Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #6

Design patterns for games

Design patterns

- OO programming can generate very large and complex projects
- Design patterns help to organize the project around predefined concepts
  - that can save man-hours and headaches
  - defining solutions for well-established software engineering problems
  - that are accepted as optimal for their efficiency, elegance and robustness

Design patterns

- Design patterns are generic solutions for common problems
- Design patterns book:
  - E. Gamma, R. Helm, R. Johnson and J. Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley
- See also:
  - Bruce Eckel, Thinking in Patterns

    http://www.mindviewinc.com/

Design principle

- Example: stepping through an array in C++

- 1st level of programming: writing code in a specific language

```
int *, [], new, delete, for, ++
// ...
```

Design principle

- 2nd level: specific design
  - the algorithm to solve the problem

```
void someFunction(int* array, int size) {
   for (int i=0; i<size; i++)
        someOtherFunction(array[i]);
}

void someOtherFunction(int x) {
   // do some stuff here
}
```

Design principle

- 3rd level: standard design
  - – the algorithm to solve this kind of problem

```
void someFunction(void** array, int size) {
   for (int i=0; i<size; i++)
        someOtherFunction(array[i]);
}

void someOtherFunction(void* x) {
   // do some stuff here
}
```

Design principle

- 4th level: design pattern
  - the algorithmic pattern to solve similar problems

```
class Container {
public:
   virtual Iterator* getIter() const = 0;
   virtual const int size() const = 0;
};

class Iterator {
public:
   virtual void begin() = 0;
   virtual void next() = 0;
   virtual bool atEnd() = 0;
   virtual Element* getCurrent() const = 0;
};
```

Design principle

- 4^{th} level: design pattern
  - the algorithmic pattern to solve similar problems

```
void someFunction(Container* c) {
   Iterator* i = c->getIter();
   i->begin();
   while (!i->atEnd()) {
        someOtherFunction(i->getCurrent());
        i->next();
   }
}
void someOtherFunction(Element* x) {
   // do some stuff here
}
```

  - Any implementation of derived classes of Container can then be parsed

Design principle

- To improve consistency
  - to avoid

```
class Warrior {
   public:
         std:string toString() const;
}
```

```
class Inventory {
   public:
         char* toString() const;
}
```

```
class Team {
   public:
         void getStr(std:string& s) const;
}
```

Design principle

- Better design

```
class Printable {
   public:
        virtual std:string toString() const = 0;
}
```

```
class Warrior : public Printable {
   public:
        std:string toString() const;
}
```

  – do the same for any other classes that want a "toString" method

Design patterns

- Higher level of functionality programming
  - higher abstraction than simple class or even library like STL
- Define subsystems composed of several classes ready and tuned for specific use
- Require experience to point out pieces of code corresponding to well-known patterns
  - some especially designed for game engine (real-time interactive) programming issues

Design patterns

- Three categories of DP
  - Creational: how is an object created
    - Singleton, Factory ...
  - Structural: designing objects to satisfy project constraint
    - Façade, Composite ...
  - Behavioral: object that handle particular types of actions within a program
    - Strategy, Spatial Index, Listener ...

Singleton

- Global object for which only one instance exists in the whole program
  – to model a single global object visible from several classes and scopes
- Two basics solutions
  – passing the object to all calls
    - very inefficient, and hard to read
  – place it in a source file as extern object
    - degrading readability, not OO spirit of encapsulation and binding, does not ensure unity if pointer

Singleton

- Using a class with one public method to request an instance of the singleton
  - all instances point at the same object
- Create the singleton at the first call, then return pointer to it in subsequent calls
  - constructor must be protected to prevent calling from outside the class
  - the public function is often named Instance()

Singleton

- 1st version

```
class Singleton {
   public:
        static Singleton* Instance(); // the public function
   protected:
        Singleton(){}; // default empty constructor
   private:
        static Singleton* _instance; // the object
};

Singleton* Singleton::_instance = NULL; // initialization

Singleton* Singleton::Instance () {
   if (_instance == NULL) _instance = new Singleton();
   return _instance;
}
```

Singleton

```
#include "Singleton.h"

int main () {
   Singleton * s1 = Singleton::Instance();
   Singleton * s2 = Singleton::Instance();
   // s1 and s2 point to the same object
   return 0;
}
```

- Life time of _instance is life time of program

Singleton

- 2nd version

```
class Singleton {
   public:
        static Singleton* Instance(); // the public function
        static void create(); // creation of _instance
        static void destroy(); // destruction of _instance
   protected:
        Singleton(); // default empty constructor
   private:
        static Singleton* _instance; // the object
};
```

Singleton

```
Singleton* Singleton::_instance = NULL; // initialization

Singleton::Singleton() {}

void Singleton::create() {
   if (_instance == NULL) _instance = new Singleton();
}

void Singleton::destroy() {
   delete _instance;
   _instance = NULL;
}

Singleton* Singleton::Instance () {
   return _instance;
}
```

Singleton

- Life time of _instance is defined by create and destroy
- create() has to be called before Instance() to ensure a non-null pointer
- create can be called with parameters to be used in the constructor

```
#include "Singleton.h"

int main () {
   Singleton::create();
   Singleton * s1 = Singleton::Instance();
   Singleton * s2 = Singleton::Instance();
   // s1 and s2 point to the same object
   Singleton::destroy();
   return 0;
}
```

Singleton

- Using Singletons in game engines
  - Game engine itself
  - Graphics renderer (if only one)
  - Resource (mesh, texture ...) managers
  - Device controller, input manager
  - And more game engine components
- An extension of Singleton: Object pool
  - Create multiple instances in a controllable fashion
  - *E.g.* to restrict the number of players on a networked game server

Strategy

- To create objects with behaviors that can be changed dynamically
  – using a family of interchangeable algorithms
- Example of enemy AI with single update function handling several strategies
- Basic solutions
  – switch statement with several functions
    - not elegant, difficult to maintain
  – derive the object with different strategies
    - too complex in practical terms

Strategy

- Strategy DP separates the class definition from one of its member algorithm
  - so they can be changed at run-time
- Implementation involves two classes
  - the abstract base class providing the strategy algorithm
  - the context class defining where the strategy should be applied and executing it

Strategy

```
class Enemy {
   public:
        Enemy(AIstrategy *);
        void update();
        void setStrategy(AIstrategy *);
   private:
        int amno;
        AIstrategy * currentStrategy;
};

Enemy::Enemy(AIstrategy * newStrategy) {
    currentStrategy = newStrategy;
}

void Enemy::update() { currentStrategy->updateAI(amno); }

void Enemy::setStrategy(AIstrategy * newStrategy) {
   if (currentStrategy != NULL) delete currentStrategy;
   currentStrategy = newStrategy;
}
```

Strategy

```
class AIstrategy {
   public:
         virtual void updateAI(int) = 0;
};
```

```
class AIAttack : public AIStrategy
{
   public:
         void updateAI(int);
};
```

```
class AIWait : public AIStrategy
{
   public:
         void updateAI(int);
};
```

```
Enemy* badGuy = new Enemy(new AIAttack());
badGuy->update();
badGuy->setStrategy(new AIWait());
badGuy->update();
// ...
```

Factory

- Games need to create and dispose of objects continuously
  - creation on demand and destruction at the end of their life cycle
  - that are spread through many classes
  - inconsistency problems in allocating memory
- The factory DP centralizes the object creation and destruction
  - universal, rock-solid method for handling objects
  - two types
    - regular factory to produce regular classes
    - abstract factory to produce abstract classes

Regular factory

```
class MovingEntity {};

class Player : public MovingEntity {};
class Enemy : public MovingEntity {};
class NPC : public MovingEntity {};

class FactoryMovingEntity {
  public:
      Player * createPlayer() {return new Player();};
      Enemy * createEnemy() {return new Enemy();};
      NPC * createNPC() {return new NPC();};
};
```

Regular factory

- As Player, Enemy and NPC are decoupled, the factory needs a method for each type
- Usage

```
FactoryMovingEntity FE;
Player * p = FE.createPlayer();
Enemy * e1 = FE.createEnemy();
Enemy * e2 = FE.createEnemy();
NPC * n = FE.createNPC();
```

- Owner of newly created object is the caller
  - If the factory stores the pointers, *e.g.* in a `vector<MovingEntity *>` then the factory is the owner

Regular factory with ID

```
class MovingEntity {};

class Player : public MovingEntity {};
class Enemy : public MovingEntity {};
class NPC : public MovingEntity {};

typedef int MovingEntityID; // or enum
#define PLAYER 0
#define ENEMY 1
#define NPC 2

class FactoryMovingEntity {
   public:
        MovingEntity * createMovingEntity(MovingEntityID id) {
                switch (id) {
                        case PLAYER : return new Player(); break;
                        case ENEMY  : return new Enemy(); break;
                        case NPC    : return new NPC();
                }
        };
};
```

Regular factory with ID

- As Player, Enemy and NPC inherit from MovingEntity, the factory centralizes the entities creation
- Usage

```
FactoryMovingEntity AFE;
Player * p = (Player *) AFE.createMovingEntity(PLAYER);
Enemy * e1 = (Enemy *) AFE.createMovingEntity(ENEMY);
MovingEntity * e2 = AFE.createMovingEntity(ENEMY);
NPC * n = (NPC *) AFE.createMovingEntity(NPC);
```

Regular factory with registration

- To avoid the factory to know the ID of each class, registration is used to create a map between class name (key) and loaders (value)
- Usage

```
FactoryMovingEntity AFE;

PlayerLoader ploader; AFE.registerLoader(&ploader);
EnemyLoader eloader; AFE.registerLoader(&eloader);

Player * p = (Player *) AFE.createMovingEntity("PLAYER");
Enemy * e1 = (Enemy *) AFE.createMovingEntity("ENEMY");
```

Regular factory with registration

```
class PlayerLoader : public MovingEntityLoader {
   public:
        string getNameID() const {return "PLAYER";}
        MovingEntity * loadMovingEntity() const {return new Player();}
};

class FactoryMovingEntity {
   public:
        MovingEntity * createMovingEntity(string nameID) {
                map<string, MovingEntityLoader *>::iterator it =
   _loaders.find(nameID);
                if (it != _loaders.end())
                        return it->second->loadMovingEntity();
                return null;
        };
        void registerLoader(MovingEntityLoader * loader);
        void unregisterLoader(string nameID);
   protected:
        map<string, MovingEntityLoader *> _loaders;
};
```

Universiteit Utrecht

Abstract factory

- Single interface for creating different "products" without specifying the concrete classes

```
class EnemyFactory {
public:
   virtual Enemy* createEnemy() = 0;
   virtual Weapon* createWeapon() = 0;
};
```

Abstract factory

```
class EasyEnemy : public Enemy {
   // ...
};
class ToughEnemy : public Enemy {
   // ...
};
class LightWeapon : public Weapon {
   // ...
};
class HeavyWeapon : public Weapon {
   // ...
};
```

Abstract factory

```
class EasyEnemyFactory : public EnemyFactory {
public:
   Enemy* createEnemy() { return new EasyEnemy(); }
   Weapon* createWeapon() { return new LightWeapon(); }
};

class ToughEnemyFactory : public EnemyFactory {
public:
   Enemy* createEnemy() { return new ToughEnemy(); }
   Weapon* createWeapon() { return new HeavyWeapon(); }
};
```

Abstract factory

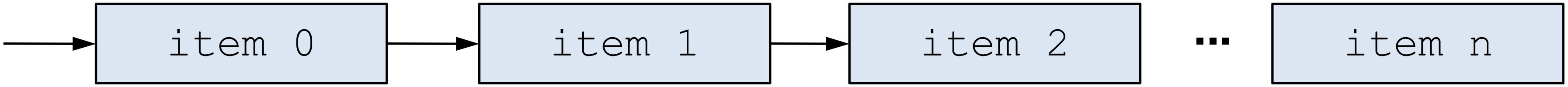
```
int main() {
   EnemyFactory* pEFactory;
   if (easyPlaying)
        pEFactory = new EasyEnemyFactory();
   else
        pEFactory = new ToughEnemyFactory();
   spawnEnemy(pEFactory);
   // ...
}

void spawnEnemy(EnemyFactory* eFac) {
   Enemy* e = eFac->createEnemy();
   Weapon* w = eFac->createWeapon();
   e->attackPlayerWithWeapon(w);
   // ...
}
```

Spatial index

- Nowadays games require fast 3D tests
  - to manipulate game world
- Spatial index DP allows to perform queries on large 3D environment
  - Examples
    - Is there 3D objects closer than N units?
    - How many primitives in that X,Y,Z direction?
  - offer almost constant cost (independent of input)
  - black box that speeds up geometric tests

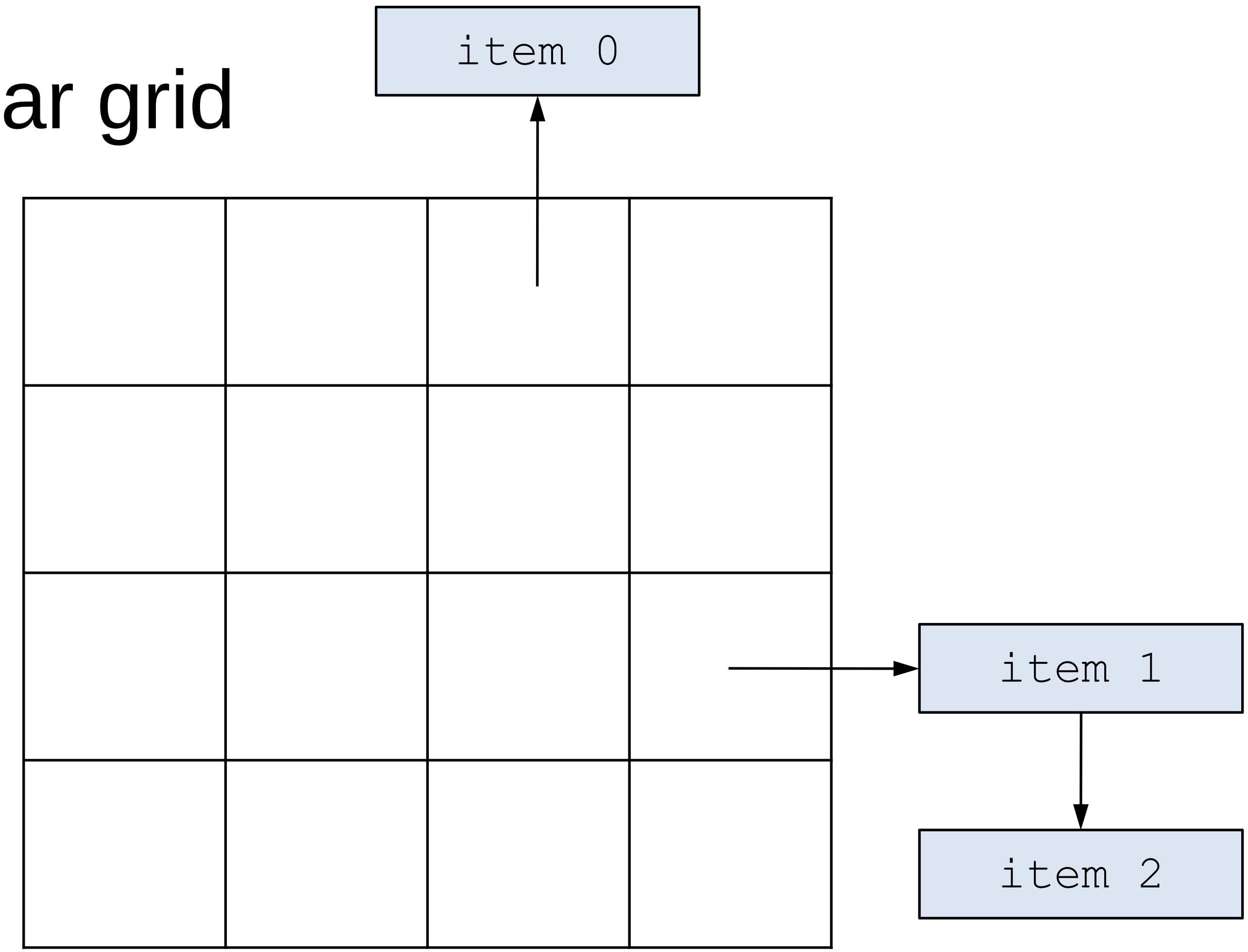
Spatial index

- As a list



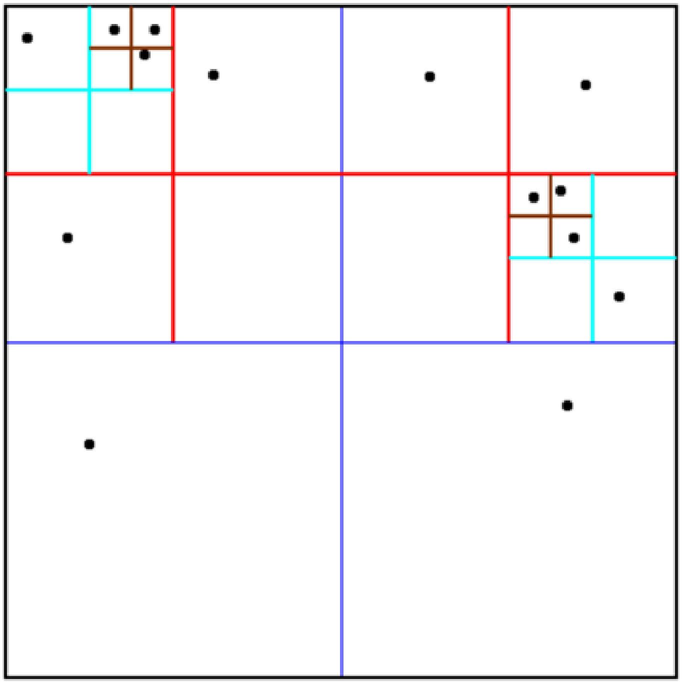
Spatial index

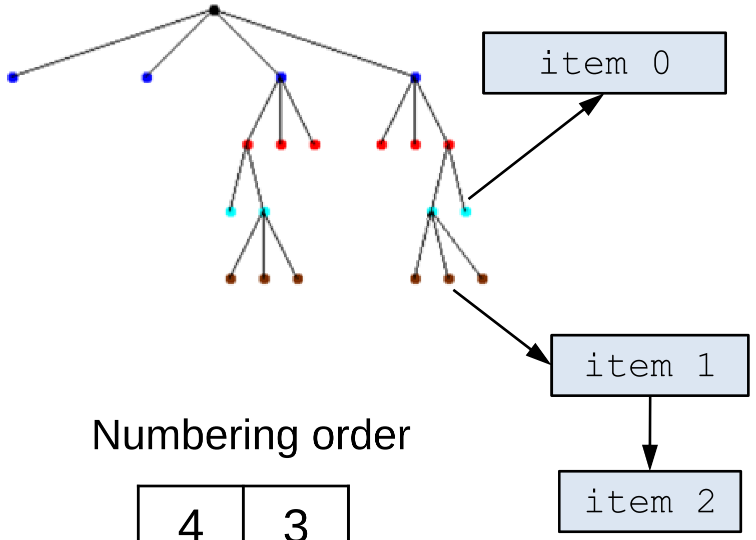
- As a list
- As a regular grid



Spatial index

- As a list
- As a regular grid
- As quadtree / octree





Numbering order

4	3
1	2

Façade

- Interface to a collection of related systems or classes
  - essentially acts as a wrapper
- Using the façade DP during the development process
  - the “under construction” façade
    - keep access to features of the system during construction phase
    - more efficient usage of time during development
  - the “refactoring” façade
    - setup a temporary façade for the old implementation
    - as the new one comes online, pipe it through the façade

Listener

- To notify related objects (listeners) that a change occurred in an object (subject)
  - (critical) update, destruction of the object *etc.*
- By using inheritance, listener/subject behavior can easily be incorporated in your class
- Listener DP is also referred to as ‘Notifier’ or ‘Observer’
- Listener examples
  - A rocket launcher will influence different visual effect objects (light, sound, particles…)
  - Use listener pattern for notification to listeners such as input events and time to update or draw entities

Listener

```
// Basic Listener class
class Listener {
   public:
         virtual ~Listener();
         virtual void update() = 0;
         void setSubject(Subject* s) {_pSubject = s;} // optional
   protected:
         Subject* _pSubject; // optional
};
```

```
// Basic subject class
class Subject {
   public:
         Subject() {};
         virtual ~Subject();
         virtual void addListener(Listener * l);
         virtual void updateListeners();
   protected:
         vector<Listener *> _listeners;
};
```

Listener

```
Subject::~Subject() {
   for (int i = 0; i < _listeners.size(); i++) {// optional
        _listeners[i]->setSubject(NULL);
   }
};

void Subject::addListener(Listener * l) {
   _listeners.push_back(l);
};

void Subject::updateListeners() {
   for (int i = 0; i < _listeners.size(); i++)
        _listeners[i]->update();
};
```

Listener

- Example

```
class Player : public Subject {
   public:
        Player();
        ~Player();
};

class Enemy : public Listener {
   public:
        ~Enemy();
        void update();
};
```

```
Subject * p = new Player();
Listener * e = new Enemy();
e->setSubject(p); // optional
p->addListener(e);
// do stuff with p and e
p->updateListeners(); // call Enemy::update() on instance e
```

Composite

- Games need to hold heterogeneous collections of data
  - ex: a game level can have sub-levels (that can have sub-levels themselves), items (like enemies and quest items) *etc.*
- Composite DP creates part-whole heterogeneous hierarchies where primitives and other composite objects are accessed using a standard interface

Composite

- Example

```
class Component
{
  public:
    virtual void traverse() = 0;
};
```

```
class Primitive: public Component {
   int value;
public:
   Primitive(int val) : value(val) { }
   void traverse() { cout << value << "  "; }
};
```

Composite

```
class Composite: public Component {
   vector<Component *> children_;
   int value;
public:
   Composite(int val) : value(val) { }

   void add (Component * c) { children_.push_back(c); }

   void traverse() {
        cout << value << "  ";
        for (int i = 0; i < children_.size(); i++)
          children[i]->traverse();
   }
};
```

Composite

- Example: 2D heterogeneous data structure

```
class Row: public Composite {
 public:
   Row(int val) : Composite(val){}
   void traverse() {
     cout << "Row";
     Composite::traverse();
    }
};

class Column: public Composite {
 public:
   Column(int val): Composite(val){}
   void traverse() {
         cout << "Col";
         Composite::traverse();
    }
};
```

Composite

- Example: 2D heterogeneous data structure

```
Row first(1);                                      // Row1
Column second(2);                                  //   |
Column third(3);                                   //   +-- Col2
Row fourth(4);                                     //   |     |
Row fifth(5);                                      //   |     +-- 7
first.add(&second);                                //   +-- Col3
first.add(&third);                                 //   |     |
third.add(&fourth);                                //   |     +-- Row4
third.add(&fifth);                                 //   |     |     |
first.add(&Primitive(6));                          //   |     |     +-- 9
second.add(&Primitive(7));                         //   |     +-- Row5
third.add(&Primitive(8));                          //   |     |     |
fourth.add(&Primitive(9));                         //   |     |     +-- 10
fifth.add(&Primitive(10));                         //   |     +-- 8
first.traverse();                                  //   +-- 6
// output: Row1  Col2  7  Col3  Row4  9  Row5  10  8  6
```

Composite

- Example of level design with
  - Composite => Level and Dungeon
  - Primitive => QuestItem and Enemy

  - int value => string name
  - void traverse() => int numberOfEnemies() and string getQuestItems()

Composite

```
void LordOfTheRings () {
    Level MiddleEarth ("Middle Earth");
    Dungeon TheShire ("The Shire");
    Dungeon Lothlorien ("Lothlorien forest");
    Dungeon Isengard ("Isengard");
    Dungeon Mordor ("Mordor");
    MiddleEarth->add(&TheShire);
    MiddleEarth->add(&Lothlorien);
    MiddleEarth->add(&Isengard);
    MiddleEarth->add(&Mordor);
    QuestItem theRing ("the One Ring");
    TheShire->add(&theRing);
    QuestItem Lembas ("Lembas");
    QuestItem EarendilLight ("The Light of Earendil");
    Lothlorien->add(&Lembas);
    Lothlorien->add(&EarendilLight);
    Enemy Saruman ("Saruman");
    QuestItem Palantir ("Palantir");
    Saruman->add(&Palantir);
    Isengard->add(&Saruman);
    Enemy Sauron ("Sauron");
    Enemy SauronArmy ("Army of Sauron");
    Mordor->add(&Sauron);
    Mordor->add(&SauronArmy);
    cout << "Number of enemies: " << MiddleEarth.NumberOfEnemies();
    cout << "List of quest items: " << MiddleEarth.getQuestItems();
}
```

More DPs

- Adapter
  - Converts existing interface into a new interface compatible with client demands
- Flyweight
  - to combine classes having many common members
- Client-server
  - to communicate data between classes
- Events (listener with parameterized update)
  - to dynamically send messages to registered classes
- Combining DPs
  - Factories are almost always singleton, ...

End of lecture #6

Next lecture

HID and Error Handling