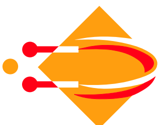
# *Game Program Design: An object-oriented approach*

Kim Steenstrup Pedersen

kimstp@itu.dk

www.itu.dk/courses/SSPG/F2005/

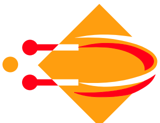
The IT University of Copenhagen



## *Code reuse*

Classes of code reuse:

- Functional:
  - Using functions in your code
- Copy-and-paste:
  - Reuse by copy-and-pasting the code you need.
- Horizontal:
  - Software with short lifespan reusable by different projects at the same time.
- Vertical:
  - Software with long lifespan reusable by different projects over a longer period of time.
- General:
  - Both horizontally and vertically reusable software.
- Pattern:
  - Pieces of code that draw upon the same basic idea (design pattern).

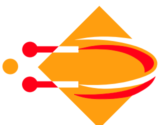


# Code reuse (Cont.)

Classes of code reuse:

- Engine:
  - Collection of libraries and code that support a major part of the game functionality.
- Component:
  - An independent library/collection of code that solves some subproblem.

Advice:
Avoiding unnecessary dependencies in code increases its reusability!



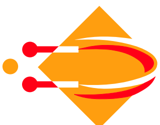
# *Reuse granularity*

Size of what we want to reuse is important (going from easy to hard to achieve reusability):

- Functions
- Classes
- Modules
- Packages

Advice:

- Write your code to be as general as possible and chances are that it can be reused.
- The object-oriented paradigm can help produce reusable code.
- Code reuse is an acquired skill, so practise this.



# *Program design process*

The design of the game program could follow this process:

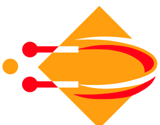
**Phase 1 - Brainstorming:** Write down whatever you think are important parts of the problem.

**Phase 2 - Prune the tree:** Remove irrelevant and redundant things.

**Phase 3 - Draw the bubbles and lines:** Form classes and their relationships and ownerships.

Example: Designing a car for our car game.

- What are the important parts of a car?
- What is not important for our game?
- What can be turned into classes?



## *Class diagram notation*

When doing the design you need some notation for your class diagrams.

Use UML diagram notation or whatever, as long as you fix and agree with the notation!

The book [JG] uses:

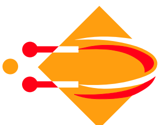
**Circled name:** Classes

**Shaded circled name:** Base classes

**Dashed arrows:** Inheritance relationship with arrows pointing to the base class.

**Fully drawn arrow:** "Has a" relationship, i.e. ownership, with indication of multiplicity (see table below).

| | |
|---|---|
| Name | Has exactly 1 |
| *Name | Has 0 to N |
| #Name | Has 1 to N |
| Name[k] | Has exactly k |



# *Program design process*

The design of the game program could follow this process:

**Phase 1 - Brainstorming:** Write down whatever you think are important parts of the problem.

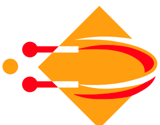
**Phase 2 - Prune the tree:** Remove irrelevant and redundant things.

**Phase 3 - Draw the bubbles and lines:** Form classes and their relationships and ownerships.

**Phase 4 - Validate the design:** Did we miss some details?

Example: Designing a car for our car game.

- What are the important parts of a car?
- What is not important for our game?
- What can be turned into classes?
- Did we miss something?

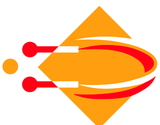


# Design patterns

The concept of design patterns was introduced by the so-called Gang of Four [Gamma, Helm, Johnson, Vlissides, 1994].

Design patterns are a set of rules/guidelines for solving object-oriented design issues that occur often in all types of programs.

These patterns can be applied to any object-oriented language (e.g. C++ or Python).



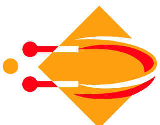
## *Using design patterns for games*

Gold [JG] mentions a couple of patterns as being important for game development:

- The interface
- Singleton
- Object factory
- Manager
- Visitor / Iterator
- Strawman
- Prototype

Lets have a closer look at those.

If you want to know more, buy the Gang of Four book or take the advanced object oriented design course here at ITU.



# The interface: Abstract class

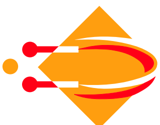
Idea:

- Ensure common methods exists for different classes.
- Interface decoupled from implementation.

Here are two solutions.

Abstract class:

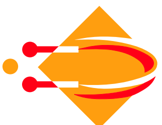
```
// Foo.h
class Foo {
public:
  Foo() {};
  virtual ~Foo() {};

  virtual void Bar() = 0;
  virtual void Pub() = 0;
};
```



# The interface: Abstract class (cont.)

Implementation:

```
// FooImpl.h
class FooImpl: public Foo {
public:
  FooImpl() {};

  /* virtual */ void Bar();
  /* virtual */ void Pub();
private:
  int m_iData;
};

// FooImpl.cpp
#include "FooImpl.h"

FooImpl::FooImpl() : Foo(), m_iData(0) { }

void FooImpl::Bar() {
  m_iData = 1;
}

void FooImpl::Pub() {
  m_iData = 2;
}
```

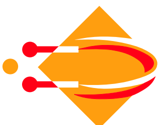


## *The interface: Abstract class (cont.)*

How to create a `FooImpl` with out knowing the implementation of `FooImpl`? The answer: A Manager.

```
// FooManager.h
class Foo; // Forward definition

class FooManager {
public:
  static Foo * CreateFoo();
};

// FooManager.cpp
#include "FooManager.h"
#include "FooImpl.h"

Foo * FooManager::CreateFoo() {
  return new FooImpl();
}
```

Now we only need to know about `Foo.h` and `FooManager.h`. We are free to make changes to `FooImpl` (will only cause a recompilation of `FooManager.cpp`).

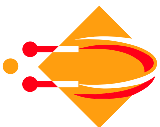


# *The interface: Pimpl = Pointer to implementation*

Here is another approach — this time in Python:

```
# FooImpl.py
# The implementation. Data hidden here
class FooImpl:
  m_iData = 0

# Foo.py
# The interface
import FooImpl

class Foo:
    def __init__(self):
        self.m_this = FooImpl.FooImpl()
    def Bar(self):
        self.m_this.m_iData = 1
    def Pub(self):
        self.m_this.m_iData = 2
```

We got rid of the extra manager class.



In Python it is not so easy to hide classes as in C++.

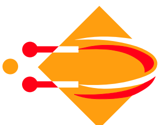
## *The interface*

Drawbacks:

- In C++ since we use abstract classes with virtual methods we cannot inline these methods (making them faster).
- In both cases an indirection is needed to access the implementation. This gives a performance penalty.

When is this useful?

- Not for “light” high performance classes such as drawing primitives.
- Good for heavy duty classes like a renderer where much time is spent in the methods of the class.



# Singleton

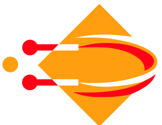
Idea:

- Only one object instance of a singleton class is allowed/logically possible. Ex. a GUI class or model database class.
- We want to prevent the user of our singleton class from creating more than one instance.

```
// ModelDatabase.h
class ModelDatabase {
public:
   // Use static methods and data (i.e. class members)
   static ModelDatabase & Instance();
private:
   // Hide the constructors and desctructors
   ModelDatabase();
   ~ModelDatabase();
};

// ModelDatabase.cpp
#include "ModelDatabase.h"
ModelDatabase & ModelDatabase::Instance() {
  static ModelDatabase anInstance; // Created first time
                                   // the method is called
  return (&anInstance);
}
```

# Using the singleton

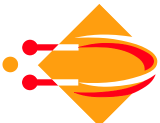
```
#include "ModelDatabase.h"

int main() {
  ModelDatabase & aDb = ModelDatabase::Instance();
  // ...
  return 0;
}
```



# Object factory

Idea:

- We would like to construct objects of various types but handle them in the same way (like the Object type in Java).
- We need a factory that can create objects of different flavors.
- We don't know in advance which objects should be created. Maybe this is read from a script file (e.g. we need 1 rocket, 2 machine guns, and a car).

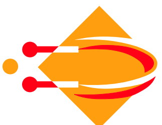


# Object factory (cont.)

```
// objectfactory.h
#include <map>
#include <string>

class Object; // Definition of object
              // without implementation

class ObjectFactory {
 public:
  typedef Object * (*tCreator)(); // Function pointer type

  typedef std::map<std::string, tCreator> tCreatorMap;

  bool Register(const char * pType, tCreator aCreator);
  Object * Create(const char * pType);
  static ObjectFactory & Instance(); // The factory is
                                     // a singleton

 private:
  ObjectFactory() {};
  ~ObjectFactory() {};

  tCreatorMap m_Creators;
};
```

## Object factory (cont.)

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

using std::string;

bool ObjectFactory::Register(const char * pType,
                             tCreator aCreator) {
  string str = string(pType);
  return m_Creators.insert(tCreatorMap::value_type(
                                     str,aCreator)).second;
}

Object * ObjectFactory::Create(const char * pType) {
  tCreatorMap::iterator i = m_Creators.find(string( pType ));

  if (i != m_Creators.end()) {
    tCreator aCreator = (*i).second;
    return aCreator();
  } else
    return 0;
}
```



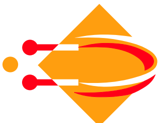
# Object factory: Creating an object for the factory

```
// rocket.cpp
#include "rocket.h"
#include "objectfactory.h"

namespace { // Define nameless namespace

  Object * createRocket() {
    return new Rocket();
  }

  // This variable definition will ensure that rocket
  // registers with the factory when the program starts
  bool s_bRegistered =
           ObjectFactory::Instance().Register("rocket",
                                              createRocket);
}

// main.cpp
#include "objectfactory.h"

int main() {
  Object * pRocket = ObjectFactory::Instance().Create("rocket");
  return 0;
}
```

## *Car Factory: An object factory in Python*

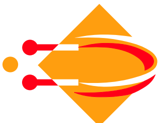
Example: An object factory that produces different types of race cars. (We do not make it singleton).

```
# car.py
class Car: # Not really necessary
    pass

# carfactory.py
class CarFactory:
    m_Creators = {}
    def Register(self, sType, aCreator):
        self.m_Creators[sType] = aCreator

    def Create(self, sType):
        f=self.m_Creators[sType]
        return f()

# Pseudo singleton
def Initialize():
    return CarFactory()

Instance = Initialize()
```



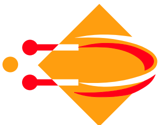
## Car Factory: Concrete cars

```
# ferrari.py
import car
import carfactory
class Ferrari(car.Car):
    def __init__(self):
        print "I am a Ferrari!"

def createFerrari():
    return Ferrari()

carfactory.Instance.Register("ferrari",createFerrari)

# audi.py
import car
import carfactory
class Audi(car.Car):
    def __init__(self):
        print "I am a Audi!"

def createAudi():
    return Audi()

carfactory.Instance.Register("audi",createAudi)
```



# Car Factory: Using the car factory

```
import ferrari
import audi
import carfactory

# Get reference to factory instance
cf = carfactory.Instance

pFerrari = cf.Create("ferrari")
pAudi = cf.Create("audi")
```



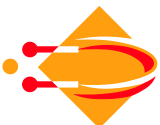
# Prototype

Idea:

- We want to create copies of objects of some type. This is done by cloning/copying from a prototype object. Similar idea an in an object factory except only one type of object is allowed at a time.
- Useful for creating objects with the same default behaviour.

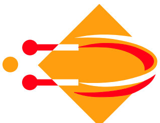
Example: A weapon with different bullet types (e.g. standard and armour piercing).

```
// weapon.h
class Bullet;

class Weapon {
public:
  Bullet * CreateBullet();
  void SetBulletPrototype( Bullet * pProto);

private:
  Bullet * m_pBulletPrototype;
};
```



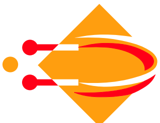
## Prototype (cont.)

```
// weapon.cpp
#include "weapon.h"
#include "bullet.h"

void Weapon::SetBulletPrototype( Bullet * pProto) {
  delete m_pBulletPrototype;
  m_pBulletPrototype = pProto;
}

Bullet * Weapon::CreateBullet() {
  return ( m_pBulletPrototype->Clone() );
}

// bullet.h
class Bullet {
public:
  Bullet() {};
  Bullet(const Bullet & that); // Copy constructor

  virtual Bullet * Clone() = 0;
}
```



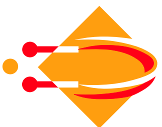
# Prototype (cont.)

```
// BulletStandard.h
#include "bullet.h"

class BulletStandard : public Bullet {
public:
  BulletStandard();
  BulletStandard(const BulletStandard & that);
  BulletStandard * Clone();
private
  Point3 m_vPosition;
};

// BulletStandard.cpp
#include "bulletstandard.h"

BulletStandard::BulletStandard(const BulletStandard & that)
       : Bullet(that) , m_vPosition(that.m_vPosition)
{
}

BulletStandard * BulletStandard::Clone() {
  return ( new BulletStandard(*this) );
}
```



# Prototype: Using the weapon with different bullet types

```
Weapon * pWeapon = GetWeapon();
pWeapon->SetBulletPrototype( new BulletStandard );

// Go shoot some ducks

pWeapon->SetBulletPrototype( new BulletArmourPiercing );

// Go shoot some tanks
```



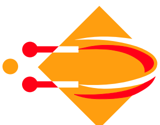
# *Visitor / Iterator, strawman, and Manager*

Visitor / Iterator allows access to a container without revealing the implementation of the container. Example: Iterators from STL that works on both `std::vector` and `std::list`.

A strawman class does nothing. It is used as a base type. Example: The `Object` class in the object factory example.

A Manager is a hybrid between a singleton and an object factory.

Lets look at an example — a state manager

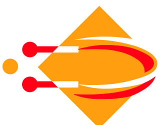


# Case Study: A Reusable State Manager in C++

Idea:
We want a state manager that can handle different game states and transitions between them. Examples of states: The game is running, paused, or in a menu mode.
Common properties of game states:

- Time evolves
- There is some visual that needs to be drawn.
- States can change into another state (transition)

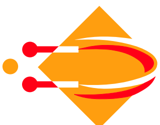
```
class state_State {
public:
  state_State();
  virtual ~state_State();

  virtual void Update(Time dt) = 0;
  virtual void Draw(Renderer * pRenderer) const = 0;
// State transitions
  virtual void OnEnter() {};
  virtual void OnLeave() {};
};
```



## *Case Study: State Manager*

The state manager should handle creation of states and state transitions and be extensible by inheritance (declare `virtual` methods). The application can subclass the state manager to add specific state transition behaviour.

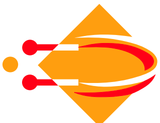
```
class state_StateManager {
public:
  state_StateManager();
  virtual ~StateManager();

  virtual void SwitchToState(state_State * pState) {
    if (m_pCurrentState != 0) {
      m_pCurrentState->OnLeave();
    }
    m_pCurrentState = pState;
    m_pCurrentState->OnEnter();
  };

private:
  state_State * m_pCurrentState;
};
```



# Case Study: State Factory

Takes care of the creation of states inside the state manager.

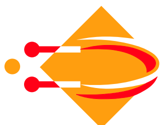
```
class state_Factory {
public:
  state_State * CreateState( const char * pName ) = 0;
};
```



# *Case Study: An application using the state manager*

The application should subclass the state manager, factory and states.

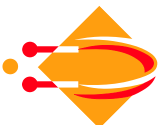
Application
Application
AppStateManager
ConcreteStateFactory
ConcreteState
State manager
State
state_StateManager
Factory
state_Factory
*States
Current
state_State



# Case Study: An application using the state manager

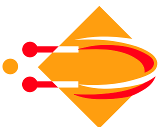
The concrete state manager and concrete states are singleton cases.

```
// ConcreteState.h
#include "state_State.h"

class ConcreteState : public state_State {
public:
  static ConcreteState & Instance();

  void Update( Time dt);
  void Draw( Renderer * pRenderer ) const;
private:
  // private data
};
```



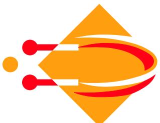
# Case Study: ConcreteState implementation

```
// ConcreteState.cpp
#include "ConcreteState.h"
#include "ConcreteStateManager.h"
#include "ConcreteStateFactory.h"

namespace {
state_State * createState() {
  return ( &ConcreteState::Instance() );
}

bool registerState() {
  ConcreteStateManager& aSM = ConcreteStateManager::Instance();
  state_Factory * pSF = aSM.GetFactory();

  return ( pSF->Register("Concrete", createState) );
}

}

ConcreteState & ConcreteState::Instance() {
  static ConcreteState anInstance;
  return ( anInstance );
}
```



# Case Study: Example of application main loop

```
Time t1 = GetTime();
Time dt = 0;

while(1) {
  ConcreteStateManager & aSM =
    ConcreteStateManager::Instance();

  state_State * pState = aSM.GetCurrentState();
  if (pState != 0) {
    pState->Update(dt);
    pState->Draw(pRenderer);
  }
  Time t2 = GetTime();
  dt = t2 - t1;
  t1 = t2;
}
```

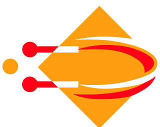


# More Python: Object references

Variables assigned an object value are references to that object. A variable-variable assignment will copy the reference not the object.

```
class MyClass:
    a = 0
    def __init__(self):
        self.a = 0

    def copy(self,A):
        self.a = A.a

# Object references
A=MyClass()
B=A
B.a=2 # Both B.a=2 and A.a=2

# How to copy an object
A=MyClass()
B=MyClass()
A.a = 2
B.copy(A) # A copy of A into the B object
```

Contrary to build-in types like numerical types and strings.



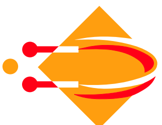
# More Python: Memory handling / reference counting

Memory handling in Python is done by reference counting:

Python counts how many variable names are referencing an object. If this count is zero the object can be removed from memory.

To delete a variable:

```
del <variableName>
```



# More Python: Errors and exceptions

Python handles program errors via exceptions.

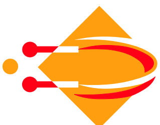
```
try:
    # Code in which an exception can occur
except <ExceptionType>:
    # Catch exception of <ExceptionType> and handle
```

An exception can be thrown by:

```
raise <ExceptionType>, <ExceptionParameter>
```

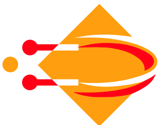
Examples of types of exceptions:

```
IOError
ValueError
ZeroDivisionError
```



# Python exceptions example

```
while 1:
    try:
        x=int(raw_input("Please enter a number: "))
        break
    except ValueError:
        print "Oops! That was not a valid number. Try again..."

print x

raise NameError, "Hi"
```



# Extending Python using C++

Assume we want to be able to call from Python a function written in C++ called `myC++Func`. What to do?

We make a Python module e.g. called `mymodule` by building a DLL called `mymodule.dll`.

The DLL has to do the following work whenever `myC++Func` is called from Python:

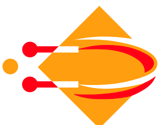
- Convert python arguments to C++ types.
- Invoke `myC++Func` with converted arguments.
- Convert C++ return value to Python types and return to Python.

Python code:

```
import mymodule
mymodule.myC++Func(args)
```

Benefits: Python produce slow programs, C++ is much more efficient.

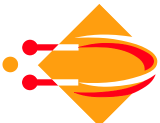
This can be solved by coding everything by hand or by a code generating tool such as SWIG.



# Extending Python with C++: A queue example

myqueue.h:

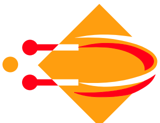
```
#ifndef __MYQUEUE__
#define __MYQUEUE__
#include <list>

class MyQueue {
public:
    MyQueue() {};
    void addElem(int elem);
    int getElem();
    void printQueue();
private:
    std::list<int> m_queue;
};
#endif // __MYQUEUE__
```



# Extending Python with C++: A queue example

myqueue.cpp:

```
#include "myqueue.h"
#include <iostream>

void MyQueue::addElem(int elem) {
   // Add code
}

int MyQueue::getElem() {
    // Add code
    return 0;
}

void MyQueue::printQueue() {
    // Add code
}
```



# Extending Python with C++ using SWIG

myqueue.i:

```
%module myqueue
%{
#include "myqueue.h"
%}

%include "myqueue.h"
```

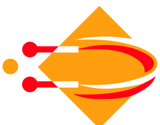
Things to note:

> `%module <modulename>`, `%{ ... %}` copied into wrapper code as is, `%include` what SWIG should parse.

Create a Visual Studio project that generates a DLL file called `_myqueue.dll` using SWIG (see `pixi book` on the course home page).

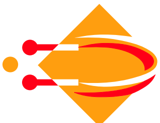
Using the `MyQueue` class in python:

```
import myqueue
q = myqueue.MyQueue()
q.addElem(1)
q.addElem(2)
q.printQueue()
```



# Overview of today

- Code reuse
- Game program design
- Design patterns
- Case study: State manager
- Python: Object references, memory handling, exceptions
- Extending Python with C++ using SWIG



## Reading material

Reading material for this lecture:

- JG: Ch. 3-4 (Excluding Case study 4.4)
- Python extension tutorial
- Optional reading:
  - JG: Ch. Case study 4.4
  - Python extension documentation

Reading material for the next lecture:

- Parts of Panda documentation (to be announced)
- Optional Reading:
  - JG: Ch. 9-10

Next week we will have a guest lecture by Maz Spork (Visionik) on the game development process.

Optional Reading for guest lecture: JG: Ch. 9-10