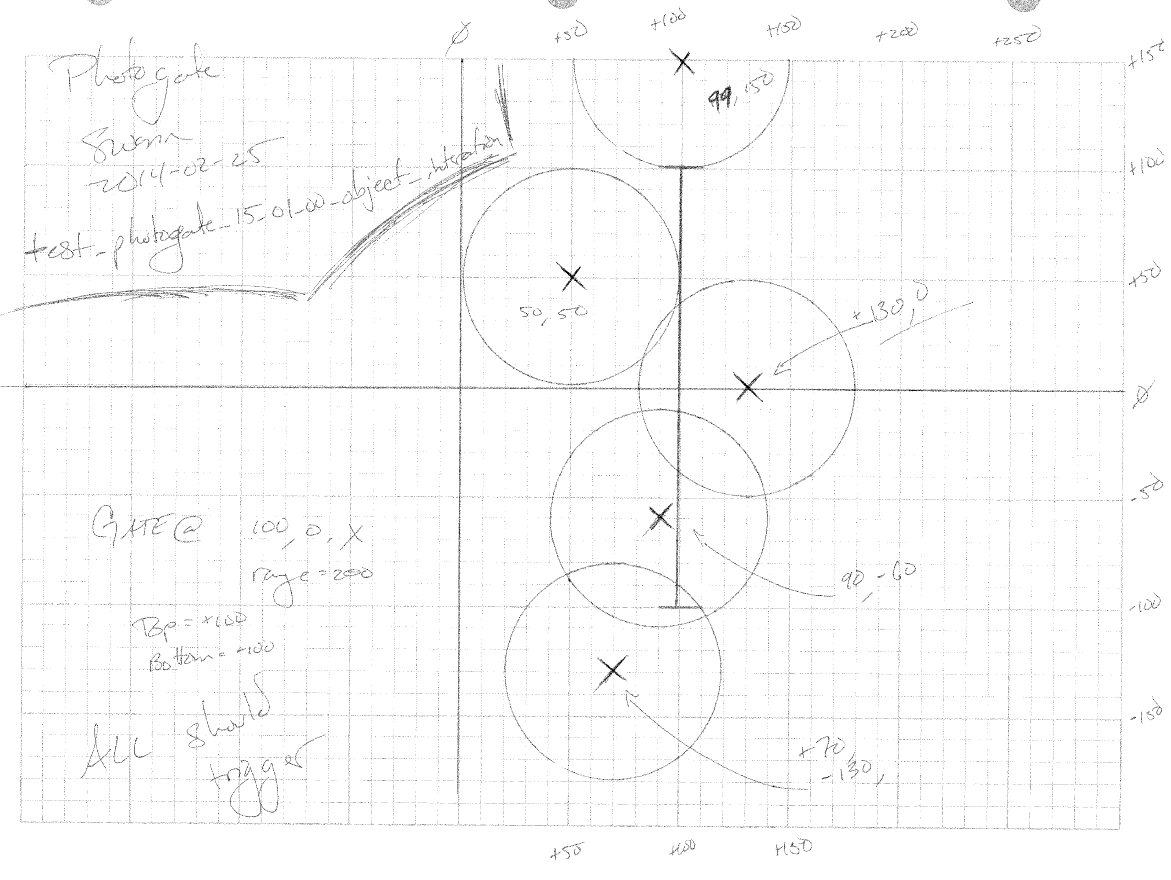
**return** **new** SwingArm( x\_prime, y\_prime, z\_prime );

} // end DropTower.swing\_arm\_generation()

module :: Item/Photogate.java

test/design :: test\_photogate\_15\_01\_00\_object\_interaction

description :: battery of checks on photogate vs circular-objects; each should trip the gate



test notes :: design of interaction of circular objects with the photogate class

// Photogate object interaction tests

@Test

**public** **void** test\_photogate\_15\_01\_00\_object\_interaction() {

// The Setup

Ball alpha = Ball.*generate\_Baseball*(99, 150, *ZERO*);

Ball beta = Ball.*generate\_Baseball*(50, 50, *ZERO*);

Ball delta = Ball.*generate\_Baseball*(130, 0, *ZERO*);

Ball gamma = Ball.*generate\_Baseball*(90, -60, *ZERO*);

Ball iota = Ball.*generate\_Baseball*(70, -130, *ZERO*);

Photogate gate\_one = **new** Photogate( 100, 0, *ZERO*,

100, Photogate.axis\_enumeration.*X*, 200);

// Each circle should trip the gate

*assertEquals*(gate\_one.is\_tripped\_by( alpha ), **true**);

*assertEquals*(gate\_one.is\_tripped\_by( beta ), **true**);

*assertEquals*(gate\_one.is\_tripped\_by( delta ), **true**);

*assertEquals*(gate\_one.is\_tripped\_by( gamma ), **true**);

*assertEquals*(gate\_one.is\_tripped\_by( iota ), **true**);

}

function notes :: implementation of related is\_tripped\_by( )

/\*\*

\* Returns true or false depending on whether or not the object tripped the gate by

\* entering the area of influence.

\*

\* **@param** { Actor\_Object } : Any Actor\_Object with a Shape

\*

\* **@return** { boolean } : True if the current placement of the Actor\_Object trips the gate.

\*/

**public** **boolean** is\_tripped\_by( Actor\_Object the\_object ) {

**boolean** output = **false**;

**switch** ( **this**.axis ) {

**case** *X*:

**if** ( the\_object.get\_y\_position() <= **this**.ranged\_axis\_top &&

the\_object.get\_y\_position() >= **this**.ranged\_axis\_bottom &&

( the\_object.shape.get\_high\_bound\_x\_projection() >= **this**.fixed\_axis\_value &&

the\_object.shape.get\_low\_bound\_x\_projection() <= **this**.fixed\_axis\_value )) {

output = **true**;

}

**else** {

Node top\_node = **new** Node(**this**.fixed\_axis\_value, **this**.ranged\_axis\_top, *ZERO*);

Node bot\_node = **new** Node(**this**.fixed\_axis\_value, **this**.ranged\_axis\_bottom, *ZERO*);

**if** ( Collision.*point\_inside\_shape*(top\_node, the\_object.get\_shape()) ) {

output = **true**;

}

**if** ( Collision.*point\_inside\_shape*(bot\_node, the\_object.get\_shape()) ) {

output = **true**;

}

}

**break**;

**case** *Y*:

**if** ( the\_object.get\_x\_position() <= **this**.ranged\_axis\_top &&

the\_object.get\_x\_position() >= **this**.ranged\_axis\_bottom &&

( the\_object.shape.get\_high\_bound\_y\_projection() >= **this**.fixed\_axis\_value &&

the\_object.shape.get\_low\_bound\_y\_projection() <= **this**.fixed\_axis\_value )) {

output = **true**;

}

**else** {

Node top\_node = **new** Node(**this**.ranged\_axis\_top, **this**.fixed\_axis\_value, *ZERO*);

Node bot\_node = **new** Node(**this**.ranged\_axis\_bottom, **this**.fixed\_axis\_value, *ZERO*);

**if** ( Collision.*point\_inside\_shape*(top\_node, the\_object.shape) ) {

output = **true**;

}

**if** ( Collision.*point\_inside\_shape*(bot\_node, the\_object.shape) ) {

output = **true**;

}

}

**break**;

} // end switch

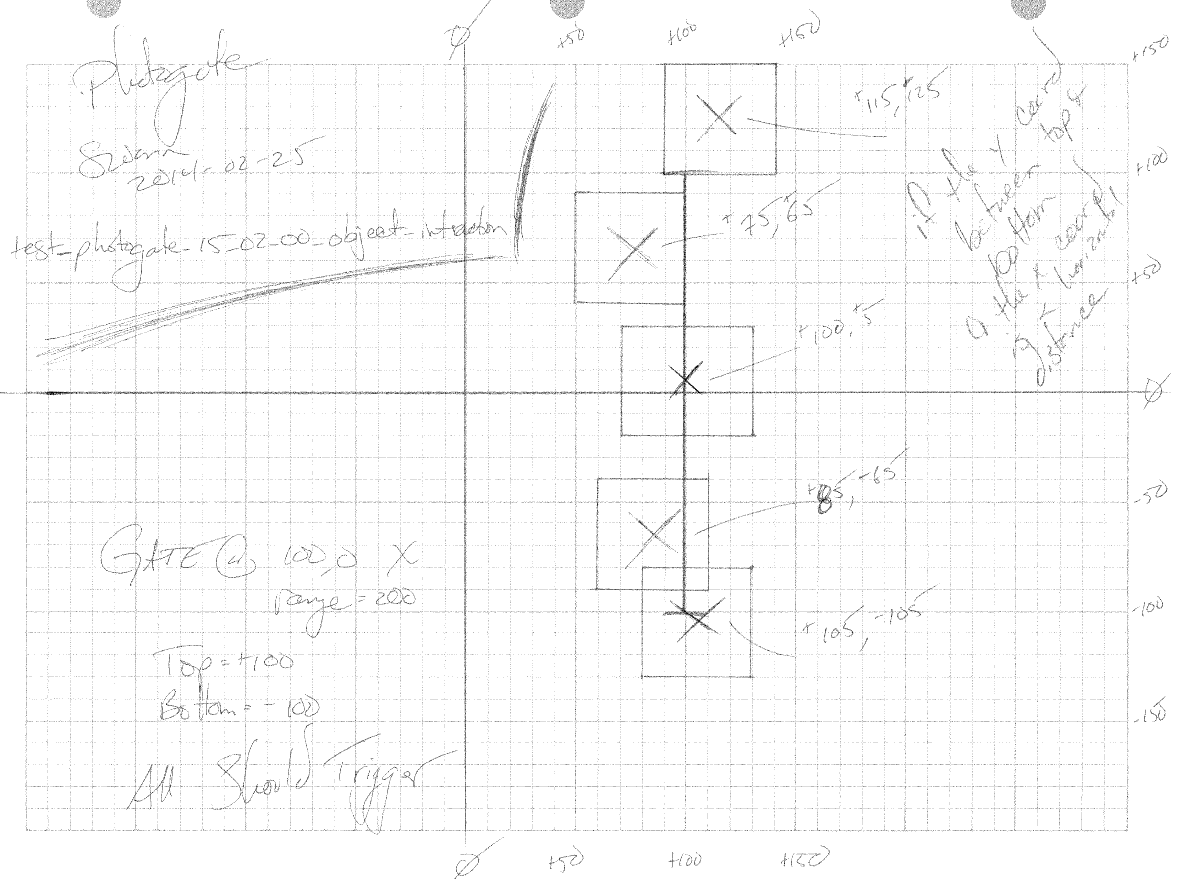
**return** output;

} // end Photogate.is\_tripped\_by()

module :: Item/Photogate.java

test/design :: test\_photogate\_15\_02\_00\_object\_interaction

description :: battery of checks on photogate vs polygon-objects; each should trip the gate



test notes :: design of interaction of polygon objects with the photogate class

// Photogate object interaction tests

@Test

**public** **void** test\_photogate\_15\_02\_00\_object\_interaction() {

// The Setup

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(115, 125, *ZERO*);

Standard\_Mass beta = Standard\_Mass.*generate\_fifty\_g\_mass*(75, 65, *ZERO*);

Standard\_Mass delta = Standard\_Mass.*generate\_fifty\_g\_mass*(100, 5, *ZERO*);

Standard\_Mass gamma = Standard\_Mass.*generate\_fifty\_g\_mass*(85, -65, *ZERO*);

Standard\_Mass iota = Standard\_Mass.*generate\_fifty\_g\_mass*(105,-105, *ZERO*);

Photogate gate\_one = **new** Photogate( 100, 0, *ZERO*,

100, Photogate.axis\_enumeration.*X*, 200);

// Each circle should trip the gate

*assertEquals*(gate\_one.is\_tripped\_by( alpha ), **true**);

*assertEquals*(gate\_one.is\_tripped\_by( beta ), **true**);

*assertEquals*(gate\_one.is\_tripped\_by( delta ), **true**);

*assertEquals*(gate\_one.is\_tripped\_by( gamma ), **true**);

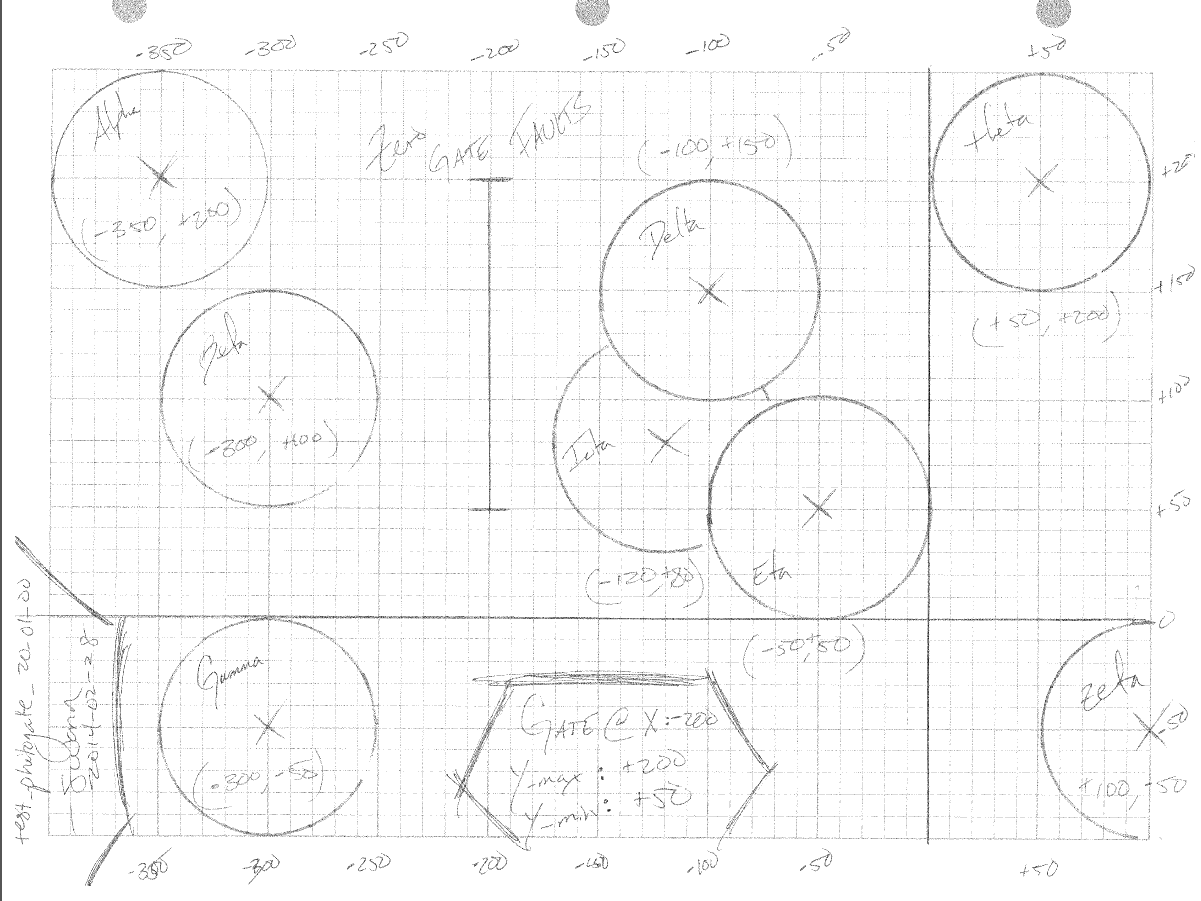
*assertEquals*(gate\_one.is\_tripped\_by( iota ), **true**);

}

module :: Item/Photogate.java

test/design :: test\_photogate\_15\_03\_00\_object\_interaction

description :: battery of checks on photogate objects; none should trip the gate



test notes :: designed for above\_list && below\_list tracking; doubled as a set of non-tripping objects

// Photogate object interaction tests

@Test

**public** **void** test\_photogate\_15\_03\_00\_object\_interaction() {

// The Setup

Ball alpha = Ball.*generate\_Baseball*( -350, 200, *ZERO* );

Ball beta = Ball.*generate\_Baseball*( -300, 100, *ZERO* );

Ball gamma = Ball.*generate\_Baseball*( -300, -50, *ZERO* );

Ball delta = Ball.*generate\_Baseball*( -100, 150, *ZERO* );

Ball iota = Ball.*generate\_Baseball*( -120, 80, *ZERO* );

Ball eta = Ball.*generate\_Baseball*( -50, 50, *ZERO* );

Ball theta = Ball.*generate\_Baseball*( 50, 200, *ZERO* );

Ball zeta = Ball.*generate\_Baseball*( 100, -50, *ZERO* );

Photogate the\_gate = **new** Photogate( -200, 125, *ZERO*,

-200, Photogate.axis\_enumeration.*X*, 150);

// Each circle should NOT trip the gate

*assertEquals*(the\_gate.is\_tripped\_by( alpha ), **false**);

*assertEquals*(the\_gate.is\_tripped\_by( beta ), **false**);

*assertEquals*(the\_gate.is\_tripped\_by( gamma ), **false**);

*assertEquals*(the\_gate.is\_tripped\_by( delta ), **false**);

*assertEquals*(the\_gate.is\_tripped\_by( iota ), **false**);

*assertEquals*(the\_gate.is\_tripped\_by( eta ), **false**);

*assertEquals*(the\_gate.is\_tripped\_by( theta ), **false**);

*assertEquals*(the\_gate.is\_tripped\_by( zeta ), **false**);

}

module :: Item/Photogate.java

test/design :: test\_photogate\_15\_04\_00\_object\_interaction

description :: battery of checks on photogate objects; none should trip the gate



test notes :: design of heterogeneous interaction with photogates; some gates tripped in static state, some not;

tracking of tripped/locked status along with the object responsible for the locked state

// Photogate object interaction tests

@Test

**public** **void** test\_photogate\_15\_04\_00\_object\_interaction() {

// The Setup

// Objects with simple objects

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(-190, 70, *ZERO*),

beta = Standard\_Mass.*generate\_fifty\_g\_mass*(-75, 75, *ZERO*),

gamma = Standard\_Mass.*generate\_fifty\_g\_mass*(-150, *ZERO*, *ZERO*);

Ball delta = Ball.*generate\_Baseball*(50, 50, *ZERO*),

eta = Ball.*generate\_Baseball*(110, 150, *ZERO*),

iota = Ball.*generate\_Baseball*(250, 70, *ZERO*);

alpha.set\_label("alpha");

beta.set\_label("beta");

gamma.set\_label("gamma");

delta.set\_label("delta");

eta.set\_label("eta");

iota.set\_label("iota");

ArrayList<Actor\_Object> interaction\_list = **new** ArrayList<Actor\_Object>();

interaction\_list.add(alpha);

interaction\_list.add(beta);

interaction\_list.add(gamma);

interaction\_list.add(delta);

interaction\_list.add(eta);

interaction\_list.add(iota);

// Gates

Photogate gate\_one = **new** Photogate( -175, 50, *ZERO*,

50, Photogate.axis\_enumeration.*Y*, 50),

gate\_two = **new** Photogate( -150, -50, *ZERO*,

-50, Photogate.axis\_enumeration.*Y*, 100),

gate\_tre = **new** Photogate( 125, 125, *ZERO*,

150, Photogate.axis\_enumeration.*X*, 150),

gate\_for = **new** Photogate( 200, 200, *ZERO*,

200, Photogate.axis\_enumeration.*X*, 100);

gate\_one.update\_information(interaction\_list, *ZERO*);

*assertEquals*(gate\_one.is\_locked(), **true**);

*assertEquals*(gate\_one.get\_object\_which\_tripped(), alpha);

gate\_two.update\_information(interaction\_list, *ZERO*);

*assertEquals*(gate\_two.is\_locked(), **false**);

*assertEquals*(gate\_two.get\_object\_which\_tripped(), **null**);

gate\_tre.update\_information(interaction\_list, *ZERO*);

*assertEquals*(gate\_tre.is\_locked(), **true**);

*assertEquals*(gate\_tre.get\_object\_which\_tripped(), eta);

gate\_for.update\_information(interaction\_list, *ZERO*);

*assertEquals*(gate\_for.is\_locked(), **false**);

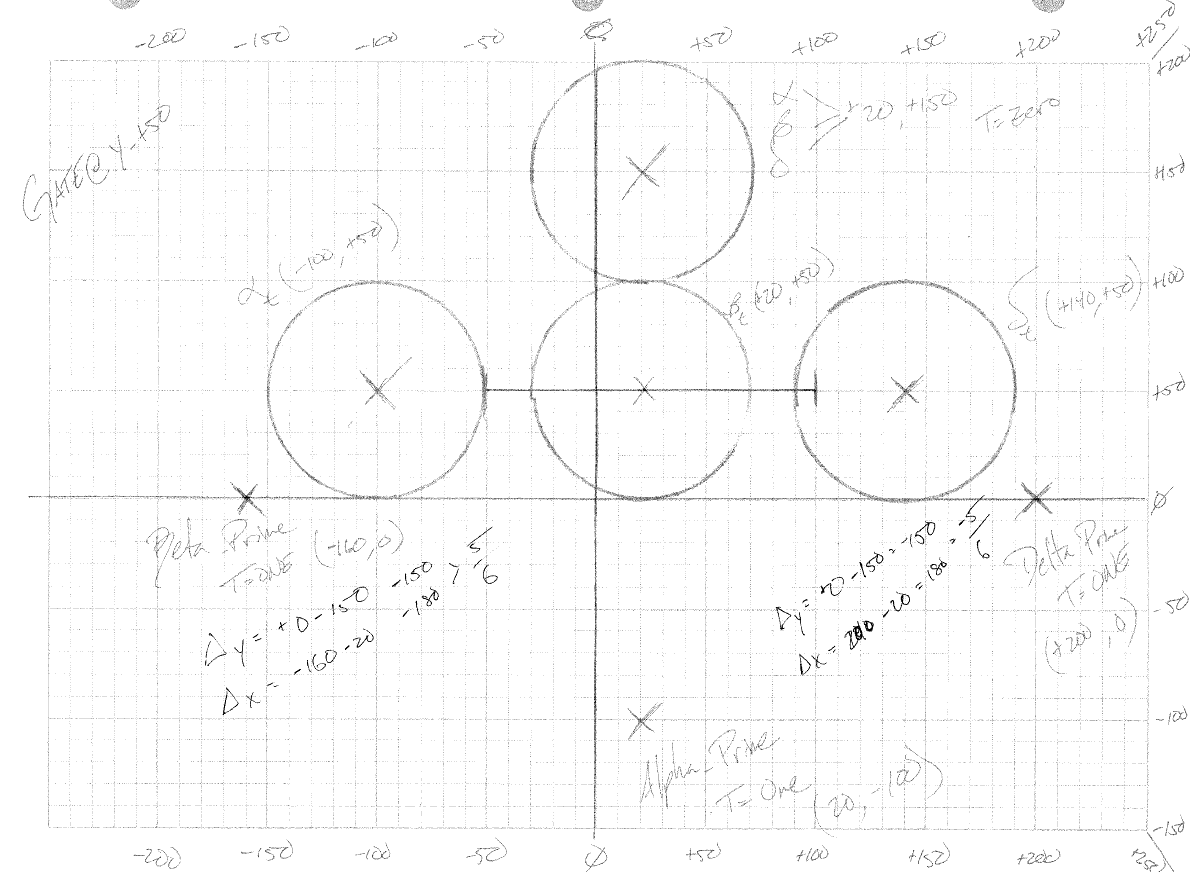
*assertEquals*(gate\_for.get\_object\_which\_tripped(), **null**);

}

module :: Item/Photogate.java

test/design :: test\_photogate\_15\_05\_00\_object\_interaction

description :: battery of checks on photogates with static state; beginning work on the ‘clipping algorithm’



test notes :: design of heterogeneous interaction with photogates; some gates tripped in static state, some not;

tracking of tripped/locked status along with the object responsible for the locked state

// Photogate object interaction tests

@Test

**public** **void** test\_photogate\_15\_05\_00\_object\_interaction() {

// The Setup

Ball alpha\_knot = Ball.*generate\_Baseball*( 20, 150, *ZERO* );

Ball beta\_knot = Ball.*generate\_Baseball*( 20, 150, *ZERO* );

Ball delta\_knot = Ball.*generate\_Baseball*( 20, 150, *ZERO* );

Ball alpha\_intra = Ball.*generate\_Baseball*((**float**)99.99999, 50, *ZERO* );

Ball beta\_intra = Ball.*generate\_Baseball*( 20, 50, *ZERO* );

Ball delta\_intra = Ball.*generate\_Baseball*( 140, 50, *ZERO* );

Ball alpha\_prime = Ball.*generate\_Baseball*( -160, 0, *ZERO* );

Ball beta\_prime = Ball.*generate\_Baseball*( 20, -100, *ZERO* );

Ball delta\_prime = Ball.*generate\_Baseball*( 200, 0, *ZERO* );

Photogate gate\_one = **new** Photogate( 25, 50, *ZERO*,

50, Photogate.axis\_enumeration.*Y*, 150);

Photogate gate\_two = **new** Photogate( 25, 50, *ZERO*,

50, Photogate.axis\_enumeration.*Y*, 150);

Photogate gate\_tre = **new** Photogate( 25, 50, *ZERO*,

50, Photogate.axis\_enumeration.*Y*, 150);

Photogate gate\_for = **new** Photogate( 25, 50, *ZERO*,

50, Photogate.axis\_enumeration.*Y*, 150);

Photogate gate\_fiv = **new** Photogate( 25, 50, *ZERO*,

50, Photogate.axis\_enumeration.*Y*, 150);

Photogate gate\_six = **new** Photogate( 25, 50, *ZERO*,

50, Photogate.axis\_enumeration.*Y*, 150);

Photogate gate\_svn = **new** Photogate( 25, 50, *ZERO*,

50, Photogate.axis\_enumeration.*Y*, 150);

Photogate gate\_ate = **new** Photogate( 25, 50, *ZERO*,

50, Photogate.axis\_enumeration.*Y*, 150);

Photogate gate\_nin = **new** Photogate( 25, 50, *ZERO*,

50, Photogate.axis\_enumeration.*Y*, 150);

// Each gate has a specific object that COULD trip it

// see designs for test\_photogate\_......................

// ~swann 2014-03-01

*assertEquals*(gate\_one.is\_tripped\_by( alpha\_knot ), **false**);

*assertEquals*(gate\_two.is\_tripped\_by( beta\_knot ), **false**);

*assertEquals*(gate\_tre.is\_tripped\_by( delta\_knot ), **false**);

*assertEquals*(gate\_for.is\_tripped\_by( alpha\_intra ), **true**);

*assertEquals*(gate\_fiv.is\_tripped\_by( beta\_intra ), **true**);

*assertEquals*(gate\_six.is\_tripped\_by( delta\_intra ), **true**);

*assertEquals*(gate\_svn.is\_tripped\_by( alpha\_prime ), **false**);

*assertEquals*(gate\_ate.is\_tripped\_by( beta\_prime ), **false**);

*assertEquals*(gate\_nin.is\_tripped\_by( delta\_prime ), **false**);

}

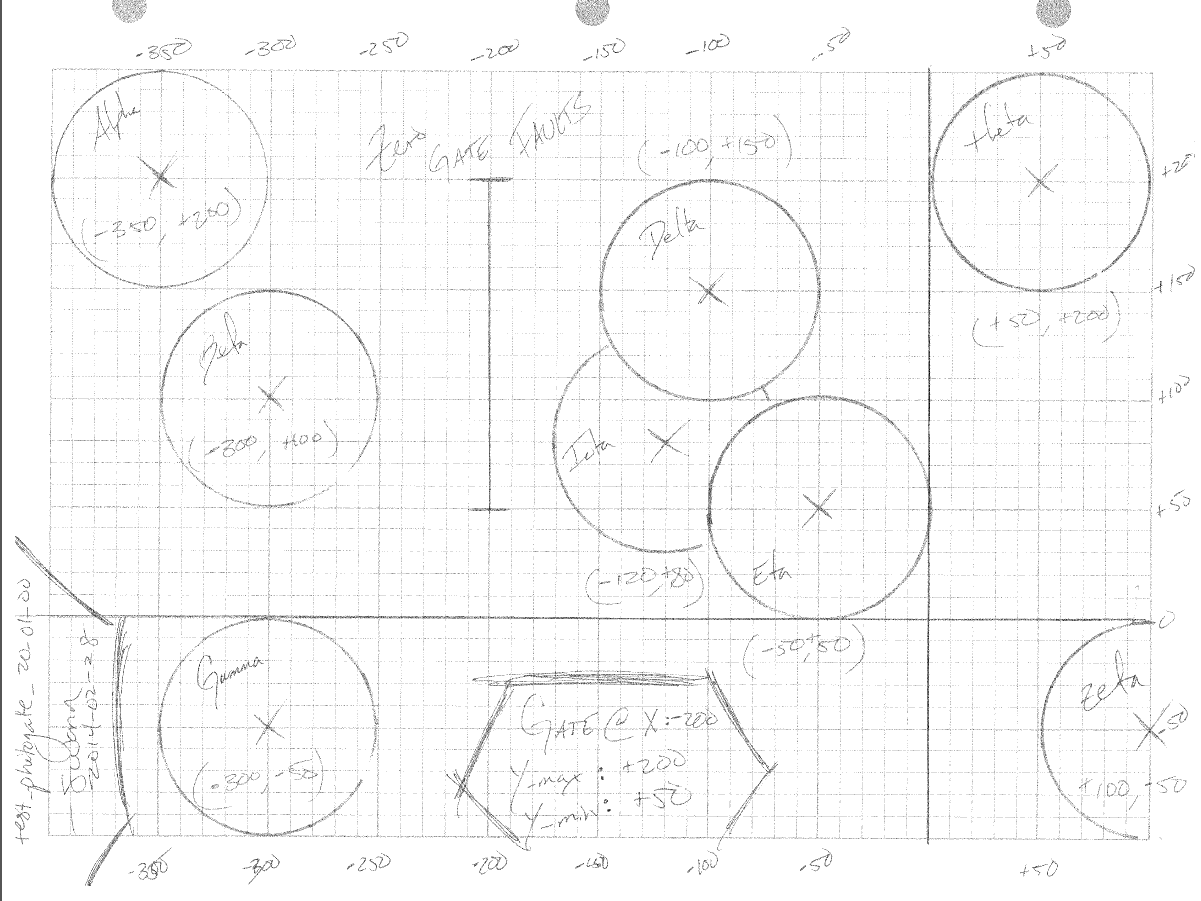
module :: Item/Photogate.java

test/design :: test\_photogate\_20\_01\_00\_populate\_new\_lists

functions :: Photogate.populate\_new\_above\_list() && Photogate.populate\_new\_below\_list() &&

Photogate.populate\_new\_swapped\_list()

description :: used to determine which objects are above or below the gate; to be used within the ‘clipping algorithm’



test notes :: algorithms used to determine which objects have switched sides in relation to the photogate; to be used as

support functionality in connection to the ‘clipping algorithm’ which determines whether an object passed

over a gate tripping it between time-steps and if multiple objects did so, which was first to trip the gate

// Photogate.populate\_new\_lists()

// --> Photogate.populate\_new\_above\_list()

// --> Photogate.populate\_new\_below\_list()

// --> Photogate.produce\_swapped\_list()

@Test

**public** **void** test\_photogate\_20\_01\_00\_populate\_new\_lists() {

// The Setup

Ball alpha = Ball.*generate\_Baseball*( -350, 200, *ZERO* );

Ball beta = Ball.*generate\_Baseball*( -300, 100, *ZERO* );

Ball gamma = Ball.*generate\_Baseball*( -300, -50, *ZERO* );

Ball delta = Ball.*generate\_Baseball*( -100, 150, *ZERO* );

Ball iota = Ball.*generate\_Baseball*( -120, 80, *ZERO* );

Ball eta = Ball.*generate\_Baseball*( -50, 50, *ZERO* );

Ball theta = Ball.*generate\_Baseball*( 50, 200, *ZERO* );

Ball zeta = Ball.*generate\_Baseball*( 100, -50, *ZERO* );

ArrayList<Actor\_Object> swapped\_list = **new** ArrayList<Actor\_Object>();

ArrayList<Actor\_Object> above\_list\_one = **new** ArrayList<Actor\_Object>();

ArrayList<Actor\_Object> below\_list\_one = **new** ArrayList<Actor\_Object>();

ArrayList<Actor\_Object> above\_list\_two = **new** ArrayList<Actor\_Object>();

ArrayList<Actor\_Object> below\_list\_two = **new** ArrayList<Actor\_Object>();

ArrayList<Actor\_Object> interaction\_list = **new** ArrayList<Actor\_Object>();

Photogate the\_gate = **new** Photogate( -200, 125, *ZERO*,

-200, Photogate.axis\_enumeration.*X*, 150);

alpha.set\_label("alpha");

beta.set\_label("beta");

gamma.set\_label("gamma");

delta.set\_label("delta");

eta.set\_label("eta");

iota.set\_label("iota");

theta.set\_label("theta");

zeta.set\_label("zeta");

interaction\_list.add( alpha );

interaction\_list.add( beta );

interaction\_list.add( gamma );

interaction\_list.add( delta );

interaction\_list.add( iota );

interaction\_list.add( eta );

interaction\_list.add( theta );

interaction\_list.add( zeta );

*assertEquals*(interaction\_list.size(), *EIGHT*, *ZERO*);

*assertEquals*(below\_list\_one.size(), *ZERO*, *ZERO*);

*assertEquals*(above\_list\_one.size(), *ZERO*, *ZERO*);

// Initial State Assertions

below\_list\_one = the\_gate.populate\_new\_below\_list(interaction\_list);

*assertEquals*(below\_list\_one.size(), *THREE*, *ZERO*);

above\_list\_one = the\_gate.populate\_new\_above\_list(interaction\_list);

*assertEquals*(above\_list\_one.size(), *FIVE*, *ZERO*);

// Secondary State Assertions

alpha.update\_location(350, 200, *ZERO*);

below\_list\_two = the\_gate.populate\_new\_below\_list(interaction\_list);

*assertEquals*(below\_list\_two.size(), *TWO*, *ZERO*);

above\_list\_two = the\_gate.populate\_new\_above\_list(interaction\_list);

*assertEquals*(above\_list\_two.size(), *SIX*, *ZERO*);

swapped\_list = the\_gate.produce\_swapped\_list( above\_list\_one, below\_list\_one,

above\_list\_two, below\_list\_two);

*assertEquals*(swapped\_list.size(), *ONE*, *ZERO*);

**if** ( !swapped\_list.contains( alpha )) {

*fail*("Secondary State Assertions");

}

// Tertiary State Assertions

theta.update\_location(-400, 200, *ZERO*);

zeta.update\_location( -500, -50, *ZERO*);

below\_list\_one = the\_gate.populate\_new\_below\_list(interaction\_list);

*assertEquals*(below\_list\_one.size(), *FOUR*, *ZERO*);

above\_list\_one = the\_gate.populate\_new\_above\_list(interaction\_list);

*assertEquals*(above\_list\_one.size(), *FOUR*, *ZERO*);

swapped\_list = the\_gate.produce\_swapped\_list( above\_list\_two, below\_list\_two,

above\_list\_one, below\_list\_one);

*assertEquals*(swapped\_list.size(), *TWO*, *ZERO*);

**if** ( !swapped\_list.contains( theta ) && !swapped\_list.contains( zeta )) {

*fail*("Tertiary State Assertions");

}

// Return to Initial State and Re-Assert

alpha.update\_location(-350, 200, *ZERO*);

theta.update\_location( 50, 200, *ZERO*);

zeta.update\_location( 100, 50, *ZERO*);

below\_list\_two = the\_gate.populate\_new\_below\_list(interaction\_list);

*assertEquals*(below\_list\_two.size(), *THREE*, *ZERO*);

above\_list\_two = the\_gate.populate\_new\_above\_list(interaction\_list);

*assertEquals*(above\_list\_two.size(), *FIVE*, *ZERO*);

swapped\_list = the\_gate.produce\_swapped\_list( above\_list\_one, below\_list\_one,

above\_list\_two, below\_list\_two);

*assertEquals*(swapped\_list.size(), *THREE*, *ZERO*);

**if** ( !swapped\_list.contains( theta ) && !swapped\_list.contains( zeta ) &&

!swapped\_list.contains( alpha )) {

*fail*("Return to Initial State and Re-Assert");

}

}

function notes :: implementation of the populate\_new\_below\_list( )

/\*\*

\* Returns an ArrayList with the objects below the fixed point axis of the current

\* photogate.

\*

\* **@param** { ArrayList<Actor\_Object> } : List of Actor Objects from the working set.

\*

\* **@return** { ArrayList<Actor\_Object> } : List of Actor Objects below the fixed point axis of the

\* photogate.

\*/

**public** ArrayList<Actor\_Object> populate\_new\_below\_list( ArrayList<Actor\_Object> interaction\_list ){

ArrayList<Actor\_Object> below\_list = **new** ArrayList<Actor\_Object>();

**for** ( **int** i = *ZERO*; i < interaction\_list.size() ; i ++ ) {

Actor\_Object the\_object = interaction\_list.get(i);

**switch** ( **this**.axis ) {

**case** *X*:

// if the x-coord is below the fixed axis value add to below list,

// otherwise put it on the above list

**if** ( the\_object.get\_x\_position() < **this**.fixed\_axis\_value ) {

below\_list.add( the\_object );

}

**break**;

**case** *Y*:

// if the y-coord is below the fixed axis value add to below list,

// otherwise put it on the above list

**if** ( the\_object.get\_y\_position() < **this**.fixed\_axis\_value ) {

below\_list.add( the\_object );

}

**break**;

} // end switch

} // end iteration

**return** below\_list;

}// end Photogate.populate\_new\_below\_list()

function notes :: implementation of the populate\_new\_above\_list( )

/\*\*

\* Returns an ArrayList with the objects above or equal to the fixed point axis of the current

\* photogate.

\*

\* **@param** { ArrayList<Actor\_Object> } : List of Actor Objects from the working set.

\*

\* **@return** { ArrayList<Actor\_Object> } : List of Actor Objects above or equal to the fixed point

\* axis of the photogate.

\*/

**public** ArrayList<Actor\_Object> populate\_new\_above\_list( ArrayList<Actor\_Object> interaction\_list ){

ArrayList<Actor\_Object> above\_list = **new** ArrayList<Actor\_Object>();

**for** ( **int** i = *ZERO*; i < interaction\_list.size() ; i ++ ) {

Actor\_Object the\_object = interaction\_list.get(i);

**switch** ( **this**.axis ) {

**case** *X*:

// if the x-coord is below the fixed axis value add to below list,

// otherwise put it on the above list

**if** ( the\_object.get\_x\_position() >= **this**.fixed\_axis\_value ) {

above\_list.add( the\_object );

}

**break**;

**case** *Y*:

// if the y-coord is below the fixed axis value add to below list,

// otherwise put it on the above list

**if** ( the\_object.get\_y\_position() >= **this**.fixed\_axis\_value ) {

above\_list.add( the\_object );

}

**break**;

} // end switch

} // end iteration

**return** above\_list;

}// end Photogate.populate\_new\_above\_list()

function notes :: implementation of the populate\_swapped\_list( )

/\*\*

\* Returns an array list of Actor Object that have swapped from a relative high or low position

\* in regards to the photogate to the other designation. For instance, if an object was initially

\* below the fixed axis point and a time step moved it to a position where it is now above that

\* fixed point axis, then the object will appear in the swapped list.

\*

\* **@param** { old\_above\_list } : List of actor objects above the fixed point axis before the time step.

\* **@param** { old\_below\_list } : List of actor objects below the fixed point axis before the time step.

\* **@param** { new\_above\_list } : List of actor objects above the fixed point axis after the time step.

\* **@param** { new\_below\_list } : List of actor objects below the fixed point axis after the time step.

\*

\* **@return** { ArrayList<Actor\_Object> } : List of objects which have crossed the fixed point axis plane.

\*/

**public** ArrayList<Actor\_Object> produce\_swapped\_list( ArrayList<Actor\_Object> old\_above\_list,

ArrayList<Actor\_Object> old\_below\_list,

ArrayList<Actor\_Object> new\_above\_list,

ArrayList<Actor\_Object> new\_below\_list) {

ArrayList<Actor\_Object> swapped\_list = **new** ArrayList<Actor\_Object>();

// Check for objects going from above to below

**for** ( **int** i = *ZERO* ; i < old\_above\_list.size(); i ++) {

**if** ( new\_below\_list.contains( old\_above\_list.get(i) )) {

swapped\_list.add( old\_above\_list.get(i) );

}

} // end new\_below\_list iteration

// Check for objects going from below to above

**for** ( **int** i = *ZERO* ; i < old\_below\_list.size(); i ++) {

**if** ( new\_above\_list.contains( old\_below\_list.get(i) )) {

swapped\_list.add( old\_below\_list.get(i) );

}

} // end new\_below\_list iteration

**return** swapped\_list;

} // end Photogate.produce\_swapped\_list()

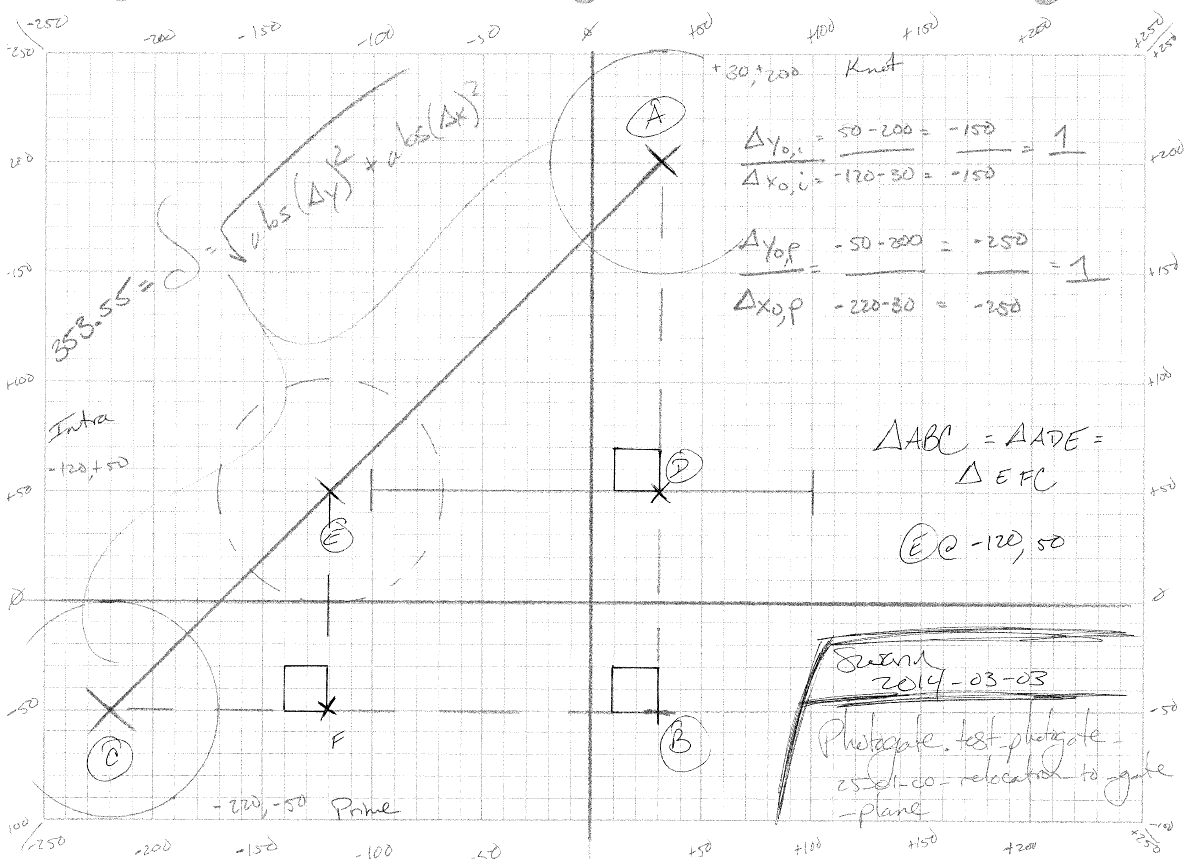
module :: Item/Photogate.java

test/design :: test\_photogate\_25\_01\_00\_relocation\_to\_gate\_plane

functions :: Photogate.relocation\_to\_gate\_plane()

description :: design of functionality scripted to return an object’s state to the point when it passed the gate’s plane;

then call of static state check on interaction with the gate itself



test notes :: object travels from point A to C in single time-step; should trip the gate as path includes location E

// Photogate.relocation\_to\_gate\_plane()

@Test

**public** **void** test\_photogate\_25\_01\_00\_relocation\_to\_gate\_plane() {

// The Setup

Ball alpha = Ball.*generate\_Baseball*(-220, -50, *ZERO*);

ArrayList<Actor\_Object> interaction\_list = **new** ArrayList<Actor\_Object>();

interaction\_list.add( alpha );

Photogate the\_gate = **new** Photogate( *ZERO*, 50, *ZERO*,

50, Photogate.axis\_enumeration.*Y*, 200);

// Pack velocity to full second

alpha.update\_velocity(-250\**TIME\_STEPS*, -250\**TIME\_STEPS*, *ZERO*);

// relocation

the\_gate.relocation\_to\_gate\_plane( alpha );

*assertEquals*(the\_gate.is\_tripped\_by( alpha ), **true**);

// intra-execute test peek

the\_gate.update\_information(interaction\_list, *ZERO*);

*assertEquals*(the\_gate.get\_object\_which\_tripped(), alpha);

}

function notes :: implementation of relocation\_to\_gate\_plane()

/\*\*

\* Repositions the passed actor object to the axis plan on which the gate itself exists.

\*

\* Nota Bene ::

\* This is only to be used with objects having been flagged and added to the swapped item list.

\* ~swann 2014-03-03

\*

\* **@param** { Actor\_Object } : The object being re-positioned.

\*/

**public** **void** relocation\_to\_gate\_plane( Actor\_Object the\_object ) {

// Establish velocity vector for intra-step calculations

Vector object\_velocity = **new** Vector( the\_object.get\_velocity() );

Vector time\_step\_velocity = Vector.*scalar\_multiply*(object\_velocity, *TIME\_STEP*);

// tracking variables

**float** x\_differential, y\_differential, proportion,

x\_position, y\_position;

// axis dependent relative math

**switch**( **this**.axis ){

**case** *X*:

x\_differential = Math.*abs*( the\_object.get\_x\_position() - **this**.fixed\_axis\_value );

proportion = x\_differential/Math.*abs*(time\_step\_velocity.get\_x\_comp());

y\_differential = proportion\*time\_step\_velocity.get\_y\_comp();

y\_position = the\_object.get\_y\_position() - y\_differential;

the\_object.update\_location(**this**.fixed\_axis\_value, y\_position, *ZERO*);

**break**;

**case** *Y*:

y\_differential = Math.*abs*( the\_object.get\_y\_position() - **this**.fixed\_axis\_value );

proportion = y\_differential/Math.*abs*(time\_step\_velocity.get\_y\_comp());

x\_differential = proportion\*time\_step\_velocity.get\_x\_comp();

x\_position = the\_object.get\_x\_position() - x\_differential;

the\_object.update\_location(x\_position, **this**.fixed\_axis\_value, *ZERO*);

**break**;

}

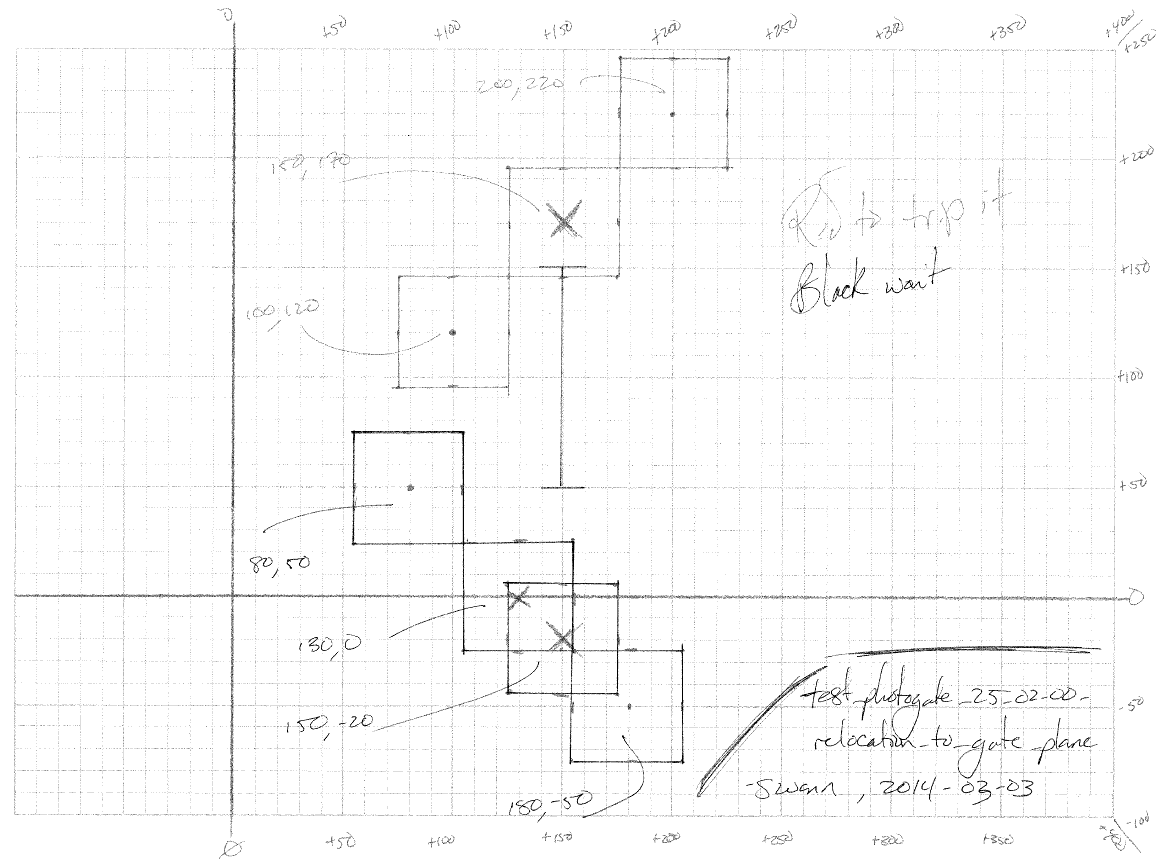
} // end Photogate.relocation\_to\_gate\_plane()

module :: Item/Photogate.java

test/design :: test\_photogate\_25\_02\_00\_relocation\_to\_gate\_plane

description :: design of functionality scripted to return an object’s state to the point when it passed the gate’s plane;

then call of static state check on interaction with the gate itself



test notes :: explores an object just barely tripping the gate and one passing by without tripping it

// Photogate.relocation\_to\_gate\_plane()

@Test

**public** **void** test\_photogate\_25\_02\_00\_relocation\_to\_gate\_plane() {

// The Setup

Standard\_Mass alpha = Standard\_Mass.*generate\_one\_g\_mass*( 180, -50, *ZERO*);

Standard\_Mass beta = Standard\_Mass.*generate\_five\_g\_mass*(200, 220, *ZERO*);

ArrayList<Actor\_Object> interaction\_list = **new** ArrayList<Actor\_Object>();

interaction\_list.add( alpha );

interaction\_list.add( beta );

Photogate the\_gate = **new** Photogate( 150, 100, *ZERO*,

150, Photogate.axis\_enumeration.*X*, 100);

// Pack velocity to full second

alpha.update\_velocity(100\**TIME\_STEPS*, -100\**TIME\_STEPS*, *ZERO*);

beta.update\_velocity( 100\**TIME\_STEPS*, 100\**TIME\_STEPS*, *ZERO*);

// relocation

the\_gate.relocation\_to\_gate\_plane( alpha );

*assertEquals*(the\_gate.is\_tripped\_by( alpha ), **false**);

*assertEquals*(alpha.get\_x\_position(), 150, *ZERO*);

*assertEquals*(alpha.get\_y\_position(), -20, *A\_THOUSANDTH*);

the\_gate.relocation\_to\_gate\_plane( beta );

*assertEquals*(the\_gate.is\_tripped\_by( beta ), **true**);

*assertEquals*(beta.get\_x\_position(), 150, *ZERO*);

*assertEquals*(beta.get\_y\_position(), 170, *ZERO*);

// intra-execute test peek

the\_gate.update\_information(interaction\_list, *ZERO*);

*assertEquals*(the\_gate.get\_object\_which\_tripped(), beta);

}

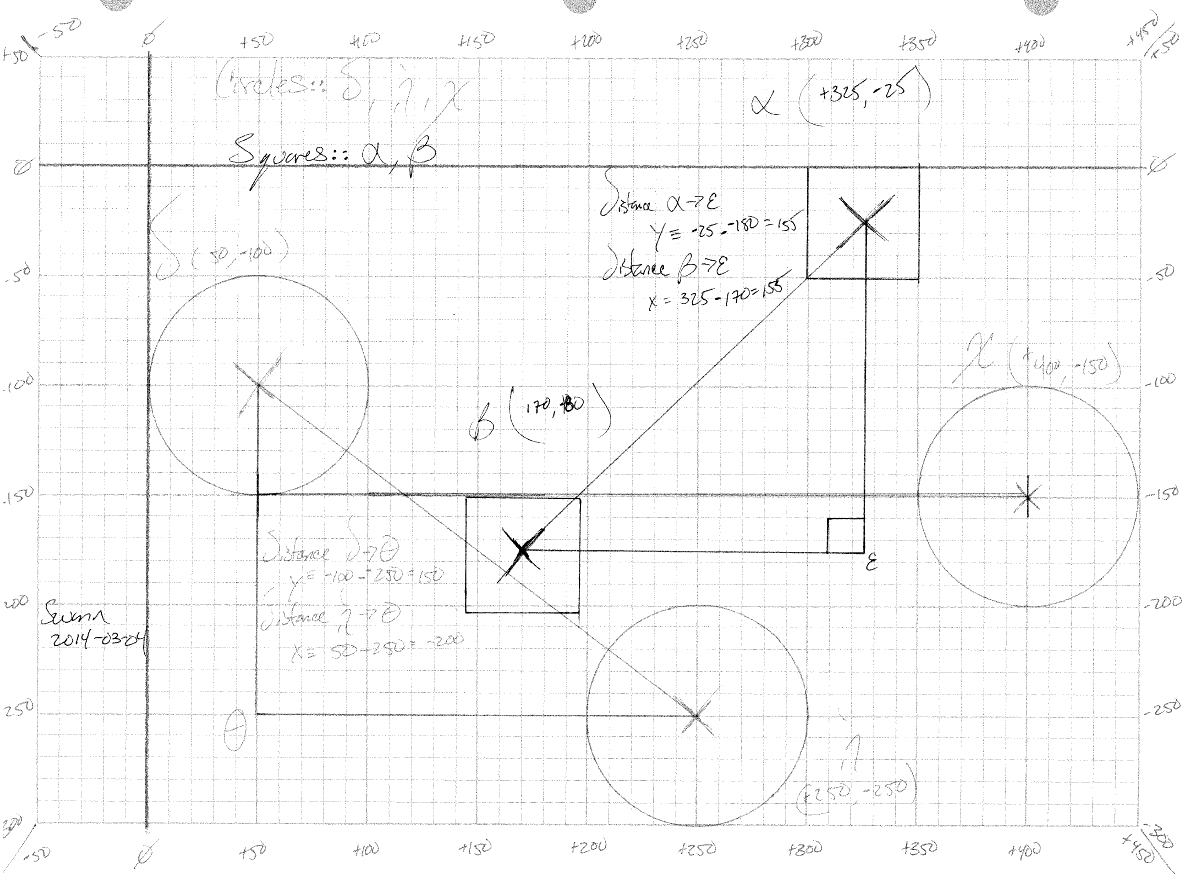
module :: Item/Photogate.java

test/design :: test\_photogate\_30\_01\_00\_correct\_trigger\_event && test\_photogate\_30\_02\_00\_correct\_trigger\_event

&& test\_photogate\_30\_03\_00\_correct\_trigger\_event

description :: design of the functionality to determine which object tripped the gate first; includes clipping and static

state objects



test notes :: explores full battery of use cases for photogate; clipping and static state objects with recognition as to

which object tripped it first

// Photogate test --> multiple objects pass through the gate, which one is first

@Test

**public** **void** test\_photogate\_30\_01\_00\_correct\_trigger\_event() {

// The Setup

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(325, -25, *ZERO*);

Ball delta = Ball.*generate\_Baseball*( 50, 99, *ZERO*);

Photogate the\_gate = **new** Photogate( 225, -150, *ZERO*,

-150, Photogate.axis\_enumeration.*Y*, 350);

ArrayList<Actor\_Object> interaction\_list = **new** ArrayList<Actor\_Object>();

interaction\_list.add( delta );

interaction\_list.add( alpha );

// Pack velocity to full second

alpha.update\_velocity( 155\**TIME\_STEPS*, 155\**TIME\_STEPS*, *ZERO*);

delta.update\_velocity(-200\**TIME\_STEPS*, 150\**TIME\_STEPS*, *ZERO*);

the\_gate.relocation\_to\_gate\_plane( alpha );

*assertEquals*(the\_gate.is\_tripped\_by( alpha ), **true**);

the\_gate.relocation\_to\_gate\_plane( delta );

*assertEquals*(the\_gate.is\_tripped\_by( delta ), **true**);

// intra-execute test peek

the\_gate.update\_information(interaction\_list, *ZERO*);

*assertEquals*(the\_gate.get\_object\_which\_tripped(), delta);

}

// Photogate test --> multiple objects pass through the gate, which one is first

@Test

**public** **void** test\_photogate\_30\_02\_00\_correct\_trigger\_event() {

// The Setup

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(325, -25, *ZERO*);

Ball delta = Ball.*generate\_Baseball*( 50, 99, *ZERO*);

Photogate the\_gate = **new** Photogate( 225, -150, *ZERO*,

-150, Photogate.axis\_enumeration.*Y*, 350);

ArrayList<Actor\_Object> interaction\_list = **new** ArrayList<Actor\_Object>();

interaction\_list.add( alpha );

interaction\_list.add( delta );

// Pack velocity to full second

alpha.update\_velocity( 155\**TIME\_STEPS*, 155\**TIME\_STEPS*, *ZERO*);

delta.update\_velocity(-200\**TIME\_STEPS*, 150\**TIME\_STEPS*, *ZERO*);

the\_gate.relocation\_to\_gate\_plane( alpha );

*assertEquals*(the\_gate.is\_tripped\_by( alpha ), **true**);

the\_gate.relocation\_to\_gate\_plane( delta );

*assertEquals*(the\_gate.is\_tripped\_by( delta ), **true**);

// intra-execute test peek

the\_gate.update\_information(interaction\_list, *ZERO*);

*assertEquals*(the\_gate.get\_object\_which\_tripped(), alpha);

}

// Photogate test --> multiple objects pass through the gate, which one is first

@Test

**public** **void** test\_photogate\_30\_03\_00\_correct\_trigger\_event() {

// The Setup

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(325, -25, *ZERO*);

Ball delta = Ball.*generate\_Baseball*( 50, 99, *ZERO*);

Ball chi = Ball.*generate\_Baseball*(400, -150, *ZERO*);

Photogate the\_gate = **new** Photogate( 225, -150, *ZERO*,

-150, Photogate.axis\_enumeration.*Y*, 350);

ArrayList<Actor\_Object> interaction\_list = **new** ArrayList<Actor\_Object>();

interaction\_list.add( alpha );

interaction\_list.add( delta );

interaction\_list.add( chi );

// Pack velocity to full second

alpha.update\_velocity( 155\**TIME\_STEPS*, 155\**TIME\_STEPS*, *ZERO*);

delta.update\_velocity(-200\**TIME\_STEPS*, 150\**TIME\_STEPS*, *ZERO*);

// intra-execute test peek

the\_gate.update\_information(interaction\_list, *ZERO*);

*assertEquals*(the\_gate.get\_object\_which\_tripped(), chi);

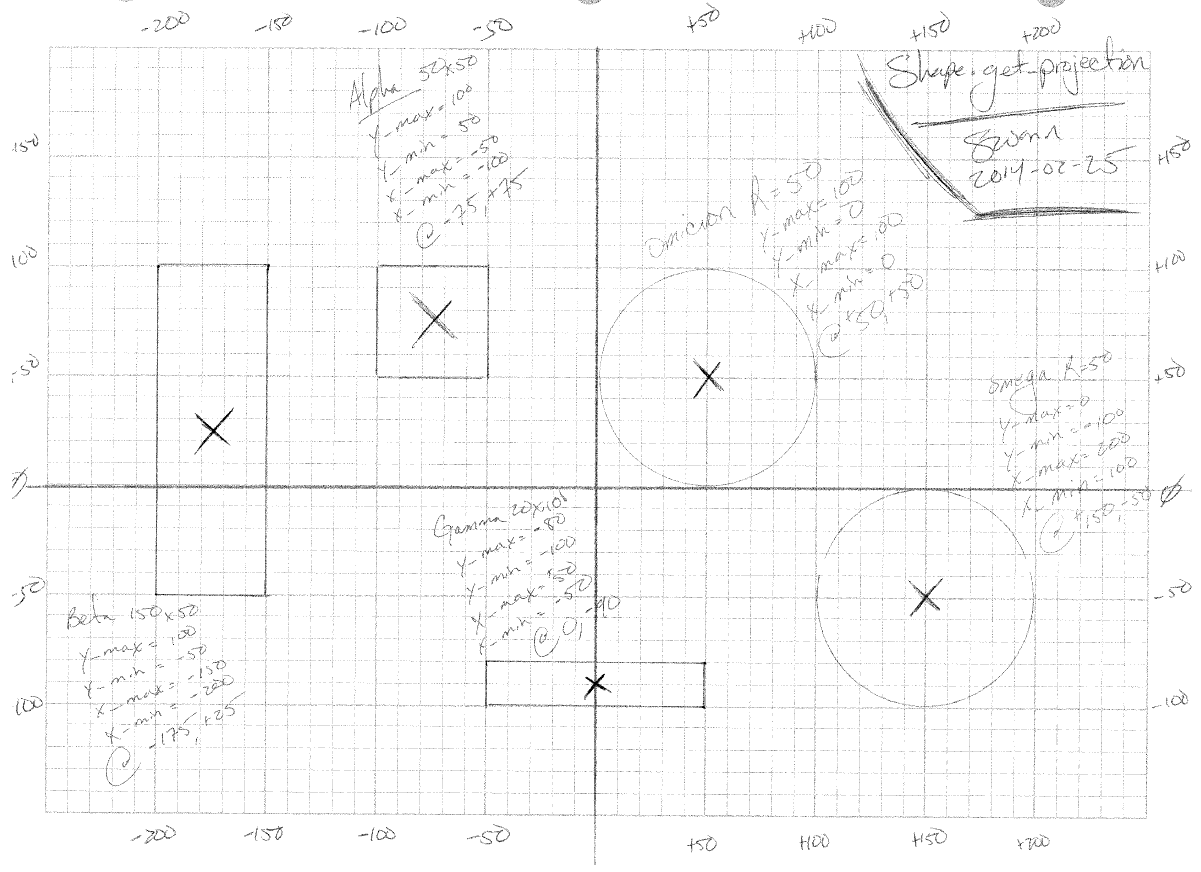
}

module :: Item/Shape.java

test/design :: test\_circle\_15\_01\_00\_projection\_returns && test\_square\_15\_01\_00\_projection\_returns

&& test\_rectangle\_15\_01\_00\_projection\_returns

description :: design of the functionality to determine the x and y axis projections of a given shape



test notes :: explores full battery of shapes and their related projections on the x and y axis

// Circle Projection F(x)'s Suite

// --> Circle.get\_x\_projection()

// --> Cricle.get\_low\_bound\_x\_projection()

// --> Cricle.get\_high\_bound\_x\_projection()

//

// --> Circle.get\_y\_projection()

// --> Cricle.get\_low\_bound\_y\_projection()

// --> Cricle.get\_low\_bound\_x\_projection()

@Test

**public** **void** test\_circle\_15\_01\_00\_projection\_returns() {

// The Setup

Circle omicron = **new** Circle(50, 50, *ZERO*, 50);

Circle omega = **new** Circle(150, -50, *ZERO*, 50);

// Omicron Battery

*assertEquals*( omicron.get\_high\_bound\_x\_projection(), 100, *ZERO* );

*assertEquals*( omicron.get\_low\_bound\_x\_projection(), 0, *ZERO* );

*assertEquals*( omicron.get\_high\_bound\_y\_projection(), 100, *ZERO* );

*assertEquals*( omicron.get\_low\_bound\_y\_projection(), 0, *ZERO* );

*assertEquals*( omicron.get\_high\_bound\_x\_projection(), omicron.get\_x\_projection()[*ONE*], *ZERO* );

*assertEquals*( omicron.get\_low\_bound\_x\_projection(), omicron.get\_x\_projection()[*ZERO*], *ZERO* );

*assertEquals*( omicron.get\_high\_bound\_y\_projection(), omicron.get\_y\_projection()[*ONE*], *ZERO* );

*assertEquals*( omicron.get\_low\_bound\_y\_projection(), omicron.get\_y\_projection()[*ZERO*], *ZERO* );

// Omega Battery

*assertEquals*( omega.get\_high\_bound\_x\_projection(), 200, *ZERO* );

*assertEquals*( omega.get\_low\_bound\_x\_projection(), 100, *ZERO* );

*assertEquals*( omega.get\_high\_bound\_y\_projection(), 0, *ZERO* );

*assertEquals*( omega.get\_low\_bound\_y\_projection(), -100, *ZERO* );

*assertEquals*( omega.get\_high\_bound\_x\_projection(), omega.get\_x\_projection()[*ONE*], *ZERO* );

*assertEquals*( omega.get\_low\_bound\_x\_projection(), omega.get\_x\_projection()[*ZERO*], *ZERO* );

*assertEquals*( omega.get\_high\_bound\_y\_projection(), omega.get\_y\_projection()[*ONE*], *ZERO* );

*assertEquals*( omega.get\_low\_bound\_y\_projection(), omega.get\_y\_projection()[*ZERO*], *ZERO* );

}

// Square Projection F(x)'s Suite

// --> Square.get\_x\_projection()

// --> Square.get\_low\_bound\_x\_projection()

// --> Square.get\_high\_bound\_x\_projection()

//

// --> Square.get\_y\_projection()

// --> Square.get\_low\_bound\_y\_projection()

// --> Square.get\_low\_bound\_x\_projection()

@Test

**public** **void** test\_square\_15\_00\_01\_projection\_returns() {

// The Setup

Square alpha = **new** Square(-75, 75, *ZERO*, 50, 50);

// Omicron Battery

*assertEquals*( alpha.get\_high\_bound\_x\_projection(), -50, *ZERO* );

*assertEquals*( alpha.get\_low\_bound\_x\_projection(), -100, *ZERO* );

*assertEquals*( alpha.get\_high\_bound\_y\_projection(), 100, *ZERO* );

*assertEquals*( alpha.get\_low\_bound\_y\_projection(), 50, *ZERO* );

*assertEquals*( alpha.get\_high\_bound\_x\_projection(), alpha.get\_x\_projection()[*ONE*], *ZERO* );

*assertEquals*( alpha.get\_low\_bound\_x\_projection(), alpha.get\_x\_projection()[*ZERO*], *ZERO* );

*assertEquals*( alpha.get\_high\_bound\_y\_projection(), alpha.get\_y\_projection()[*ONE*], *ZERO* );

*assertEquals*( alpha.get\_low\_bound\_y\_projection(), alpha.get\_y\_projection()[*ZERO*], *ZERO* );

}

// Rectangle Projection F(x)'s Suite

// --> Rectangle.get\_x\_projection()

// --> Rectangle.get\_low\_bound\_x\_projection()

// --> Rectangle.get\_high\_bound\_x\_projection()

//

// --> Rectangle.get\_y\_projection()

// --> Rectangle.get\_low\_bound\_y\_projection()

// --> Rectangle.get\_low\_bound\_x\_projection()

@Test

**public** **void** test\_rectangle\_15\_00\_01\_projection\_returns() {

// The Setup

Rectangle beta = **new** Rectangle(-175, 25, *ZERO*, 150, 50);

Rectangle gamma = **new** Rectangle( 0, -90, *ZERO*, 20, 100);

// Omicron Battery

*assertEquals*( beta.get\_high\_bound\_x\_projection(), -150, *ZERO* );

*assertEquals*( beta.get\_low\_bound\_x\_projection(), -200, *ZERO* );

*assertEquals*( beta.get\_high\_bound\_y\_projection(), 100, *ZERO* );

*assertEquals*( beta.get\_low\_bound\_y\_projection(), -50, *ZERO* );

*assertEquals*( beta.get\_high\_bound\_x\_projection(), beta.get\_x\_projection()[*ONE*], *ZERO* );

*assertEquals*( beta.get\_low\_bound\_x\_projection(), beta.get\_x\_projection()[*ZERO*], *ZERO* );

*assertEquals*( beta.get\_high\_bound\_y\_projection(), beta.get\_y\_projection()[*ONE*], *ZERO* );

*assertEquals*( beta.get\_low\_bound\_y\_projection(), beta.get\_y\_projection()[*ZERO*], *ZERO* );

// Omega Battery

*assertEquals*( gamma.get\_high\_bound\_x\_projection(), 50, *ZERO* );

*assertEquals*( gamma.get\_low\_bound\_x\_projection(), -50, *ZERO* );

*assertEquals*( gamma.get\_high\_bound\_y\_projection(), -80, *ZERO* );

*assertEquals*( gamma.get\_low\_bound\_y\_projection(), -100, *ZERO* );

*assertEquals*( gamma.get\_high\_bound\_x\_projection(), gamma.get\_x\_projection()[*ONE*], *ZERO* );

*assertEquals*( gamma.get\_low\_bound\_x\_projection(), gamma.get\_x\_projection()[*ZERO*], *ZERO* );

*assertEquals*( gamma.get\_high\_bound\_y\_projection(), gamma.get\_y\_projection()[*ONE*], *ZERO* );

*assertEquals*( gamma.get\_low\_bound\_y\_projection(), gamma.get\_y\_projection()[*ZERO*], *ZERO* );

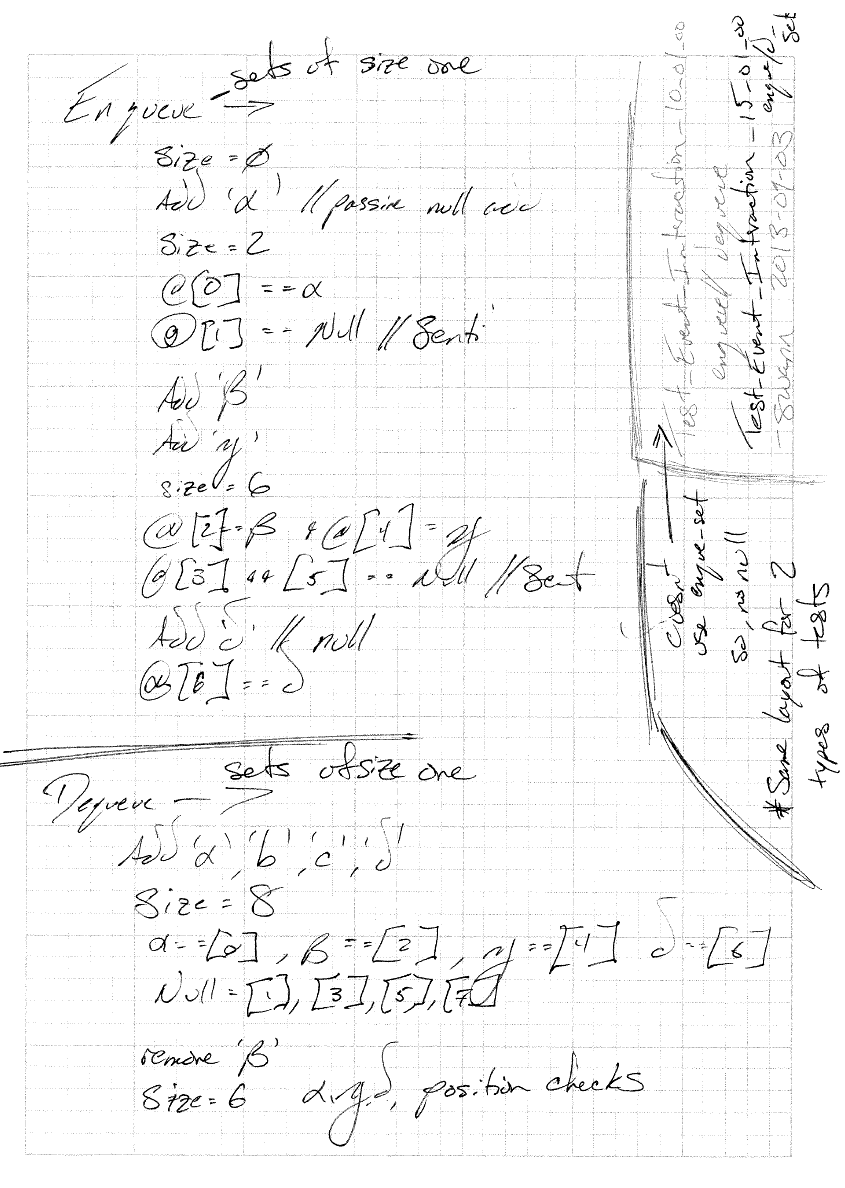
}

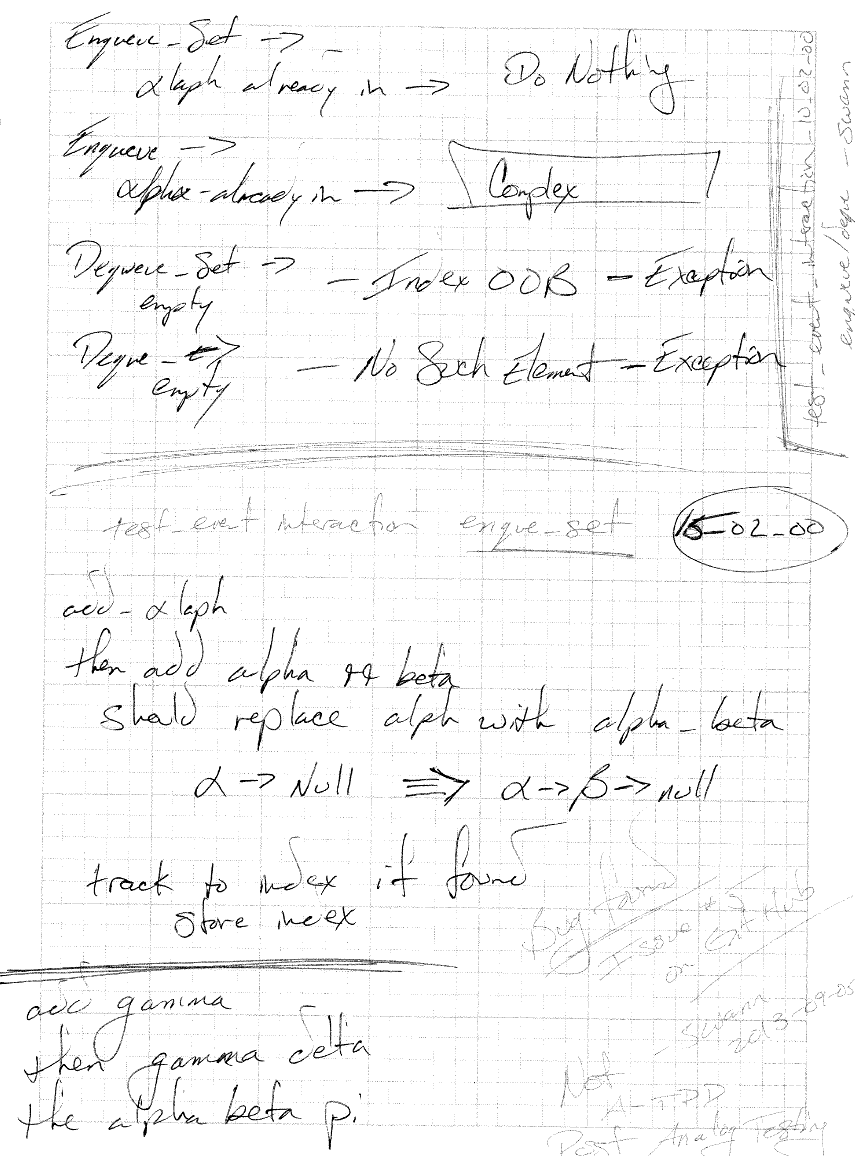
module :: Item/Event\_Interaction.java

test/design :: test\_event\_interaction\_10\_01/02\_00\_enqueue && test\_event\_interaction\_10\_01/02\_00\_dequeue

&& test\_event\_interaction\_15\_01/02\_00\_enqueue\_dequeue\_set

description :: design of the functionality to determine, queue and dequeue groups of actors involved in an event





test notes :: explores full battery of enqueue, enqueue\_set, dequeue, and dequeue\_set functions of customized

event interaction grouping logic

// Event\_Interaction.enqueue()

@Test

**public** **void** test\_event\_interaction\_10\_01\_00\_enqueue() {

// corresponding test draft in notebook

// The Setup

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(*FIVE*, *FIVE*, *ZERO*);

Cannon beta = **new** Cannon(*TEN*, *NEGATIVE\_ONE*, *ZERO*);

DropTower gamma = **new** DropTower(*TWO*, *THREE*, *ZERO*);

Ball delta = Ball.*generate\_Baseball*(*NEGATIVE\_ONE*, *SEVEN*, *ZERO*);

Collision collision = **new** Collision();

LinkedList<Actor\_Object> the\_list = collision.get\_list();

// alpha add

*assertEquals*(*ZERO*, the\_list.size(), *ZERO*);

collision.enqueue(alpha);

the\_list = collision.get\_list();

*assertEquals*(*ONE*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

// beta add

collision.enqueue(beta);

the\_list = collision.get\_list();

*assertEquals*(*TWO*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

*assertEquals*(beta, the\_list.get(*ONE*));

// gamma add

collision.enqueue(gamma);

the\_list = collision.get\_list();

*assertEquals*(*THREE*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

*assertEquals*(beta, the\_list.get(*ONE*));

*assertEquals*(gamma, the\_list.get(*TWO*));

// delta add

collision.enqueue(delta);

the\_list = collision.get\_list();

*assertEquals*(*FOUR*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

*assertEquals*(beta, the\_list.get(*ONE*));

*assertEquals*(gamma, the\_list.get(*TWO*));

*assertEquals*(delta, the\_list.get(*THREE*));

}

//Event\_Interaction.enqueue()

@Test

**public** **void** test\_event\_interaction\_10\_02\_00\_enqueue() {

// Re-adding already added

// The Setup

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(*FIVE*, *FIVE*, *ZERO*);

Collision collision = **new** Collision();

LinkedList<Actor\_Object> the\_list = collision.get\_list();

// alpha add

*assertEquals*(*ZERO*, the\_list.size(), *ZERO*);

collision.enqueue(alpha);

the\_list = collision.get\_list();

*assertEquals*(*ONE*, the\_list.size(), *ZERO*);

collision.enqueue(alpha);

the\_list = collision.get\_list();

*assertEquals*(*ONE*, the\_list.size(), *ZERO*);

}

// Event\_Interaction.dequeue()

@Test

**public** **void** test\_event\_interaction\_10\_01\_00\_dequeue() {

// The Setup

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(*FIVE*, *FIVE*, *ZERO*);

Cannon beta = **new** Cannon(*TEN*, *NEGATIVE\_ONE*, *ZERO*);

DropTower gamma = **new** DropTower(*TWO*, *THREE*, *ZERO*);

Ball delta = Ball.*generate\_Baseball*(*NEGATIVE\_ONE*, *SEVEN*, *ZERO*);

Collision collision = **new** Collision();

LinkedList<Actor\_Object> the\_list = collision.get\_list();

// all add

collision.enqueue(alpha);

collision.enqueue(beta);

collision.enqueue(gamma);

collision.enqueue(delta);

the\_list = collision.get\_list();

*assertEquals*(*FOUR*, the\_list.size(), *ZERO*);

// alpha remove

collision.dequeue();

the\_list = collision.get\_list();

*assertEquals*(*THREE*, the\_list.size(), *ZERO*);

*assertEquals*(beta, the\_list.get(*ZERO*));

*assertEquals*(gamma, the\_list.get(*ONE*));

*assertEquals*(delta, the\_list.get(*TWO*));

// beta remove

collision.dequeue();

the\_list = collision.get\_list();

*assertEquals*(*TWO*, the\_list.size(), *ZERO*);

*assertEquals*(gamma, the\_list.get(*ZERO*));

*assertEquals*(delta, the\_list.get(*ONE*));

// gamma remove

collision.dequeue();

the\_list = collision.get\_list();

*assertEquals*(*ONE*, the\_list.size(), *ZERO*);

*assertEquals*(delta, the\_list.get(*ZERO*));

// gamma remove

collision.dequeue();

the\_list = collision.get\_list();

*assertEquals*(*ZERO*, the\_list.size(), *ZERO*);

}

// Event\_Interaction.dequeue()

@Test

**public** **void** test\_event\_interaction\_10\_02\_00\_dequeue() {

// The Setup

Collision collision = **new** Collision();

//LinkedList<Actor\_Object> the\_list = collision.get\_list();

**try** {

collision.dequeue();

}

**catch** (NoSuchElementException e){

// do nothing just pass

}

}

// Event\_Interaction.enqueue\_set()

@Test

**public** **void** test\_event\_interaction\_15\_01\_00\_enqueue\_set() {

// corresponding test draft in notebook

// The Setup

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(*FIVE*, *FIVE*, *ZERO*);

Cannon beta = **new** Cannon(*TEN*, *NEGATIVE\_ONE*, *ZERO*);

DropTower gamma = **new** DropTower(*TWO*, *THREE*, *ZERO*);

Ball delta = Ball.*generate\_Baseball*(*NEGATIVE\_ONE*, *SEVEN*, *ZERO*);

ArrayList<Actor\_Object> alpha\_list = **new** ArrayList<Actor\_Object>();

alpha\_list.add((Actor\_Object)alpha);

ArrayList<Actor\_Object> beta\_list = **new** ArrayList<Actor\_Object>();

beta\_list.add((Actor\_Object)beta);

ArrayList<Actor\_Object> gamma\_list = **new** ArrayList<Actor\_Object>();

gamma\_list.add((Actor\_Object)gamma);

ArrayList<Actor\_Object> delta\_list = **new** ArrayList<Actor\_Object>();

delta\_list.add((Actor\_Object)delta);

Collision collision = **new** Collision();

LinkedList<Actor\_Object> the\_list = collision.get\_list();

// alpha add

*assertEquals*(*ZERO*, the\_list.size(), *ZERO*);

collision.enqueue\_set(alpha\_list);

the\_list = collision.get\_list();

*assertEquals*(*ONE*\**TWO*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

// beta add

collision.enqueue\_set(beta\_list);

the\_list = collision.get\_list();

*assertEquals*(*TWO*\**TWO*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

*assertEquals*(beta, the\_list.get(*ONE*\**TWO*));

// gamma add

collision.enqueue\_set(gamma\_list);

the\_list = collision.get\_list();

*assertEquals*(*THREE*\**TWO*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

*assertEquals*(beta, the\_list.get(*ONE*\**TWO*));

*assertEquals*(gamma, the\_list.get(*TWO*\**TWO*));

// delta add

collision.enqueue\_set(delta\_list);

the\_list = collision.get\_list();

*assertEquals*(*FOUR*\**TWO*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

*assertEquals*(beta, the\_list.get(*ONE*\**TWO*));

*assertEquals*(gamma, the\_list.get(*TWO*\**TWO*));

*assertEquals*(delta, the\_list.get(*THREE*\**TWO*));

// null sentinel checks

Ball SENTINEL = **null**;

*assertEquals*(SENTINEL, the\_list.get(*ONE*));

*assertEquals*(SENTINEL, the\_list.get(*THREE*));

*assertEquals*(SENTINEL, the\_list.get(*FIVE*));

*assertEquals*(SENTINEL, the\_list.get(*SEVEN*));

}

// Event\_Interaction.enqueue\_set()

@Test

**public** **void** test\_event\_interaction\_15\_02\_00\_enqueue\_set() {

// corresponding test draft in notebook

// The Setup

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(*FIVE*, *FIVE*, *ZERO*);

Cannon beta = **new** Cannon(*TEN*, *NEGATIVE\_ONE*, *ZERO*);

DropTower gamma = **new** DropTower(*TWO*, *THREE*, *ZERO*);

Ball delta = Ball.*generate\_Baseball*(*NEGATIVE\_ONE*, *SEVEN*, *ZERO*);

Ball eta = Ball.*generate\_Baseball*(*NEGATIVE\_ONE*, *SEVEN*, *ZERO*);

// primary lists

ArrayList<Actor\_Object> alpha\_list = **new** ArrayList<Actor\_Object>();

alpha\_list.add((Actor\_Object)alpha);

ArrayList<Actor\_Object> beta\_list = **new** ArrayList<Actor\_Object>();

beta\_list.add((Actor\_Object)beta);

ArrayList<Actor\_Object> gamma\_list = **new** ArrayList<Actor\_Object>();

gamma\_list.add((Actor\_Object)gamma);

ArrayList<Actor\_Object> delta\_list = **new** ArrayList<Actor\_Object>();

delta\_list.add((Actor\_Object)delta);

// Misc. Overhead

Collision collision = **new** Collision();

LinkedList<Actor\_Object> the\_list = collision.get\_list();

Ball SENTINEL = **null**;

// add alpha

collision.enqueue\_set(alpha\_list);

the\_list = collision.get\_list();

*assertEquals*(*ONE*\**TWO*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

*assertEquals*(SENTINEL, the\_list.get(*ONE*));

// Transmogrification of alpha\_list and re-test

alpha\_list.add(beta);

alpha\_list.add(alpha);

collision.enqueue\_set(alpha\_list);

the\_list = collision.get\_list();

//

// **ANALOG**

// Bug found here from **ANALOG** test design

// improper coagulation of set data for a given event

// issue #5 on github.

// ~swann 2013-11-14

*assertEquals*(*THREE*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

*assertEquals*(beta, the\_list.get(*ONE*));

*assertEquals*(SENTINEL, the\_list.get(*TWO*));

// add gamma, then gamma-delta

collision.enqueue\_set(gamma\_list);

*assertEquals*(*FIVE*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

*assertEquals*(beta, the\_list.get(*ONE*));

*assertEquals*(SENTINEL, the\_list.get(*TWO*));

*assertEquals*(gamma, the\_list.get(*THREE*));

*assertEquals*(SENTINEL, the\_list.get(*FOUR*));

// double add

gamma\_list.add(gamma);

gamma\_list.add(delta);

collision.enqueue\_set(gamma\_list);

*assertEquals*(*SIX*, the\_list.size(), *ZERO*);

*assertEquals*(alpha, the\_list.get(*ZERO*));

*assertEquals*(beta, the\_list.get(*ONE*));

*assertEquals*(SENTINEL, the\_list.get(*TWO*));

*assertEquals*(gamma, the\_list.get(*THREE*));

*assertEquals*(delta, the\_list.get(*FOUR*));

*assertEquals*(SENTINEL, the\_list.get(*FIVE*));

// Triple add to alpha list

alpha\_list.add(beta);

alpha\_list.add(alpha);

alpha\_list.add(eta);

collision.enqueue\_set(alpha\_list);

the\_list = collision.get\_list();

*assertEquals*(*SEVEN*, the\_list.size(), *ZERO*);

*assertEquals*(gamma, the\_list.get(*ZERO*));

*assertEquals*(delta, the\_list.get(*ONE*));

*assertEquals*(SENTINEL, the\_list.get(*TWO*));

*assertEquals*(alpha, the\_list.get(*THREE*));

*assertEquals*(beta, the\_list.get(*FOUR*));

*assertEquals*(eta, the\_list.get(*FIVE*));

*assertEquals*(SENTINEL, the\_list.get(*SIX*));

}

// Event\_Interaction.dequeue\_set()

@Test

**public** **void** test\_event\_interaction\_15\_01\_00\_dequeue\_set() {

// The Setup

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(*FIVE*, *FIVE*, *ZERO*);

Cannon beta = **new** Cannon(*TEN*, *NEGATIVE\_ONE*, *ZERO*);

DropTower gamma = **new** DropTower(*TWO*, *THREE*, *ZERO*);

Ball delta = Ball.*generate\_Baseball*(*NEGATIVE\_ONE*, *SEVEN*, *ZERO*);

Collision collision = **new** Collision();

ArrayList<Actor\_Object> alpha\_list = **new** ArrayList<Actor\_Object>();

alpha\_list.add((Actor\_Object)alpha);

ArrayList<Actor\_Object> beta\_list = **new** ArrayList<Actor\_Object>();

beta\_list.add((Actor\_Object)beta);

ArrayList<Actor\_Object> gamma\_list = **new** ArrayList<Actor\_Object>();

gamma\_list.add((Actor\_Object)gamma);

ArrayList<Actor\_Object> delta\_list = **new** ArrayList<Actor\_Object>();

delta\_list.add((Actor\_Object)delta);

LinkedList<Actor\_Object> the\_list = collision.get\_list();

// all add

collision.enqueue\_set(alpha\_list);

collision.enqueue\_set(beta\_list);

collision.enqueue\_set(gamma\_list);

collision.enqueue\_set(delta\_list);

the\_list = collision.get\_list();

*assertEquals*(*FOUR*\**TWO*, the\_list.size(), *ZERO*);

// null sentinel checks

Ball SENTINEL = **null**;

*assertEquals*(SENTINEL, the\_list.get(*ONE*));

*assertEquals*(SENTINEL, the\_list.get(*THREE*));

*assertEquals*(SENTINEL, the\_list.get(*FIVE*));

*assertEquals*(SENTINEL, the\_list.get(*SEVEN*));

// alpha remove

collision.dequeue\_set();

the\_list = collision.get\_list();

*assertEquals*(*THREE*\**TWO*, the\_list.size(), *ZERO*);

*assertEquals*(beta, the\_list.get(*ZERO*));

*assertEquals*(gamma, the\_list.get(*ONE*\**TWO*));

*assertEquals*(delta, the\_list.get(*TWO*\**TWO*));

// beta remove

collision.dequeue\_set();

the\_list = collision.get\_list();

*assertEquals*(*TWO*\**TWO*, the\_list.size(), *ZERO*);

*assertEquals*(gamma, the\_list.get(*ZERO*));

*assertEquals*(delta, the\_list.get(*ONE*\**TWO*));

// gamma remove

collision.dequeue\_set();

the\_list = collision.get\_list();

*assertEquals*(*ONE*\**TWO*, the\_list.size(), *ZERO*);

*assertEquals*(delta, the\_list.get(*ZERO*\**TWO*));

// gamma remove

collision.dequeue\_set();

the\_list = collision.get\_list();

*assertEquals*(*ZERO*, the\_list.size(), *ZERO*);

}

// Event\_Interaction.dequeue\_set()

@Test

**public** **void** test\_event\_interaction\_15\_02\_00\_dequeue\_set() {

// The Setup

Collision collision = **new** Collision();

//LinkedList<Actor\_Object> the\_list = collision.get\_list();

**try** {

collision.dequeue\_set();

}

**catch** (IndexOutOfBoundsException e){

// do nothing just pass

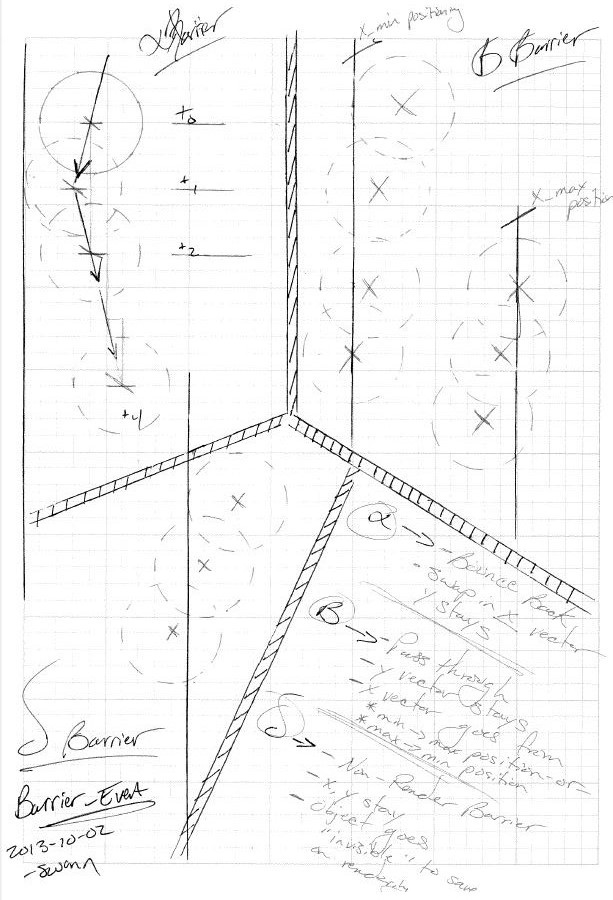
}

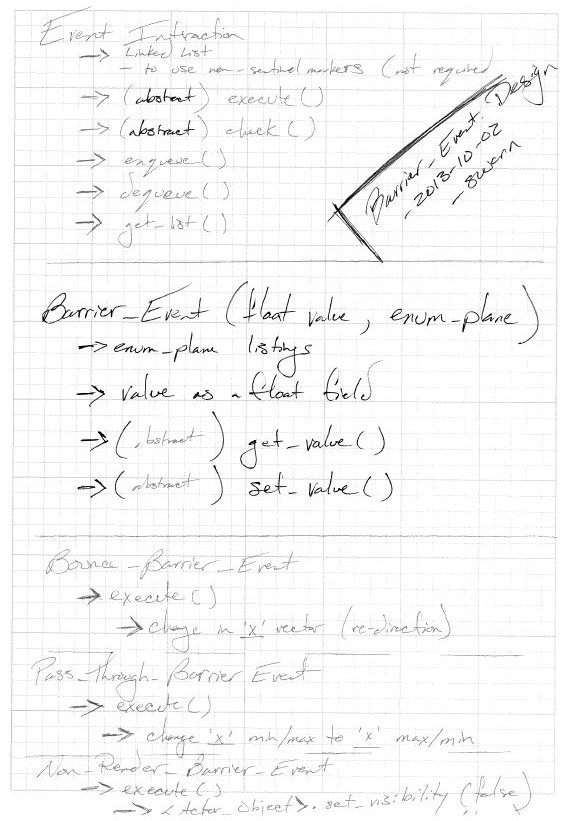
}

module :: Item/Barrier\_Event.java

test/design :: test\_barrier\_general\_10\_01\_00\_constructor\_equivalence

description :: design of the normalized behaviors of all barrier events





test notes :: explores full battery of normalized behaviors across the three barrier event classes

// Barrier\_Event Constructor equivalence

// --> set\_value()

// --> get\_value()

// --> get\_axis()

// --> Pass\_Through

@Test

**public** **void** test\_barrier\_general\_10\_01\_00\_constructor\_equivalence() {

// The Setup

Bounce\_Barrier\_Event alpha = **new** Bounce\_Barrier\_Event(*ZERO*, Barrier\_Event.axis\_enumeration.*X*,

Barrier\_Event.axis\_control.*STAY\_HIGHER*);

Pass\_Through\_Barrier\_Event beta = **new** Pass\_Through\_Barrier\_Event(*ONE*, *TEN*, Barrier\_Event.axis\_enumeration.*X*,

Barrier\_Event.axis\_control.*STAY\_HIGHER*);

Non\_Render\_Barrier\_Event delta = **new** Non\_Render\_Barrier\_Event(*TWO*, Barrier\_Event.axis\_enumeration.*X*,

Barrier\_Event.axis\_control.*STAY\_HIGHER*);

Bounce\_Barrier\_Event gamma = **new** Bounce\_Barrier\_Event((**float**) .2, Barrier\_Event.axis\_enumeration.*Y*,

Barrier\_Event.axis\_control.*STAY\_LOWER*);

Pass\_Through\_Barrier\_Event iota = **new** Pass\_Through\_Barrier\_Event((**float**) .009, *TEN*,

Barrier\_Event.axis\_enumeration.*Y*, Barrier\_Event.axis\_control.*STAY\_LOWER*);

Non\_Render\_Barrier\_Event eta = **new** Non\_Render\_Barrier\_Event((**float**) 14.77,

Barrier\_Event.axis\_enumeration.*Y*, Barrier\_Event.axis\_control.*STAY\_LOWER*);

// Battery One

*assertEquals*(*ZERO*, alpha.get\_value(), *A\_THOUSANDTH*);

*assertEquals*(*ONE*, beta.get\_value(), *A\_THOUSANDTH*);

*assertEquals*(*TWO*, delta.get\_value(), *A\_THOUSANDTH*);

*assertEquals*((**float**) .2, gamma.get\_value(), *A\_THOUSANDTH*);

*assertEquals*((**float**) .009, iota.get\_value(), *A\_THOUSANDTH*);

*assertEquals*((**float**) 14.77, eta.get\_value(), *A\_THOUSANDTH*);

*assertEquals*(Barrier\_Event.axis\_enumeration.*X*, alpha.get\_axis());

*assertEquals*(Barrier\_Event.axis\_enumeration.*X*, beta.get\_axis());

*assertEquals*(Barrier\_Event.axis\_enumeration.*X*, delta.get\_axis());

*assertEquals*(Barrier\_Event.axis\_enumeration.*Y*, gamma.get\_axis());

*assertEquals*(Barrier\_Event.axis\_enumeration.*Y*, iota.get\_axis());

*assertEquals*(Barrier\_Event.axis\_enumeration.*Y*, eta.get\_axis());

*assertEquals*(Barrier\_Event.axis\_control.*STAY\_HIGHER*, alpha.get\_axis\_control());

*assertEquals*(Barrier\_Event.axis\_control.*STAY\_HIGHER*, beta.get\_axis\_control());

*assertEquals*(Barrier\_Event.axis\_control.*STAY\_HIGHER*, delta.get\_axis\_control());

*assertEquals*(Barrier\_Event.axis\_control.*STAY\_LOWER*, gamma.get\_axis\_control());

*assertEquals*(Barrier\_Event.axis\_control.*STAY\_LOWER*, iota.get\_axis\_control());

*assertEquals*(Barrier\_Event.axis\_control.*STAY\_LOWER*, eta.get\_axis\_control());

// Battery Two

alpha.set\_value((**float**) .7);

beta.set\_value( (**float**) .7);

delta.set\_value((**float**) .7);

gamma.set\_value((**float**) .7);

iota.set\_value( (**float**) .7);

eta.set\_value( (**float**) .7);

*assertEquals*((**float**) .7, gamma.get\_value(), *A\_THOUSANDTH*);

*assertEquals*((**float**) 0.7, iota.get\_value(), *A\_THOUSANDTH*);

*assertEquals*((**float**) .7, eta.get\_value(), *A\_THOUSANDTH*);

*assertEquals*((**float**) .7, alpha.get\_value(), *A\_THOUSANDTH*);

*assertEquals*((**float**) 0.7, beta.get\_value(), *A\_THOUSANDTH*);

*assertEquals*((**float**) 0.7, delta.get\_value(), *A\_THOUSANDTH*);

// Battery\_Three

*assertEquals*(*TEN*, iota.get\_opposing\_value(), *A\_THOUSANDTH*);

*assertEquals*(*TEN*, beta.get\_opposing\_value(), *A\_THOUSANDTH*);

iota.set\_opposing\_value( (**float**) -.001 );

beta.set\_opposing\_value( (**float**) 12345.6789 );

*assertEquals*((**float**) -.001, iota.get\_opposing\_value(), *A\_THOUSANDTH*);

*assertEquals*((**float**) 12345.6789, beta.get\_opposing\_value(), *A\_THOUSANDTH*);

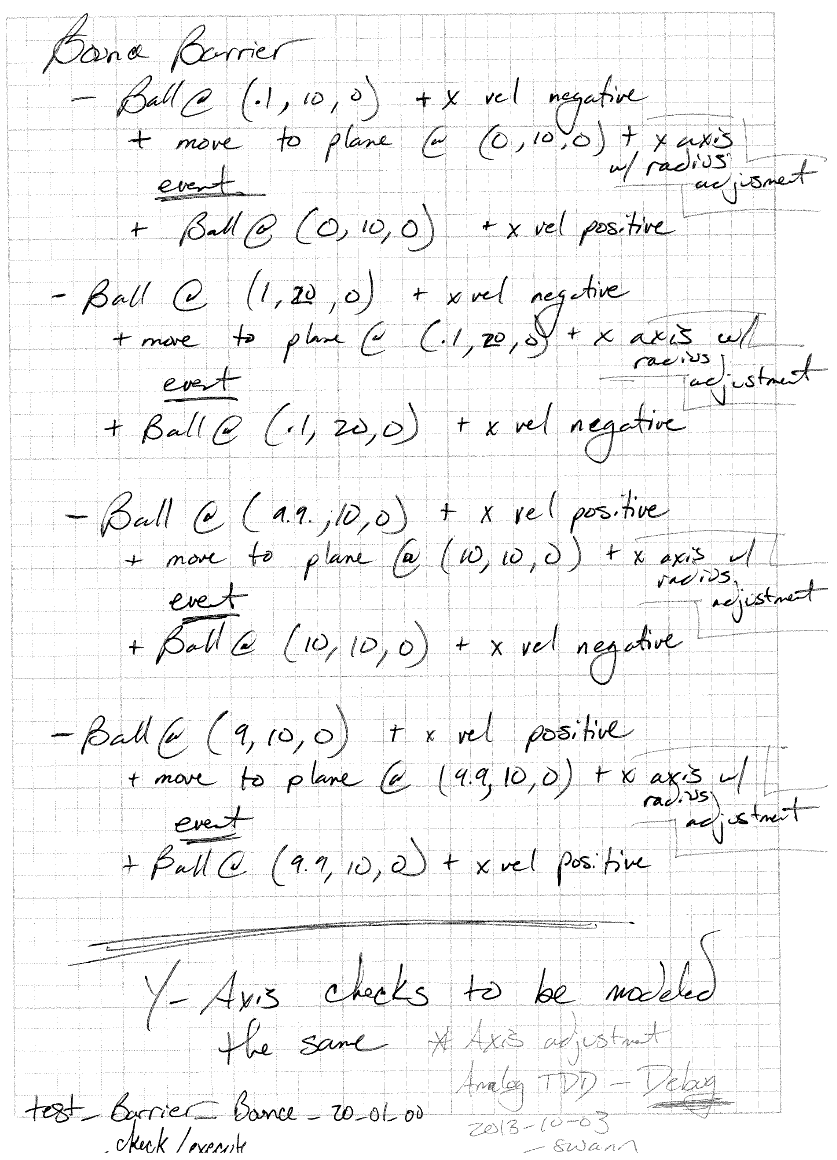
}

module :: Item/Bounce\_Barrier\_Event.java && Item.Pass\_Through\_Barrier.java

test/design :: test\_bounce\_barrier\_20\_01\_00\_check && test\_bouce\_barrier\_20\_02\_00\_execute &&

test\_pass\_through\_barrier\_20\_01\_00\_check && test\_pass\_through\_barrier\_20\_02\_00\_execute

description :: design of the specific check and execute behaviors for bounce and pass through barriers



test notes :: explores functionality associated with both bounce barrier and pass through barrier; above test scenario

recycled for pass through barriers with slight logic modifications

// Bounce\_Barrier\_Event.check/execute()

@Test

**public** **void** test\_barrier\_bounce\_20\_01\_00\_check() {

// The setup

Ball alpha;

Ball beta;

**float**[] velocity\_spread;

Bounce\_Barrier\_Event omicron;

LinkedList<Actor\_Object> flag\_list;

ArrayList<Actor\_Object> the\_list = **new** ArrayList<Actor\_Object>();

//

// X Axis Sweep

//

// Battery One --> No changes yet

alpha = Ball.*generate\_Baseball*((**float**) .1, *TEN*, *ZERO*);

beta = Ball.*generate\_Baseball*(*ONE*, *TEN*, *ZERO*);

omicron = **new** Bounce\_Barrier\_Event(*ZERO*-alpha.get\_shape().get\_radius(),

Barrier\_Event.axis\_enumeration.*Y*,

Barrier\_Event.axis\_control.*STAY\_HIGHER*);

alpha.update\_velocity(*ZERO*-*FIVE*\**TEN*, *ZERO*, *ZERO*);

beta.update\_velocity (*ZERO*-*FIVE*\**TEN*, *ZERO*, *ZERO*);

the\_list.add(alpha);

the\_list.add(beta);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ZERO*, flag\_list.size());

// alpha has changed vel, beta does not

alpha.update\_location(*ZERO*, *TEN*, *ZERO*);

beta.update\_location((**float**) .1, *TEN*, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(alpha, flag\_list.get(*ZERO*));

velocity\_spread = alpha.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ONE*], *ZERO*);

*assertEquals*(*FIVE*\**TEN*\*alpha.get\_coefficient\_of\_restitution(), velocity\_spread[*ZERO*], *ZERO*);

// beta has changed vel, alpha does not

omicron.clean\_list();

beta.update\_location(*ZERO*, *TEN*, *ZERO*);

alpha.update\_location(*ONE*, *TEN*, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(beta, flag\_list.get(*ZERO*));

velocity\_spread = beta.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ONE*], *ZERO*);

*assertEquals*(*FIVE*\**TEN*\*beta.get\_coefficient\_of\_restitution(), velocity\_spread[*ZERO*], *ZERO*);

velocity\_spread = alpha.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ONE*], *ZERO*);

*assertEquals*(*FIVE*\**TEN*\*alpha.get\_coefficient\_of\_restitution(), velocity\_spread[*ZERO*], *ZERO*);

// Battery Two --> No changes yet

alpha = Ball.*generate\_Baseball*((**float**) 9.9, *TEN*, *ZERO*);

beta = Ball.*generate\_Baseball*(*NINE*, *TEN*, *ZERO*);

omicron = **new** Bounce\_Barrier\_Event(*TEN*+alpha.get\_shape().get\_radius(),

Barrier\_Event.axis\_enumeration.*Y*,

Barrier\_Event.axis\_control.*STAY\_LOWER*);

the\_list = **new** ArrayList<Actor\_Object>();

alpha.update\_velocity(*FIVE*\**TEN*, *ZERO*, *ZERO*);

beta.update\_velocity (*FIVE*\**TEN*, *ZERO*, *ZERO*);

the\_list.add(alpha);

the\_list.add(beta);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ZERO*, flag\_list.size());

// alpha has changed vel, beta does not

alpha.update\_location(*TEN*, *TEN*, *ZERO*);

beta.update\_location((**float**) 9.9, *TEN*, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(alpha, flag\_list.get(*ZERO*));

velocity\_spread = alpha.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ONE*], *ZERO*);

*assertEquals*(*ZERO*-*FIVE*\**TEN*\*alpha.get\_coefficient\_of\_restitution(), velocity\_spread[*ZERO*], *ZERO*);

// beta has changed vel, alpha does not

omicron.clean\_list();

beta.update\_location(*TEN*, *TEN*, *ZERO*);

alpha.update\_location(*NINE*, *TEN*, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(beta, flag\_list.get(*ZERO*));

velocity\_spread = beta.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ONE*], *ZERO*);

*assertEquals*(*ZERO*-*FIVE*\**TEN*\*beta.get\_coefficient\_of\_restitution(), velocity\_spread[*ZERO*], *ZERO*);

velocity\_spread = alpha.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ONE*], *ZERO*);

*assertEquals*(*ZERO*-*FIVE*\**TEN*\*alpha.get\_coefficient\_of\_restitution(), velocity\_spread[*ZERO*], *ZERO*);

//

// Y Axis Sweep

//

// Battery Three --> No changes yet

alpha = Ball.*generate\_Baseball*(*TEN*, (**float**) .1, *ZERO*);

beta = Ball.*generate\_Baseball*(*TEN*, *ONE*, *ZERO*);

omicron = **new** Bounce\_Barrier\_Event(*ZERO*-alpha.get\_shape().get\_radius(),

Barrier\_Event.axis\_enumeration.*X*,

Barrier\_Event.axis\_control.*STAY\_HIGHER*);

alpha.update\_velocity(*ZERO*, *ZERO*-*FIVE*\**TEN*, *ZERO*);

beta.update\_velocity (*ZERO*, *ZERO*-*FIVE*\**TEN*, *ZERO*);

the\_list.add(alpha);

the\_list.add(beta);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ZERO*, flag\_list.size());

// alpha has changed vel, beta does not

alpha.update\_location(*TEN*, *ZERO*, *ZERO*);

beta.update\_location(*TEN*, (**float**) .1, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(alpha, flag\_list.get(*ZERO*));

velocity\_spread = alpha.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ZERO*], *ZERO*);

*assertEquals*(*FIVE*\**TEN*\*alpha.get\_coefficient\_of\_restitution(), velocity\_spread[*ONE*], *ZERO*);

// beta has changed vel, alpha does not

omicron.clean\_list();

beta.update\_location(*TEN*, *ZERO*, *ZERO*);

alpha.update\_location(*TEN*, *ONE*, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(beta, flag\_list.get(*ZERO*));

velocity\_spread = beta.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ZERO*], *ZERO*);

*assertEquals*(*FIVE*\**TEN*\*beta.get\_coefficient\_of\_restitution(), velocity\_spread[*ONE*], *ZERO*);

velocity\_spread = alpha.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ZERO*], *ZERO*);

*assertEquals*(*FIVE*\**TEN*\*alpha.get\_coefficient\_of\_restitution(), velocity\_spread[*ONE*], *ZERO*);

// Battery Four --> No changes yet

alpha = Ball.*generate\_Baseball*(*TEN*, (**float**) 9.9, *ZERO*);

beta = Ball.*generate\_Baseball*(*TEN*, *NINE*, *ZERO*);

omicron = **new** Bounce\_Barrier\_Event(*TEN*+alpha.get\_shape().get\_radius(),

Barrier\_Event.axis\_enumeration.*X*,

Barrier\_Event.axis\_control.*STAY\_LOWER*);

the\_list = **new** ArrayList<Actor\_Object>();

alpha.update\_velocity(*ZERO*, *FIVE*\**TEN*, *ZERO*);

beta.update\_velocity (*ZERO*, *FIVE*\**TEN*, *ZERO*);

the\_list.add(alpha);

the\_list.add(beta);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ZERO*, flag\_list.size());

// alpha has changed vel, beta does not

alpha.update\_location(*TEN*, *TEN*, *ZERO*);

beta.update\_location(*TEN*, (**float**) 9.9, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(alpha, flag\_list.get(*ZERO*));

velocity\_spread = alpha.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ZERO*], *ZERO*);

*assertEquals*(*ZERO*-*FIVE*\**TEN*\*alpha.get\_coefficient\_of\_restitution(), velocity\_spread[*ONE*], *ZERO*);

// beta has changed vel, alpha does not

omicron.clean\_list();

beta.update\_location(*TEN*, *TEN*, *ZERO*);

alpha.update\_location(*TEN*, *NINE*, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(beta, flag\_list.get(*ZERO*));

velocity\_spread = beta.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ZERO*], *ZERO*);

*assertEquals*(*ZERO*-*FIVE*\**TEN*\*beta.get\_coefficient\_of\_restitution(), velocity\_spread[*ONE*], *ZERO*);

velocity\_spread = alpha.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ZERO*], *ZERO*);

*assertEquals*(*ZERO*-*FIVE*\**TEN*\*alpha.get\_coefficient\_of\_restitution(), velocity\_spread[*ONE*], *ZERO*);

}

// Bounce\_Barrier\_Event.check/execute()

@Test

**public** **void** test\_barrier\_bounce\_20\_02\_00\_check\_execute() {

// The setup

Standard\_Mass gamma;

Standard\_Mass delta;

Cannon the\_cannon;

**float**[] velocity\_spread;

Bounce\_Barrier\_Event omicron;

LinkedList<Actor\_Object> flag\_list;

ArrayList<Actor\_Object> the\_list = **new** ArrayList<Actor\_Object>();

//

// X Axis Sweep

//

// Battery One --> No changes yet

gamma = Standard\_Mass.*generate\_fifty\_g\_mass*((**float**) .1, *TEN*, *ZERO*);

delta = Standard\_Mass.*generate\_fifty\_g\_mass*(*ONE*, *TEN*, *ZERO*);

the\_cannon = **new** Cannon(*ZERO*, *ZERO*, *ZERO*);

omicron = **new** Bounce\_Barrier\_Event((**float**) (*ZERO*-*FIVE*\**TEN*/*TWO*),

Barrier\_Event.axis\_enumeration.*Y*,

Barrier\_Event.axis\_control.*STAY\_HIGHER*);

gamma.update\_velocity(*ZERO*-*FIVE*\**TEN*, *ZERO*, *ZERO*);

delta.update\_velocity(*ZERO*-*FIVE*\**TEN*, *ZERO*, *ZERO*);

the\_list.add(gamma);

the\_list.add(delta);

the\_list.add(the\_cannon);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ZERO*, flag\_list.size());

// alpha has changed vel, beta does not

gamma.update\_location((**float**) -.1, *TEN*, *ZERO*);

delta.update\_location((**float**) .1, *TEN*, *ZERO*);

// # **ANALOG** -->

// Found update\_location bug... does not update radius or points

// ~swann 2013-10-07

// -------------------------------

// Logged into github; issue #19

// Solved via <Shape.set\_location() Override>

// Time to fix :: 45-60 seconds

// test re-run and test pass

// ~swann 2013-10-07

//gamma.get\_shape().set\_radius(gamma.get\_shape().get\_radius());

//delta.get\_shape().set\_radius(delta.get\_shape().get\_radius());

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(gamma, flag\_list.get(*ZERO*));

velocity\_spread = gamma.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ONE*], *ZERO*);

*assertEquals*(*FIVE*\**TEN*\*gamma.get\_coefficient\_of\_restitution(), velocity\_spread[*ZERO*], *ZERO*);

// beta has changed vel, alpha does not

omicron.clean\_list();

delta.update\_location((**float**) -.1, *TEN*, *ZERO*);

gamma.update\_location(*ONE*, *TEN*, *ZERO*);

// DO NOT DELETE --> needed for above Analog tag

// ~swann 2013-10-07

//gamma.get\_shape().set\_radius(gamma.get\_shape().get\_radius());

//delta.get\_shape().set\_radius(delta.get\_shape().get\_radius());

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(delta, flag\_list.get(*ZERO*));

velocity\_spread = delta.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ONE*], *ZERO*);

*assertEquals*(*FIVE*\**TEN*\*delta.get\_coefficient\_of\_restitution(), velocity\_spread[*ZERO*], *ZERO*);

velocity\_spread = gamma.get\_velocity();

*assertEquals*(*ZERO*, velocity\_spread[*TWO*], *ZERO*);

*assertEquals*(*ZERO*, velocity\_spread[*ONE*], *ZERO*);

*assertEquals*(*FIVE*\**TEN*\*gamma.get\_coefficient\_of\_restitution(), velocity\_spread[*ZERO*], *ZERO*);

}

// Pass\_Through\_Barrier\_Event.check/execute()

@Test

**public** **void** test\_barrier\_pass\_through\_20\_01\_00\_check\_execute() {

// not designed or installed

// ~swann 2013-10-03

// The setup

Ball alpha;

Ball beta;

**float**[] location\_spread;

Pass\_Through\_Barrier\_Event omicron;

LinkedList<Actor\_Object> flag\_list;

ArrayList<Actor\_Object> the\_list = **new** ArrayList<Actor\_Object>();

//

// X Axis Sweep

//

// Battery One --> No changes yet

alpha = Ball.*generate\_Baseball*((**float**) .1, *TEN*, *ZERO*);

beta = Ball.*generate\_Baseball*(*ONE*, *TEN*, *ZERO*);

omicron = **new** Pass\_Through\_Barrier\_Event(*ZERO*,

*TEN*,

Barrier\_Event.axis\_enumeration.*Y*,

Barrier\_Event.axis\_control.*STAY\_HIGHER*);

the\_list.add(alpha);

the\_list.add(beta);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ZERO*, flag\_list.size());

// alpha has changed vel, beta does not

alpha.update\_location(*ZERO*, *TEN*, *ZERO*);

beta.update\_location((**float**) .1, *TEN*, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(alpha, flag\_list.get(*ZERO*));

location\_spread = alpha.get\_location();

*assertEquals*(*TEN*, location\_spread[*ZERO*], *ZERO*);

// beta has changed vel, alpha does not

omicron.clean\_list();

beta.update\_location(*ZERO*, *TEN*, *ZERO*);

alpha.update\_location(*ONE*, *TEN*, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(beta, flag\_list.get(*ZERO*));

location\_spread = beta.get\_location();

*assertEquals*(*TEN*, location\_spread[*ZERO*], *ZERO*);

//

// Y Axis Sweep

//

// Battery One --> No changes yet

alpha = Ball.*generate\_Baseball*(*ONE*, (**float**) 9.9, *ZERO*);

beta = Ball.*generate\_Baseball*(*ONE*, *NINE*, *ZERO*);

omicron = **new** Pass\_Through\_Barrier\_Event(*TEN*,

*ZERO*,

Barrier\_Event.axis\_enumeration.*X*,

Barrier\_Event.axis\_control.*STAY\_LOWER*);

the\_list = **new** ArrayList<Actor\_Object>();

the\_list.add(alpha);

the\_list.add(beta);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ZERO*, flag\_list.size());

// alpha has changed vel, beta does not

alpha.update\_location(*ONE*, *TEN*, *ZERO*);

beta.update\_location( *ONE*, (**float**) 9.9, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*ONE*, flag\_list.size());

*assertEquals*(alpha, flag\_list.get(*ZERO*));

location\_spread = alpha.get\_location();

*assertEquals*(*ZERO*, location\_spread[*ONE*], *ZERO*);

// beta has changed vel, alpha does not

omicron.clean\_list();

beta.update\_location(*ZERO*, *TEN*, *ZERO*);

alpha.update\_location(*ONE*, *TEN*, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*TWO*, flag\_list.size());

*assertEquals*(alpha, flag\_list.get(*ZERO*));

*assertEquals*(beta, flag\_list.get(*ONE*));

location\_spread = beta.get\_location();

*assertEquals*(*ZERO*, location\_spread[*ONE*], *ZERO*);

location\_spread = alpha.get\_location();

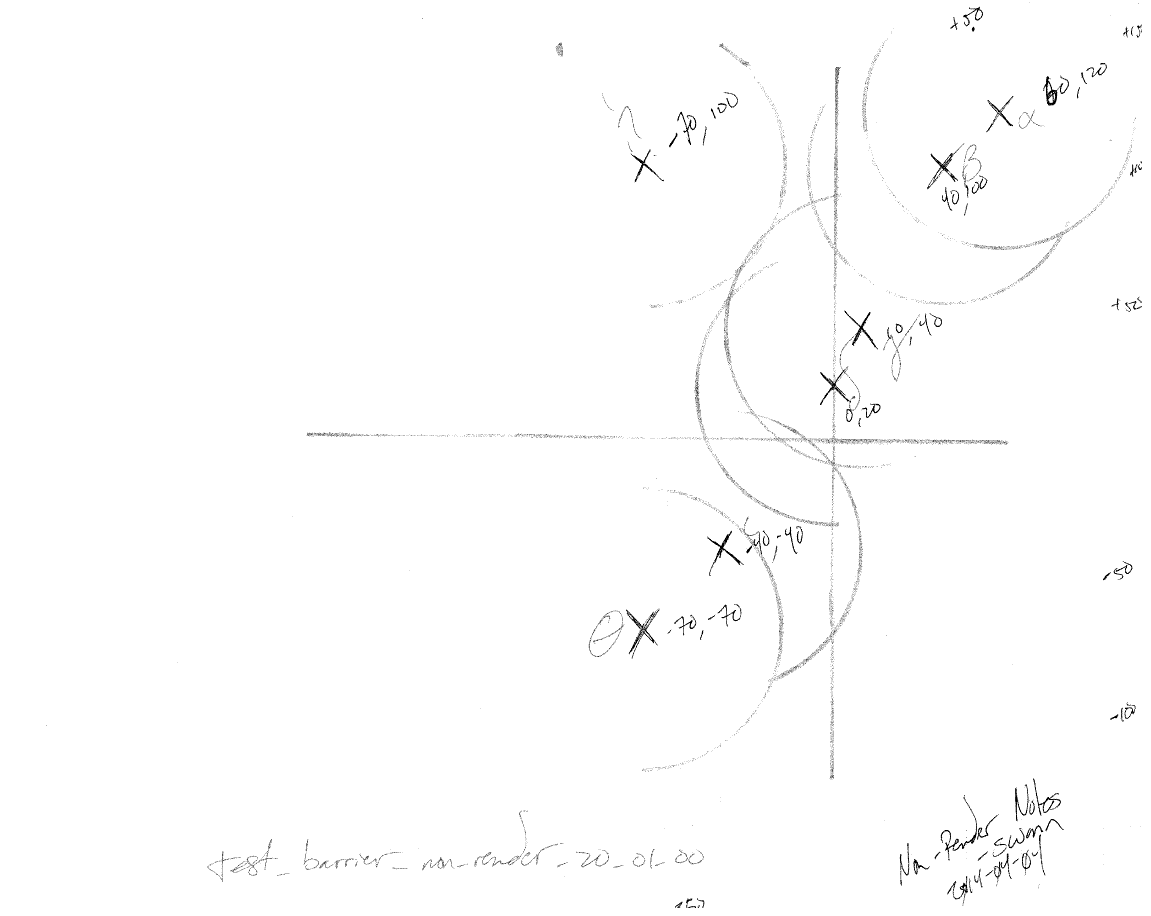
*assertEquals*(*ZERO*, location\_spread[*ONE*], *ZERO*);

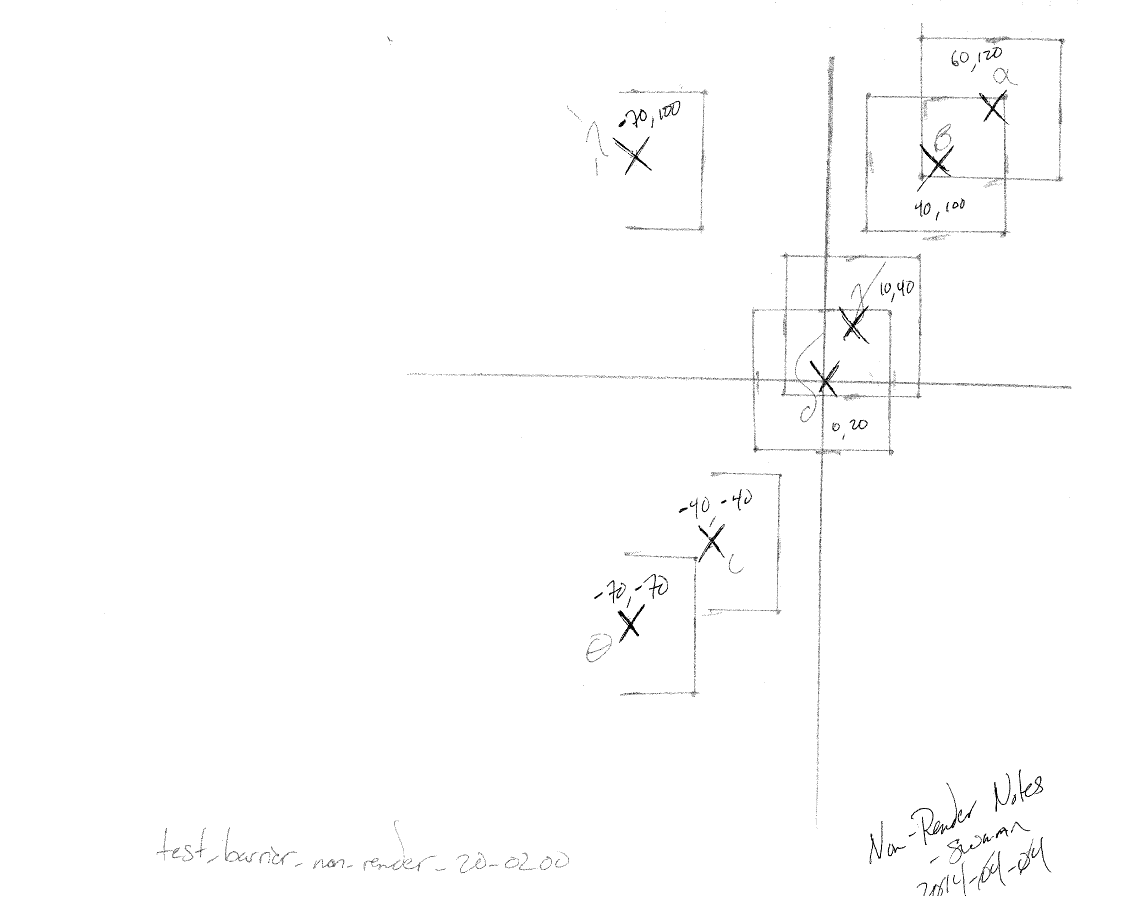
}

module :: Item/Non\_Render\_Barrier\_Event.java

test/design :: test\_non\_render\_barrier\_20\_01\_00\_check\_execute && test\_non\_render\_barrier\_20\_02\_00\_check\_execute

description :: design of the specific check and execute behaviors for bounce and pass through barriers





test notes :: explores functionality associated with non render barriers

// Non\_Render\_Barrier\_Event.check\_execute()

@Test

**public** **void** test\_barrier\_non\_render\_20\_01\_00\_check\_execute() {

// The setup

Ball alpha, beta, gamma, delta, iota, theta, eta;

Cannon the\_cannon;

Non\_Render\_Barrier\_Event omicron;

LinkedList<Actor\_Object> flag\_list;

ArrayList<Actor\_Object> the\_list = **new** ArrayList<Actor\_Object>();

//

// X Axis Sweep

//

// Battery One --> No changes yet

alpha = Ball.*generate\_Baseball*(60, 120, *ZERO*);

beta = Ball.*generate\_Baseball*(40, 100, *ZERO*);

gamma = Ball.*generate\_Baseball*(10, 40, *ZERO*);

delta = Ball.*generate\_Baseball*(0, 20, *ZERO*);

iota = Ball.*generate\_Baseball*(-40, -40, *ZERO*);

theta = Ball.*generate\_Baseball*(-70, -70, *ZERO*);

eta = Ball.*generate\_Baseball*(-70, 100, *ZERO*);

the\_cannon = **new** Cannon(*ZERO*, *ZERO*, *ZERO*);

omicron = **new** Non\_Render\_Barrier\_Event( *ZERO*,

Barrier\_Event.axis\_enumeration.*Y*,

Barrier\_Event.axis\_control.*STAY\_HIGHER*);

the\_list.add(alpha);

the\_list.add(beta);

the\_list.add(gamma);

the\_list.add(delta);

the\_list.add(iota);

the\_list.add(theta);

the\_list.add(eta);

the\_list.add(the\_cannon);

// Should have two objects not visible

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*THREE*, flag\_list.size());

*assertEquals*(alpha.get\_visibility(), **true**);

*assertEquals*(beta.get\_visibility(), **true**);

*assertEquals*(gamma.get\_visibility(), **true**);

*assertEquals*(delta.get\_visibility(), **true**);

*assertEquals*(iota.get\_visibility(), **false**);

*assertEquals*(theta.get\_visibility(), **false**);

*assertEquals*(eta.get\_visibility(), **false**);

*assertEquals*(the\_cannon.get\_visibility(), **true**);

// Movement and re-check

omicron.clean\_list();

alpha.update\_location(10, 1000, *ZERO*);

beta.update\_location(-400, 35, *ZERO*);

gamma.update\_location(-50, *ZERO*, *ZERO*);

delta.update\_location(-24, *ZERO*, *ZERO*);

iota.update\_location(-51, *ZERO*, *ZERO*);

the\_cannon.update\_location(-1000, *ZERO*, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*FIVE*, flag\_list.size());

*assertEquals*(alpha.get\_visibility(), **true**);

*assertEquals*(beta.get\_visibility(), **false**);

*assertEquals*(gamma.get\_visibility(), **false**);

*assertEquals*(delta.get\_visibility(), **true**);

*assertEquals*(iota.get\_visibility(), **false**);

*assertEquals*(theta.get\_visibility(), **false**);

*assertEquals*(eta.get\_visibility(), **false**);

}

// Non\_Render\_Barrier\_Event.check\_execute()

@Test

**public** **void** test\_barrier\_non\_render\_20\_02\_00\_check\_execute() {

// The setup

Standard\_Mass alpha, beta, gamma, delta, iota, theta, eta;

Cannon the\_cannon;

Non\_Render\_Barrier\_Event omicron;

LinkedList<Actor\_Object> flag\_list;

ArrayList<Actor\_Object> the\_list = **new** ArrayList<Actor\_Object>();

//

// X Axis Sweep

//

// Battery One --> No changes yet

alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(60, 120, *ZERO*);

beta = Standard\_Mass.*generate\_fifty\_g\_mass*(40, 100, *ZERO*);

gamma = Standard\_Mass.*generate\_fifty\_g\_mass*(10, 40, *ZERO*);

delta = Standard\_Mass.*generate\_fifty\_g\_mass*(0, 20, *ZERO*);

iota = Standard\_Mass.*generate\_fifty\_g\_mass*(-40, -40, *ZERO*);

theta = Standard\_Mass.*generate\_fifty\_g\_mass*(-70, -70, *ZERO*);

eta = Standard\_Mass.*generate\_fifty\_g\_mass*(-70, 100, *ZERO*);

the\_cannon = **new** Cannon(*ZERO*, *ZERO*, *ZERO*);

omicron = **new** Non\_Render\_Barrier\_Event( *ZERO*,

Barrier\_Event.axis\_enumeration.*Y*,

Barrier\_Event.axis\_control.*STAY\_HIGHER*);

the\_list.add(alpha);

the\_list.add(beta);

the\_list.add(gamma);

the\_list.add(delta);

the\_list.add(iota);

the\_list.add(theta);

the\_list.add(eta);

the\_list.add(the\_cannon);

// Should have two objects not visible

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*THREE*, flag\_list.size());

*assertEquals*(alpha.get\_visibility(), **true**);

*assertEquals*(beta.get\_visibility(), **true**);

*assertEquals*(gamma.get\_visibility(), **true**);

*assertEquals*(delta.get\_visibility(), **true**);

*assertEquals*(iota.get\_visibility(), **false**);

*assertEquals*(theta.get\_visibility(), **false**);

*assertEquals*(eta.get\_visibility(), **false**);

*assertEquals*(the\_cannon.get\_visibility(), **true**);

// Movement and re-check

omicron.clean\_list();

alpha.update\_location(10, 1000, *ZERO*);

beta.update\_location(-400, 35, *ZERO*);

gamma.update\_location(-50, *ZERO*, *ZERO*);

delta.update\_location(-10, *ZERO*, *ZERO*);

iota.update\_location(-51, *ZERO*, *ZERO*);

the\_cannon.update\_location(-1000, *ZERO*, *ZERO*);

omicron.check(the\_list);

omicron.execute();

flag\_list = omicron.get\_list();

*assertEquals*(*FIVE*, flag\_list.size());

*assertEquals*(alpha.get\_visibility(), **true**);

*assertEquals*(beta.get\_visibility(), **false**);

*assertEquals*(gamma.get\_visibility(), **false**);

*assertEquals*(delta.get\_visibility(), **true**);

*assertEquals*(iota.get\_visibility(), **false**);

*assertEquals*(theta.get\_visibility(), **false**);

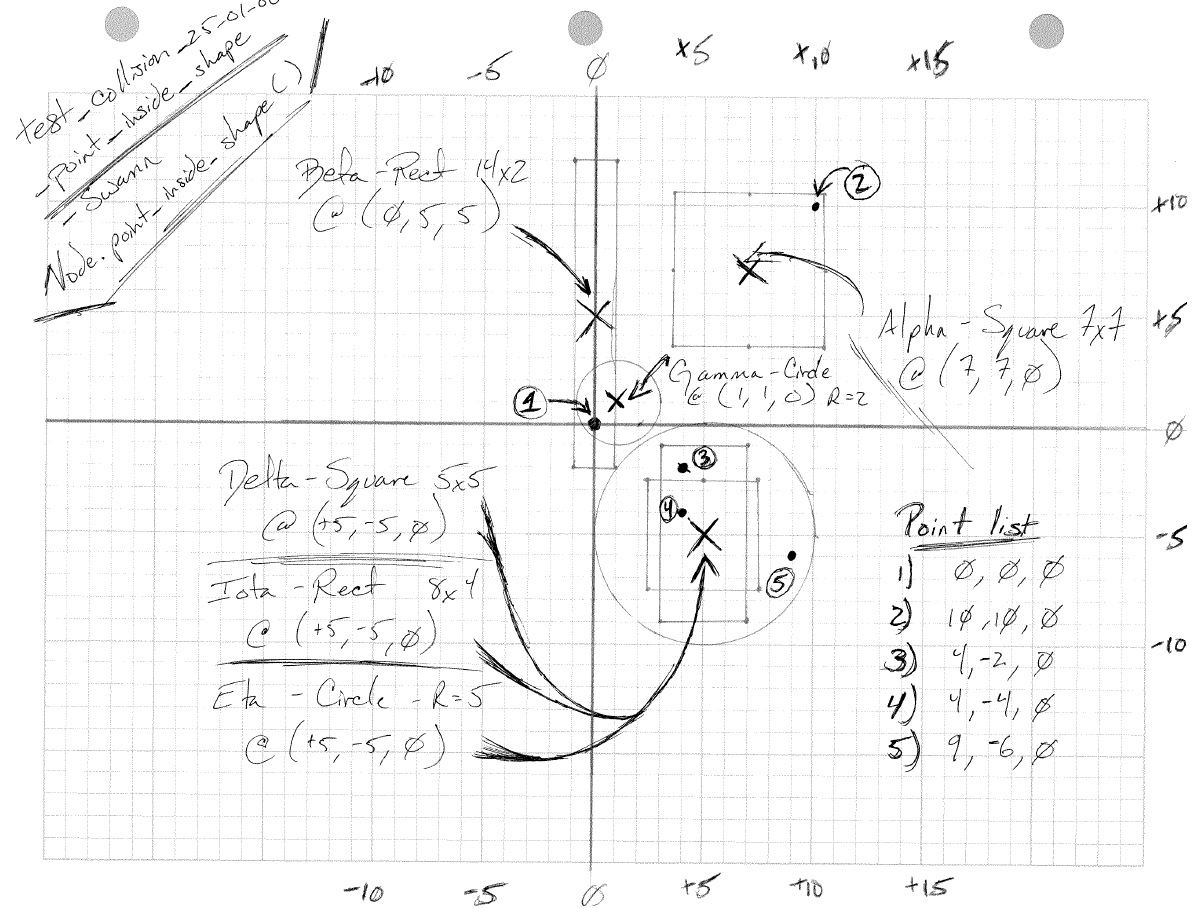
*assertEquals*(eta.get\_visibility(), **false**);

}

module :: Item/Collision.java

test/design :: test\_collision\_25\_01\_00\_point\_inside\_shape

description :: design of the specific check indicating whether a given node is within another shape



test notes :: explores support functionality determining whether a given shape’s node exists within another shape

// Collision.point\_inside\_shape()

// --> Heterogeneous Test

@Test

**public** **void** test\_collision\_25\_01\_00\_point\_inside\_shape() {

// The setup

Node one = **new** Node(0, 0, 0),

two = **new** Node(10, 10, 0),

tre = **new** Node(4, -2, 0),

four = **new** Node(4, -4, 0),

five = **new** Node(9, -6, 0);

// Simple Shapes

Square alpha = **new** Square( 7, 7, 7, 7, 7);

Square delta = **new** Square( 5, -5, 0, 5, 5);

Rectangle beta = **new** Rectangle( 0, 5, 0, 14, 2);

Rectangle iota = **new** Rectangle( 5, -5, 0, 8, 4);

Circle gamma = **new** Circle( 1, 1, 0, 2);

Circle eta = **new** Circle( 5, -5, 0, 5);

// Complex Shape

Shape omega = **new** Shape( 5, -5, 0);

omega.add\_shape(delta);

omega.add\_shape(iota);

omega.add\_shape(eta);

// Alpha Shape

*assertEquals*(Collision.*point\_inside\_shape*(one, alpha), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(two, alpha), **true**);

*assertEquals*(Collision.*point\_inside\_shape*(tre, alpha), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(four, alpha), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(five, alpha), **false**);

// Beta Shape

*assertEquals*(Collision.*point\_inside\_shape*(one, beta), **true**);

*assertEquals*(Collision.*point\_inside\_shape*(two, beta), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(tre, beta), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(four, beta), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(five, beta), **false**);

// Gamma Shape

*assertEquals*(Collision.*point\_inside\_shape*(one, gamma), **true**);

*assertEquals*(Collision.*point\_inside\_shape*(two, gamma), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(tre, gamma), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(four, gamma), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(five, gamma), **false**);

// Delta Shape

*assertEquals*(Collision.*point\_inside\_shape*(one, delta), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(two, delta), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(tre, delta), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(four, delta), **true**);

*assertEquals*(Collision.*point\_inside\_shape*(five, delta), **false**);

// Iota Shape

*assertEquals*(Collision.*point\_inside\_shape*(one, iota), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(two, iota), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(tre, iota), **true**);

*assertEquals*(Collision.*point\_inside\_shape*(four, iota), **true**);

*assertEquals*(Collision.*point\_inside\_shape*(five, iota), **false**);

// Eta Shape

*assertEquals*(Collision.*point\_inside\_shape*(one, eta), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(two, eta), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(tre, eta), **true**);

*assertEquals*(Collision.*point\_inside\_shape*(four, eta), **true**);

*assertEquals*(Collision.*point\_inside\_shape*(five, eta), **true**);

// Omega Shape

*assertEquals*(Collision.*point\_inside\_shape*(one, omega), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(two, omega), **false**);

*assertEquals*(Collision.*point\_inside\_shape*(tre, omega), **true**);

*assertEquals*(Collision.*point\_inside\_shape*(four, omega), **true**);

*assertEquals*(Collision.*point\_inside\_shape*(five, omega), **true**);

}

function notes :: implementation of point\_inside\_shape()

/\*\*

\* Controls the logic path for determining whether or not a

\*

\* **@param** vertex\_node : The point/node for comparison

\* **@param** target\_shape : The show for comparison

\*

\* **@return** { boolean } : Boolean variable representing whether the point is

\* inside the shape.

\*/

**public** **static** **boolean** point\_inside\_shape( Node vertex\_node, Shape target\_shape ) {

**if** ( target\_shape.is\_composite() ) {

**return** Collision.*point\_inside\_complex\_shape*( vertex\_node, target\_shape );

}

**else** {

**return** Collision.*point\_inside\_simple\_shape*( vertex\_node, target\_shape );

}

} // end Collision.point\_inside\_shape()

/\*\*

\* Determines whether or not a point is within the bounds of a simple shape.

\*

\* **@param** vertex\_node : The point/node for comparison

\* **@param** target\_shape : The show for comparison

\*

\* **@return** { boolean } : Boolean variable representing whether the point is

\* inside the shape.

\*/

**public** **static** **boolean** point\_inside\_simple\_shape( Node vertex\_node, Shape target\_shape ) {

**boolean** output = **false**;

// What to do if Circle shape.

**if** (target\_shape **instanceof** Circle){

// Point must be within radius only.

**if** (target\_shape.get\_radius() > Node.*distance\_between\_nodes*(vertex\_node,

target\_shape.get\_primary\_node())) {

output = **true**;

} // end internal IF

} // end instanceof IF

// What to do if NOT Circle shape.

**else** {

Node head\_node = target\_shape.get\_head\_point();

// tracking variable : if NOT ~ 360 at the end, node is not inside

**float** tracker = *ZERO*;

**while**( head\_node.has\_next() ) {

**if** ( head\_node.is\_edge\_closed() ) {

tracker += Node.*get\_angle*(vertex\_node,

head\_node,

head\_node.get\_next());

} // end internal IF

head\_node = head\_node.get\_next();

} // end WHILE

Node top\_node = target\_shape.get\_head\_point();

// Wrap around to front node.

**if** ( head\_node.is\_edge\_closed() ) {

tracker += Node.*get\_angle*( vertex\_node,

head\_node,

top\_node);

} // end IF

// If angle summation ~360 --> node is within the shape

//

// This therom was originally grabbed from Wolfram Alpha's website

// I have since not been able to locate the function. If I can find

// another reference, I will supply it.

// ~ swann 2013-11-19

//

**if** ( tracker < *TOP* && tracker > *BOTTOM* ) {

output = **true**;

}

} // end ELSE

**return** output;

} // end Collision.point\_inside\_simple\_shape()

/\*\*

\* Determines whether or not a point is within the bounds of a complex shape.

\*

\* **@param** vertex\_node : The point/node for comparison

\* **@param** target\_shape : The show for comparison

\*

\* **@return** output : Boolean variable representing whether the point is

\* inside the shape.

\*/

**public** **static** **boolean** point\_inside\_complex\_shape( Node vertex\_node, Shape target\_shape ) {

**boolean** output = **false**;

**for** (**int** i = *ZERO* ; i < target\_shape.get\_composite\_list().size() ; i++ ) {

**if** (Collision.*point\_inside\_shape*(vertex\_node,

target\_shape.get\_composite\_list().get(i))) {

output = **true**;

}

} // end FOR

**return** output;

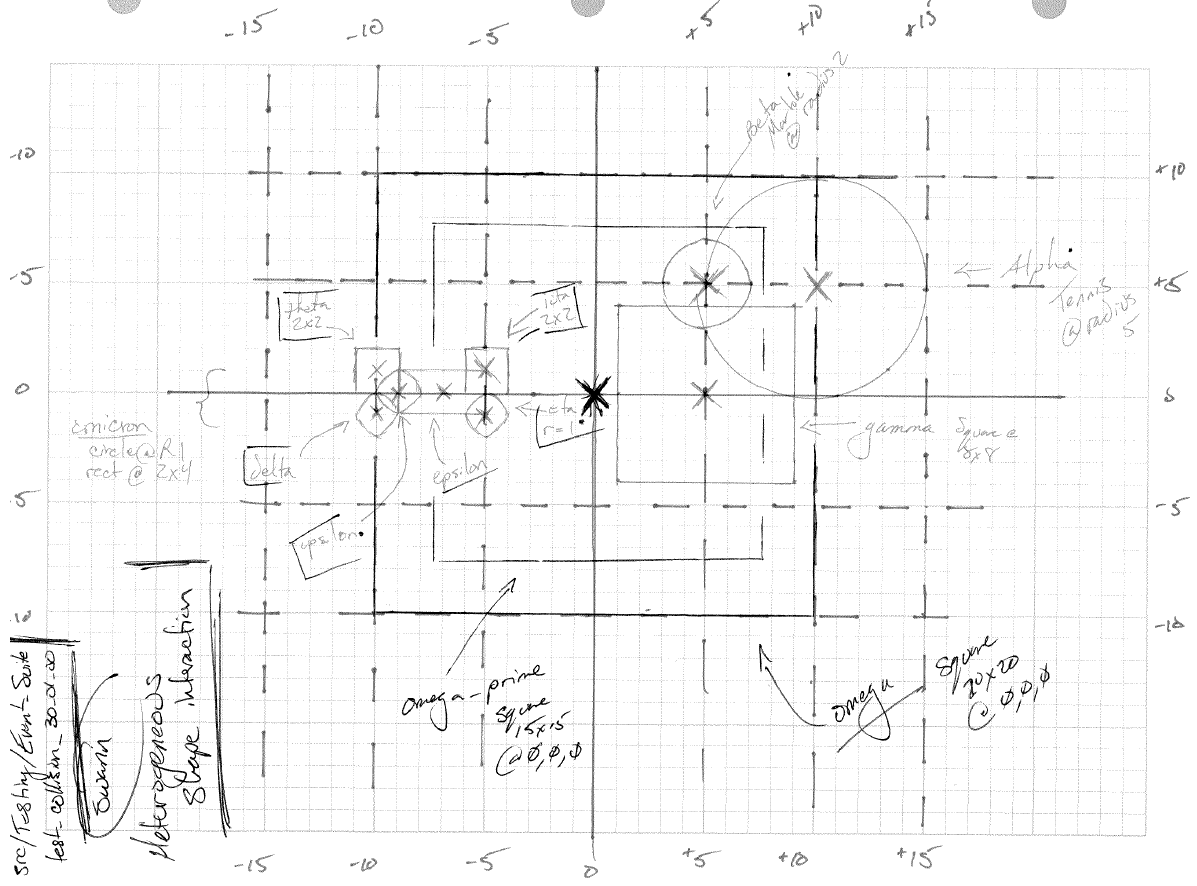
} // end Collision.point\_inside\_complex\_shape()

module :: Item/Collision.java

test/design :: test\_collision\_30\_01\_00\_collision\_check

function :: Collision.check() && Collision.execute()

description :: design of the specific check determining whether or not different shape objects collide



test notes :: explores functionality determining whether different shapes collide

// Collision.collision\_check()

// --> Heterogeneous Test

@Test

**public** **void** test\_collision\_30\_01\_00\_collision\_check() {

// The Setup

// Simple Shapes

Circle alpha = **new** Circle(*TEN*, *FIVE*, *FIVE*, *FIVE*);

Circle beta = **new** Circle(*FIVE*, *FIVE*, *FIVE*, *TWO*);

Circle delta = **new** Circle(-10, -1, -1, *ONE*);

Circle eta = **new** Circle(-5, -1, -1, *ONE*);

Circle upsilon = **new** Circle(-9, *ZERO*, *ZERO*, *ONE*);

Square gamma = **new** Square(*FIVE*, *ZERO*, *ZERO*, *EIGHT*, *EIGHT*);

Square theta = **new** Square(-10, *ONE*, *ONE*, *TWO*, *TWO*);

Square iota = **new** Square(-5, *ONE*, *ONE*, *TWO*, *TWO*);

Square omega = **new** Square(*ZERO*, *ZERO*, *ZERO*, 20, 20);

Square omega\_p = **new** Square(*ZERO*, *ZERO*, *ZERO*, 15, 15);

Rectangle epsilon = **new** Rectangle(-7, *ZERO*, *ZERO*, *TWO*, *FOUR*);

// Complex Shapes

Shape omicron = **new** Shape(-9, *ZERO*, *ZERO*);

omicron.add\_shape( upsilon );

omicron.add\_shape( epsilon );

omicron.calculate\_radius();

*assertEquals*(omicron.get\_radius(), Math.*pow*(17, *A\_HALF*), *A\_HUNDREDTH*);

// Alpha Shape

*assertEquals*(Collision.*collision\_check*( alpha, alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( alpha, beta ), **true**);

*assertEquals*(Collision.*collision\_check*( alpha, gamma ), **true**);

*assertEquals*(Collision.*collision\_check*( alpha, delta ), **false**);

*assertEquals*(Collision.*collision\_check*( alpha, eta ), **false**);

*assertEquals*(Collision.*collision\_check*( alpha, upsilon ), **false**);

*assertEquals*(Collision.*collision\_check*( alpha, epsilon ), **false**);

*assertEquals*(Collision.*collision\_check*( alpha, iota ), **false**);

*assertEquals*(Collision.*collision\_check*( alpha, theta ), **false**);

*assertEquals*(Collision.*collision\_check*( alpha, omega ), **true**);

*assertEquals*(Collision.*collision\_check*( alpha, omega\_p ), **true**);

// Beta Shape

*assertEquals*(Collision.*collision\_check*( beta, alpha ), **true**);

*assertEquals*(Collision.*collision\_check*( beta, beta ), **false**);

*assertEquals*(Collision.*collision\_check*( beta, gamma ), **true**);

*assertEquals*(Collision.*collision\_check*( beta, delta ), **false**);

*assertEquals*(Collision.*collision\_check*( beta, eta ), **false**);

*assertEquals*(Collision.*collision\_check*( beta, upsilon ), **false**);

*assertEquals*(Collision.*collision\_check*( beta, epsilon ), **false**);

*assertEquals*(Collision.*collision\_check*( beta, iota ), **false**);

*assertEquals*(Collision.*collision\_check*( beta, theta ), **false**);

*assertEquals*(Collision.*collision\_check*( beta, omega ), **true**);

*assertEquals*(Collision.*collision\_check*( beta, omega\_p ), **true**);

// Gamma Shape

*assertEquals*(Collision.*collision\_check*( gamma, alpha ), **true**);

*assertEquals*(Collision.*collision\_check*( gamma, beta ), **true**);

*assertEquals*(Collision.*collision\_check*( gamma, gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( gamma, delta ), **false**);

*assertEquals*(Collision.*collision\_check*( gamma, eta ), **false**);

*assertEquals*(Collision.*collision\_check*( gamma, upsilon ), **false**);

*assertEquals*(Collision.*collision\_check*( gamma, epsilon ), **false**);

*assertEquals*(Collision.*collision\_check*( gamma, iota ), **false**);

*assertEquals*(Collision.*collision\_check*( gamma, theta ), **false**);

*assertEquals*(Collision.*collision\_check*( gamma, omega ), **true**);

*assertEquals*(Collision.*collision\_check*( gamma, omega\_p ), **true**);

// Delta Shape

*assertEquals*(Collision.*collision\_check*( delta, alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( delta, beta ), **false**);

*assertEquals*(Collision.*collision\_check*( delta, gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( delta, delta ), **false**);

*assertEquals*(Collision.*collision\_check*( delta, eta ), **false**);

*assertEquals*(Collision.*collision\_check*( delta, upsilon ), **true**);

*assertEquals*(Collision.*collision\_check*( delta, epsilon ), **false**);

*assertEquals*(Collision.*collision\_check*( delta, iota ), **false**);

*assertEquals*(Collision.*collision\_check*( delta, theta ), **true**);

*assertEquals*(Collision.*collision\_check*( delta, omega ), **true**);

*assertEquals*(Collision.*collision\_check*( delta, omega\_p ), **false**);

// Eta Shape

*assertEquals*(Collision.*collision\_check*( eta, alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( eta, beta ), **false**);

*assertEquals*(Collision.*collision\_check*( eta, gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( eta, delta ), **false**);

*assertEquals*(Collision.*collision\_check*( eta, eta ), **false**);

*assertEquals*(Collision.*collision\_check*( eta, upsilon ), **false**);

*assertEquals*(Collision.*collision\_check*( eta, epsilon ), **true**);

*assertEquals*(Collision.*collision\_check*( eta, iota ), **true**);

*assertEquals*(Collision.*collision\_check*( eta, theta ), **false**);

*assertEquals*(Collision.*collision\_check*( eta, omega ), **true**);

*assertEquals*(Collision.*collision\_check*( eta, omega\_p ), **true**);

// Iota Shape

*assertEquals*(Collision.*collision\_check*( iota, alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( iota, beta ), **false**);

*assertEquals*(Collision.*collision\_check*( iota, gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( iota, delta ), **false**);

*assertEquals*(Collision.*collision\_check*( iota, eta ), **true**);

*assertEquals*(Collision.*collision\_check*( iota, upsilon ), **false**);

*assertEquals*(Collision.*collision\_check*( iota, epsilon ), **true**);

*assertEquals*(Collision.*collision\_check*( iota, iota ), **false**);

*assertEquals*(Collision.*collision\_check*( iota, theta ), **false**);

*assertEquals*(Collision.*collision\_check*( iota, omega ), **true**);

*assertEquals*(Collision.*collision\_check*( iota, omega\_p ), **true**);

// Theta Shape

*assertEquals*(Collision.*collision\_check*( theta, alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( theta, beta ), **false**);

*assertEquals*(Collision.*collision\_check*( theta, gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( theta, delta ), **true**);

*assertEquals*(Collision.*collision\_check*( theta, eta ), **false**);

*assertEquals*(Collision.*collision\_check*( theta, upsilon ), **true**);

*assertEquals*(Collision.*collision\_check*( theta, epsilon ), **true**);

*assertEquals*(Collision.*collision\_check*( theta, iota ), **false**);

*assertEquals*(Collision.*collision\_check*( theta, theta ), **false**);

*assertEquals*(Collision.*collision\_check*( theta, omega ), **true**);

*assertEquals*(Collision.*collision\_check*( theta, omega\_p ), **false**);

// Upsilon Shape

*assertEquals*(Collision.*collision\_check*( upsilon, alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( upsilon, beta ), **false**);

*assertEquals*(Collision.*collision\_check*( upsilon, gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( upsilon, delta ), **true**);

*assertEquals*(Collision.*collision\_check*( upsilon, eta ), **false**);

*assertEquals*(Collision.*collision\_check*( upsilon, upsilon ), **false**);

*assertEquals*(Collision.*collision\_check*( upsilon, epsilon ), **true**);

*assertEquals*(Collision.*collision\_check*( upsilon, iota ), **false**);

*assertEquals*(Collision.*collision\_check*( upsilon, theta ), **true**);

*assertEquals*(Collision.*collision\_check*( upsilon, omega ), **true**);

*assertEquals*(Collision.*collision\_check*( upsilon, omega\_p ), **false**);

// Epsilon Shape

*assertEquals*(Collision.*collision\_check*( epsilon, alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( epsilon, beta ), **false**);

*assertEquals*(Collision.*collision\_check*( epsilon, gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( epsilon, delta ), **false**);

*assertEquals*(Collision.*collision\_check*( epsilon, eta ), **true**);

*assertEquals*(Collision.*collision\_check*( epsilon, upsilon ), **true**);

*assertEquals*(Collision.*collision\_check*( epsilon, epsilon ), **false**);

*assertEquals*(Collision.*collision\_check*( epsilon, iota ), **true**);

*assertEquals*(Collision.*collision\_check*( epsilon, theta ), **true**);

*assertEquals*(Collision.*collision\_check*( epsilon, omega ), **true**);

*assertEquals*(Collision.*collision\_check*( epsilon, omega\_p ), **true**);

// Omicron Shape --> Forward

*assertEquals*(Collision.*collision\_check*( omicron, alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( omicron, beta ), **false**);

*assertEquals*(Collision.*collision\_check*( omicron, gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( omicron, delta ), **true**);

*assertEquals*(Collision.*collision\_check*( omicron, eta ), **true**);

*assertEquals*(Collision.*collision\_check*( omicron, iota ), **true**);

*assertEquals*(Collision.*collision\_check*( omicron, theta ), **true**);

*assertEquals*(Collision.*collision\_check*( omicron, omega ), **true**);

*assertEquals*(Collision.*collision\_check*( omicron, omega\_p ), **true**);

// Omicron Shape --> Backward

*assertEquals*(Collision.*collision\_check*( alpha, omicron ), **false**);

*assertEquals*(Collision.*collision\_check*( beta , omicron ), **false**);

*assertEquals*(Collision.*collision\_check*( gamma, omicron ), **false**);

*assertEquals*(Collision.*collision\_check*( delta, omicron ), **true**);

*assertEquals*(Collision.*collision\_check*( eta, omicron ), **true**);

*assertEquals*(Collision.*collision\_check*( iota, omicron ), **true**);

*assertEquals*(Collision.*collision\_check*( theta, omicron ), **true**);

*assertEquals*(Collision.*collision\_check*( omega, omicron ), **true**);

*assertEquals*(Collision.*collision\_check*( omega\_p, omicron ), **true**);

}

Fuction notes :: implementation of primary and support logic for execute() and check()

/\*\*

\* This function controls the routing logic as to which version of the execute functions

\* ought be called. A set of size two can use a simple update mechanism. A set of larger

\* sizes requires a more complex method.

\*/

**public** **void** execute( ) **throws** CloneNotSupportedException, RuntimeException {

// A working set is needed.

ArrayList<Actor\_Object> working\_set;

// While there are more collision sets to execute.

**while** ( **this**.flagged\_object\_list.size() > *ZERO* ){

working\_set = dequeue\_set();

// Shouldn't have a set of size zero

**if** ( working\_set.size() == *ZERO* ) {

**throw** **new** RuntimeException("working set size is ZERO, too low for processing a COLLISION");

}

// Shouldn't have a set of size one

**else** **if** ( working\_set.size() == *ONE* ) {

**throw** **new** RuntimeException("working set size is ONE, too low for processing a COLLISION");

}

// For size two, use the simpler calculation method

**else** **if** ( working\_set.size() == *TWO* ) {

**this**.execute\_collision\_size\_two( working\_set );

}

// For size three or larger, use the more complex method

**else** {

**this**.execute\_collision\_size\_n( working\_set );

} // end IF-ELSE Chain

} // end while connected to this.flagged\_object\_list

} // end Collision.execute()

/\*\*

\* This function handles the execution of a collision event on a pair of objects.

\*

\* **@param** set : ArrayList of two Actor\_Objects

\*/

**public** **void** execute\_collision\_size\_two( ArrayList<Actor\_Object> set ) **throws** CloneNotSupportedException, RuntimeException {

Iterator<Actor\_Object> working\_set\_iterator = set.iterator();

// Get the two Actor\_Objects for this collision.

Actor\_Object actor\_one;

Actor\_Object actor\_two;

// Initialize both to the first to allow 2-at-a-time iteration.

actor\_one = actor\_two = working\_set\_iterator.next();

**while** ( working\_set\_iterator.hasNext() ){

// Pull the new actor from the working set.

actor\_one = actor\_two;

actor\_two = working\_set\_iterator.next();

Vector[] vector\_list = **this**.execute\_collision\_support(actor\_one, actor\_two, actor\_one.get\_mass() + actor\_two.get\_mass());

Vector new\_a = Vector.*add*(**new** Vector(actor\_one.get\_velocity()), vector\_list[*ZERO*]);

Vector new\_b = Vector.*add*(**new** Vector(actor\_two.get\_velocity()), vector\_list[*ONE*]);

// Update each actor with the adjusted velocities.

actor\_one.update\_velocity(new\_a.get\_x\_comp(), new\_a.get\_y\_comp(), new\_a.get\_z\_comp());

actor\_two.update\_velocity(new\_b.get\_x\_comp(), new\_b.get\_y\_comp(), new\_b.get\_z\_comp());

// Calculate the popout distance between the two objects.

**float** diff = actor\_one.get\_shape().get\_radius() +

actor\_two.get\_shape().get\_radius() -

Node.*distance\_between\_nodes*( actor\_one.get\_shape().get\_primary\_node(),

actor\_two.get\_shape().get\_primary\_node() );

Vector a\_pos = **new** Vector(actor\_one.get\_location());

Vector b\_pos = **new** Vector(actor\_two.get\_location());

// Calculate the ratio of the diff distance which should be applied to each object.

**float** diff\_one = diff \* (actor\_one.get\_shape().get\_radius() / (

actor\_one.get\_shape().get\_radius() +

actor\_two.get\_shape().get\_radius()));

**float** diff\_two = diff \* (actor\_two.get\_shape().get\_radius() / (

actor\_one.get\_shape().get\_radius() +

actor\_two.get\_shape().get\_radius()));

// Calculate the normal of impact for this collision (Normal vector for collision plane).

Vector norm\_of\_impact = Vector.*normalize*(Vector.*subtract*(b\_pos, a\_pos));

a\_pos = Vector.*add*(a\_pos, Vector.*scalar\_multiply*(norm\_of\_impact, -diff\_one));

b\_pos = Vector.*add*(b\_pos, Vector.*scalar\_multiply*(norm\_of\_impact, diff\_two));

actor\_one.update\_location(a\_pos.get\_x\_comp(), a\_pos.get\_y\_comp(), a\_pos.get\_z\_comp());

actor\_two.update\_location(b\_pos.get\_x\_comp(), b\_pos.get\_y\_comp(), b\_pos.get\_z\_comp());

} // end while

} // end Collision.execute\_collision\_size\_two()

/\*\*

\* This function handles the execution of a collision event on a number of objects

\* greater than two.

\*

\* **@param** set : ArrayList of three or more Actor\_Objects

\*/

**public** **void** execute\_collision\_size\_n( ArrayList<Actor\_Object> set )

**throws** CloneNotSupportedException, RuntimeException {

// Working object memory reservations

Actor\_Object first\_object, second\_object;

Vector[] vector\_list;

// Make a clone copy

ArrayList<Actor\_Object> copied\_set = Event\_Interaction.*clone\_list*( set );

**float** total\_mass = *ZERO*;

**for** ( **int** i = *ZERO* ; i < set.size() ; i++ ){

total\_mass += set.get(i).get\_mass();

}

// External loop

**for** ( **int** i = *ZERO* ; i <= set.size() - *TWO* ; i++ ) {

first\_object = set.get(i);

// Internal loop

**for** ( **int** j = i + *ONE* ; j <= set.size() - *ONE* ; j++ ) {

second\_object = set.get(j);

**float**[] a = first\_object.get\_velocity();

**float**[] b = second\_object.get\_velocity();

vector\_list = **this**.execute\_collision\_support(first\_object, second\_object, total\_mass);

Actor\_Object first\_objectx = copied\_set.get(i);

Actor\_Object second\_objectx = copied\_set.get(j);

a = first\_objectx.get\_velocity();

b = second\_objectx.get\_velocity();

Vector new\_a = Vector.*add*(**new** Vector(first\_objectx.get\_velocity()), vector\_list[*ZERO*]);

Vector new\_b = Vector.*add*(**new** Vector(second\_objectx.get\_velocity()), vector\_list[*ONE*]);

// Update each actor with the adjusted velocities.

first\_objectx.update\_velocity(new\_a.get\_x\_comp(), new\_a.get\_y\_comp(), new\_a.get\_z\_comp());

second\_objectx.update\_velocity(new\_b.get\_x\_comp(), new\_b.get\_y\_comp(), new\_b.get\_z\_comp());

} // end internal loop

} // end external loop

// Unpacking the clone

**for** ( **int** i = *ZERO* ; i < set.size() ; i++ ){

Actor\_Object the\_obj = set.get(i);

Actor\_Object updated\_obj = copied\_set.get(i);

//Vector v = Vector.scalar\_multiply(new Vector(updated\_obj.get\_velocity()), the\_obj.get\_mass() / total\_mass );

Vector v = **new** Vector(updated\_obj.get\_velocity());

the\_obj.update\_velocity(v.get\_x\_comp(), v.get\_y\_comp(), v.get\_z\_comp());

} // end for loop

} // end Collision.execute\_collision\_size\_n()

/\*\*

\* This function handles the execution of a collision event on a number of objects

\* greater than two.

\*

\* **@param** alpha : First of two Actor\_Objects

\* **@param** beta : Second of two Actor\_Objects

\*

\* **@return** {Vector[] } : Array of two Vector Adjustments

\*/

**public** Vector[] execute\_collision\_support( Actor\_Object alpha, Actor\_Object beta, **float** total\_mass )

**throws** CloneNotSupportedException, RuntimeException {

Actor\_Object actor\_one = (Actor\_Object)alpha.clone();

Actor\_Object actor\_two = (Actor\_Object) beta.clone();

// Get the actors' locations as vectors.

Vector a\_pos = **new** Vector(actor\_one.get\_location());

Vector b\_pos = **new** Vector(actor\_two.get\_location());

// Calculate the normal of impact for this collision (Normal vector for collision plane).

Vector norm\_of\_impact = Vector.*normalize*(Vector.*subtract*(b\_pos, a\_pos));

// Get the actors' velocities as vectors.

Vector a\_vel = **new** Vector(actor\_one.get\_velocity());

Vector b\_vel = **new** Vector(actor\_two.get\_velocity());

// Calculate the relative velocity between the two actors.

Vector relative\_velocity = Vector.*subtract*(b\_vel, a\_vel);

// Get the magnitude of the relative velocity along the collision normal.

**float** relative\_velocity\_along\_normal = Vector.*dot\_product*(relative\_velocity, norm\_of\_impact);

// Find the lower coefficient of restitution (elasticity)

**float** coeff\_of\_restitution = Math.*min*(actor\_one.get\_coefficient\_of\_restitution(), actor\_two.get\_coefficient\_of\_restitution());

// Calculate the impulse magnitude.

**float** impulse\_scalar = -(1+coeff\_of\_restitution) \* relative\_velocity\_along\_normal;

impulse\_scalar = impulse\_scalar / (1/actor\_one.get\_mass() + 1/actor\_two.get\_mass());

// Scale the normal of collision by the impulse magnitude.

Vector impulse\_along\_normal = Vector.*scalar\_multiply*(norm\_of\_impact, impulse\_scalar);

// Using mass ratios (to conserve mass / energy in the system), calculate the velocity adjustments for each actor.

Vector a\_adjust = Vector.*scalar\_multiply*(impulse\_along\_normal, -1/actor\_one.get\_mass());

Vector b\_adjust = Vector.*scalar\_multiply*(impulse\_along\_normal, +1/actor\_two.get\_mass());

Vector[] vector\_list = **new** Vector[*TWO*];

vector\_list[*ZERO*] = a\_adjust;

vector\_list[*ONE*] = b\_adjust;

**return** vector\_list;

} // end Collision.execute\_collision\_support()

/\*\*

\* Checks each object within the sim-space and determines if a boundary

\*

\* **@param** list\_of\_objects : All interactive objects within the sim-space.

\*

\* **@return** { boolean } : Representing whether or not something has an collision event.

\*/

**public** **boolean** check( ArrayList<Actor\_Object> list\_of\_objects ) {

ArrayList<Actor\_Object> flex\_list;

**boolean** checked = **false**;

**for** (**int** i = *ZERO* ; i < list\_of\_objects.size() ; i++) {

**for** (**int** j = i + *ONE* ; j < list\_of\_objects.size() ; j++) {

Actor\_Object first\_object = list\_of\_objects.get(i);

Actor\_Object second\_object = list\_of\_objects.get(j);

// short circuit operation if non-interactive event

**if** ( first\_object.get\_interactive() == **true** &&

second\_object.get\_interactive() == **true**) {

**if** ( Collision.*collision\_check*( first\_object.get\_shape(),

second\_object.get\_shape() )) {

flex\_list = **new** ArrayList<Actor\_Object>();

flex\_list.add( first\_object );

flex\_list.add(second\_object );

**this**.enqueue\_set( flex\_list );

checked = **true**;

} // end IF

} // end interactive check

} // end internal for-loop --> J var

} // end external for-loop --> I var

**return** checked;

} // end Collision.check()

/\*\*

\* Ascertains whether a point exists within a given shape.

\*

\* **@param** vertex\_node : The point/node for comparison

\* **@param** target\_shape : The show for comparison

\*

\* **@return** output : Boolean variable representing whether the point is

\* inside the shape.

\*/

**public** **static** **boolean** collision\_check( Shape shape\_one, Shape shape\_two ) {

**boolean** result = **false**;

// Identity Check should end quickly

**if** (shape\_one == shape\_two) {

result = **false**;

} // end short circuit

// Circle vs Circle Check

**else** **if** (shape\_one **instanceof** Circle && shape\_two **instanceof** Circle) {

result = Collision.*boundary\_sphere\_incident*( shape\_one, shape\_two );

} // end IF

**else** {

// Checks boundary spheres. Only enters point by point check if a sphere

// incident has taken place.

**if** ( Collision.*boundary\_sphere\_incident*(shape\_one, shape\_two) ){

**if** (shape\_one **instanceof** Circle) {

shape\_one.set\_head\_node(shape\_two);

}

**if** (shape\_two **instanceof** Circle) {

shape\_two.set\_head\_node(shape\_one);

}

// START FORWARD CHECK

**if** ( ! shape\_one.is\_composite() ){

Node head\_node = shape\_one.get\_head\_point();

**if** ( Collision.*point\_inside\_shape*(head\_node, shape\_two) ) {

result = **true**;

}

**while** ( result == **false** && head\_node.has\_next() ) {

**if** ( Collision.*point\_inside\_shape*(head\_node, shape\_two) ) {

result = **true**;

}

head\_node = head\_node.get\_next();

}// end while

} // end if

**else** {

ArrayList<Shape> the\_list = shape\_one.get\_composite\_list();

**for** (**int** i = *ZERO*; i < the\_list.size(); i++) {

Node head\_node = the\_list.get(i).get\_head\_point();

**if** ( Collision.*point\_inside\_shape*(head\_node, shape\_two) ) {

result = **true**;

}

**while** ( result == **false** && head\_node.has\_next() ) {

**if** ( Collision.*point\_inside\_shape*(head\_node, shape\_two) ) {

result = **true**;

}

head\_node = head\_node.get\_next();

}// end while

} // end for-loop

} // end else

// END FORWARD CHECK

// START REVERSE CHECK

**if** ( ! shape\_two.is\_composite() ){

Node head\_node = shape\_two.get\_head\_point();

**if** ( Collision.*point\_inside\_shape*(head\_node, shape\_one) ) {

result = **true**;

}

**while** ( result == **false** && head\_node.has\_next() ) {

**if** ( Collision.*point\_inside\_shape*(head\_node, shape\_one) ) {

result = **true**;

}

head\_node = head\_node.get\_next();

}// end while

} // end if

**else** {

ArrayList<Shape> the\_list = shape\_two.get\_composite\_list();

**for** (**int** i = *ZERO*; i < the\_list.size(); i++) {

Node head\_node = the\_list.get(i).get\_head\_point();

**if** ( Collision.*point\_inside\_shape*(head\_node, shape\_one) ) {

result = **true**;

}

**while** ( result == **false** && head\_node.has\_next() ) {

**if** ( Collision.*point\_inside\_shape*(head\_node, shape\_one) ) {

result = **true**;

}

head\_node = head\_node.get\_next();

}// end while

} // end for-loop

} // end else

// END REVERSE CHECK

} // end Boundary Sphere incident check

} // end ELSE

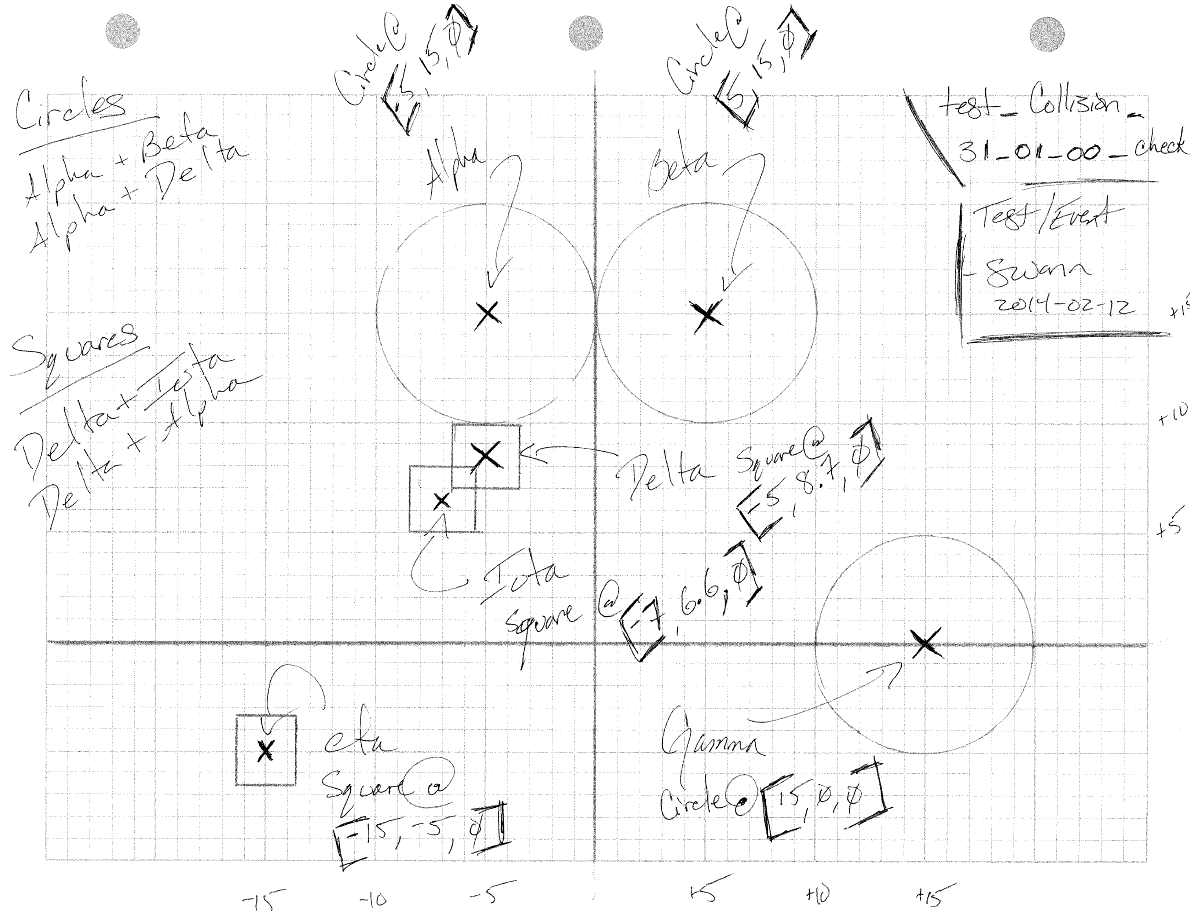
**return** result;

} // end Collision.collision\_check()

module :: Item/Collision.java

test/design :: test\_collision\_31\_01\_00\_collision\_check

description :: design of the specific check determining whether or not different shape objects collide



test notes :: explores functionality determining whether different shapes collide

// Collision.collision\_check()

// --> Heterogeneous Test

@Test

**public** **void** test\_collision\_31\_01\_00\_collision\_check() {

// The Setup

Circle Alpha = **new** Circle( -5, 15, 0),

Beta = **new** Circle( 5, 15, 0),

Gamma = **new** Circle( 15, 0, 0);

Square Delta = **new** Square( -5, (**float**)8.7, 0, 3, 3),

Iota = **new** Square( -7, (**float**)6.6, 0, 3, 3),

Eta = **new** Square(-15, -5, 0, 3, 3);

// Alpha Checks

*assertEquals*(Collision.*collision\_check*( Alpha, Alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( Alpha, Beta ), **false**);

*assertEquals*(Collision.*collision\_check*( Alpha, Gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( Alpha, Delta ), **false**);

*assertEquals*(Collision.*collision\_check*( Alpha, Iota ), **false**);

*assertEquals*(Collision.*collision\_check*( Alpha, Eta ), **false**);

// Beta Checks

*assertEquals*(Collision.*collision\_check*( Beta, Alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( Beta, Beta ), **false**);

*assertEquals*(Collision.*collision\_check*( Beta, Gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( Beta, Delta ), **false**);

*assertEquals*(Collision.*collision\_check*( Beta, Iota ), **false**);

*assertEquals*(Collision.*collision\_check*( Beta, Eta ), **false**);

// Gamma Checks

*assertEquals*(Collision.*collision\_check*( Gamma, Alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( Gamma, Beta ), **false**);

*assertEquals*(Collision.*collision\_check*( Gamma, Gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( Gamma, Delta ), **false**);

*assertEquals*(Collision.*collision\_check*( Gamma, Iota ), **false**);

*assertEquals*(Collision.*collision\_check*( Gamma, Eta ), **false**);

// Delta Checks

*assertEquals*(Collision.*collision\_check*( Delta, Alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( Delta, Beta ), **false**);

*assertEquals*(Collision.*collision\_check*( Delta, Gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( Delta, Delta ), **false**);

*assertEquals*(Collision.*collision\_check*( Delta, Iota ), **true**);

*assertEquals*(Collision.*collision\_check*( Delta, Eta ), **false**);

// Iota Checks

*assertEquals*(Collision.*collision\_check*( Iota, Alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( Iota, Beta ), **false**);

*assertEquals*(Collision.*collision\_check*( Iota, Gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( Iota, Delta ), **true**);

*assertEquals*(Collision.*collision\_check*( Iota, Iota ), **false**);

*assertEquals*(Collision.*collision\_check*( Iota, Eta ), **false**);

// Eta Checks

*assertEquals*(Collision.*collision\_check*( Eta, Alpha ), **false**);

*assertEquals*(Collision.*collision\_check*( Eta, Beta ), **false**);

*assertEquals*(Collision.*collision\_check*( Eta, Gamma ), **false**);

*assertEquals*(Collision.*collision\_check*( Eta, Delta ), **false**);

*assertEquals*(Collision.*collision\_check*( Eta, Iota ), **false**);

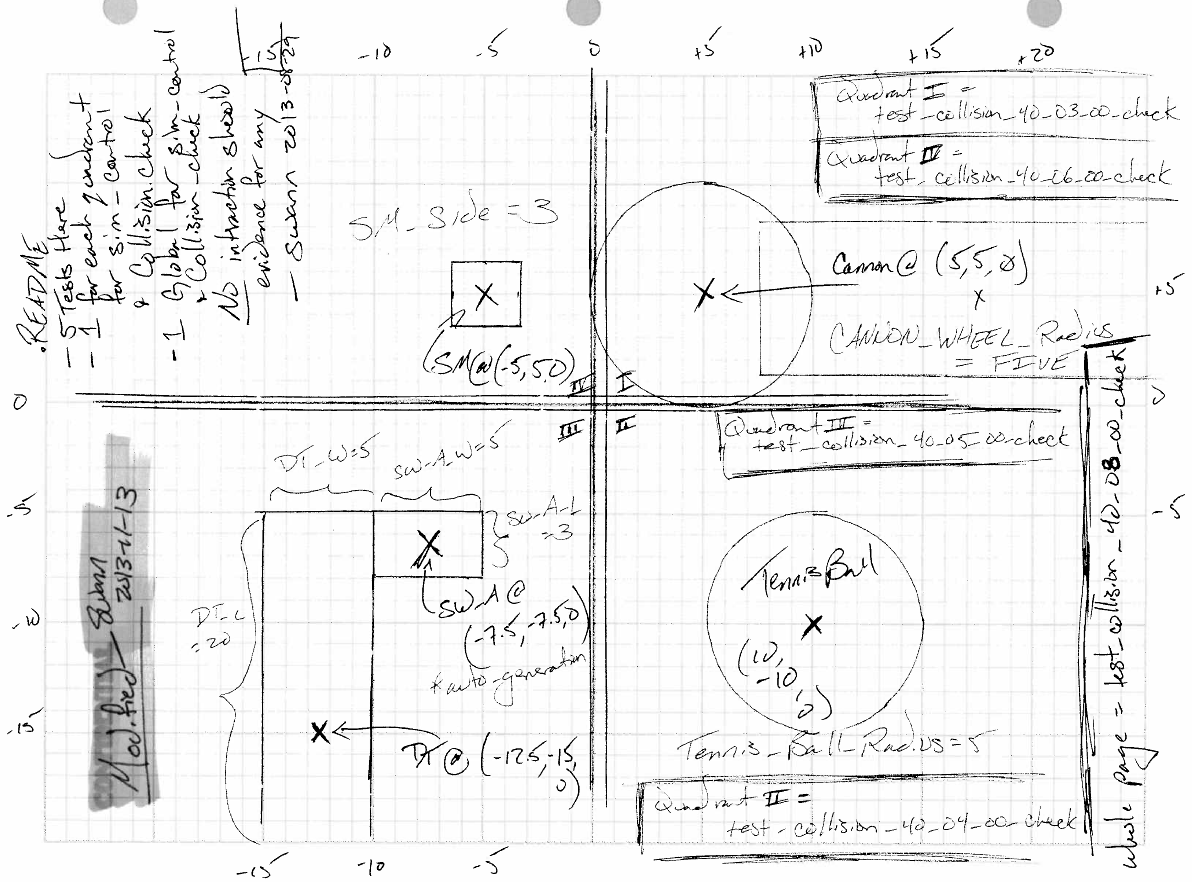
*assertEquals*(Collision.*collision\_check*( Eta, Eta ), **false**);

}

module :: Item/Collision.java

test/design :: test\_collision\_40\_(03-08)\_00\_collision\_check

description :: design of the specific check determining whether or not different shape objects collide; single sets



test notes :: explores functional validity of check algorithm against single sets of objects

// Collision.check()

@Test

**public** **void** test\_collision\_40\_03\_00\_check() {

ArrayList<Actor\_Object> test\_list = **new** ArrayList<Actor\_Object>();

// Quadrant One Test of related analog test design --> see notebook

// ~swann, 2013-08-29

test\_list.add( **new** Cannon(*FIVE*, *FIVE*, *ZERO*) );

*assertEquals*(test\_list.size(), *ONE*, *ZERO*);

collider.check(test\_list);

LinkedList<Actor\_Object> THE\_list = collider.get\_list();

*assertEquals*(*ZERO*, THE\_list.size(), *ZERO*);

}

// Collision.check()

@Test

**public** **void** test\_collision\_40\_04\_00\_check() {

ArrayList<Actor\_Object> test\_list = **new** ArrayList<Actor\_Object>();

// Quadrant Two Test of related analog test design --> see notebook

// ~swann, 2013-08-29

test\_list.add( Ball.*generate\_Tennis\_Ball*(*TEN*, *ZERO*-*TEN*, *ZERO*) );

*assertEquals*(test\_list.size(), *ONE*, *ZERO*);

collider.check(test\_list);

LinkedList<Actor\_Object> THE\_list = collider.get\_list();

*assertEquals*(*ZERO*, THE\_list.size(), *ZERO*);

}

// Collision.check()

@Test

**public** **void** test\_collision\_40\_05\_00\_check() {

ArrayList<Actor\_Object> test\_list = **new** ArrayList<Actor\_Object>();

// Quadrant Three Test of related analog test design --> see notebook

// ~swann, 2013-08-29

DropTower drop\_tower = **new** DropTower((**float**) (*ZERO*-12.5), *ZERO*-15, *ZERO*);

test\_list.add( drop\_tower );

test\_list.add( drop\_tower.swing\_arm\_generation() );

*assertEquals*(test\_list.size(), *TWO*, *ZERO*);

collider.check(test\_list);

LinkedList<Actor\_Object> THE\_list = collider.get\_list();

*assertEquals*(*ZERO*, THE\_list.size(), *ZERO*);

}

// Collision.check()

@Test

**public** **void** test\_collision\_40\_06\_00\_check() {

ArrayList<Actor\_Object> test\_list = **new** ArrayList<Actor\_Object>();

// Quadrant Four Test of related analog test design --> see notebook

// ~swann, 2013-08-29

test\_list.add( Standard\_Mass.*generate\_one\_g\_mass*(*ZERO*-*FIVE*, *FIVE*, *ZERO*) );

*assertEquals*(test\_list.size(), *ONE*, *ZERO*);

collider.check(test\_list);

LinkedList<Actor\_Object> THE\_list = collider.get\_list();

*assertEquals*(*ZERO*, THE\_list.size(), *ZERO*);

}

// Collision.check()

@Test

**public** **void** test\_collision\_40\_07\_00\_check() {

//ArrayList<Actor\_Object> test\_list = new ArrayList<Actor\_Object>();

// Undefined test number/test structure

// ~swann, 2013-08-29

}

// Collision.check()

@Test

**public** **void** test\_collision\_40\_08\_00\_check() {

ArrayList<Actor\_Object> test\_list = **new** ArrayList<Actor\_Object>();

// ALL Quadrant Test of related analog test design --> see notebook

// ~swann, 2013-08-29

DropTower drop\_tower = **new** DropTower((**float**) (*ZERO*-12.5), *ZERO*-15, *ZERO*);

test\_list.add( drop\_tower );

test\_list.add( drop\_tower.swing\_arm\_generation() );

test\_list.add( **new** Cannon(*FIVE*, *FIVE*, *ZERO*) );

test\_list.add( Standard\_Mass.*generate\_one\_g\_mass*(*ZERO*-*FIVE*, *FIVE*, *ZERO*) );

test\_list.add( Ball.*generate\_Tennis\_Ball*(*TEN*, *ZERO*-*TEN*, *ZERO*) );

*assertEquals*(test\_list.size(), *FIVE*, *ZERO*);

collider.check(test\_list);

LinkedList<Actor\_Object> THE\_list = collider.get\_list();

// console feed of linked list internals.

// save for now. **ANALOG**

// ~swann, 2013-11-13

//

// list has 3 elements, the mass, the ball and a NULL sentinel

// cannons, swing\_arms and drop\_towers are ALL NON-interactive at this time

**for** ( **int** i = *ZERO* ; i < THE\_list.size() ; i++ ) {

**if** ( THE\_list.get(i) == **null** ) {

System.*out*.println( "null" );

} **else** {

System.*out*.println( THE\_list.get(i).toString() );

}

}

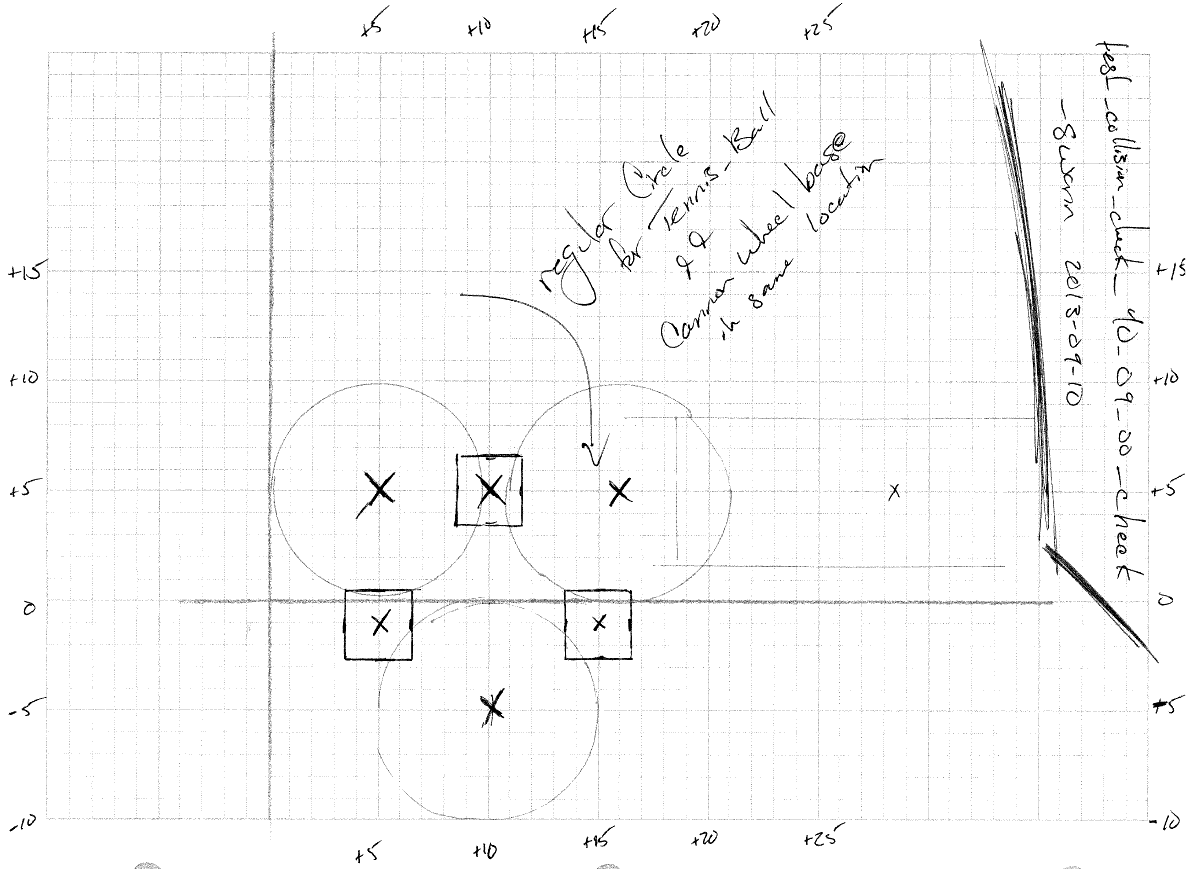
*assertEquals*(*THREE*, THE\_list.size(), *ZERO*);

} // test\_collision\_40\_08\_00\_check

module :: Item/Collision.java

test/design :: test\_collision\_40\_09\_00\_collision\_check

description :: design of the first heterogenous shape collision check; testing functional validity of enqueue/dequeue



test notes :: heterogenous check on shape collisions; also a functional validity test on enqueue and dequeue

// Collision.check()

@Test

**public** **void** test\_collision\_40\_09\_00\_check() {

ArrayList<Actor\_Object> test\_list = **new** ArrayList<Actor\_Object>();

// Related to analog test design --> see notebook

// ~swann, 2013-09-10

test\_list.add( Standard\_Mass.*generate\_one\_g\_mass*(*TEN*, *FIVE*, *ZERO*) );

test\_list.add( Standard\_Mass.*generate\_one\_g\_mass*(*FIVE*, *NEGATIVE\_ONE*, *ZERO*) );

test\_list.add( Standard\_Mass.*generate\_one\_g\_mass*(*TEN*+*FIVE*, *NEGATIVE\_ONE*, *ZERO*) );

test\_list.add( Ball.*generate\_Tennis\_Ball*(16, *FIVE*, *ZERO*) );

test\_list.add( Ball.*generate\_Tennis\_Ball*(*FIVE*, *FIVE*, *ZERO*) );

test\_list.add( Ball.*generate\_Tennis\_Ball*(*TEN*, *ZERO*-*FIVE*, *ZERO*) );

test\_list.add( **new** Cannon(16, *FIVE*, *ZERO*) );

*assertEquals*(test\_list.size(), *SEVEN*, *ZERO*);

collider.check(test\_list);

LinkedList<Actor\_Object> THE\_list = collider.get\_list();

// 7 Objects, 1 Non-Interactive, 1 Sentinel = 7 total in list

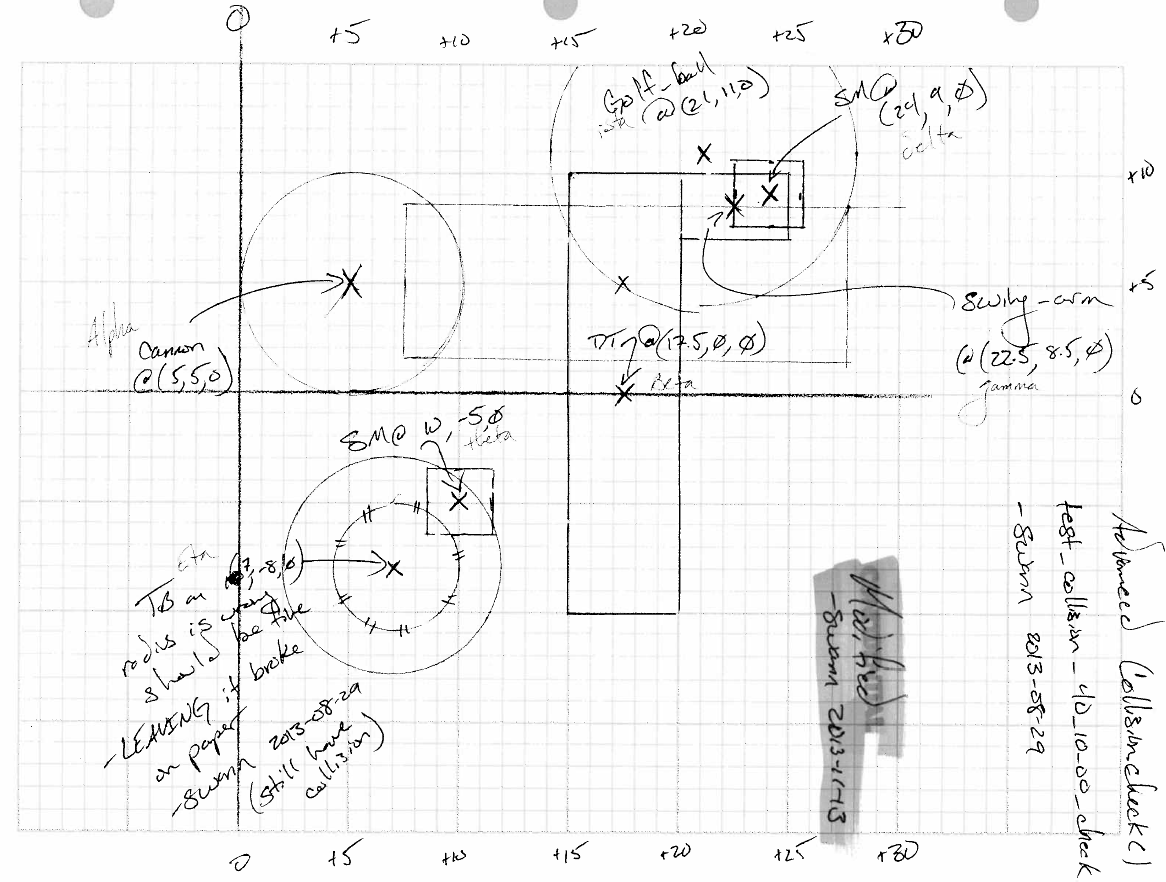
*assertEquals*(*SEVEN*, THE\_list.size(), *ZERO*);

}

module :: Item/Collision.java

test/design :: test\_collision\_40\_10\_00\_collision\_check

description :: design of the first heterogenous interactivity collision check; testing functional validity of enqueue/dequeue



test notes :: heterogenous check on shape collisions; also a functional validity test on interactivity logic within enqueue and

dequeue

// Collision.check()

@Test

**public** **void** test\_collision\_40\_10\_00\_check() {

ArrayList<Actor\_Object> test\_list = **new** ArrayList<Actor\_Object>();

// Related to analog test design --> see notebook

// ~swann, 2013-08-29

DropTower beta = **new** DropTower((**float**) (17.5), *ZERO*, *ZERO*);

SwingArm gamma = beta.swing\_arm\_generation();

Standard\_Mass delta = Standard\_Mass.*generate\_one\_g\_mass*(24, *NINE*, *ZERO*);

Standard\_Mass theta = Standard\_Mass.*generate\_one\_g\_mass*(*TEN*, *ZERO*-*FIVE*, *ZERO*);

Cannon alpha = **new** Cannon(*FIVE*, *FIVE*, *ZERO*);

Ball eta = Ball.*generate\_Tennis\_Ball*(*SEVEN*, *ZERO*-*EIGHT*, *ZERO*);

Ball iota = Ball.*generate\_Golf\_Ball*(21, 11, *ZERO*);

// Battery One

test\_list.add( eta );

test\_list.add( theta );

*assertEquals*(test\_list.size(), *TWO*, *ZERO*);

collider.check(test\_list);

LinkedList<Actor\_Object> THE\_list = collider.get\_list();

// eta + beta + null = 3

*assertEquals*(*THREE*, THE\_list.size(), *ZERO*);

// Battery Two

test\_list = **new** ArrayList<Actor\_Object>();

test\_list.add( delta ); // Standard\_Mass

test\_list.add( iota ); // Ball

test\_list.add( eta ); // Ball

test\_list.add( theta ); // Standard\_Mass

*assertEquals*(test\_list.size(), *FOUR*, *ZERO*);

collider.check(test\_list);

THE\_list = collider.get\_list();

// ORIGINAL NOTATIONS

//

// eta + beta + null = 3

// delta + itoa + null = 3 ==> 6

//

// ---------------------------------------------

// console feed of linked list internals.

// save for now. **ANALOG**

// ~swann 2013-11-13

//

// list has 5 elements, after re-sizing, the collision sets

// converge into ONE larger set. the size drops one because of the lost

// of a flagged sentinel value

//

// eta + beta + delta + itoa + null = 5

//

**for** ( **int** i = *ZERO* ; i < THE\_list.size() ; i++ ) {

**if** ( THE\_list.get(i) == **null** ) {

System.*out*.println( "null" );

} **else** {

System.*out*.println( THE\_list.get(i).toString() );

}

}

*assertEquals*(*FIVE*, THE\_list.size(), *ZERO*);

// Battery Three - part A

//

// ASSUMES SWING ARM NEVER INTERACTIVE

// ASSUMES CANNON NEVER INTERACTIVE

test\_list = **new** ArrayList<Actor\_Object>();

test\_list.add( alpha ); // Cannon

test\_list.add( beta ); // DropTower

test\_list.add( gamma ); // SwingArm

test\_list.add( delta ); // Standard\_Mass

test\_list.add( iota ); // Ball

test\_list.add( eta ); // Ball

test\_list.add( theta ); // Standard\_Mass

*assertEquals*(test\_list.size(), *SEVEN*, *ZERO*);

collider.check(test\_list);

THE\_list = collider.get\_list();

collider.clean\_list();

// ORIGINAL NOTATIONS

//

// eta + beta + null = 3

// delta + itoa + gamma + null = 4 ==> 7

//

// ---------------------------------------------

// console feed of linked list internals.

// save for now. **ANALOG**

// ~swann 2013-11-03

//

// list has 5 elements, after re-sizing, the collision sets

// converge into ONE larger set. the size drops one because of the lost

// of a flagged sentinel value

//

System.*out*.println("----Battery Three A---------");

**for** ( **int** i = *ZERO* ; i < THE\_list.size() ; i++ ) {

**if** ( THE\_list.get(i) == **null** ) {

System.*out*.println( "null" );

} **else** {

System.*out*.println( THE\_list.get(i).toString() );

}

}

*assertEquals*(*FIVE*, THE\_list.size(), *ZERO*);

// Battery Three - part B

//

// ASSUMES SWING ARM CAN MAYBE BE INTERACTIVE

// ASSUMES CANNON NEVER INTERACTIVE

test\_list = **new** ArrayList<Actor\_Object>();

gamma.set\_interactive(**true**);

*assertEquals*(gamma.get\_interactive(), **true**);

test\_list.add( alpha ); // Cannon

test\_list.add( beta ); // DropTower

test\_list.add( gamma ); // SwingArm

test\_list.add( delta ); // Standard\_Mass

test\_list.add( iota ); // Ball

test\_list.add( eta ); // Ball

test\_list.add( theta ); // Standard\_Mass

*assertEquals*(test\_list.size(), *SEVEN*, *ZERO*);

collider.check(test\_list);

THE\_list = collider.get\_list();

collider.clean\_list();

// eta + beta + null = 3

// delta + itoa + null = 3 ==> 6

System.*out*.println("----Battery Three B---------");

**for** ( **int** i = *ZERO* ; i < THE\_list.size() ; i++ ) {

**if** ( THE\_list.get(i) == **null** ) {

System.*out*.println( "null" );

} **else** {

System.*out*.println( THE\_list.get(i).toString() );

}

}

*assertEquals*(*SIX*, THE\_list.size(), *ZERO*);

// Battery Four

//

//

// **ANALOG**

// Bug found here from analog test design.

// SwingArm.Swing() appearing to mis-behave.

// Isolated and vixed via <Event\_Interaction>.clean\_list() installation

// ~swann 2013-09-10

//

test\_list = **new** ArrayList<Actor\_Object>();

gamma.swing();

*assertEquals*(gamma.get\_interactive(), **false**);

test\_list.add( alpha ); // Cannon

test\_list.add( beta ); // DropTower

test\_list.add( gamma ); // SwingArm

test\_list.add( delta ); // Standard\_Mass

test\_list.add( iota ); // Ball

test\_list.add( eta ); // Ball

test\_list.add( theta ); // Standard\_Mass

*assertEquals*(test\_list.size(), *SEVEN*, *ZERO*);

collider.check(test\_list);

THE\_list = collider.get\_list();

collider.clean\_list();

// eta + beta + null = 3

// delta + itoa + null = 3 ==> 6

// --------------------------------------

// eta + beta + delta + itoa + null = 5

// Changes made to reflect the above on 2013-11-13

System.*out*.println("----Battery Four---------");

**for** ( **int** i = *ZERO* ; i < THE\_list.size() ; i++ ) {

**if** ( THE\_list.get(i) == **null** ) {

System.*out*.println( "null" );

} **else** {

System.*out*.println( THE\_list.get(i).toString() );

}

}

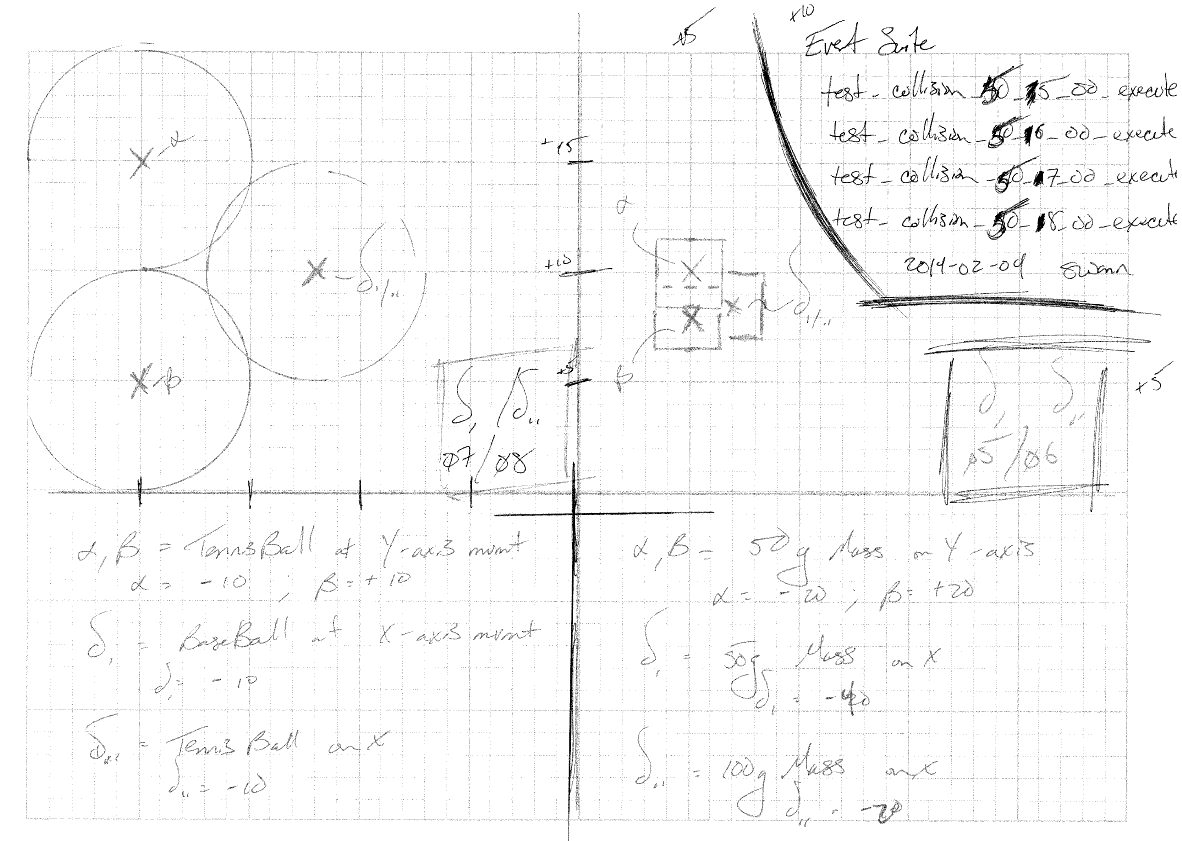
*assertEquals*(*FIVE*, THE\_list.size(), *ZERO*);

} // test\_collision\_40\_10\_00\_check

module :: Item/Collision.java

test/design :: test\_collision\_50\_(05-08)\_00\_ execute

description :: design of the first heterogenous shape collision execution with calculated velocity vectors



test notes :: heterogenous check on shape collisions related to the mathematics of calculating velocity vectors; new vector

values pushed to screen output in this version of the test for manual floating point analysis

// Collision.execute()

@Test

**public** **void** test\_collision\_50\_05\_00\_execute() **throws** CloneNotSupportedException{

// The Setup

ArrayList<Actor\_Object> test\_list = **new** ArrayList<Actor\_Object>();

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(*FIVE*, *TEN*, *ZERO*);

alpha.update\_velocity(*ZERO*, -20, *ZERO*);

Standard\_Mass beta = Standard\_Mass.*generate\_fifty\_g\_mass*(*FIVE*, *EIGHT*, *ZERO*);

beta.update\_velocity(*ZERO*, 20, *ZERO*);

Standard\_Mass delta = Standard\_Mass.*generate\_fifty\_g\_mass*(*SEVEN*, *NINE*, *ZERO*);

delta.update\_velocity(-40, *ZERO*, *ZERO*);

test\_list.add(alpha);

test\_list.add( beta);

test\_list.add(delta);

*assertEquals*(test\_list.size(), *THREE*, *ZERO*);

collider.clean\_list();

// Battery

collider.check(test\_list);

LinkedList<Actor\_Object> THE\_list = collider.get\_list();

*assertEquals*(*FOUR*, THE\_list.size(), *ZERO*);

*assertEquals*(alpha, THE\_list.get(*ZERO*));

*assertEquals*(beta, THE\_list.get(*ONE*));

*assertEquals*(delta, THE\_list.get(*TWO*));

*assertEquals*(**null**, THE\_list.get(*THREE*));

// execute

collider.execute();

} // test\_collision\_50\_05\_00\_execute

// Collision.execute()

@Test

**public** **void** test\_collision\_50\_06\_00\_execute() **throws** CloneNotSupportedException {

// The Setup

ArrayList<Actor\_Object> test\_list = **new** ArrayList<Actor\_Object>();

Standard\_Mass alpha = Standard\_Mass.*generate\_fifty\_g\_mass*(*FIVE*, *TEN*, *ZERO*);

alpha.update\_velocity(*ZERO*, -20, *ZERO*);

Standard\_Mass beta = Standard\_Mass.*generate\_fifty\_g\_mass*(*FIVE*, *EIGHT*, *ZERO*);

beta.update\_velocity(*ZERO*, 20, *ZERO*);

Standard\_Mass delta = Standard\_Mass.*generate\_one\_hundred\_g\_mass*(*SEVEN*, *NINE*, *ZERO*);

delta.update\_velocity(-20, *ZERO*, *ZERO*);

test\_list.add(alpha);

test\_list.add( beta);

test\_list.add(delta);

*assertEquals*(test\_list.size(), *THREE*, *ZERO*);

collider.clean\_list();

// Battery

collider.check(test\_list);

LinkedList<Actor\_Object> THE\_list = collider.get\_list();

*assertEquals*(*FOUR*, THE\_list.size(), *ZERO*);

*assertEquals*(alpha, THE\_list.get(*ZERO*));

*assertEquals*(beta, THE\_list.get(*ONE*));

*assertEquals*(delta, THE\_list.get(*TWO*));

*assertEquals*(**null**, THE\_list.get(*THREE*));

// execute

collider.execute();

} //test\_collision\_50\_06\_00\_execute

// Collision.execute()

@Test

**public** **void** test\_collision\_50\_07\_00\_execute() **throws** CloneNotSupportedException{

// The Setup

ArrayList<Actor\_Object> test\_list = **new** ArrayList<Actor\_Object>();

Ball alpha = Ball.*generate\_Tennis\_Ball*(-200, 50, *ZERO*);

alpha.update\_velocity(*ZERO*, -10, *ZERO*);

Ball beta = Ball.*generate\_Tennis\_Ball*(-200, 149, *ZERO*);

beta.update\_velocity(*ZERO*, 10, *ZERO*);

Ball delta = Ball.*generate\_Baseball*(-140, (**float**) 99.5, *ZERO*);

delta.update\_velocity(-140, (**float**) 99.5, *ZERO*);

test\_list.add(alpha);

test\_list.add( beta);

test\_list.add(delta);

*assertEquals*(test\_list.size(), *THREE*, *ZERO*);

collider.clean\_list();

// Battery

collider.check(test\_list);

LinkedList<Actor\_Object> THE\_list = collider.get\_list();

*assertEquals*(*FOUR*, THE\_list.size(), *ZERO*);

*assertEquals*(alpha, THE\_list.get(*ZERO*));

*assertEquals*(beta, THE\_list.get(*ONE*));

*assertEquals*(delta, THE\_list.get(*TWO*));

*assertEquals*(**null**, THE\_list.get(*THREE*));

// execute

collider.execute();

} // test\_collision\_50\_07\_00\_execute

// Collision.execute()

@Test

**public** **void** test\_collision\_50\_08\_00\_execute() **throws** CloneNotSupportedException {

// The Setup

ArrayList<Actor\_Object> test\_list = **new** ArrayList<Actor\_Object>();

Ball alpha = Ball.*generate\_Tennis\_Ball*(-200, 149, *ZERO*);

alpha.update\_velocity(*ZERO*, -10, *ZERO*);

Ball beta = Ball.*generate\_Tennis\_Ball*(-200, 50, *ZERO*);

beta.update\_velocity(*ZERO*, 10, *ZERO*);

Ball delta = Ball.*generate\_Tennis\_Ball*(-140, (**float**) 99.5, *ZERO*);

delta.update\_velocity(-10, *ZERO*, *ZERO*);

test\_list.add(alpha);

test\_list.add( beta);

test\_list.add(delta);

*assertEquals*(test\_list.size(), *THREE*, *ZERO*);

// Battery

collider.check(test\_list);

LinkedList<Actor\_Object> THE\_list = collider.get\_list();

*assertEquals*(*FOUR*, THE\_list.size(), *ZERO*);

*assertEquals*(alpha, THE\_list.get(*ZERO*));

*assertEquals*(beta, THE\_list.get(*ONE*));

*assertEquals*(delta, THE\_list.get(*TWO*));

*assertEquals*(**null**, THE\_list.get(*THREE*));

// execute

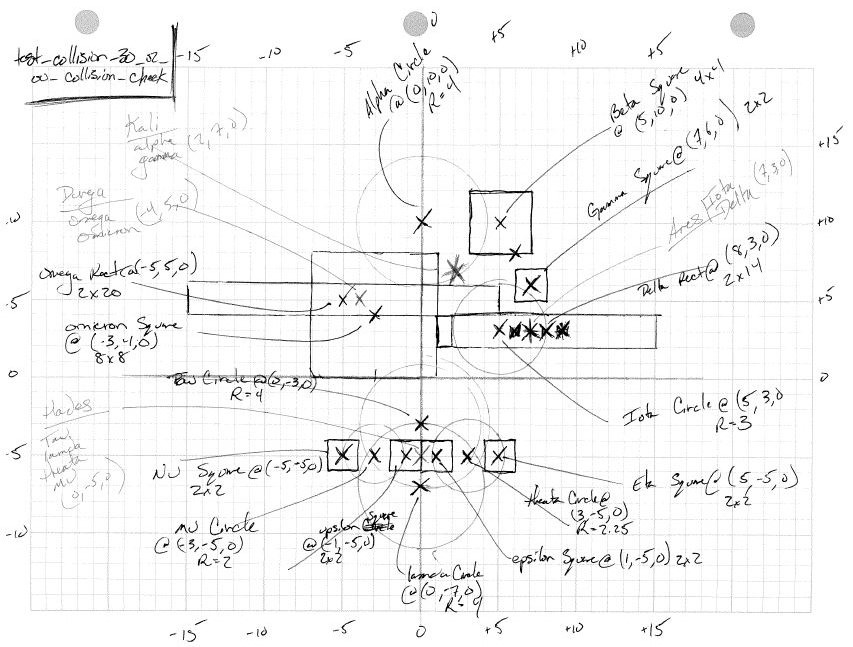
collider.execute();

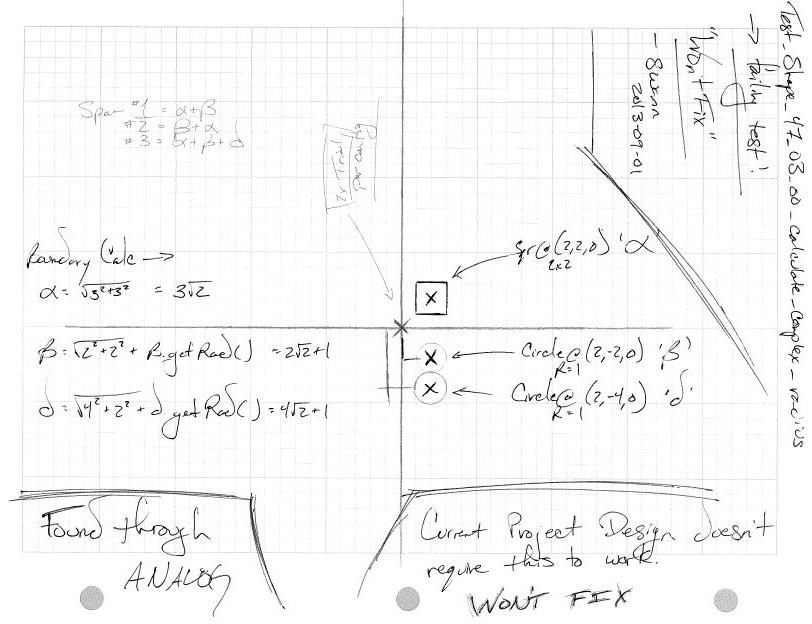
} // test\_collision\_50\_08\_00\_execute

module :: Item/Collision.java

test/design :: test\_collision\_30\_02\_00\_collision\_check && test\_shape\_47\_03\_00\_calcuate\_complex\_radius

description :: design of the logic stress test for logic exercise; notation of won’t fix designations outside of project scope





test notes :: logic stress test of several constructs outside of project scope to preform self-reconnassiance; won’t fix

designation associated with boundary sphere calcuations outside scope

// Collision.collision\_check()

// --> Heterogeneous Test

@Test

**public** **void** test\_collision\_30\_02\_00\_collision\_check() {

// The Setup

// Simple Shapes

Circle alpha = **new** Circle(*ZERO*, *TEN*, *ZERO*, *FOUR*);

Circle iota = **new** Circle(*FIVE*, *THREE*, *ZERO*, *THREE*);

Circle theta = **new** Circle(*THREE*, -5, *ZERO*, (**float**) 2.25);

Circle lamda = **new** Circle(*ZERO*, -7, *ZERO*, *FOUR*);

Circle mu = **new** Circle(-3, -5, *ZERO*, *TWO*);

Circle tau = **new** Circle(*ZERO*, -3, *ZERO*, *FOUR*);

//Square beta = new Square(FIVE, TEN, ZERO, FOUR, FOUR);

Square gamma = **new** Square(*SEVEN*, *SIX*, *ZERO*, *TWO*, *TWO*);

//Square eta = new Square(FIVE, -5, ZERO, TWO, TWO);

//Square epsilon = new Square(ONE, -5, ZERO, TWO, TWO);

//Square upsilon = new Square(-1, -5, ZERO, TWO, TWO);

//Square nu = new Square(-5, -5, ZERO, TWO, TWO);

Square omicron = **new** Square(-3, -4, *ZERO*, *EIGHT*, *EIGHT*);

Rectangle delta = **new** Rectangle(*EIGHT*, *THREE*, *ZERO*, *TWO*, 14);

Rectangle omega = **new** Rectangle(-5, -5, *ZERO*, *TWO*, 20);

// Complex Shapes

Shape hades = **new** Shape(*ZERO*, -5, *ZERO*);

hades.add\_shape( mu );

hades.add\_shape( theta );

hades.add\_shape( tau );

hades.add\_shape( lamda );

hades.calculate\_radius();

Shape ares = **new** Shape(*SEVEN*, *THREE*, *ZERO*);

ares.add\_shape( iota );

ares.add\_shape( delta );

ares.calculate\_radius();

Shape durga = **new** Shape(-4, *FIVE*, *ZERO*);

durga.add\_shape( omega );

durga.add\_shape( omicron );

durga.calculate\_radius();

Shape kali = **new** Shape(*ZERO*, *TEN*, *ZERO*);

kali.add\_shape( alpha );

kali.add\_shape( gamma );

kali.calculate\_radius();

// **ANALOG**: TEST DESIGN

// -----------------------

// Calc radius may need a re-working... maybe

// keep this note if so... Analog test design evidencing bugs before

// digital testing even in place.

// ~swann 2013-08-29

// -----------------------

// projected error --> boundary sphere will not calc full boundary sphere

// -----------------------

// error confirmed ~ swann, 2013-08-29

// WON'T FIX --> unless I have to in the future

// commented tests are those that won't pass -- not needed for reqs

// Kali vs. Complex Shapes

*assertEquals*(Collision.*collision\_check*( kali, kali ), **false**);

//assertEquals(Collision.collision\_check( kali, durga ), true);

*assertEquals*(Collision.*collision\_check*( kali, hades ), **false**);

*assertEquals*(Collision.*collision\_check*( kali, ares ), **true**);

// Durga vs. Complex Shapes

*assertEquals*(Collision.*collision\_check*( durga, durga ), **false**);

*assertEquals*(Collision.*collision\_check*( durga, hades ), **true**);

//assertEquals(Collision.collision\_check( durga, kali ), true);

//assertEquals(Collision.collision\_check( durga, ares ), true);

// Hades vs. Complex Shapes

*assertEquals*(Collision.*collision\_check*( hades, hades ), **false**);

*assertEquals*(Collision.*collision\_check*( hades, kali ), **false**);

*assertEquals*(Collision.*collision\_check*( hades, ares ), **false**);

*assertEquals*(Collision.*collision\_check*( hades, durga ), **true**);

// Ares vs. Complex Shapes

*assertEquals*(Collision.*collision\_check*( ares, ares ), **false**);

*assertEquals*(Collision.*collision\_check*( ares, hades ), **false**);

*assertEquals*(Collision.*collision\_check*( ares, kali ), **true**);

//assertEquals(Collision.collision\_check( ares, durga ), true);

}

// Shape.calculate\_complex\_radius()

//

@Test

**public** **void** test\_shape\_47\_03\_00\_calculate\_complex\_radius() {

// Battery One

// --> Setup

Shape outer\_one = **new** Shape(*ZERO*, *ZERO*, *ZERO*);

Shape outer\_two = **new** Shape(*ZERO*, *ZERO*, *ZERO*);

Square inner\_one = **new** Square(*TWO*, *TWO*, *TWO*, *TWO*, *TWO*);

Circle inner\_two = **new** Circle(*TWO*, *NEGATIVE\_TWO*, *ZERO*, *ONE*);

Circle inner\_tre = **new** Circle(*TWO*, *ZERO*-*FOUR*, *ZERO*, *ONE*);

// Shape One

*assertEquals*(outer\_one.get\_radius(), *ZERO*, *ZERO*);

*assertEquals*(outer\_one.is\_composite(), **false**);

outer\_one.add\_shape( inner\_two );

outer\_one.calculate\_radius();

*assertEquals*(outer\_one.get\_radius(), *TWO*\*Math.*pow*(*TWO*, *A\_HALF*)+1, *A\_THOUSANDTH*);

outer\_one.add\_shape( inner\_one );

outer\_one.calculate\_radius();

*assertEquals*(outer\_one.get\_radius(), *THREE*\*Math.*pow*(*TWO*, *A\_HALF*), *A\_THOUSANDTH*);

outer\_one.add\_shape( inner\_tre );

outer\_one.calculate\_radius();

//assertEquals(outer\_one.get\_radius(), THREE\*Math.pow(TWO, A\_HALF), A\_THOUSANDTH);

// TEST END HERE --> KNOWN BUG ON RADIUS CALCULATION WHEN CIRCLES ADDED AFTER AND

// OUTSIDE

// WON'T FIX FOR NOW. NO INTERNAL ITEMS REQUIRE THIS FIX

// ~ swann, 2013-09-01

//

// FOUND THROUGH **ANALOG**:

}