Chapter 3.3
Programming Fundamentals

- Languages
- Paradigms
- Basic Data Types
- Data Structures
- OO in Game Design
- Component Systems
- Design Patterns

Languages

- Language: A system composed of signs (symbols, indices, icons) and axioms (rules) used for encoding and decoding information.
- Syntax: Refers to rules of a language, in particular the structure and punctuation.
- Semantics: Refers to the meaning given to symbols (and combinations of symbols).

Programming Languages

- A language for creating programs (giving instructions to a computer).
- Computers are dumb... no, really, they are.
- A computer (at the lowest level) is simply a powerful adding machine.
- A computer stores and manipulates numbers (which can represent other things) in binary.
- I don't know about you, buy I don't speak binary.

Programming Paradigms

- Paradigm – approach, method, thought pattern used to seek a solution to a problem.
- There are dozens of (often overlapping) programming paradigms including:
  - Logical (declarative, recursive, **Prolog**)
  - Functional (declarative, immutable, stateless, **Haskell**)
  - Imperative (linear, state-full, **VAST MAJORITY OF POPULAR PROGAMNING LANGUAGES**)

Imperative Programming

- Imperative programs define sequences of commands for the computer to perform.
- Structured Programming (subcategory of Imperative) requires 3 things:
  - Sequence
  - Selection
  - Repetition

Procedural Programming

- Another subcategory of Imperative.
- Uses procedures (subroutines, methods, or functions) to contain computational steps to be carried out.
- Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

Object Oriented

- Subcategory of structured programming.
- Uses "objects" – customized data structures consisting of data fields and methods – to design applications and computer programs.
- As with procedures (in procedural programming) any given objects methods or variables might be referred to at any point during a program's execution, including by other objects or itself.

Popular Languages

- Understanding programming paradigms can help you approach learning new programming languages (if they are within a paradigm you are familiar with).
- Most popular languages: C++, C, Java, PHP, Perl, C#, Python, JavaScript, Visual Basic, Shell, Delphi, Ruby, ColdFusion, D, Actionscript, Pascal, Lua, Lisp, Assembly, Objective C, etc.

Data Types

- Primitive data types: integers, booleans, characters, floating-point numbers (decimals), alphanumeric strings.
- Pointers: (void* q = &x;)
- Variables: a symbolic name associated with a value (value may be changed).
  - Strong vs. Weak typing
  - Implicit vs. Explicit type conversion

Data Structures

- Arrays
  - Elements are adjacent in memory (great cache consistency)
  - They never grow or get reallocated
  - In C++ there's no check for going out of bounds
  - Inserting and deleting elements in the middle is expensive
  - Consider using the STL Vector in C++

Data Structures

- Linked lists
  - Very fast and cheap to add/remove elements.
  - Available in the STL (std::list)
  - Every element is allocated separately
    - Lots of little allocations
  - Not placed contiguously in memory

Data Structures

- Dictionaries (hash maps)
  - Maps a set of keys to some data.
  - std::map, std::multimap, std::hash
  - Very fast access to data
    - Underlying structure varies, but is ordered in some way.
  - Perfect for mapping IDs to pointers, or resource handles to objects

Data Structures

- Stacks (LIFO)
  - Last in, first out
  - std::stack adaptor in STL
  - parsing
- Queues (FIFO)
  - First in, first out
  - std::deque
  - Priority queues for timing issues.

Data Structures

- Bit packing
  - Fold all necessary data into small number of bits
  - Very useful for storing boolean flags
    - (pack 32 in a double word)
  - Possible to apply to numerical values if we can give up range or accuracy
  - Very low level trick
    - Only use when absolutely necessary
    - Used OFTEN in networking/messaging scenarios

Bit Shifting

The bitwise operators

Operator	Name	Description
a&*b*	and	1 if both bits are 1. 3 & 5 is 1.
a|*b*	or	1 if either bit is 1. 3 | 5 is 7.
a^*b*	xor	1 if both bits are different. 3 ^ 5 is 6.
~*a*	not	This unary operator inverts the bits. If ints are stored as 32-bit integers, ~3 is 11111111111111111111111111111100.
n<<*p*	left shift	shifts the bits of *n* left *p* positions. Zero bits are shifted into the low-order positions. 3 << 2 is 12.
n>>*p*	right shift	shifts the bits of *n* right *p* positions. If *n* is a 2's complement signed number, the sign bit is shifted into the high-order positions. 5 >> 2 is 1.

```
int age, gender, height, packed_info;
. . .   // Assign values

// Pack as AAAAAAAA G HHHHHHH using shifts and "or"
packed_info = (age << 8) | (gender << 7) | height;

// Unpack with shifts and masking using "and"
height = packed_info & 0x7F;   // This is binary 0000000001111111
gender = (packed_info >> 7) & 1;
age    = (packed_info >> 8);
```

Union Bitpacking (C++)

```
union Packed_Info {
    int age : 8;
    int gender: 1;
    int height:  7;
}

Packed_Info playercharacter;

playercharacter.age = 255;
```

Object Oriented Design

- Concepts
  - Class
    - Abstract specification of a data type; a pattern or template of an object we would like to create.
  - Instance
    - A region of memory with associated semantics to store all the data members of a class; something created using our pattern.
  - Object
    - Another name for an instance of a class

Classes

```
#include <iostream>
using namespace std;

Class Enemy {
    int height, weight;
    public: void set_values (int,int);
}  ;

void Enemy::set_values (int a, int b)  { height = a; weight = b; }

int main () {
    Enemy enemy1;
    enemy1.set_values (36,350);
    return 0;
}
```

Object Oriented Design

- Inheritance
  - Models “is-a” relationship
  - Extends behaviour of existing classes by making minor changes in a newly created class.
- Example:

```
... // adding a AI function
 public:
       void RunAI();
```

Inheritance

class derived_class: *public* base_class

{ /*…*/ };

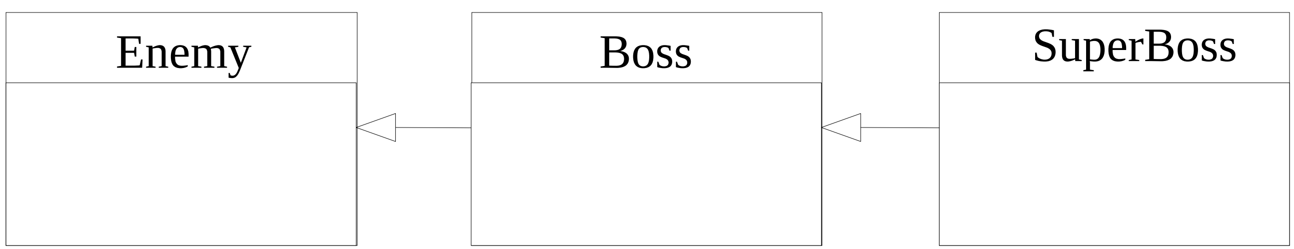
The public access specifier may be replaced protected or private. This access specifier limits the most accessible level for the members inherited from the base class

```
class Boss: public Enemy {
   private: int damage_resitance;
   public: void RunAI();
} ;

class SuperBoss: public Boss {
   public: void RunAI();
} ;
```

Object Oriented Design

- Inheritance
  - UML diagram representing inheritance



Object Oriented Design

- Polymorphism
  - The ability to refer to an object through a reference (or pointer) of the type of a parent class
  - Key concept of object oriented design
  - Allow (among other things) for me to keep an array of pointers to all objects in a particular derivation tree.

```
Enemy* enemies[256];

enemies[0] = new Enemy;
enemies[1] = new Enemy;
enemies[2] = new Enemy;
enemies[3] = new Boss;
enemies[4] = new SuperBoss;
```

Object Oriented Design

- Multiple Inheritance
  - Allows a class to have more than one base class
  - Derived class adopts characteristics of all parent classes
  - Huge potential for problems (clashes, casting, etc)
  - Multiple inheritance of abstract interfaces is much less error prone
  - Use pure virtual functions to create abstract interfaces.

```
class Planet {
    private:
        double gravitationalmass;
    public:
        void WarpTimeSpace() = 0;
        // Note pure virtual function
} ;

class SuperBoss: public Enemy, public Planet
 {    };
```

Component Systems

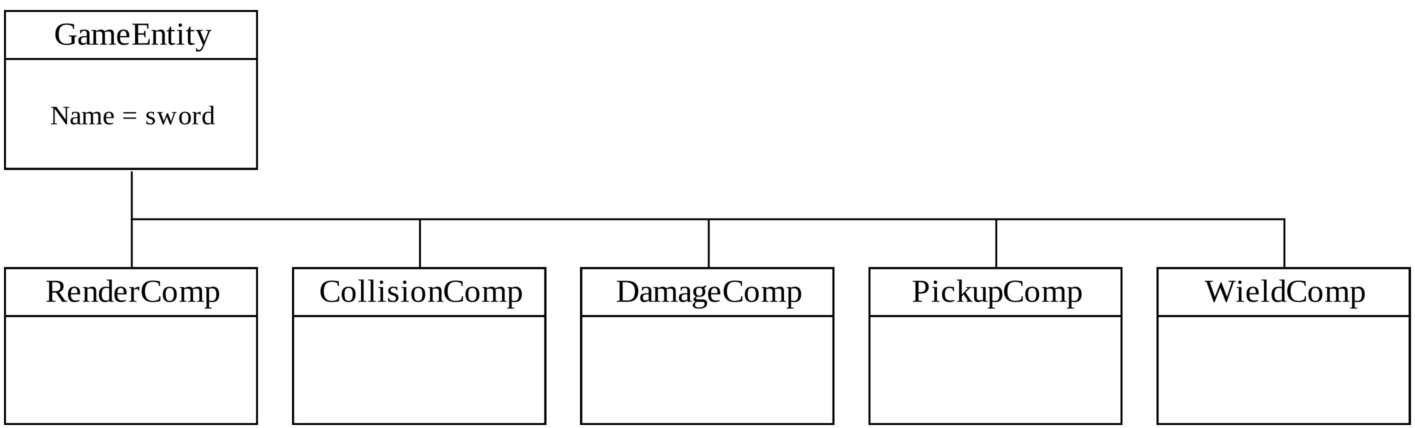
- Limitations of inheritance
  - Tight coupling
  - Unclear flow of control
  - Not flexible enough
  - Static hierarchy

Component Systems

- Component system organization
  - Use aggregation (composition) instead of inheritance
  - A game entity can "own" multiple components that determine its behavior
  - Each component can execute whenever the entity is updated
  - Messages can be passed between components and to other entities

Component Systems

- Component system organization



Component Systems

- Data-Driven Composition
  - The structure of the game entities can be specified in data
  - Components are created and loaded at runtime
  - Very easy to change (which is very important in game development)
  - Easy to implement with XML (go hierarchical databases) which has excellent parser utilities.

Component Systems

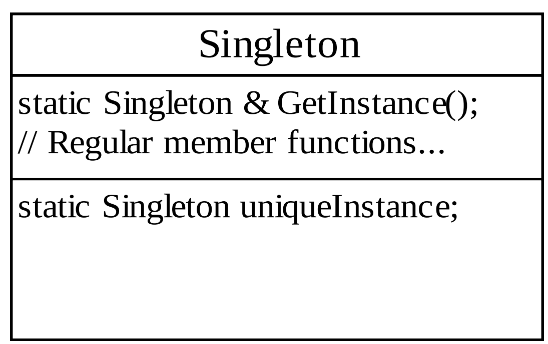
- Analysis
  - Very hard to debug
  - Performance can be a bottleneck
  - Keeping code and data synchronized can be a challenge
  - Extremely flexible
    - Great for experimentation and varied gameplay
  - Not very useful if problem/game is very well known ahead of time

Design Patterns

- General solutions that to specific problems/situations that come up often in software development.
- Deal with high level concepts like program organization and architecture.
- Not usually provided as library solutions, but are implemented as needed.
- They are the kinds of things that you would expect a program lead, or project manager to know how to use.

Design Patterns

- Singleton
  - Implements a single instance of a class with global point of creation and access
  - Don't overuse it!!!
  - http://www.yolinux.com/TUTORIALS/C++Singleton.html

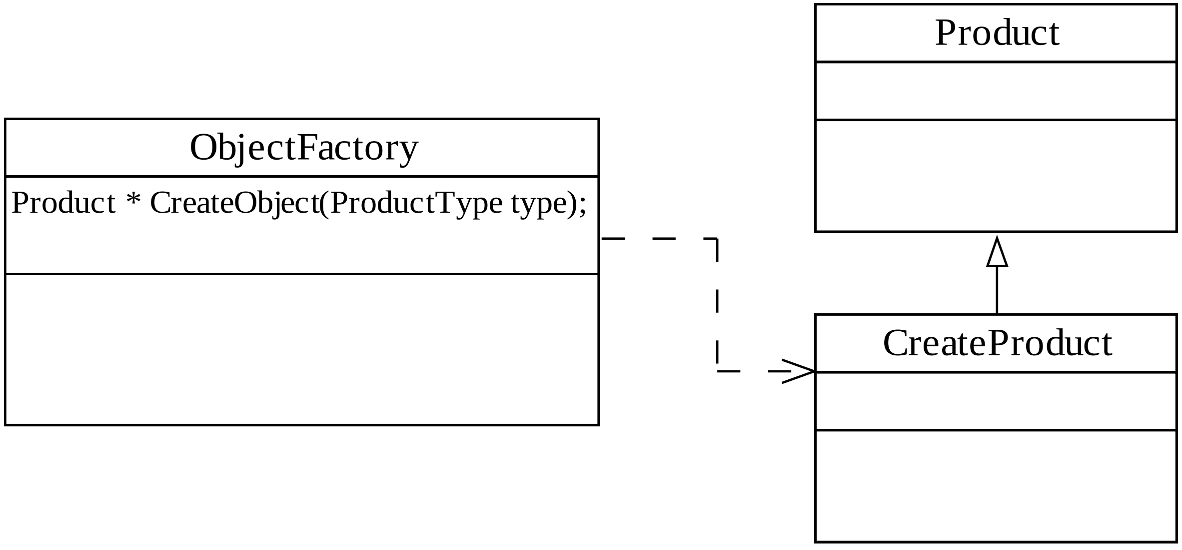


Design Patterns

- Object Factory
  - Creates objects by name
  - Pluggable factory allows for new object types to be registered at runtime
  - Extremely useful in game development for creating new objects, loading games, or instantiating new content after game ships
  - Extensible factory allows new objects to be registered at runtime (see book.)

Design Patterns

- Object factory

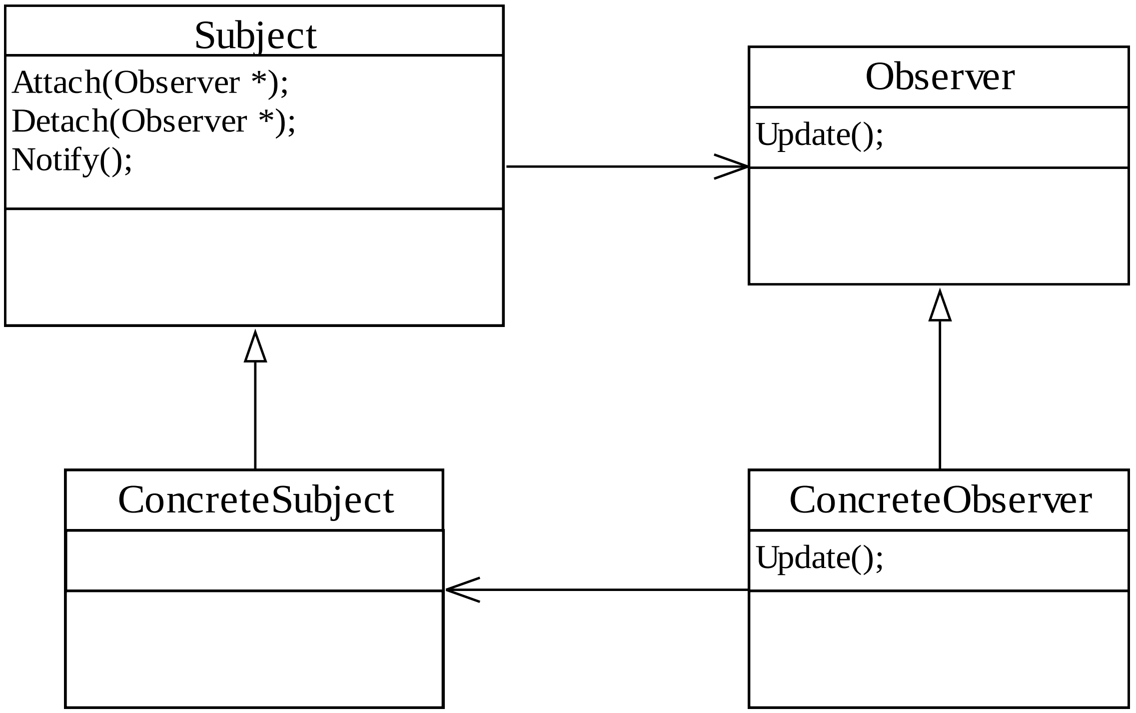


Design Patterns

- Observer
  - Allows objects to be notified of specific events with minimal coupling to the source of the event
  - Two parts
    - subject and observer
  - Observers register with a subject to so that they can be notified when certain events happen to the subject.

Design Patterns

- Observer

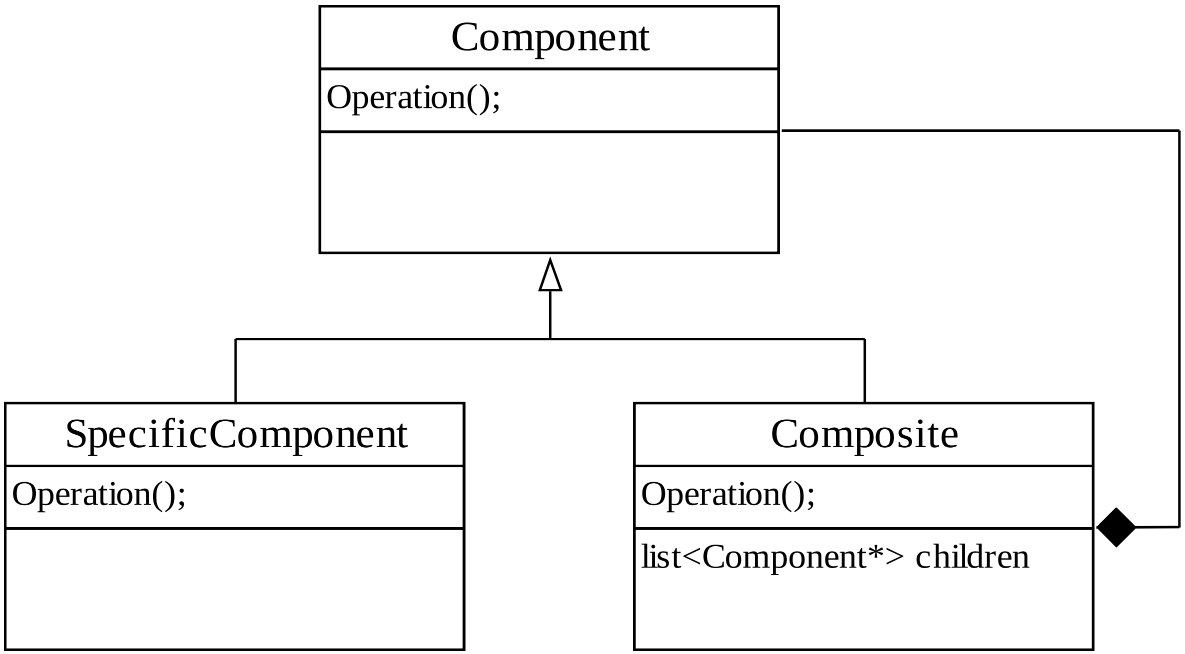


Design Patterns

- Composite
  - Allow a group of objects to be treated as a single object
  - Very useful for GUI elements, hierarchical objects, inventory systems, etc

Design Patterns

- Composite



Other Design Patterns

- Decorator
- Façade
- Visitor
- Adapter
- Flyweight
- Command

The End