Game Programming in C++

Arjan Egges

Lecture #6: Design patterns

Overview

- What are design patterns?
- Singleton
- Abstract Factory
- Façade
- Observer
- Strategy
- Adapter

What are design patterns?

- Design patterns are generic solutions for common problems.
- Design patterns book:

  Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, hardcover, 395 pages, Addison-Wesley. ISBN 0-201-63361-2.
- See also:

  Bruce Eckel. Thinking in Patterns.
  http://www.mindviewinc.com/

Example

- 1st level: writing code in a particular language to do a particular thing
  - Coding the process of stepping through an array in C++

```
int*, [], new, delete, for, ++
operator…
```

Example

- 2nd level: specific design, or: the solution we came up with to solve this 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
}
```

Example

- 3rd level: standard design, or: a way 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
}
```

Example

- 4th level: design pattern: how to solve an entire class of similar problems

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

Example

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

- Iterator allows for separation of datastructure and the algorithms used

Example

```
void someFunction(Container* c) {
   Iterator* i = c->getIter();
   i->begin();
   while (!i->atEnd()) {
      someOtherFunction(i->getCurrent());
      i->next();
}

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

Design principles

- **Make common things easy, and rare things possible**
  - Adding a special feature to Container::size():

```
int size(const bool arrayFromOne) const;
```

  - A better way to do this:

```
int size(const bool arrayFromOne=false) const;
```

Design principles

- **Consistency**
  - To avoid:

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

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

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

Design principles

  - A better solution in this case:

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

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

And do the same for any other classes that need a “toString” method

Design principles

Other suggestions to think about…

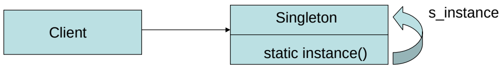
- *Subtraction*: a design is finished when you cannot take anything else away
- *Simplicity vs. Generality*
- *Independence*: express independent ideas independently
- *Singularity*: avoid duplication of structure and logic

Design patterns

Three categories:

- **Creational**: how is an object created (Singleton, Factory, etc.)
- **Structural**: designing objects to satisfy certain project constraints (Façade, …)
- **Behavioral**: object that handle particular types of actions within a program (Iterator, Observer, Visitor, etc.)

Singleton



Singleton

```
class GlobalClass {
public:
   static GlobalClass* instance();
   static void create();
   static void destroy();
private:
   static GlobalClass* s_instance;
   GlobalClass();
};
```

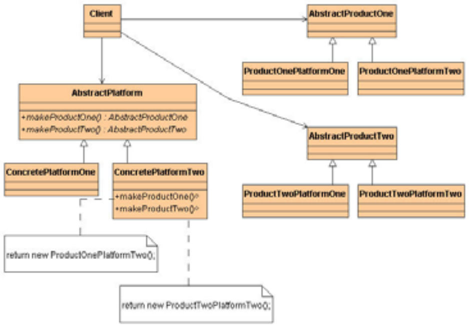
Singleton

```
GlobalClass* GlobalClass::s_instance = NULL;

GlobalClass* GlobalClass::instance() {
   return s_instance;
}

void GlobalClass::create() {
   if (!s_instance)
      s_instance = new GlobalClass();
}

void GlobalClass::destroy() {
   delete s_instance;
   s_instance = NULL;
}
```

Singleton

- Using Singletons in game engines:
  - GraphicsRenderer (normally there is only one)
  - Script reading interface
  - File manager
- An extension of Singleton: Object pool
  - Create multiple instances in a controllable fashion
    - E.g. restrict the amount of connections to a database (players on a networked game server)

Abstract Factory



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();
   Weapon* createWeapon();
};

Enemy* EasyEnemyFactory::createEnemy() {
   return new EasyEnemy();
}

Weapon* EasyEnemyFactory::createWeapon() {
   return new LightWeapon();
}
```

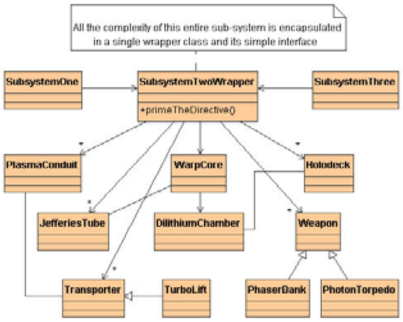
Abstract Factory

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

Enemy* ToughEnemyFactory::createEnemy() {
   return new ToughEnemy();
}

Weapon* ToughEnemyFactory::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);
   …
}
```

Façade



Façade

- Interface to a collection of (loosely) related systems or classes
  - Essentially acts as a “wrapper”

Example: displaying a text overlay:

```
Texture* pTexture =
GfxTextureMgr::GetTexture(“CoolTexture.tif”);

Font* pFont = FontMgr::GetFont(“Roman.fnt”);

Overlay2D pMsgBox =
OverlayManager::CreateOverlay(pTexture, pFont,
“Hello World”);

GfxRenderer::AddScreenElement(pMsgBox);
```

Façade

- Encapsulating the system in a higher-level “wrapper” class:

```
// GraphicsInterface acts as façade for hiding
// the detailed operations involved in creating
// a messagebox.
GraphicsInterface::displayMsgBox(“CoolTexture”,
“Roman.fnt”, “Hello World”);
```

Façade

Using the façade during the development process:

- The “Under Construction” façade
  - Keep access to the features of your system during the construction phase
  - More efficient usage of time during project development
- The “Refactoring” façade
  - Setup a temporary façade for the old implementation while working on a new one
  - As new implementation comes online → pipe it through the façade

Observer

- Example: references to objects that go out of scope

```
class ObjectA {
public:
   void doSomething();
};

class ObjectB {
public:
   ObjectB(ObjectA* pA);
   void update();
private:
   ObjectA* pA_;
};
```

Observer

```
ObjectB::ObjectB(ObjectA* pA) : pA_(pA) {
}

void ObjectB::update() {
   if (pA_ != NULL)
      pA_->doSomething();
}
```

Observer

```
// Create an ObjectA
ObjectA* pA = new ObjectA();

// Create an ObjectP, passing pA to it
ObjectB* pB = new ObjectB(pA);

// Update B
pB->update(); // works fine

// Destroy A
delete pA;

// B's update will fail…
pB->update(); // uh-oh…
```

Observer

- Adding destruction notification

```
class ObjectB {
public:
   ObjectB(ObjectA* pA);
   void update();
   void NotifyObjectADestruction();
private:
   ObjectA* pA_;
};
```

```
void ObjectB::NotifyObjectADestruction() {
   // set pointer to NULL
   pA_ = NULL;
}
```

Observer

- And in ObjectA

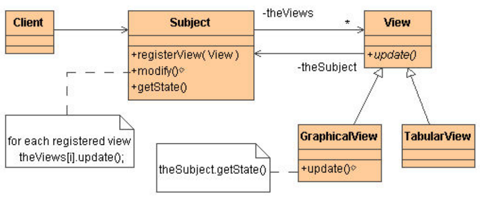
```
class ObjectA {
public:
   // Everything else here, and…
   ~ObjectA();
   void setOwner(ObjectB* pOwner);
private:
   ObjectB* pOwner_;
};
```

```
ObjectA::~ObjectA() {
   pOwner_->NotifyObjectADestruction();
}
```

Observer

- Generalizing this approach:
  - ObjectB also needs to be notified of changes other than ObjectA's destruction
  - Multiple objects need to be informed of these changes, not only ObjectB
- Solution: the *Observer* pattern

Observer



Observer

```
// Basic observer class
class Observer {
public:
   virtual ~Observer();
   virtual void update() = 0;
   void setSubject(Subject* s);
protected:
   Subject* pSubject_;
};
```

```
void Observer::setSubject(Subject* s) {
   pSubject_ = s;
}
```

Observer

```
// Basic subject class
class Subject {
public:
   Subject();
   virtual ~Subject();
   virtual void addObserver(Observer* o);
   virtual void updateObservers();
protected:
   Observer** observers_;
   int size_;
};
```

Observer

```
Subject::Subject() {
   observers_ = new Observer*[MAX_OBS];
   size_ = 0;
}

Subject::~Subject() {
   for (int i=0; i<size_; i++) {
      observers_[i]->setSubject(NULL);
   }
   delete [] observers_;
}
```

Observer

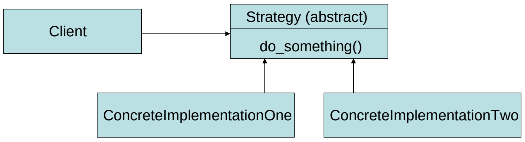
```
void Subject::addObserver(Observer* o) {
   if (size_ < MAX_OBS) {
      observers_[size_] = o;
      size_++;
   }
}

void Subject::updateObservers() {
   for (int i=0; i<size_; i++)
      observers_[i]->update();
}
```

Observer

- By using inheritance, observer/subject behaviour can easily be incorporated in your class
- Observer pattern is sometimes also refered to as ‘Notifier’
- Observer example in the book:
  - A rocket launcher will influence different visual effect objects (light, sound, particles…)
  - Use observer pattern for notification of subjects

Strategy

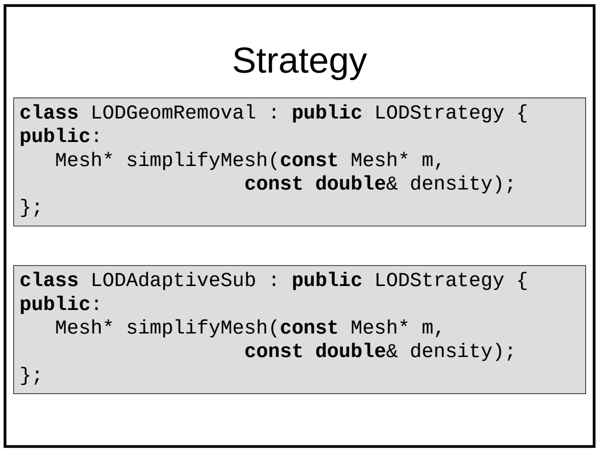
- Using a family of interchangeable algorithms
- Strategy lets the algorithms vary independent of the clients using them



Strategy

- Example: A level-of-detail management system
- Mesh simplification can be done with different algorithms

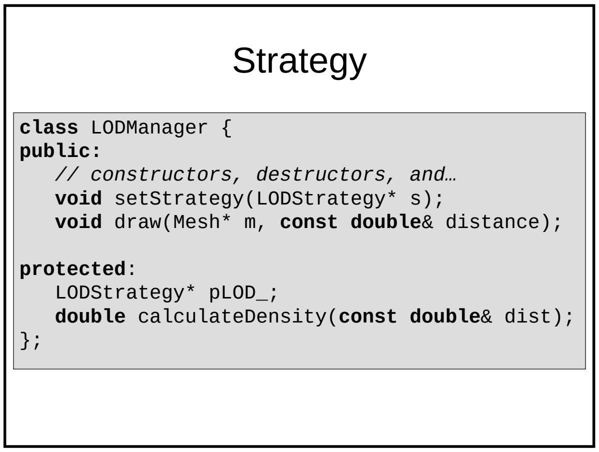
```
class LODStrategy {
public:
   virtual Mesh* simplifyMesh(const Mesh* m,
                    const double& density) = 0;
};
```



Strategy

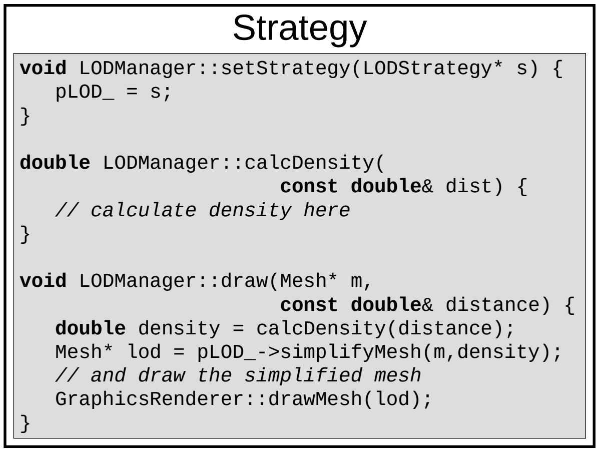
```
class LODGeomRemoval : public LODStrategy {
public:
   Mesh* simplifyMesh(const Mesh* m,
                  const double& density);
};
```

```
class LODAdaptiveSub : public LODStrategy {
public:
   Mesh* simplifyMesh(const Mesh* m,
                  const double& density);
};
```



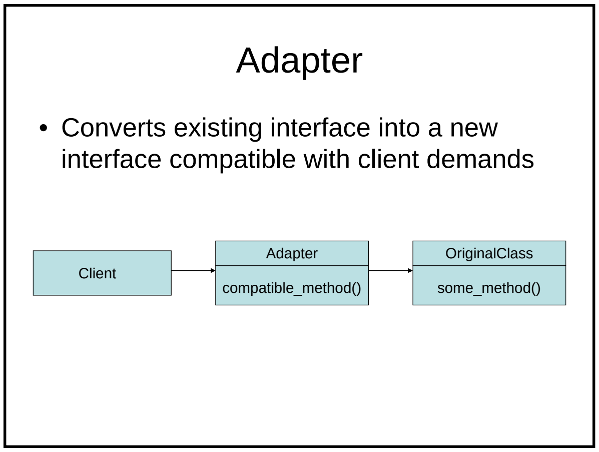
Strategy

```
class LODManager {
public:
   // constructors, destructors, and…
   void setStrategy(LODStrategy* s);
   void draw(Mesh* m, const double& distance);

protected:
   LODStrategy* pLOD_;
   double calculateDensity(const double& dist);
};
```



Strategy

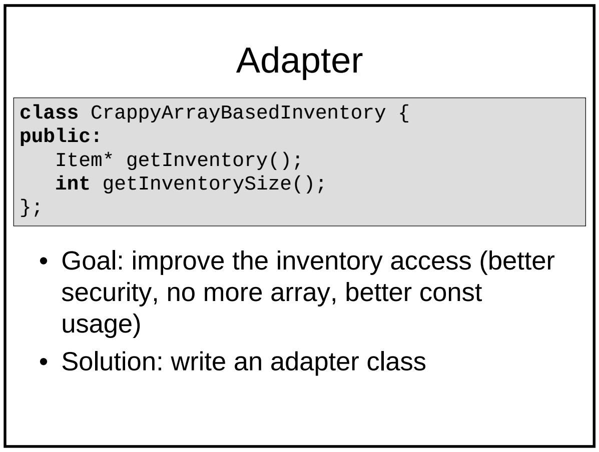
```
void LODManager::setStrategy(LODStrategy* s) {
   pLOD_ = s;
}

double LODManager::calcDensity(
                      const double& dist) {
   // calculate density here
}

void LODManager::draw(Mesh* m,
                      const double& distance) {
   double density = calcDensity(distance);
   Mesh* lod = pLOD_->simplifyMesh(m,density);
   // and draw the simplified mesh
   GraphicsRenderer::drawMesh(lod);
}
```



Adapter

- Converts existing interface into a new interface compatible with client demands

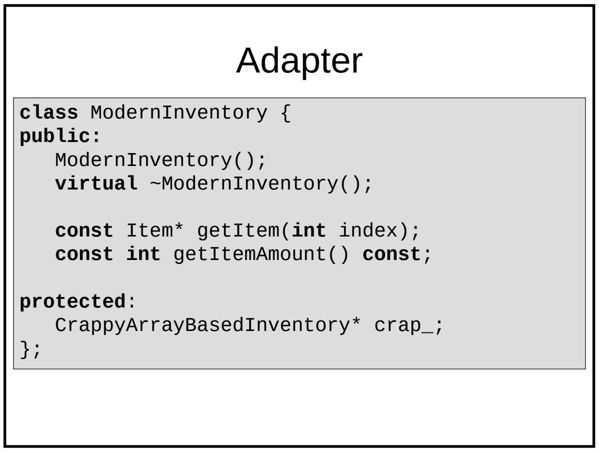
Client → Adapter (compatible_method()) → OriginalClass (some_method())



Adapter

```
class CrappyArrayBasedInventory {
public:
   Item* getInventory();
   int getInventorySize();
};
```

- Goal: improve the inventory access (better security, no more array, better const usage)
- Solution: write an adapter class



Adapter

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

   const Item* getItem(int index);
   const int getItemAmount() const;

protected:
   CrappyArrayBasedInventory* crap_;
};
```

Adapter

```
const Item* ModernInventory::getItem(int index) {
   if (index > crap_->getInventorySize())
      return NULL;
   else
      return crap_->getInventory()[index];
}

const int ModernInventory::getItemAmount() const {
    return crap_->getInventorySize();
}
```

Summary

- Design Patterns
- Singleton, Factory, Adapter, etc.
- How they are used in games

Next course:

- Game Engines