Object Oriented Design

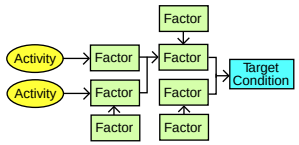
Design - the creative process of transforming the problem into a solution.

Object Oriented Design - Determining a set of components (objects, classes, etc.) and interfaces between those components that will satisfy the requirements and solve the problem.

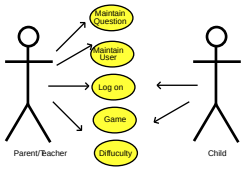
Object Oriented Design

Outputs from the OO Analysis Workflow
become inputs for the OO Design Workflow

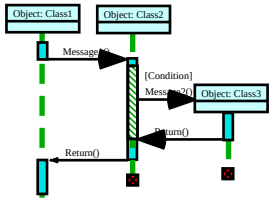
- **1. The Conceptual Model**



- **2. Use Cases**



- **3. System Sequence Diagram**



- **4. User Interface documentation**

- **5. Relational Data model**

Object Oriented Thinking

Putting the pieces together...

1. Understand the problem.

2. Figure out a solution to the problem.

3. Outline what objects would be needed to solve the problem.

4. Plan how these objects should interact to solve the problem.

5. Create a design for how these objects will be simulated in a computer program.

Object Oriented Thinking

Model - a set of UML diagrams that represent one or more aspects of the software product being developed. UML is used to represent (model) the software product being developed.

Can you grasp a "picture" of the entire design of your software at one time?

Psychologist George A. Miller

Miller's Law

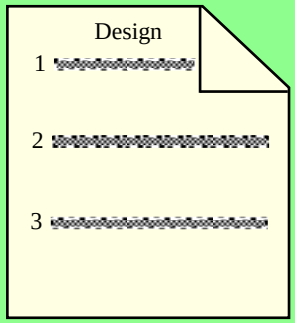
At any one time humans are capable of concentrating on only approximately seven chunks of information.

Object Oriented Thinking

Iteration and Incrementation using...

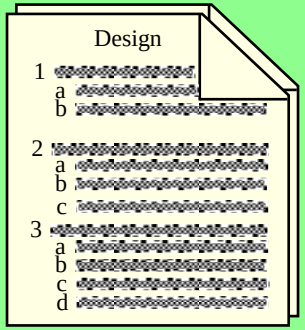
Step-wise Refinement

The process of creating the design by starting at a high level of abstraction and working down to more detail.



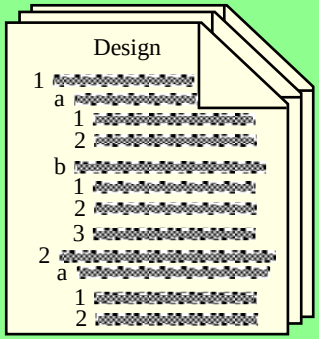
Iteration 1

List all possible objects (classes) that might be needed. Create a basic UML class diagram.



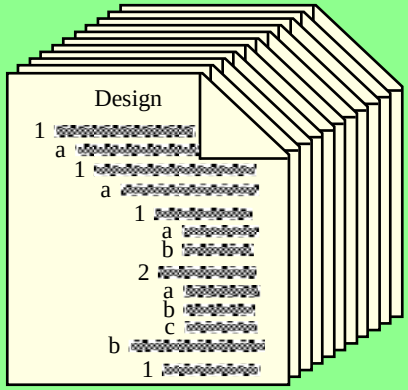
Iteration 2

For each class list what it should do, i.e. services provided, behaviors performed, states to exist in, etc. Add this info to the UML diagram.



Iteration 3

For each service/behavior in each class list the functions that might be needed to provide that service or perform that behavior and the variables (attributes) that might be needed for this. Add this information to the UML diagram.



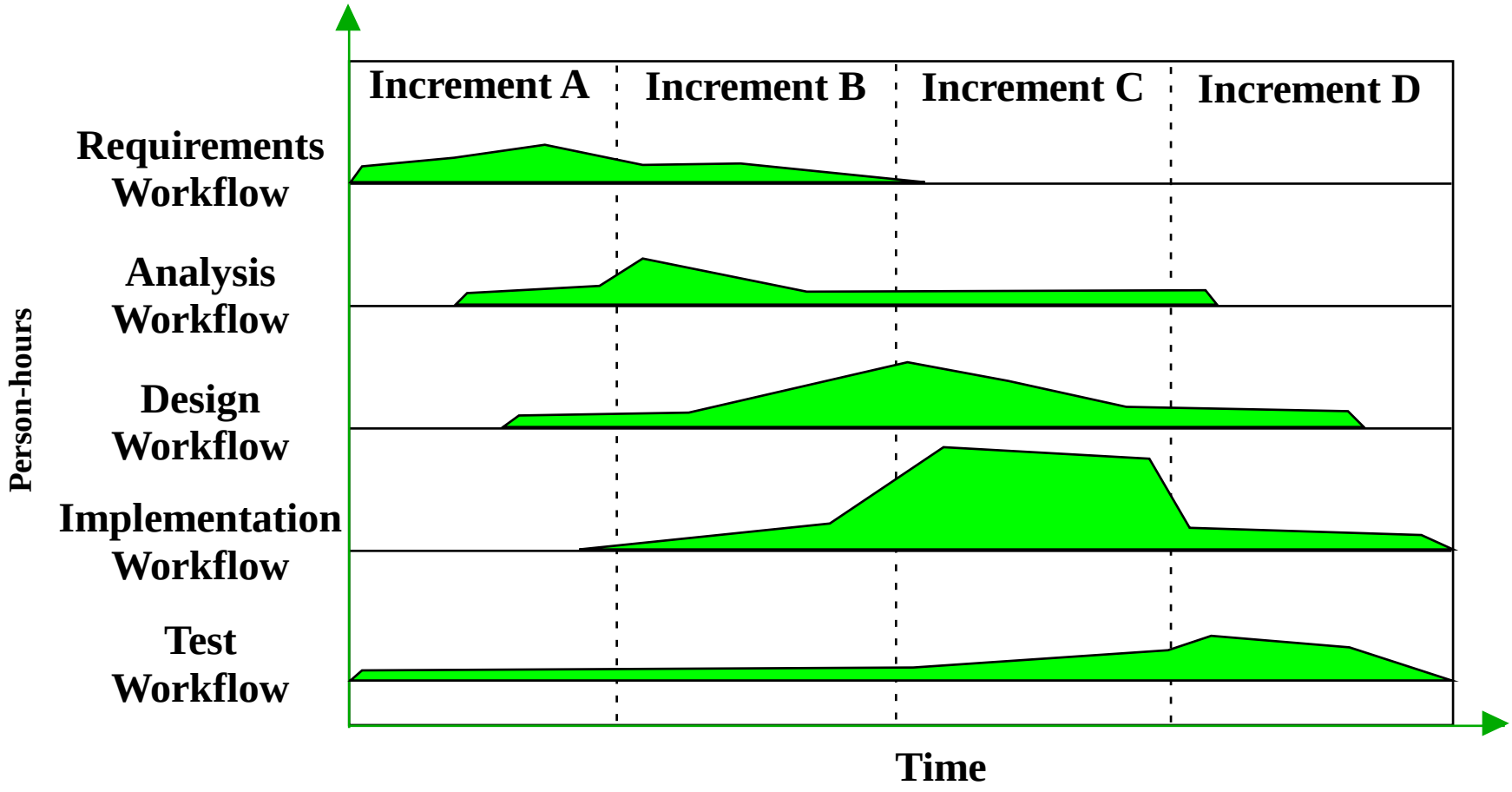
Iteration 4

For each function in each service/behavior in each class work out the details of the algorithm in pseudocode to perform this action.

And so on...

Object Oriented Thinking

Iteration and Incrementation



From Schach, Stephen R. *Object-Oriented and Classical Software Engineering*, 8th ed.

Design Principles

De-sign, yeah!

Design Principles

Design Principles

- ***Identify the aspects of your application that vary and separate them from what stays the same.***

Remember this?

Change

Remember Doug?

Remember Harry and Blanche?

We have a great idea. How about adding a bark recognizer to the dog door so it knows Fido's bark?

They are never satisfied!

Design your software so that when changes happen they are easy to implement because the places where changes need to be made are isolated into separate classes.

Design Principles

Design Principles

- *Identify the aspects of your application that vary and separate them from what stays the same.*
- *Program to an interface, not an implementation.*

Suppose you have a class...

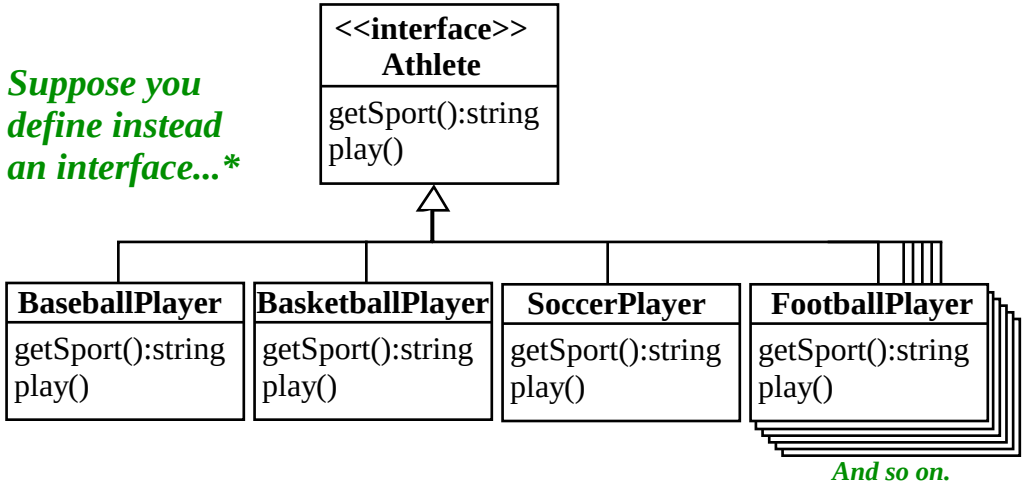
Athlete
getSport():string
play()

which implements...

```
void Athlete:play()
{
   if(sport==baseball)
      // Play baseball
   else if(sport==basketball)
      // Play basketball
   else if(sport==soccer)
      // Play soccer
   else if(sport==football)
      // Play football
```

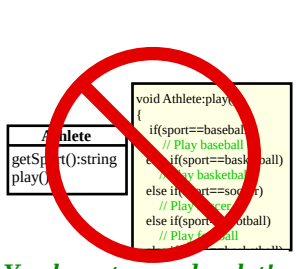
This is NOT good!!

*Suppose you define instead an interface...**



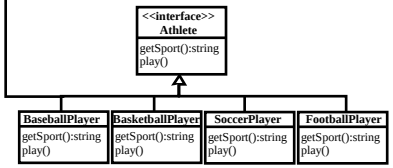
And so on.

What if you need to add a new player type?



You have to recode a lot!

QuidditchPlayer
getSport():string
play()



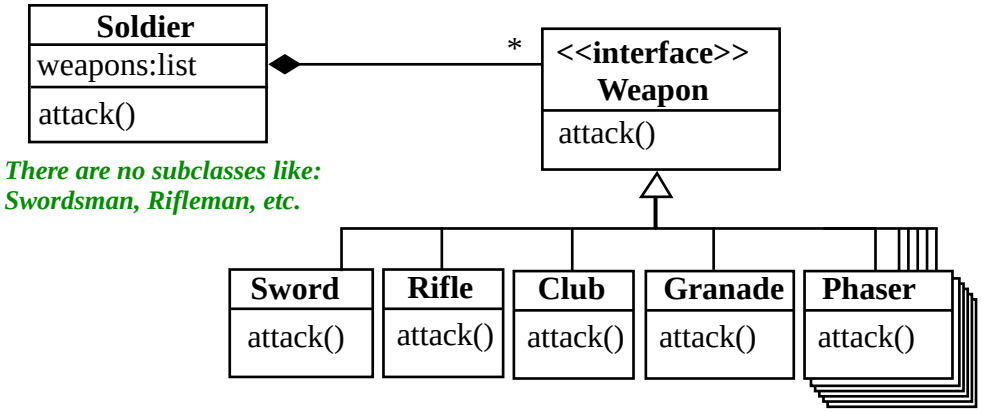
No problemo!!!

** In Java this is an Interface object. In C++ this is a parent class defining the "interface".*

Design Principles

Design Principles

- *Identify the aspects of your application that vary and separate them from what stays the same.*
- *Program to an interface, not an implementation.*
- *Favor composition over inheritance.*



There are no subclasses like: Swordsman, Rifleman, etc.

And so on.

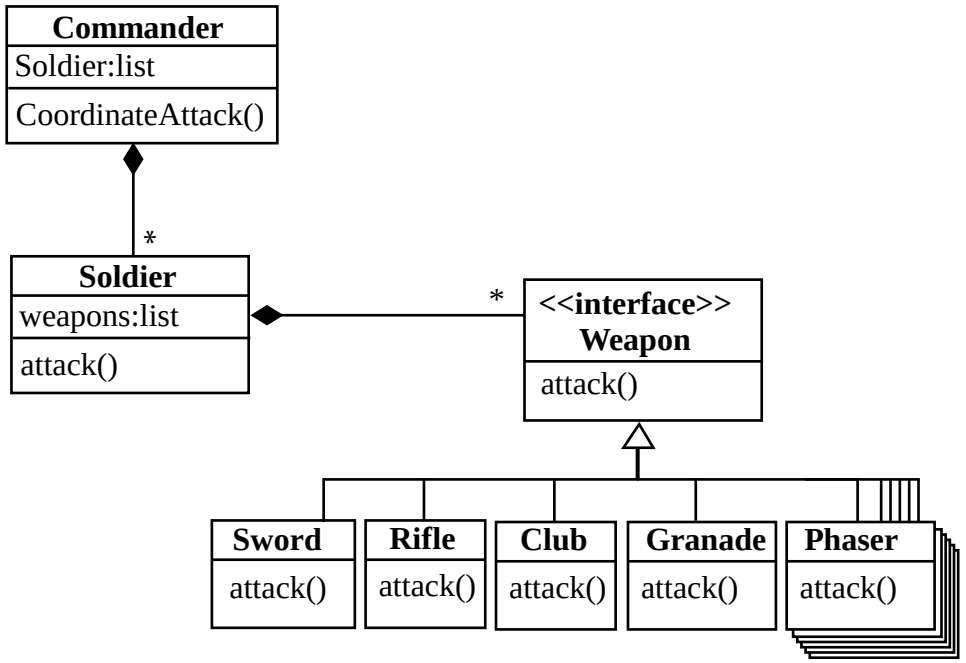
Here we don't have subclasses of Soldier we compose the appropriate behavior of a Soldier by adding instances of Weapon.

Design Principles

Design Principles

- *Identify the aspects of your application that vary and separate them from what stays the same.*
- *Program to an interface, not an implementation.*
- *Favor composition over inheritance.*
- *Strive for loosely coupled designs between objects that interact.*

These are loosely coupled objects.



The class that tells a soldier to attack knows nothing about how the soldier will perform that action.

The soldier class knows nothing about how the caller or how it performs its actions.

The soldier class tells each of its weapons to "attack", but it knows nothing about how that action is performed.

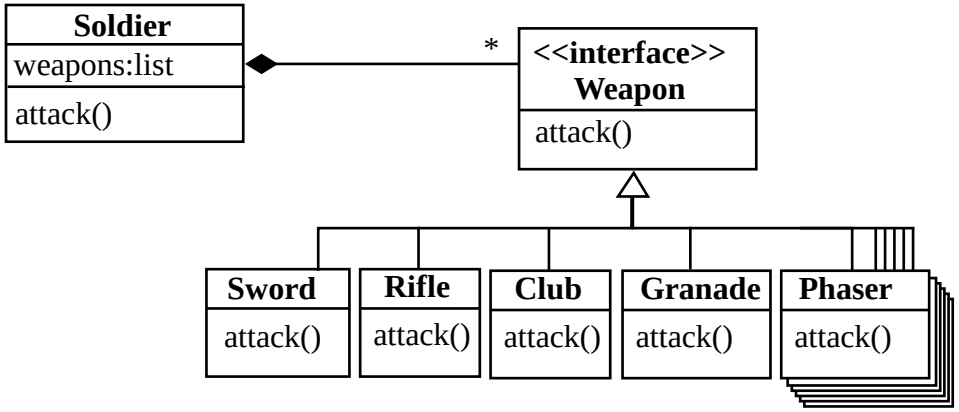
Design Principles

Design Principles

- *Identify the aspects of your application that vary and separate them from what stays the same.*
- *Program to an interface, not an implementation.*
- *Favor composition over inheritance.*
- *Strive for loosely coupled designs between objects that interact.*
- *Classes should be open for extension, but closed for modification.*

Also known as the Open-Closed Principle (OCP)

Take another look at the soldier...



Soldier and Weapon are closed for modification.

Soldier is open for extension through composition.

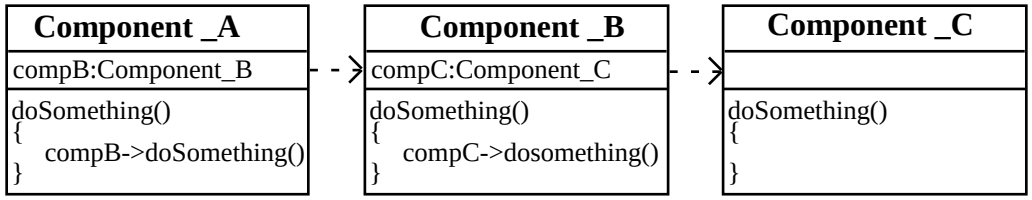
Weapon is open for extension through subclasses.

Design Principles

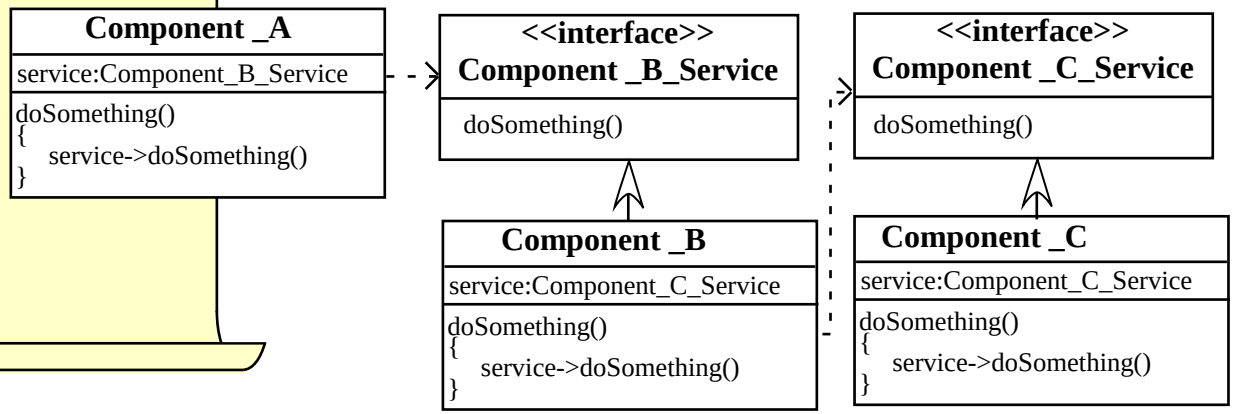
Design Principles

- *Identify the aspects of your application that vary and separate them from what stays the same.*
- *Program to an interface, not an implementation.*
- *Favor composition over inheritance.*
- *Strive for loosely coupled designs between objects that interact.*
- *Classes should be open for extension, but closed for modification.*
- *Depend on abstractions. Do not depend on concrete classes.*

Not good. Component_A depends on Component_B which also depends on Component_C



Much better.

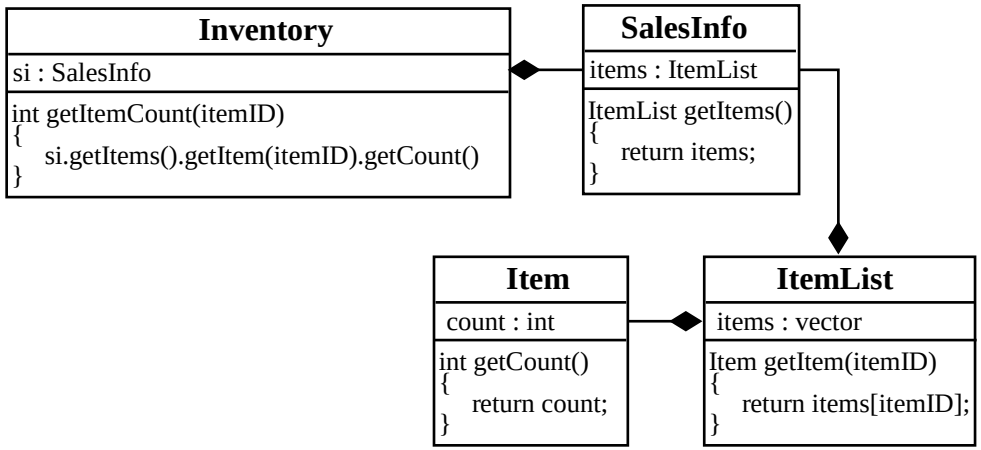


Design Principles

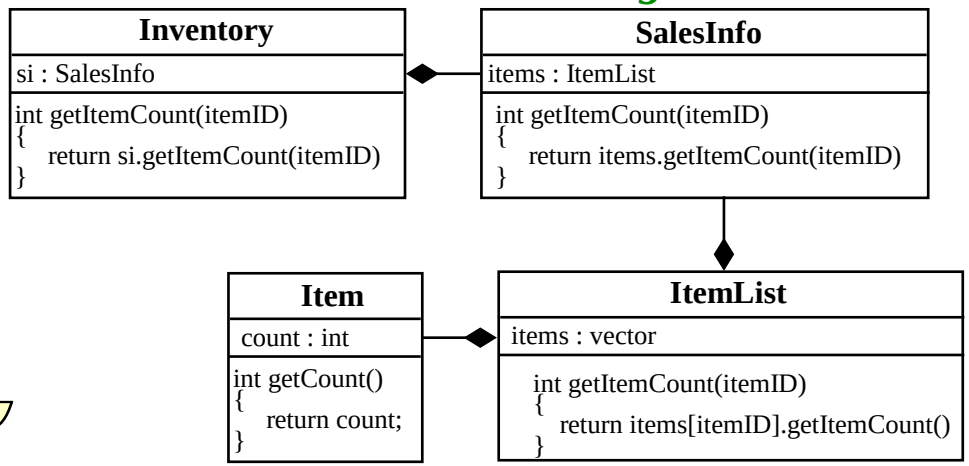
Design Principles

- *Identify the aspects of your application that vary and separate them from what stays the same.*
- *Program to an interface, not an implementation.*
- *Favor composition over inheritance.*
- *Strive for loosely coupled designs between objects that interact.*
- *Classes should be open for extension, but closed for modification.*
- *Depend on abstractions. Do not depend on concrete classes.*
- *Principle of Least Knowledge - talk only to your immediate friends.*

Bad: too many links.



Better: each class calls only functions in its immediate neighbor



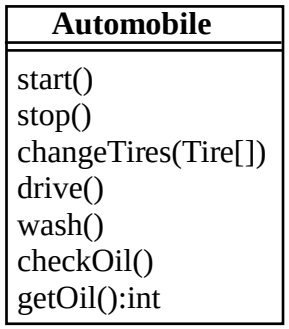
Design Principles

Design Principles

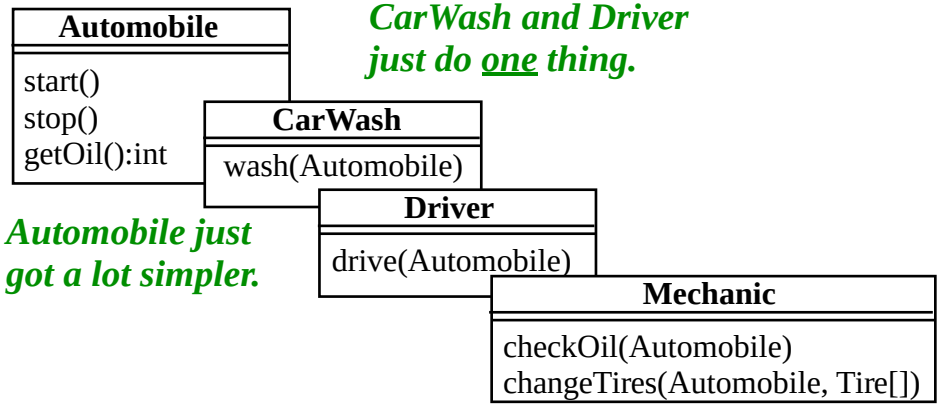
- *Identify the aspects of your application that vary and separate them from what stays the same.*
- *Program to an interface, not an implementation.*
- *Favor composition over inheritance.*
- *Strive for loosely coupled designs between objects that interact.*
- *Classes should be open for extension, but closed for modification.*
- *Depend on abstractions. Do not depend on concrete classes.*
- *Principle of Least Knowledge - talk only to your immediate friends.*
- *A class should have only one reason to change.*

Change

Not again?



Look at all the things that could cause this class to have to change!

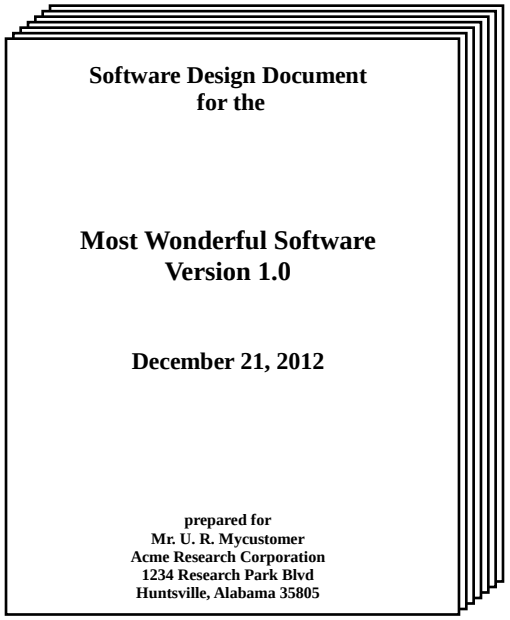


CarWash and Driver just do one thing.

Automobile just got a lot simpler.

You could even break this up into two behavior classes.

Design Document

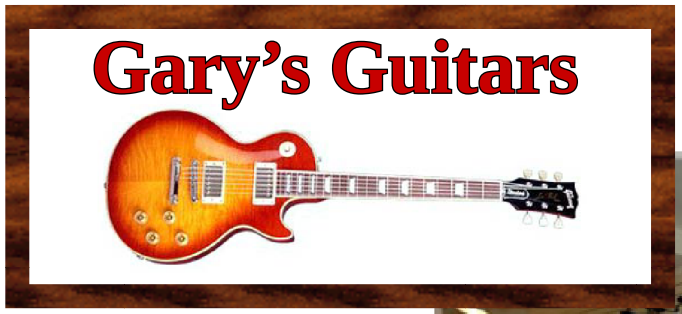


Only after you have worked out all the details of the design do you write the Design Document

Time Out!!

Let's pause to look at an example of why good Object Oriented Software Design is so important.

Object Oriented Design



Each guitar in the inventory is represented by an instance of this class

Guitar

serialNumber: string
price: double
builder: string
model: string
type: string
backWood: string
topWood: string

Variables for the Guitar class

getSerialNumber(): string
getPrice(): double
setPrice(float)
getBuilder(): string
getModel(): string
getType(): string
getBackWood(): string
getTopWood(): string

Methods for the Guitar class

Here's Gary's inventory as well as a way to search for guitars.

Inventory

guitars: list

Here's a list of all guitars in stock.

addGuitar(string, double, string, string, string, string, string)
getGuitar(string): Guitar
search(Guitar): Guitar

This takes all the specs and adds a Guitar to the inventory.

This takes a guitar's serial number and returns that Guitar object.

This takes a client's description of an ideal guitar, finds it in the inventory and returns that Guitar object.

A Fantastic piece of software! Now everythings' rosey...except for one thing...

Object Oriented Design

Gary is now loosing customers and money!

This is part of the code that sets up Gary's inventory. Looks like he has just the perfect guitar for Jimi.

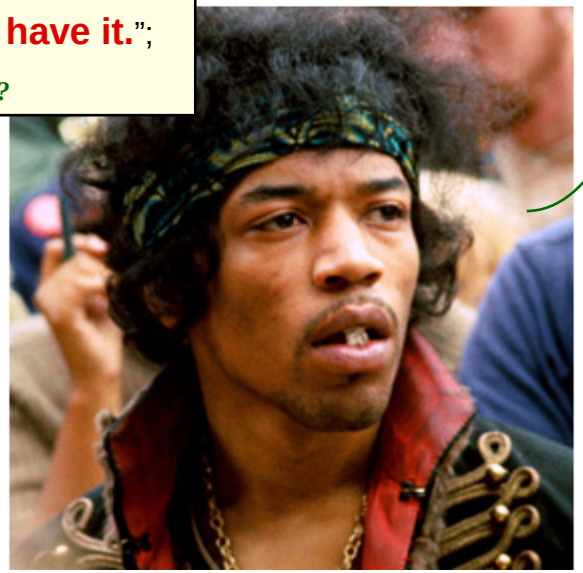
```
inventory.addGuitar("V95693", 4499.95, "Fender",
        "Stratocaster", "electric", "Alder", "Alder");
```

```
Guitar whatJimiWants = new Guitar("", 0, "fender",
    "Stratocaster", "electric", "Alder", "Alder");

Guitar guitar = inventory.search(whatJimiWants);

if(guitar == NULL)
{
   cout << "Sorry, Jimi. We don't have it.";
}
```

But, the specs match so what's up?



Sucks dude! I'll go somewhere else.

Gary: "Huh? I don't get it. I know we have one and its right here.

Object Oriented Design

Were you thinking "Object-Oriented"?
Did you notice any of these problems?

Look at all those strings. Too many ways to screw up man! Can't we use constants or objects instead?

Guitar
serialNumber: string price: double builder: string model: string type: string backWood: string topWood: string
getSerialNumber(): string getPrice(): double setPrice(float) getBuilder(): string getModel(): string getType(): string getBackWood(): string getTopWood(): string

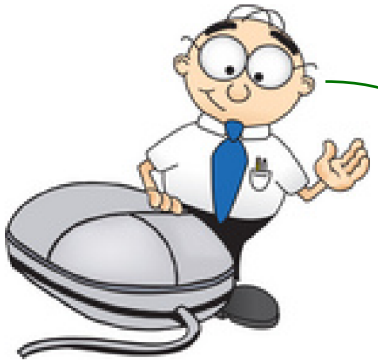
Whoa... these notes from the owner says he wants his clients to have multiple choices. Shouldn't the search() method return a list of matches?

Inventory
guitars: list
addGuitar(string, double, string, string, string, string, string) getGuitar(string): Guitar search(Guitar): Guitar

This design is terrible! The Inventory and Guitar classes depend on each other too much, and I can't see how this is an architecture that you'd ever be able to build upon. We need some restructuring!

Object Oriented Design

What is really great software?



The Customer Friendly Programmer

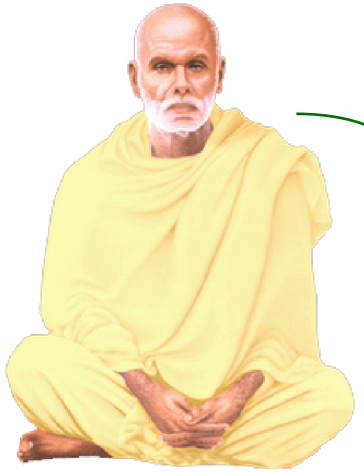
"Great software always does what the customer wants it to. So even if customers think of new ways to use the software, it doesn't break or give them unexpected results."

"You got to keep the customer satisfied! Satisfied!"
Simon & Garfunkle

The Object-Oriented Programmer

"Great software is code that is object-oriented. So there's not a bunch of duplicate code, and each object pretty much controls its own behavior. It's also easy to extend because your design is really solid and flexible."

Sounds geeky! But, good OO programmers are always looking for ways to make their code more flexible.



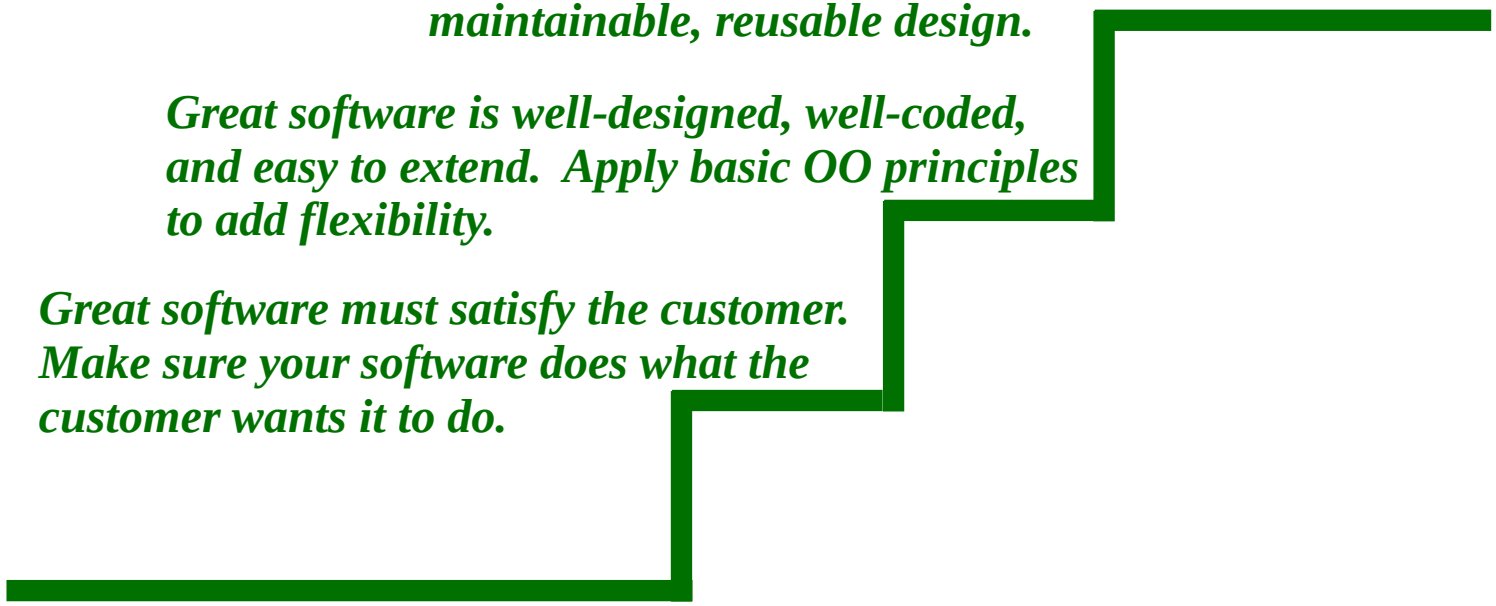
The Design-Guru Programmer

"Great software is when you use tried-and-true design patterns and principles. You've kept your object loosely coupled, and your code open for extension but closed for modification. That also helps make the code more reusable, so you don't have to rework everything to use parts of your application over and over again."

Do you understand all of this, Grasshopper?
Don't worry, we'll talk more about this later.

Object Oriented Design

Great Software in Three Easy Steps



So, how would you change Gary's software to make it do what it is supposed to do?

Object Oriented Design

Problem 1. Notice that we are using Guitar for two different purposes
a. To define the specs that a customer wants in a guitar
b. To define an actual guitar in the inventory

Guitar
serialNumber: string price: double builder: string model: string type: string backWood: string topWood: string
getSerialNumber(): string getPrice(): double setPrice(float) getBuilder(): string getModel(): string getType(): string getBackWood(): string getTopWood(): string

Inventory
guitars: list
addGuitar(string, double, string, string, string, string, string) getGuitar(string): Guitar search(Guitar): Guitar

This defines the guitar a customer is searching for, but the customer doesn't bring in a sample guitar for comparison.

This defines the actual guitar that is found for the customer.

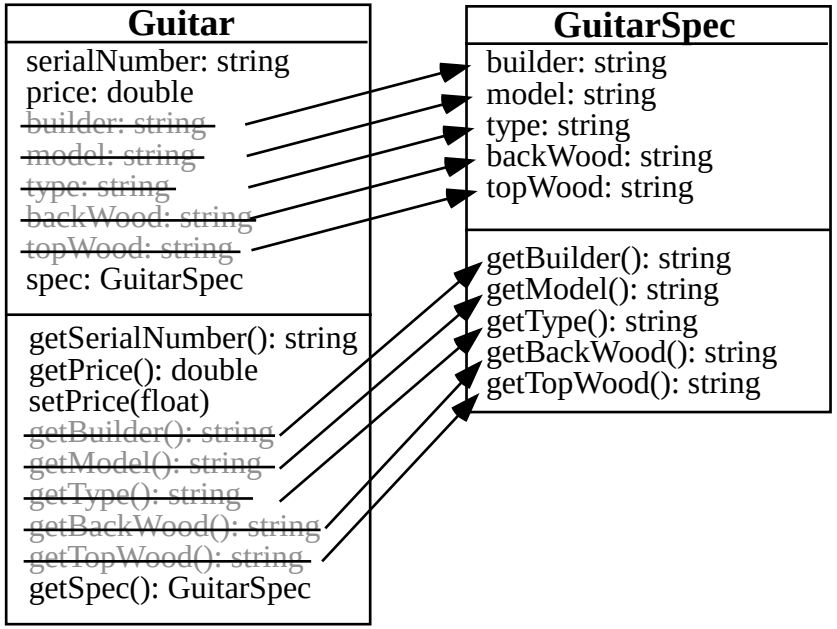
1. Objects should do what their names indicate and only that.

2. Each object should represent a single concept.

3. Unused properties are a dead giveaway

Object Oriented Design

A possible solution to the “Guitar has 2 uses” problem



Encapsulate the specifications for a guitar so it can be reused in different places and so we do not duplicate code.

- **Use Encapsulation to hide the inner workings of your application‘s parts.**
- **Encapsulation is about breaking your application into logical parts and then keeping those parts separate.**
- **Any time you see duplicate code in different parts of your application look for a way to encapsulate that so it appears only once and is used from different places.**

Object Oriented Design

Problem 2. Look at how many specifications for a guitar are defined using strings.

GuitarSpec
builder: string model: string type: string backWood: string topWood: string
getBuilder(): string getModel(): string getType(): string getBackWood(): string getTopWood(): string

Oops: ***Fender, fender, FENDER, fendr, fendour...***

These all get interpreted, or misinterpreted as different things.

Object Oriented Design

A possible solution to specs defined as strings

Enumerated data types in C++ or Type-safe Enums in Java

enum Wood {ALDER, MAPLE, OAK, ELDER, CHERRY, FIR, CEDAR, SPRUCE};

enum Numbers {ONE=1, TWO, THREE, FOUR, FIVE, TEN=10, ELEVEN};

By defining enum data types the GuitarSpec now looks like this...

GuitarSpec
builder: Builder model: string type: Type backWood: Wood topWood: Wood
getBuilder(): Builder getModel(): string getType(): Type getBackWood(): Wood getTopWood(): Wood

Defining an *enum* variable

```
Numbers num = ONE;
```

Using an *enum* variable

```
if(num == ONE)
    num = TWO;

switch(num)
{
        case ONE    :  /* do something*/  break;
        case TWO    :  /* do something*/  break;
        case THREE  :  /* do something*/  break;
}
```

Object Oriented Design

Problem 3. The search algorithm in the Inventory class.

```
Guitar Inventory::search(GuitarSpec *searchGuitar)
{
  for(list<guitars>::interator i=guitars.begin(); i<guitars.end(); i++)
  {
    if((i->getBuilder()!=NULL) && (!(i->getBuilder()= =searchGuitar->getBuilder())))
       continue;
    if((i->getModel()!=NULL) && (i->getModel()!="") && (!(i->getModel()= =searchGuitar->getModel())))
       continue;
    if((i->getType()!=NULL) && (!(i->getType()= =searchGuitar->getType())))
       continue;
    if((i->getBackWood()!=NULL) && (!(i->getBackWood()= =searchGuitar->getBackWood())))
       continue;
    if((i->getTopWood()!=NULL) && (!(i->getTopWood()= =searchGuitar->getTopWood())))
       continue;
    return *i;  // Found one that matches so return it
  }
  return NULL; // No match found
}
```

And note that this search reflects the addition of the enumerated data types instead of having to compare all strings.

But what happens if Gary decides he wants to add 12-string guitars to the inventory?

Object Oriented Design

Problem 3. The search algorithm in the Inventory class.

```
list *Inventory::search(GuitarSpec *searchGuitar)
{
  matchingGuitars.clear();
  for(list<guitars>::interator i=guitars.begin(); i<guitars.end(); i++)
   {
     if(guitar->getSpec().matches(searchSpec)
                          matchingGuitars->pushback(guitar)
   }
   return matchingGuitars; // Return list of all matches
}
```

We have moved the code to find a match out of the search algorithm.

Loose coupling

Delegation

Object Oriented Design

This is what all this has been leading up to...

Object Oriented Analysis and Design

You got to keep the customer satisfied...

... and customers are satisfied when their applications:

- **WORK.**
- **KEEP WORKING.**
- **CAN BE UPGRADED.**
- **CAN BE REUSED.**
- **ARE FLEXIBLE.**