WELCOME TO
TUTORIAL 8
MORE DESIGN PATTERNS
TOPICS TODAY

• Strategy Pattern
• Factory Pattern
• Template Functions/Classes
STRATEGY PATTERN

Goal:

• Strategy pattern gives several algorithms that can be used to perform particular operation or task.
• The strategy pattern allows programmers to change the implementation of something used at runtime.
• Strategy lets the algorithm vary independently from the clients that use it.
STRATEGY PATTERN

How:

• Define a family of algorithms, encapsulate each one, and make them interchangeable

• Capture the abstraction in an interface, bury implementation details in derived classes

Example: strategy.cc
FACTORY

A Factory is an object that makes other objects. With a factory, you can use:

• Encapsulation: client code is not directly tied to specific classes, so classes can be changed, added, or refactored without changing client code

• Polymorphism: delegating the creation of an object to the right factory means that we can decide what kind of object to create at runtime.
A simple factory’s “create” method can look like this:

```cpp
Enemy* Factory::createEnemy(Difficulty difficulty) {
    if (difficulty == BEGINNER)
        return new LittleDude();
    if (difficulty == STANDARD)
        return new BigDude();
    if (difficulty == EXPERT)
        return new WhatsThatThing();
}
```

This design allows us to change what enemy corresponds with what difficulty as the design of the game changes.
FACTORY METHOD PATTERN

- The previous example is simple, practical, and commonly used, but does not take advantage of Polymorphism. The Factory Method pattern does this by having a separate factory for each subclass, then using a pointer to a base factory class. This allows us to use Polymorphism, swapping factories at runtime as necessary.
FACTORY METHOD PATTERN

The same factory from before can be refactored using the Factory Method pattern like so:

```cpp
Enemy* EnemyFactory::createEnemy { return create(); }
Enemy* EnemyFactory::create() = 0;
Enemy* LittleDude::create() { return new LittleDude; }
Enemy* BigDude::create() { return new BigDude; }
```

Note that this is also an example of the Template Method pattern. The `createEnemy` function is not abstract, but the private `create()` method is abstract.
ABSTRACT FACTORY

• After finding a good way to produce a single class, the next logical step is to create a factory that produces multiple classes. We can create an Abstract Factory that can create multiple types of abstract products, then each Concrete Factory can define which concrete products it wants to produce. This lets each Concrete Factory represent a family of related concrete objects.
FACTORY METHOD PATTERN

• Now we still want to generate enemies (WeakEnemy or StrongEnemy) based on difficulty levels (BEGINNER or EXPERT).
• For each type of enemies, we need two types of spawners (FireLevelSpawner and IceLevelSpawner).
• Fire Elemental and Ice Elemental are for WeakEnemy.
• Dragon and Ice Golem are for StrongEnemy.
Enemy* EnemySpawner::createEnemy(Difficulty difficulty) {
    if (difficulty == BEGINNER)
        return createWeak();
    if (difficulty == EXPERT)
        return createStrong();
}
TEMPLATE FUNCTIONS

• The simple idea is to pass data type as a parameter so that we don’t need to write same code for different data types. For example a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

• C++ adds two new keywords to support templates: ‘template’ and ‘typename’. The second keyword can always be replaced by keyword ‘class’.
HOW THEY WORK

• Templates are expanded at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

• **Function Templates** We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray()

• funcTemplate.cc
CLASS TEMPLATES

• Like function templates, class templates are useful when a class defines something that is independent of data type. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

• classTemplate.cc