Section 3.4
Polymorphism

1. Overview
2. Pointers and class hierarchy
3. Dynamic binding
4. Abstract classes
5. Dynamic casting
6. Encapsulating behaviour
7. Strategy design pattern

3.4.1 Overview

- What is polymorphism?
  - in general
    - *poly*: many
    - *morph*: shape
  - in programming
    - serves to isolate client classes from derived class implementations
      - *client classes* are the class users (classes that use our classes)
    - client classes use base class interface to manipulate derived objects
      - they do **not** know the kind of derived object
    - requires the manipulation of derived objects using base class pointers
    - it *uses* inheritance
    - it is *not* the same as inheritance

Overview (cont.)

- Why polymorphism?
  - enables the generalized use of a class hierarchy
    - the interface is provided at the base class, as a set of public functions
    - derived classes override these functions with specialized behaviour
  - allows each class in hierarchy to implement its own behaviour
  - adding new derived classes has no impact on existing classes
  - client classes do **not** need to know the kind of derived object
    - if your classes check the type of object, then it's **not** polymorphism

Overview (cont.)

- How does polymorphism work?
  - use base class pointers to manipulate derived class objects
  - invoke generalized function, overridden by specialized one
    - each object knows its own specialized behaviour
  - the correct function on the correct class is selected at runtime
    - this is called *dynamic binding*
    - ... more on this later ...

Overview (cont.)

- Two unbreakable rules of polymorphism:
  - it only works on objects related to each other by *inheritance*
  - it only works using *pointers* to the objects
    - not the objects themselves
- Underlying idea:
  - derived class object can be treated as a base class object
    - because of the derived class object “is-a” kind of base class object
  - **`coding example <p1>`**

3.4.2 Pointers and Class Hierarchy

- Pointer and object combinations

		Type of object	
		base class	**derived class**
Type of pointer	**base class**		
	derived class		

3.4.3 Dynamic Binding

- Terminology:
  - handle: identifier used to access an object
    - the variable name of the object
    - a reference to the object
    - a pointer to the object
- Problem:
  - how do we invoke derived class behaviour, given a base class handle?
- Solution:
  - dynamic binding
  - **`coding example <p2>`**

Dynamic Binding (cont.)

- What is function binding:
  - on a function call, it's selecting the correct function to execute
    - a specific function must be chosen
  - static binding:
    - the selection made at compile time
    - it's always used when called using an object variable name
    - it *can* be used when called using object pointer (non-virtual functions)
  - dynamic binding:
    - the selection made at runtime
    - it's never used when called using an object variable name
    - it *can* be used when called using object pointer (virtual functions)

Dynamic Binding (cont.)

- Advantages of dynamic binding
  - function to call is selected based on the **type of object**
    - it is **not** selected based on the type of handle
- Implementing dynamic binding
  - use virtual functions

Virtual Functions

- What is a virtual function?
  - a function is selected for execution at runtime
  - the selection is based on the type of *object*, not the type of *handle*
    - with a base class object, the base class function is called
    - with a derived class object, the derived class function is called
  - “virtual-ness” is inherited all the way down the hierarchy
    - by convention, we repeat the `virtual` keyword in every class definition

Virtual Functions (cont.)

- Non-virtual function:
  - a function is selected for execution at compile time
  - the selection is hard-coded to the type of *handle*
  - **`coding example <p3>`**

Virtual Functions (cont.)

- Virtual destructor:
  - it is used to deallocate memory using a base class pointer
  - it calls the correct destructor based on the type of object
- Non-virtual destructor:
  - it calls the destructor based on the type of handle
  - this can result in unpredictable behaviour
- **coding example <p4>**

Virtual Functions (cont.)

- Which function is called?

			Type of object	
			base class	derived class
NON VIRTUAL FUNCTION	Type of pointer	base class		
		derived class		
VIRTUAL FUNCTION	Type of pointer	base class		
		derived class		

3.4.4 Abstract Classes

- What is an abstract class?
  - a class that is too general to have instances
- Characteristics:
  - no objects of the class can be created
  - typically used as a base class to provide a generalized interface
  - in C++, an abstract class is specified using a *pure virtual function*
  - in UML, the name of an abstract class is represented in *italics*

Abstract Classes (cont.)

- Pure virtual function
  - it is a virtual function that is given *no implementation*
    - a special syntax is used for this
  - it guarantees that all concrete classes **must** override it
- Classes and pure virtual functions
  - having at least one pure virtual function makes a class abstract
  - all concrete derived classes must provide an implementation
  - **`coding example <p5>`**

3.4.5 Dynamic Casting

- Polymorphism is the best way to program for general case
- Bad second-place alternatives
  - runtime type information (RTTI)
  - dynamic casting
  - do not use these unless absolutely necessary

Dynamic Casting (cont.)

- RTTI
  - `typeid` operator used on an object
    - returns reference to `type_info` object
    - can be queried for class name
  - not all compilers enable this by default
  - bad substitute for good design techniques

Dynamic Casting (cont.)

- Dynamic casting
  - `dynamic_cast` operator is used to *downcast* base class pointer
    - used when we need to access derived class member of an object
    - returns 0 if the downcast fails
  - it's a safe way to check if an object is of a specific type
  - **coding example <p6>**
- Polymorphism example: race between Tortoise and Hare
  - **coding example <p7>**

3.4.6 Encapsulating Behaviour

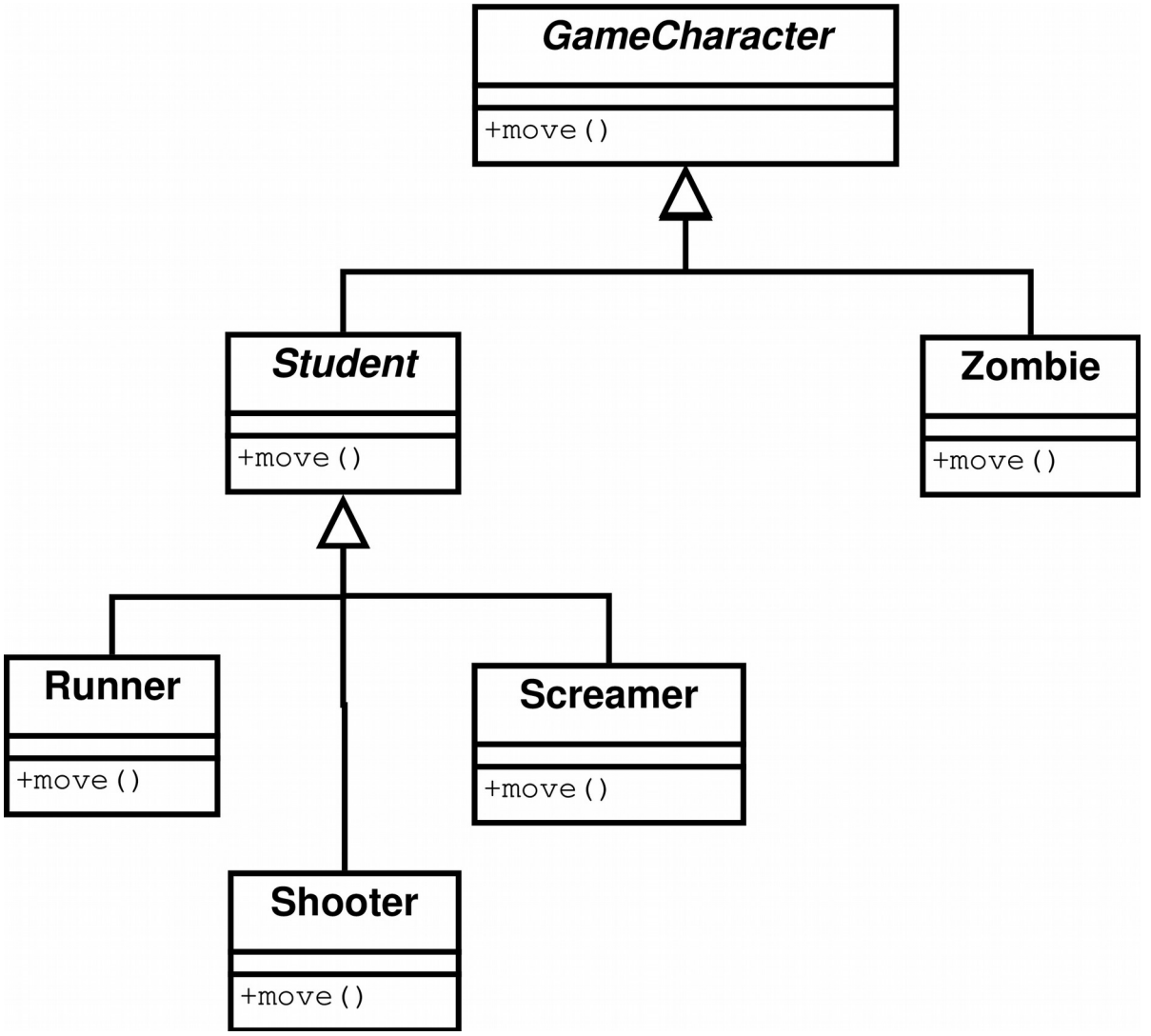
- What is a behaviour class?
  - a class that encapsulates a set of behaviours or algorithms
- Purpose
  - it allows client class to dynamically change which code executes
  - it treats behaviour as an object that can be switched at runtime
- Why?
  - planning for changes

Encapsulating Behaviour (cont.)

- Walk a mile in the user's shoes
  - let's say you paid good money for a game
  - you're playing, and your character undergoes a major change
    - for example, dies and becomes a zombie
  - your character's behaviour has to change too
    - for example, chase people instead of running from zombies
  - does the game force you to:
    - quit the game
    - change the source code to change the behaviour
    - recompile the game and restart
  - how much would you pay for this game?

Traditional Approach: Using Entity Classes

- Encapsulating behaviour in the entity classes



Using Entity Classes (cont.)

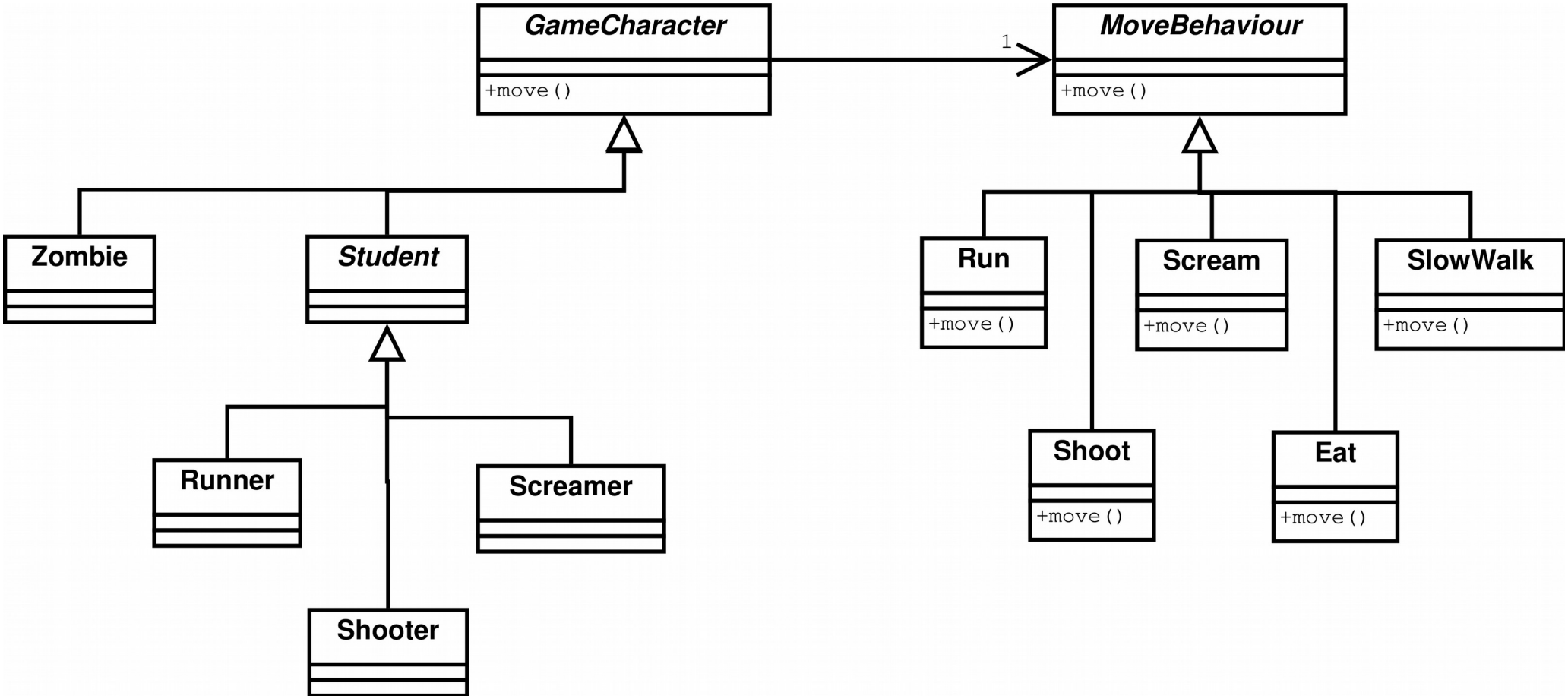
- Traditional approach
  - promotes data abstraction, encapsulation
  - uses polymorphism
- Problem
  - changing behaviour at runtime
    - for example, how does a screamer become a brain eater?

Using Behaviour Classes

- New solution
  - encapsulate behaviour inside a behaviour class
  - entity object delegates responsibility to behaviour object
  - entity object can point to different behaviour object, as needed

Using Behaviour Classes (cont.)

- Encapsulating behaviour in a behaviour class



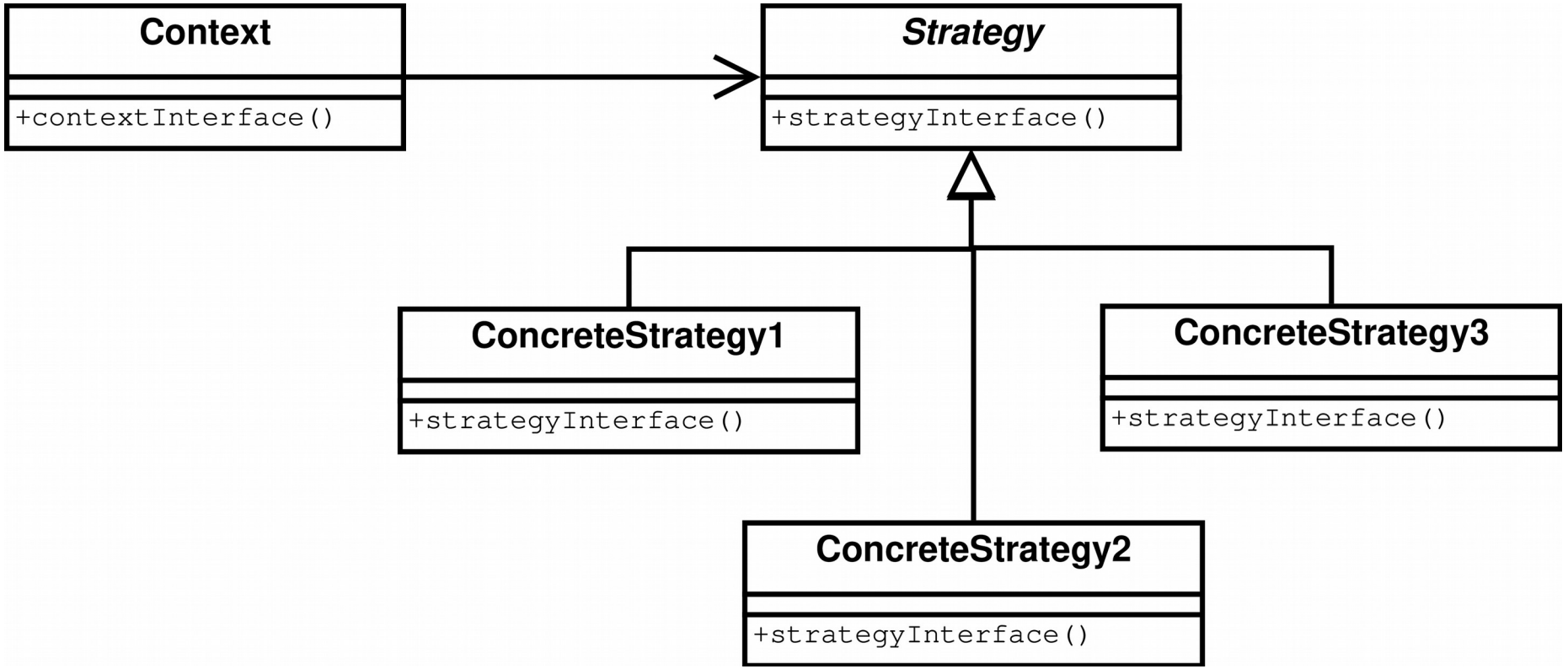
Using Behaviour Classes (cont.)

- Advantages:
  - adding new behaviour
    - add new behaviour class
    - no need to change entity objects
  - reusing behaviours
    - zombies with guns? they can reuse existing shooting behaviour
  - behaviour can change at runtime
    - semester is over? zombies can become students again
  - still promotes encapsulation and uses polymorphism
  - **coding example <p8>**

3.4.7 Strategy Design Pattern

- Behavioural design pattern
- Provides a family of algorithms
  - it defines an abstract interface for a family of algorithms
  - it encapsulates each type of behaviour (each algorithm)
  - the concrete implementations are interchangeable at runtime

Strategy Design Pattern (cont.)



Strategy Design Pattern (cont.)