Unit 5

Increment 2 Design and Implementation

Objectives for this Unit

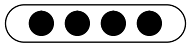
At the end of this unit you will have:

- Participated in the design of Increment 2 of Brickles, applying design patterns and newer features of C++.
- Completed coding and testing of Increment 2 of Brickles.
- Discussed and resolved some design and coding issues that were raised during the design and implementation.

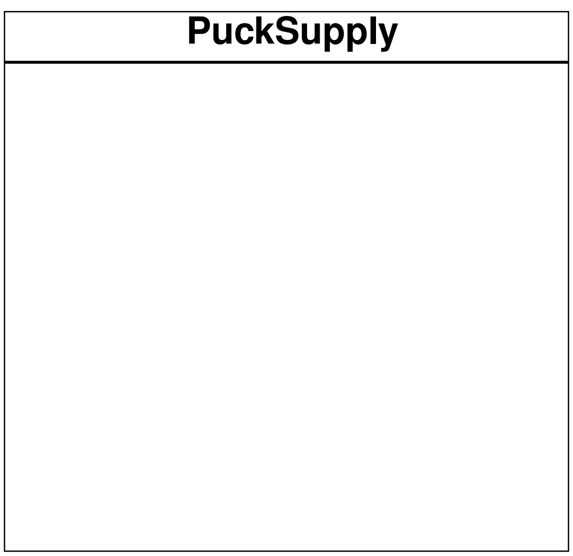
Brickles: Increment 2

- What is left?
  - Incorporate a puck supply.
  - Incorporate a brick pile.
  - Recognize and process collisions.
  - Putting a new puck in play when one hits the floor.
  - End the game (win/lose).
- In this unit, we will address design and implementation issues associated with completing Increment 2.

PuckSupply Specification

- The puck supply  is intended to hold the pucks that are not yet in play. Draw a dynamic model for a puck supply?

- Should there be a way to add a puck to a supply?
- Design the protocol for class PuckSupply?



*A **puck supply** is a collection of pucks.*

Pay special attention to the constructors and destructors.

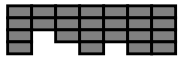
PuckSupply Implementation

- Sketch out an implementation for class PuckSupply. [Do not code it in the Visual C++ environment.]

```
// PkSpply.h
#ifndef PUCKSUPPLY_H
#define PUCKSUPPLY_H


class PuckSupply {
  public:









  private:







};
#endif
```

```
// PkSpply.cpp
#include "PkSpply.h"
```

BrickPile Specification

- The brick pile  is intended to hold the bricks that are not yet destroyed. Draw a dynamic model for a brick pile?

- Should there be a way to add a brick to a brick pile?
- Design the protocol for class BrickPile?

*A **brick pile** is a collection of bricks.*

BrickPile

Again, pay special attention to the constructors and destructors.

Should we use a template here?

Are there constraints on our design of this class?

BrickPile Implementation

- Sketch out an implementation for class BrickPile. [Do not code it in the Visual C++ environment.]

```
// BrckPile.h
#ifndef BRICKPILE_H
#define BRICKPILE_H


class BrickPile{
  public:







  private:






};
#endif
```

```
// BrckPile.cpp
#include "BrckPile.h"
```

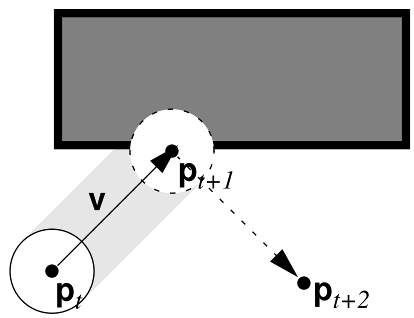
- Should we try to exploit commonality (if any) between this class and PuckSupply?

Collisions

- How can we handle collisions?
  - Who recognizes them?
  - Who handles them?
- In Increment 1, a puck checks its new position on a move against the playing field boundaries and adjusts its direction as appropriate. What will happen now that there are different kinds of arcade game pieces a puck can collide with?
  - a puck hits a brick, breaks it, and rebounds
  - a puck hits the paddle and rebounds at an angle determined by the collision point
  - a puck hits the floor, disappears, and another replaces it
  - a puck hits a wall or the ceiling and rebounds

Analysis of a Collision

- Consider a puck hitting a brick at time $t+1$:

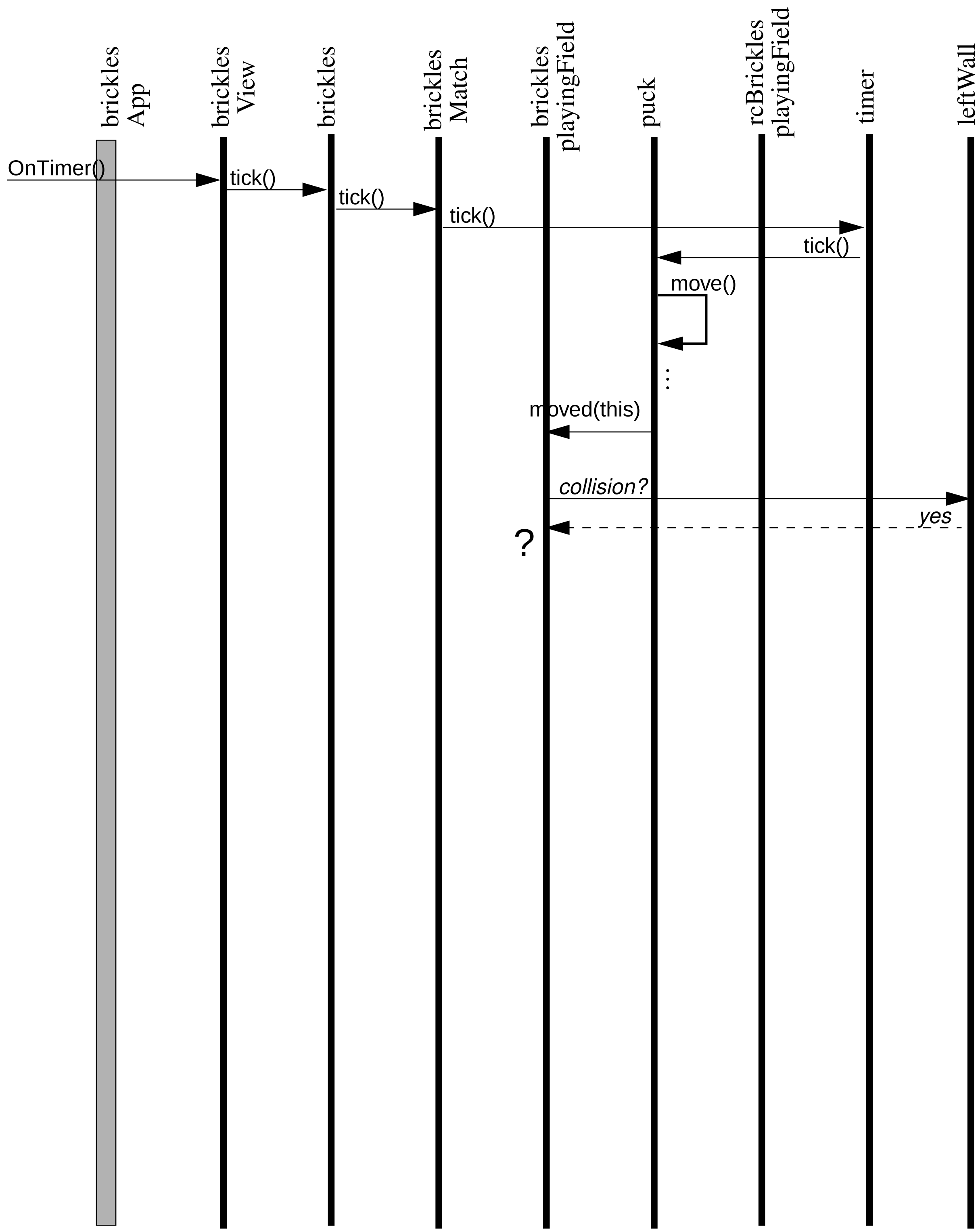


A collision occurs when two objects attempt to occupy the same space at some time t.

- What objects are involved in a collision?

- Are there different kinds of collisions, depending on the objects involved?

Context for a Collision



Observations

- Since the Brickles playing field is already being notified that the puck has changed its position—`moved()`, and a playing field knows the arcade game pieces that occupy it, then it seems reasonable for the playing field to detect a collision.
- So then what happens? Some ideas:

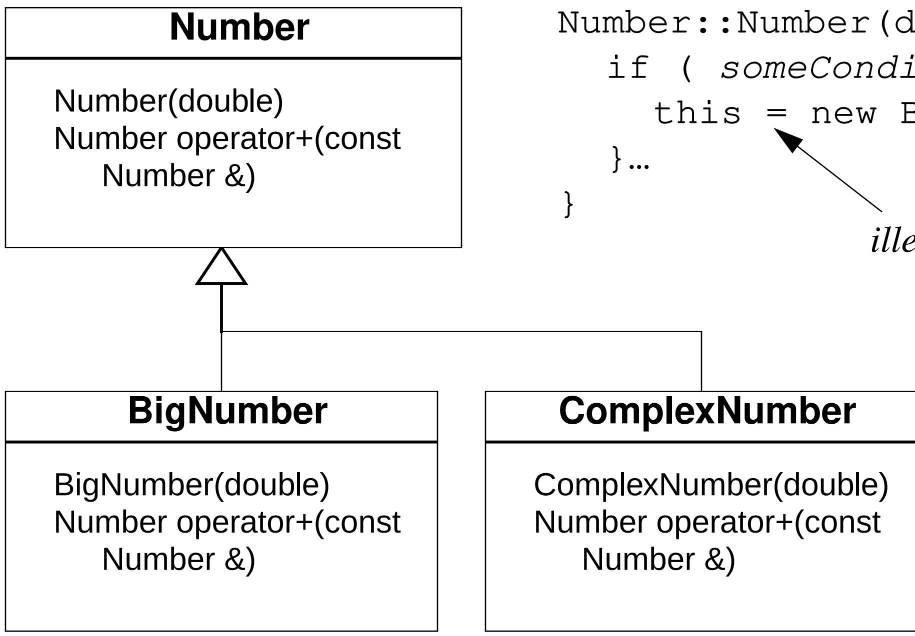
What are the implications of each of these?

Potential Implementation Techniques

- Virtual Constructor Idiom (Envelope and Letter Classes) [Coplien, Section 5.5]
- Defensive programming/Contract programming
- Exceptions

Virtual Constructors

- Simulating the notion of "virtual constructors"



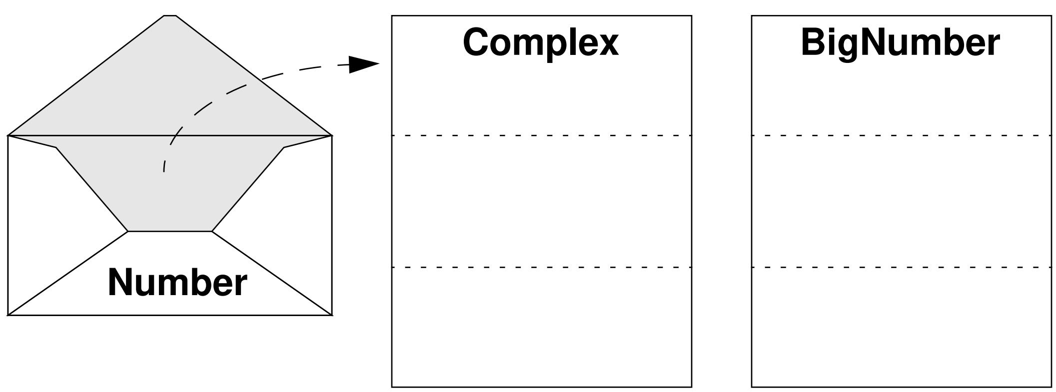

```
Number::Number(double d) {
  if ( someCondition(d) ) {
    this = new BigNumber(d);
  }...
}
```

illegal

Letters and Envelopes

An envelope delegates some of its functionality to a collaborating letter object

The letter can be though of as being in the envelope's employ.



Implementation

```
// An instance of an envelope class contains a
pointer to an instance of a letter class and the
member functions in the envelope forward requests
to the letter:

class Number {
  public:
    virtual Number operator+(const Number &);
    …
  private:
    Number *rep;
};

Number::Number(double d) {
  if ( someCondition(d) ) {
    rep = new BigNumber(d);
  } …
};

Number operator+(const Number &n) {
  return rep->operator+(n);
};
```

- “When to use this idiom: The virtual constructor idiom is used when the type of an object needs to be determined from the context in which the object is constructed. …”
- Does it apply to collisions?

Defensive Programming/ Contract Programming

- Defensive programming:
  - Member functions check all requests—for example,

    ```
    BOOL PlayingField::moved(MovablePiece &)
    ```

    returns a condition code that indicates the success of an operation. If a requested operation is nonsensical, it is rejected

- Contract programming:
  - No nonsensical operation is ever applied—for example, before a movable piece tells a playing field that it has moved, the piece verifies that the move it is making is valid:

    ```
    class PlayingField {
      public:
        void PlayingField::moved(MovablePiece &m);
          // precondition: (this->isWithin(m.boundingRect()))
        BOOL isWithin(const CRect &) const;
        ...
    }
    ```

Exceptions

- Most people think of an exception in terms of an error that occurs during processing—for example, arithmetic overflow.
- Consider that an exception may be an *exceptional occurrence* rather than an error—for example, a queue overflow or a file system full.

Exception Handling

- Exception handling in C++ provides a standard, language-level facility for responding to execution time errors.
- The exception mechanism provides an alternative path for exiting a method. The path taken for most computation can remain simple, easy to understand, and efficient. The path taken in the event of the exceptional occurence is separated and used only when necessary.
- In C++, the raising of an exception is called *throwing* an exception and is done with the throw command.
- An exception handler is written to *catch* each thrown exception.
- Throws and catches are grouped within *try* blocks which define the scope of the error handling.

C++ Exception Handling

- The “try” keyword, followed by a number of statements enclosed within braces defines the scope of throws and catches.

```
      try {
          ...
          ...doSomething()
          ...
      }
      catch (const char * string) { ... }
      catch (InitErr &ie) { ... }
      catch (ArrayErr &ae) { ... }
//    catch (...) { handle any other exceptions }
```

- In this example, three handlers for exceptions that may be raised in the try block are defined. Each catch statement defines a handler that processes an exception. [The commented-out handler catches any other exception.
- Any exception raised in the try block is compared to the catch statements in the order written. If a “match” occurs, the associated handler code is executed. If there is no match, the call stack is unwound until another try block is encountered. If no matching handler is found, the terminate() function is called.

Exception Handling

- Exceptions are thrown in the following way:

```
template <class Type>
Type &Array<Type>::operator[](int i)
{
  if ( (i < _low) || (_high < i) ) {
    throw IndexViolation(this, i);
  };
  return _store[i - _low];
}
```

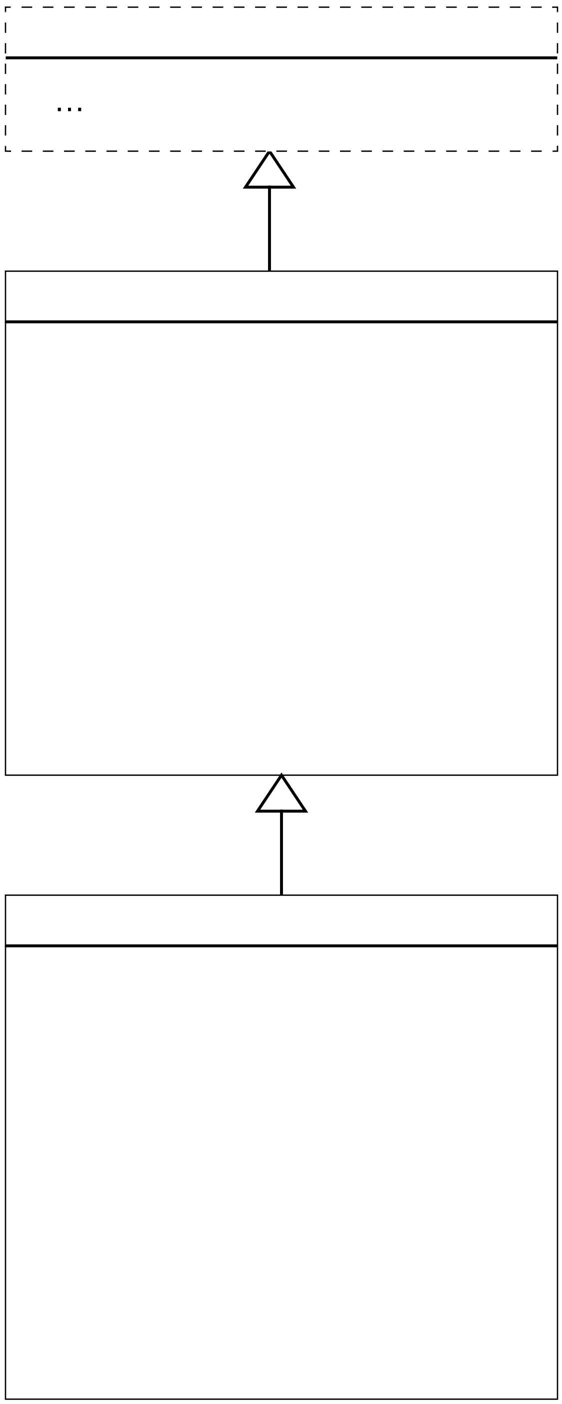
- The set of catch handlers resembles a set of overloaded functions where the appropriate handler is determined by matching the type of the exception thrown with the parameter defined by the catch statements.
- On completion of the catch handler, if the execution of the program is allowed to continue, it does so at the point following the try block of the catch handler, **not** at the point following where the throw expression occurred.
- As the call stack unwinds, destructors are called for objects created in the scope being exited.
- Are these useful for recognizing or processing collisions?

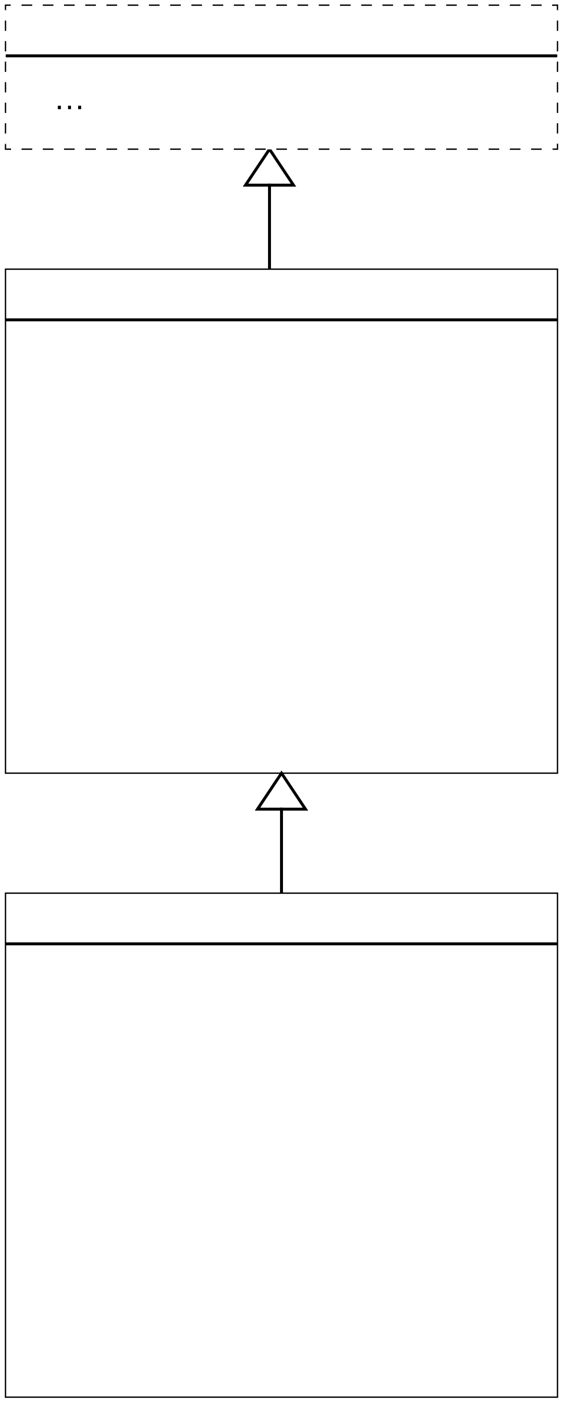
Visual C++ Exception Handling

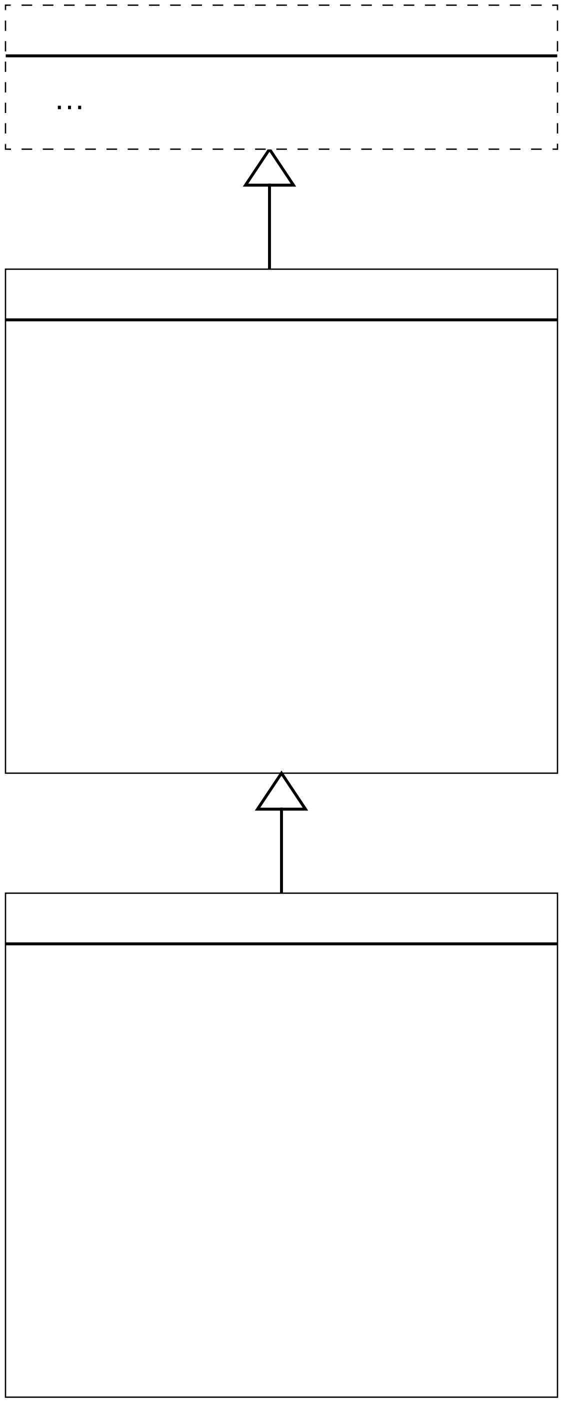
- Visual C++ version 1.51 does not support exceptions as part of the language syntax.
- The Microsoft Foundation Classes do provide an exception macro facility that we will use.
- The syntax is the same as the standard syntax but the names are in all capital letters.
- Also, each exception class must be derived from CException.

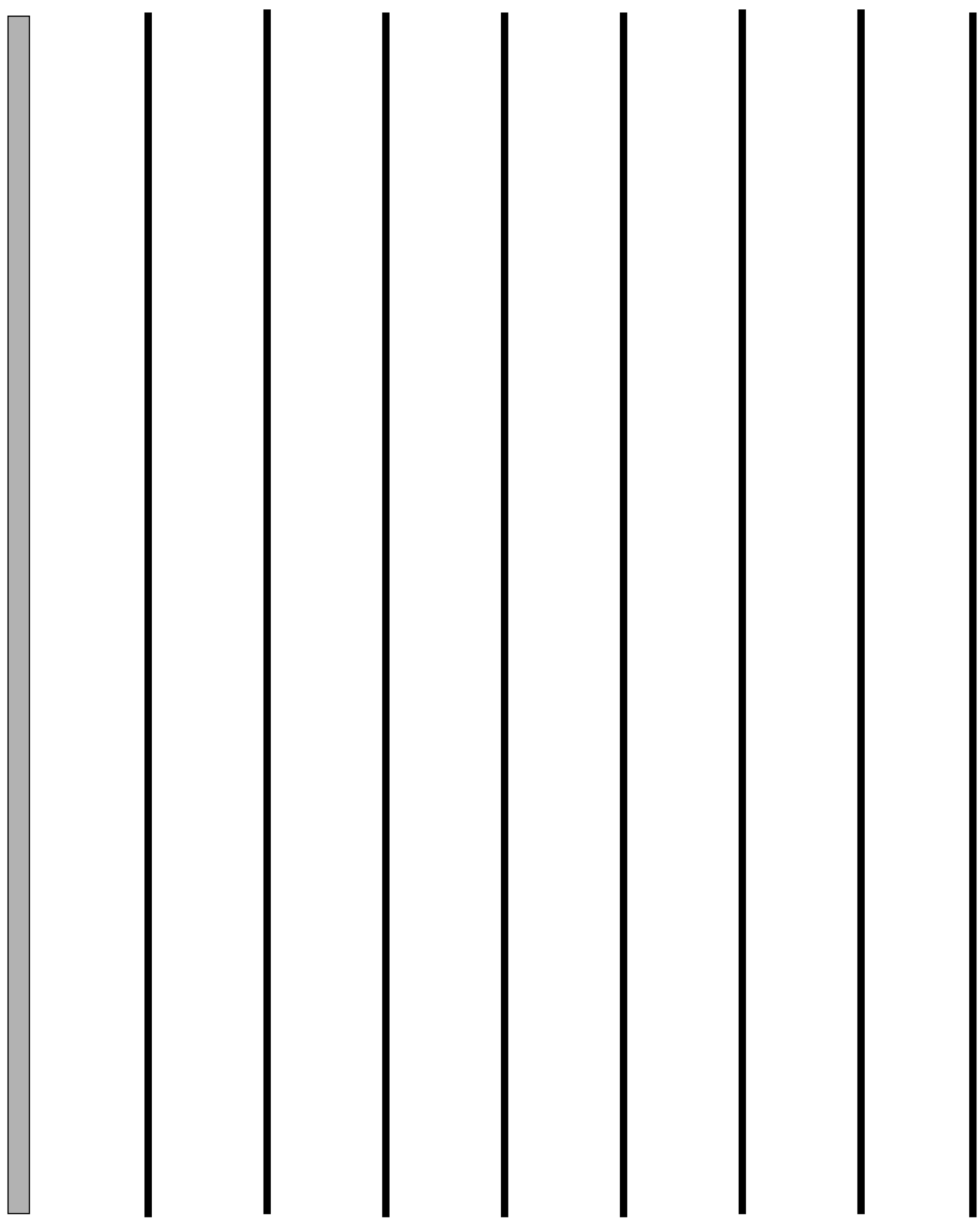
Back to Collisions

- Use the following pages to sketch out a design for handling collisions in Increment 2.
- Consider that future enhancements to Brickles might introduce new kinds of pieces, or perhaps variations on existing pieces. We would like to make those enhancements quickly and easily.



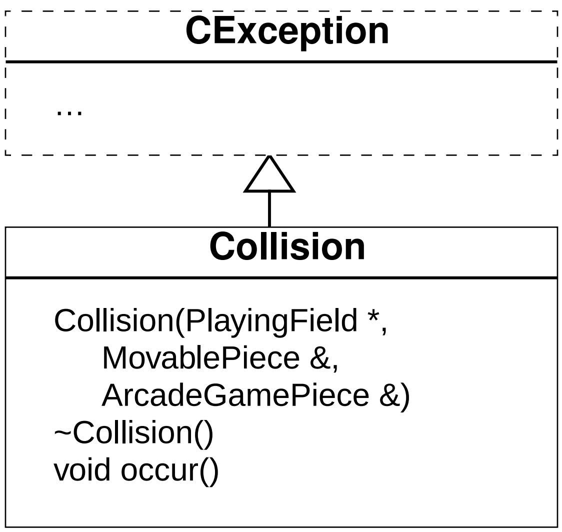






A Design

- Consider the following two classes of objects:
  - Trajectory—the path followed by a movable piece in a playing field during one unit interval of time
  - Collision—an act of a movable piece and an arcade game piece coming together—that is, occupying the same space in a playing field
- If the playing field detects a collision[1] when a movable piece notifies it of a move, then the move is rejected by throwing a Collision exception, which can then be processed and result in some change in the state of the pieces involved in the collision.



Design note: We have implemented this design in Visual C++ as a class CollisionException, which is a subclass of CException, and an instance of class Collision is contained within a CollisionException instance.

1. A playing field distributes responsibilities to the pieces in it to detect collisions to see if they are along trajectories.

Collision::occur()

- A collision “occurs” when an `occur()` (by a puck in our system). Thus, we have an exception object which has responsibilities in our system!
- How does a collision occur—that is, how does it create the effect of a collision in its component pieces? Recall that each pair of objects produces a different effect.

  We identified two approaches:

  - Use the virtual constructor idiom (letters and envelopes) to model the various kinds of collisions: Puck/Paddle, Puck/Brick, Puck/Floor, etc. These would comprise the letter classes, all subclasses of Collision.
  - Apply the Visitor pattern—in effect, the puck is visiting a brick, for example, so between the two of them they can work out what the effect of a collision is.
- What are some of the trade-offs?

Revised ArcadeGamePiece Hierarchy

ArcadeGamePiece

ArcadeGamePiece(PlayingField*,
    CPoint &)
ArcadeGamePiece(
    ArcadeGamePiece &)
~ArcadeGamePiece()
CPoint getPosition()

MovablePiece

MovablePiece(PlayingField*,
    CPoint &)
MovablePiece(MovablePiece &)
~MovablePiece()
void stop()
void resume()
void resume(Velocity &)
BOOL isInMotion() const
Velocity *getVelocity()
void setVelocity(Velocity *)
void changeSpeed(Speed)
void changeDirection(Direction)
Speed getSpeed();
Direction getDirection()
void move()
void tick();
\
void collideWith(
    ArcadeGamePiece &, CPoint
    &) = 0
void collideWithPuck(Puck &,
    CPoint &) = 0
void collideWithPaddle(Paddle &,
    CPoint &) = 0

StationaryPiece

StationaryPiece(PlayingField*,
    CPoint &)
StationaryPiece(StationaryPiece &)
~StationaryPiece()

void collideWithPuck(Puck &,
    CPoint &) = 0

The use of visitor implies that we set up a protocol that ensures pair-wise colisios can be handled. Every movable piece can collide with any arcade game piece, any movable piece can collide (explicitly) with a paddle or a puck, and any arcade game piece can collide (explicitly) with a puck. Note that a collision of a paddle and a brick is precluded, but a collision of a puck with a puck is a possibility—with this design.

Visitor (Double Dispatching)

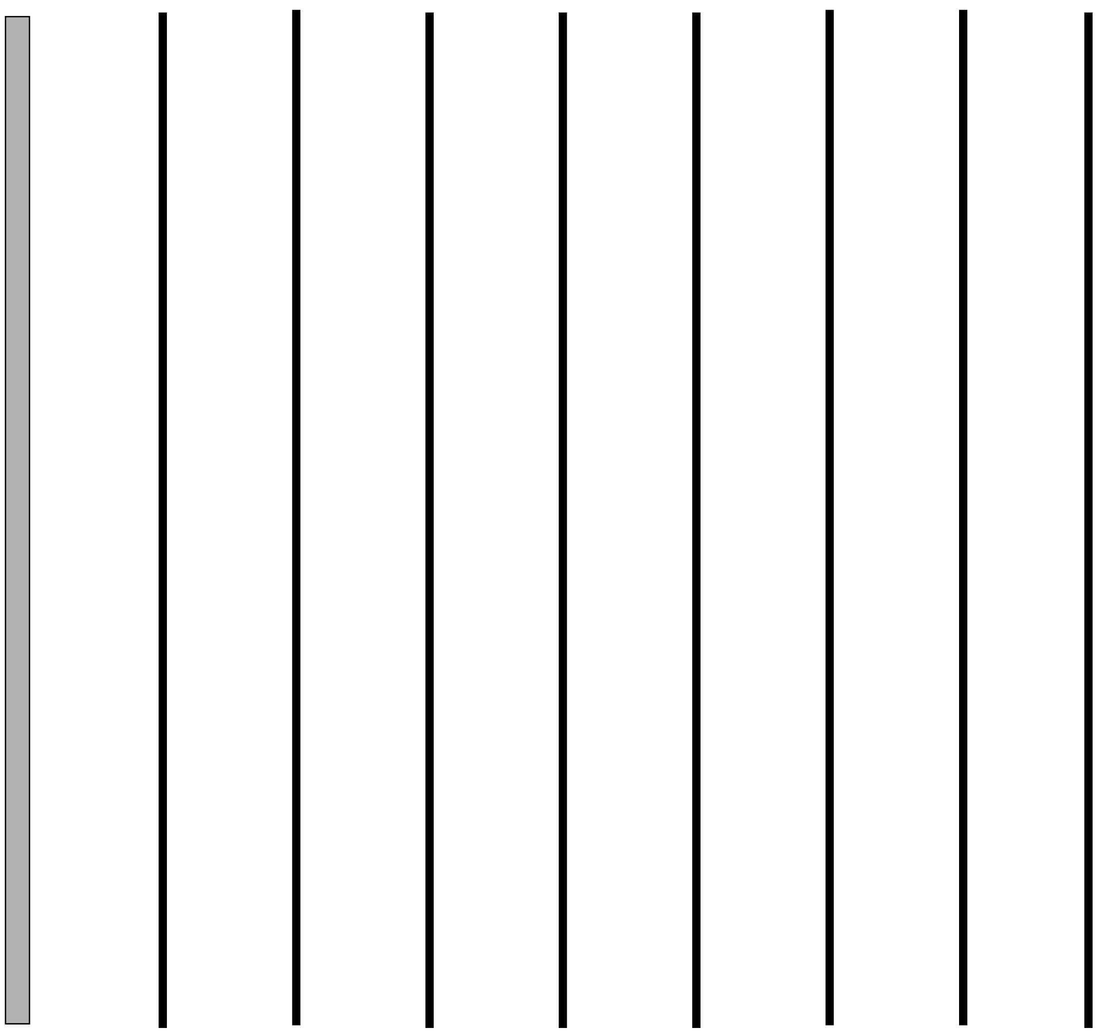
- `Collision::occur()` tells the movable piece *m* to collide with an arcade game piece *x* at a collision point *p*, and then *m* tells *x* to collide with an object of the specific class of *m* at *p*.

```
void Collision::occur() {
  _movablePiece.collideWith(_arcadeGamePiece, _point);
};

void Puck::collideWith(ArcadeGamePiece &piece, CPoint p)
{
  piece.collideWithPuck(*this);
};

void Brick::collideWithPuck(Puck &puck, CPoint p) {
  puck->reverseY();
  delete this;
};
```

- Note that if another kind of movable piece X were added, then we would have to add a method `collideWithX(...)` in each class in the ArcadeGamePiece hierarchy. The pure virtual member function concept is useful for ensuring that happens.

Putting Another Puck in Play

- When a puck hits the floor, it should be replaced with another from the puck supply.
- Where should that happen? Here’s the scenario:

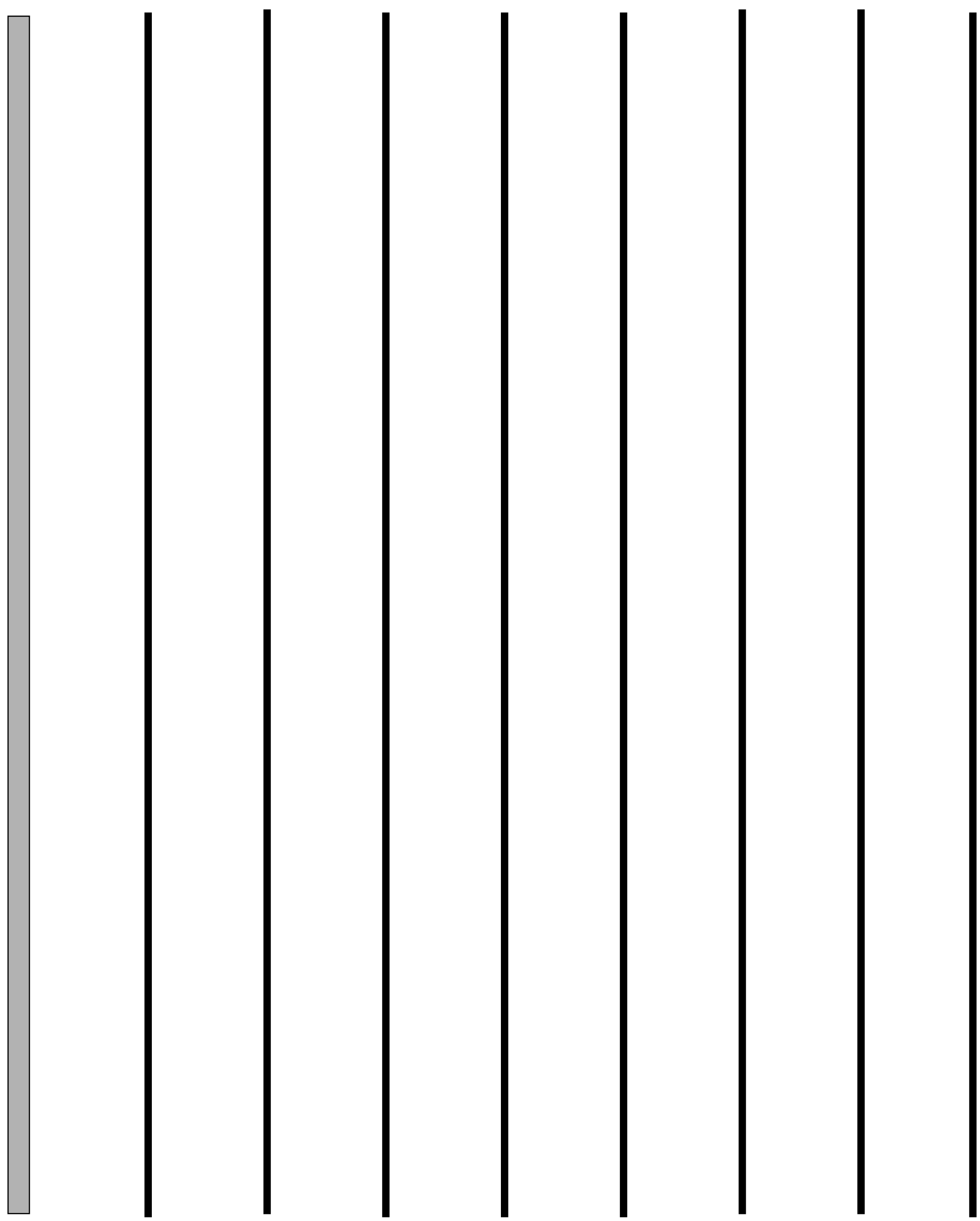


Ending the Match

- Now, consider the problem of ending a match:
  - When all the bricks are gone, the match is won,
  - When all the pucks are gone, the match is lost.
- Having the program exit at the point one of these conditions occurs would preclude the selection of “play again.”
- Clearly, the “win” condition can be detected by the brick pile, while the “lose” condition can be detected by the puck supply.

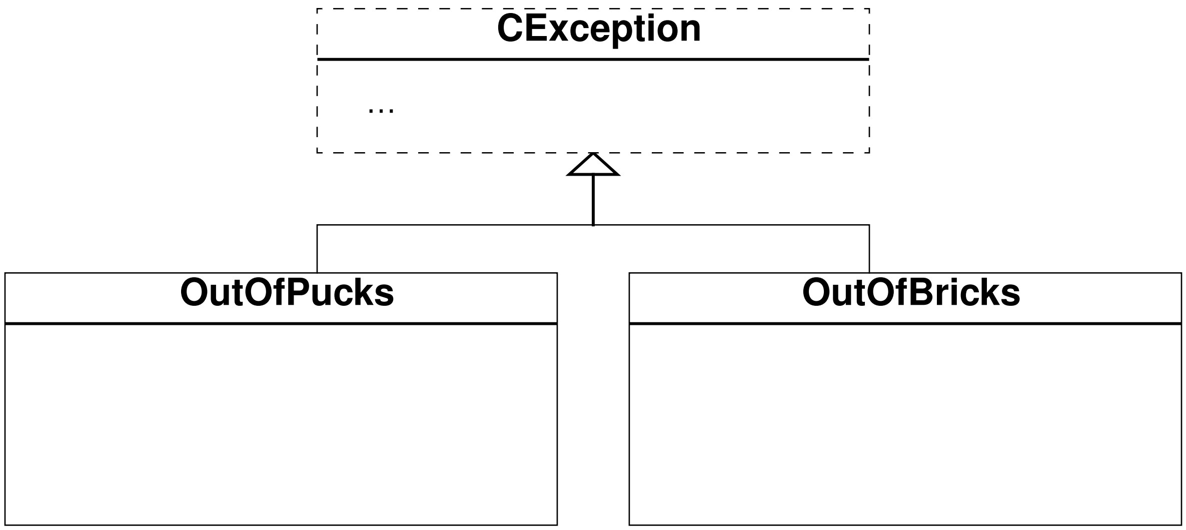
  What design changes are necessary to accommodate the processing associated with these conditions? [The scenario for a lose situation is shown on the next page.]

The “Lose” Scenario



Our Design

- We use another pair of exceptions: OutOfPucks and OutOfBricks that, when propagated to the match, terminate play and produces a dialog box to congratulate (or console) the player.



- Note that the root class CException is required by Visual C++. However, polymorphism can be exploited in exception handler parameters just as it can be in member function parameters. In this case, however, it is important for our code to distinguish the two, and their relationship to each other is really just coincidental—"out of something."

Summary

-
-
-
-

title

Text

StationaryPiece Hierarchy

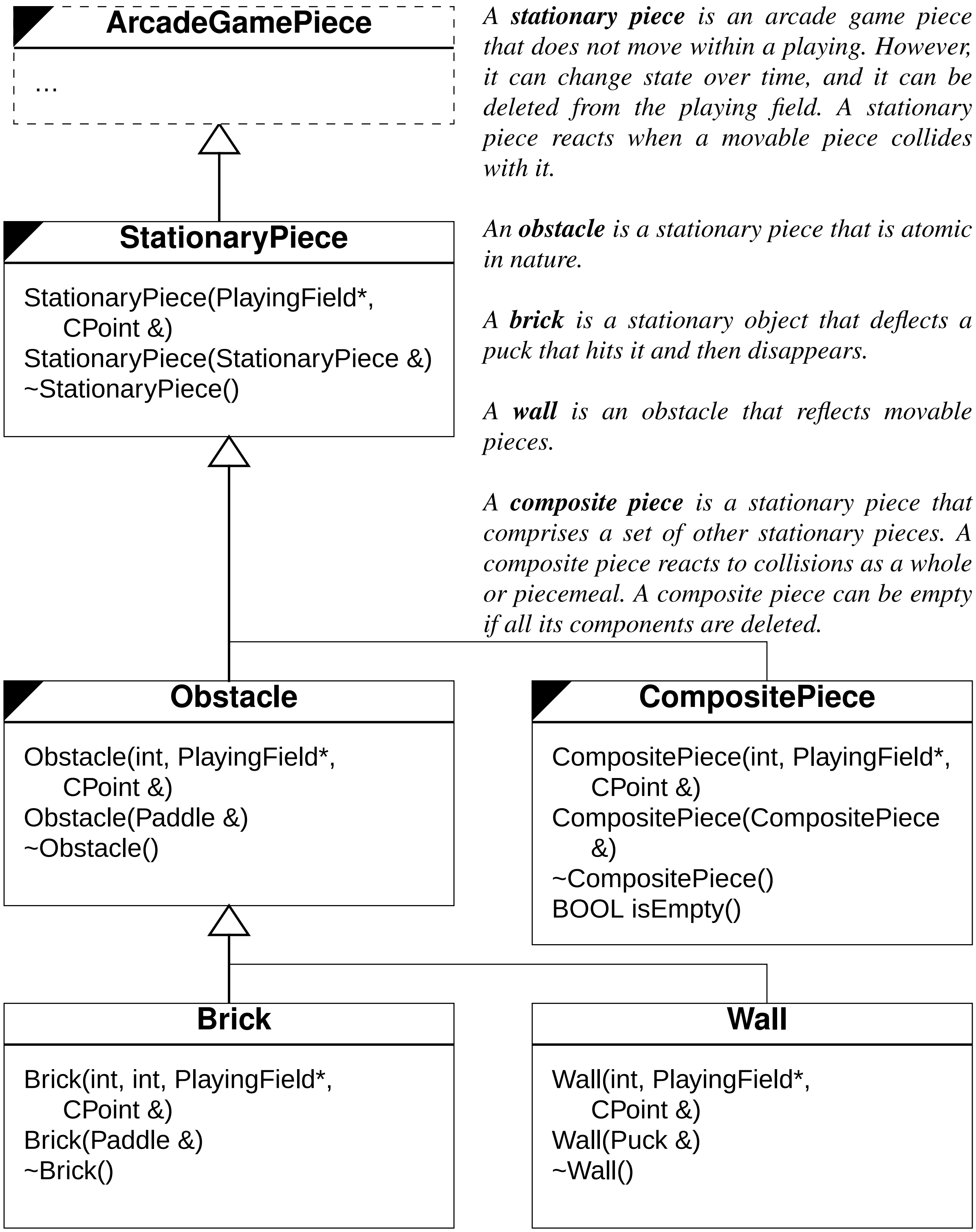
*A **stationary piece** is an arcade game piece that does not move within a playing. However, it can change state over time, and it can be deleted from the playing field. A stationary piece reacts when a movable piece collides with it.*

*An **obstacle** is a stationary piece that is atomic in nature.*

*A **brick** is a stationary object that deflects a puck that hits it and then disappears.*

*A **wall** is an obstacle that reflects movable pieces.*

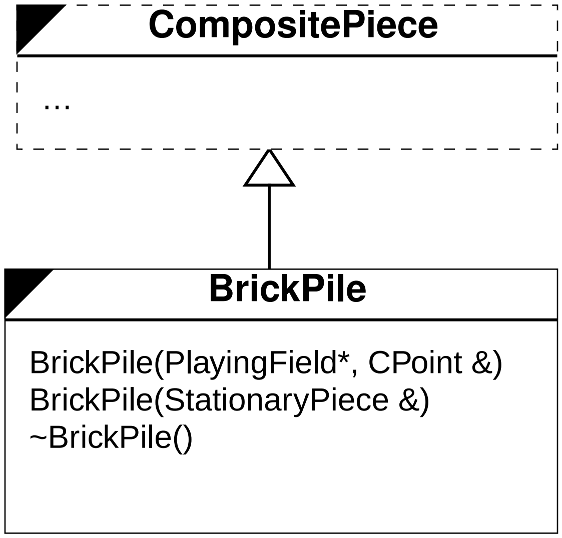
*A **composite piece** is a stationary piece that comprises a set of other stationary pieces. A composite piece reacts to collisions as a whole or piecemeal. A composite piece can be empty if all its components are deleted.*



title

Text

CompositePiece Hierarchy



*A **brick pile** is a composite piece that comprises a two-dimensional array of bricks. When a movable piece enters a brick pile, it might strike (and break) one of the bricks.*

title

Text

PuckSupply and Velocity

PuckSupply
PuckSupply(PlayingField *) PuckSupply(PuckSupply &) ~PuckSupply() int numberLeft() const Puck *getPuck()

A ***puck supply*** *holds pucks until they are removed for use in a playing field.*

Velocity
Velocity() Velocity(const Speed, const Direction) ~Velocity() Speed getSpeed() const Speed getSpeedX() const Speed getSpeedY() const void setSpeed(Speed const) Direction getDirection() const void setDirection(Direction const)

A ***velocity*** *is a rate of change in position made by an object over a unit interval of time, characterized by a speed and a direction in some Cartesian coordinate system. A velocity is a vector, having a direction and magnitude in the coordinate system.*

•