G64OOS (Spring 2014)

Lecture 04 r02

Object Oriented Programming (Principles)

Peer-Olaf Siebers

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Motivation

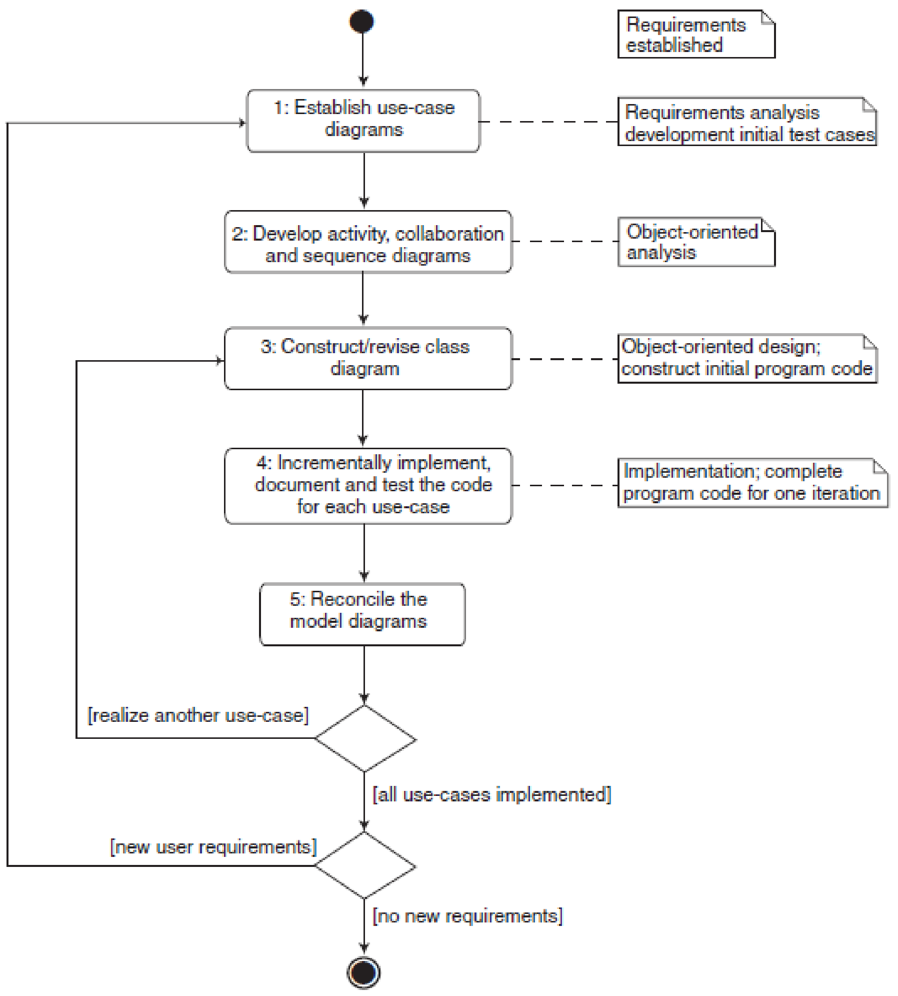
- Learn about strategic aspects of the OOA/D process
- Become acquainted with the ideas of the object oriented philosophy
- Introduce object oriented design principles and object oriented design patterns

Motivation

- OO Basics
  - Object model (abstraction, encapsulation, modularity, hierarchy); data abstraction; inheritance; polymorphism; interface
- OO Design Principles
  - Encapsulate what varies; favour composition over inheritance; program to interfaces, not implementations; ...
- OO Design Patterns:
  - Show how to build systems with good OO design qualities (reusable; extensible; maintainable)

OOA/D Process

OOA/D Process [Barclay and Savage 2004]



Requirements established
1: Establish use-case diagrams
Requirements analysis development initial test cases
2: Develop activity, collaboration and sequence diagrams
Object-oriented analysis
3: Construct/revise class diagram
Object-oriented design; construct initial program code
4: Incrementally implement, document and test the code for each use-case
Implementation; complete program code for one iteration
5: Reconcile the model diagrams
[realize another use-case]
[all use-cases implemented]
[new user requirements]
[no new requirements]

OOA/D Process [Barclay and Savage 2004]

- Requirements
  - Develop a set of use-cases that describe the capabilities we expect from the system; use-cases can also operate as the basis for test-cases since they define the functionality sought from the software
- Analysis
  - These analysis views reveal how a set of interacting objects deliver the functionality described by the use-cases
  - Establishing a number of sequence, object, collaboration and activity diagrams that demonstrate how each use-case is realized

OOA/D Process [Barclay and Savage 2004]

- Design
  - Formulate a class diagram for the system's architecture
- Implementation
  - We can augment the initial code with method bodies and start the process of implementing and testing the development
  - Here it is best to conduct this work incrementally; often it is possible to fully develop one class in isolation
- Refactoring
  - Refactoring a class diagram to obtain a more elegant solution often produces cleaner code that is simpler to enhance and maintain

OOA/D Process [Barclay and Savage 2004]

- What is so cool about the OO idea?
  - The development of OO systems is both: iterative and incremental
    - Iteration is a complete development cycle in which we deliver some subset of the overall product such as a single use-case; we then grow toward the final product by adding more use-cases
    - Increments introduce small changes as the software is developed
  - Such an approach can lead to a significant reduction in the risks involved in software development
  - We can constantly keep the customer and all other stakeholders informed at each iteration; with the customer always involved we can be sure that we deliver a product that meets their requirements

General Programming Philosophy

General Programming Philosophy [Raymond 2003]

- Basics of the Unix Philosophy (1/4)
  - Rule of Modularity
    - Write simple parts connected by clean interfaces
  - Rule of Clarity
    - Clarity is better than cleverness
  - Rule of Composition
    - Design programs to be connected to other programs
  - Rule of Separation
    - Separate policy from mechanism; separate interfaces from engines
  - Rule of Simplicity
    - Design for simplicity; add complexity only where you must.

General Programming Philosophy [Raymond 2003]

- Basics of the Unix Philosophy (2/4)
  - Rule of Parsimony
    - Write a big program only when it is clear by demonstration that nothing else will do
  - Rule of Transparency
    - Design for visibility to make inspection and debugging easier
  - Rule of Robustness
    - Robustness is the child of transparency and simplicity
  - Rule of Representation
    - Fold knowledge into data so program logic can be stupid and robust
  - Rule of Least Surprise
    - In interface design, always do the least surprising thing

General Programming Philosophy [Raymond 2003]

- Basics of the Unix Philosophy (3/4)
  - Rule of Silence
    - When a program has nothing surprising to say, it should say nothing
  - Rule of Repair
    - When you must fail, fail noisily and as soon as possible
  - Rule of Economy
    - Programming time is expensive; conserve it in preference to machine time
  - Rule of Generation
    - Avoid hand-hacking; write programs to write programs when you can
  - Rule of Optimization
    - Prototype before polishing; get it working before you optimize it

General Programming Philosophy [Raymond 2003]

- Basics of the Unix Philosophy (4/4)
  - Rule of Diversity
    - Distrust all claims for "one true way"
  - Rule of Extensibility
    - Design for the future, because it will be here sooner than you think

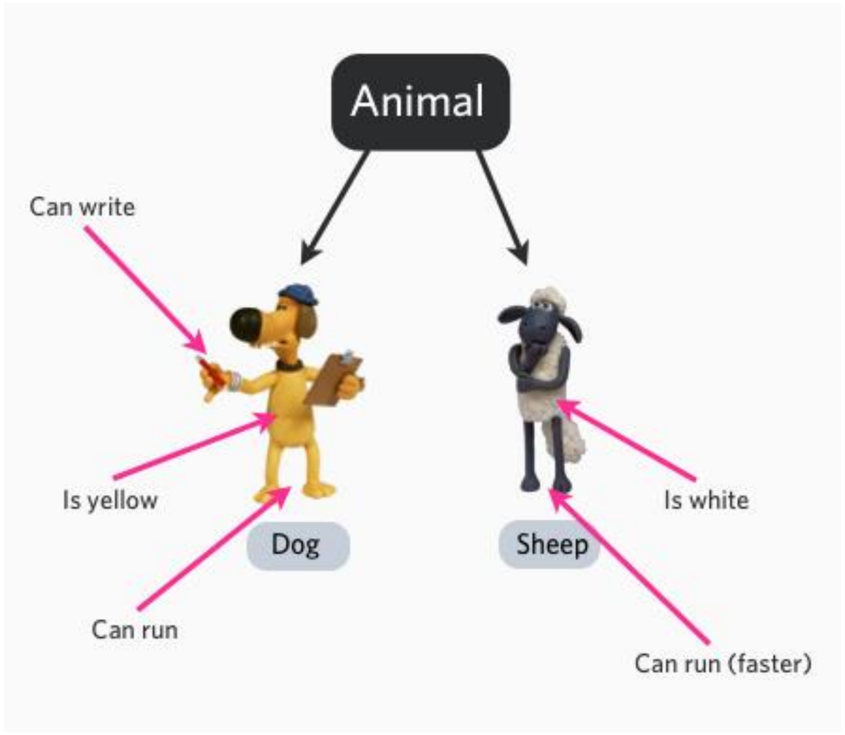
OO Basics: The Object Model

The Object Model [Booch 1994]

- Object Orientation
  - A way of thinking about a problem
    - Regard the world as consisting of interacting objects
    - Aim to understand objects, their properties, operations, and interactions with other objects
  - A way of solving problems
    - Build systems consisting of representations of objects
    - Objects interact with each other

The Object Model [Booch 1994]

- "A sound engineering foundation"
  - Four basic principles:
    - Abstraction
    - Encapsulation
    - Modularity
    - Hierarchy

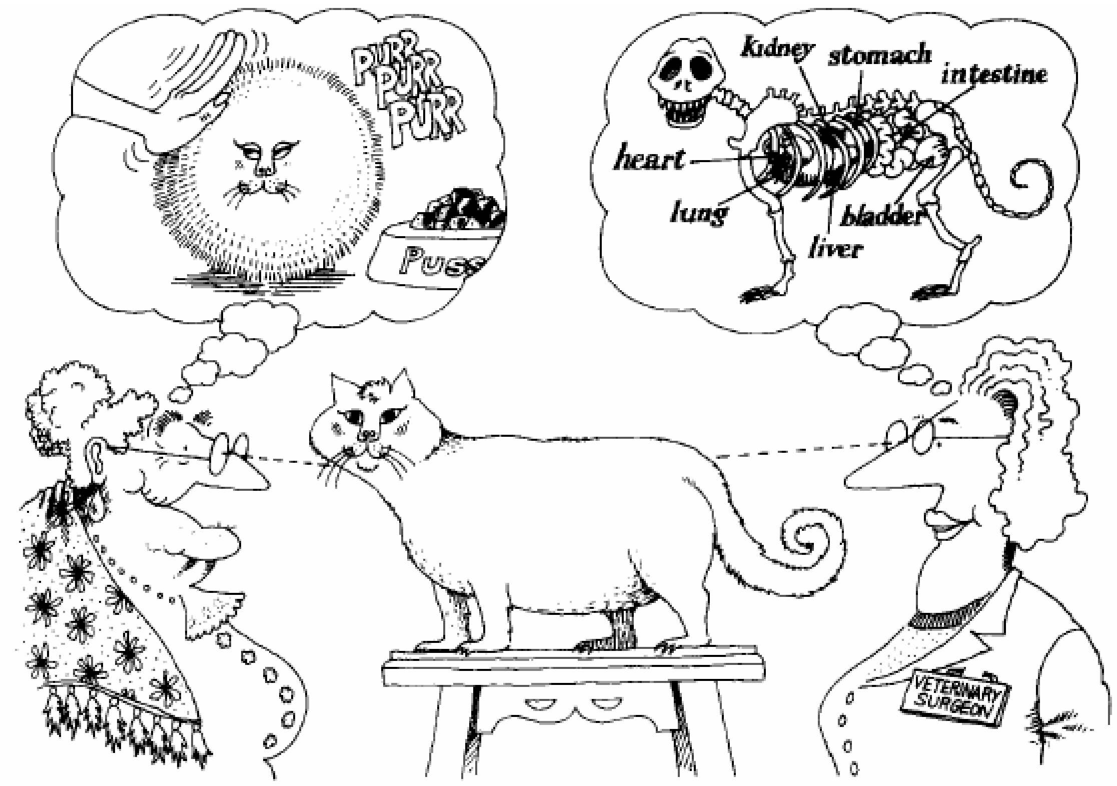


The Object Model [Booch 1994]

- Abstraction
  - "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."
  - Key concepts:
    - Concentrating only on essential characteristics: Allows complexity to be more easily managed
    - Abstraction is relative to the perspective of the viewer: Many different views of the same object are possible

The Object Model [Booch 1994]

- Abstraction



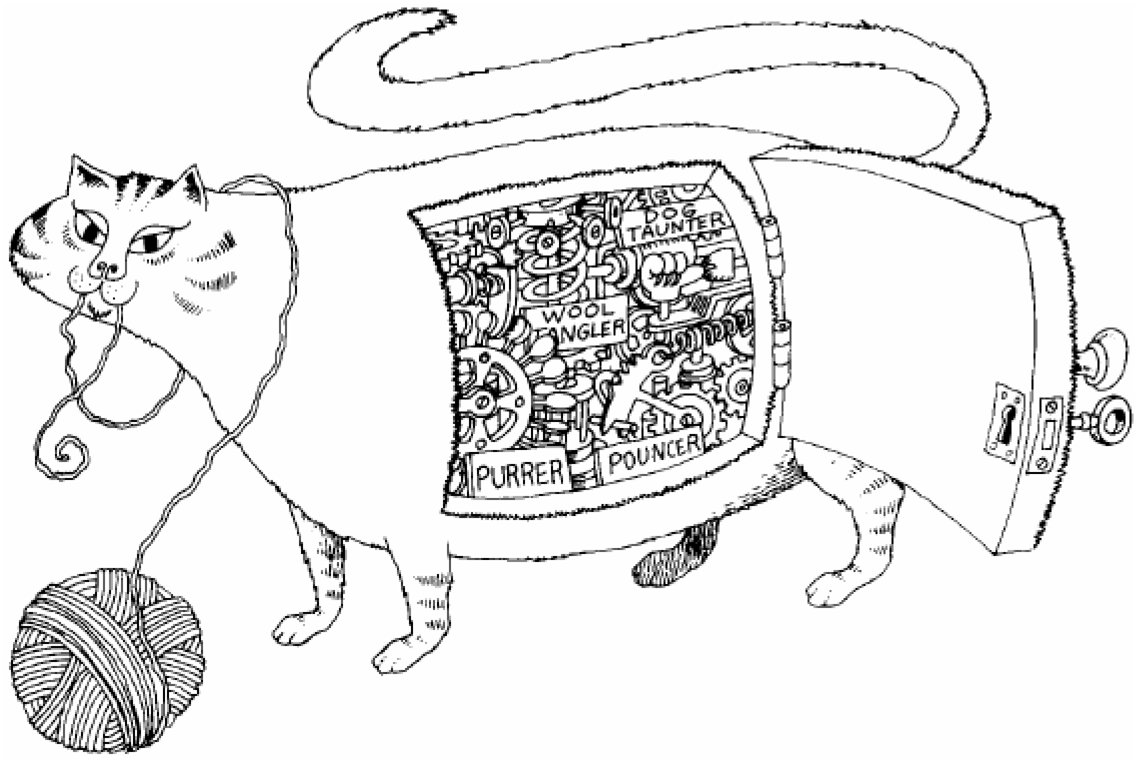
Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

The Object Model [Booch 1994]

- Encapsulation:
  - "Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation"
  - It associates the code and the data it manipulates into a single unit; it keeps them save from external interference and misuse
  - Key concepts:
    - Packaging structure and behaviour together in one unit: Makes objects more independent
    - Objects exhibit an interface through which others can interact with it: Hides complexity from an object's clients

The Object Model [Booch 1994]

- Encapsulation



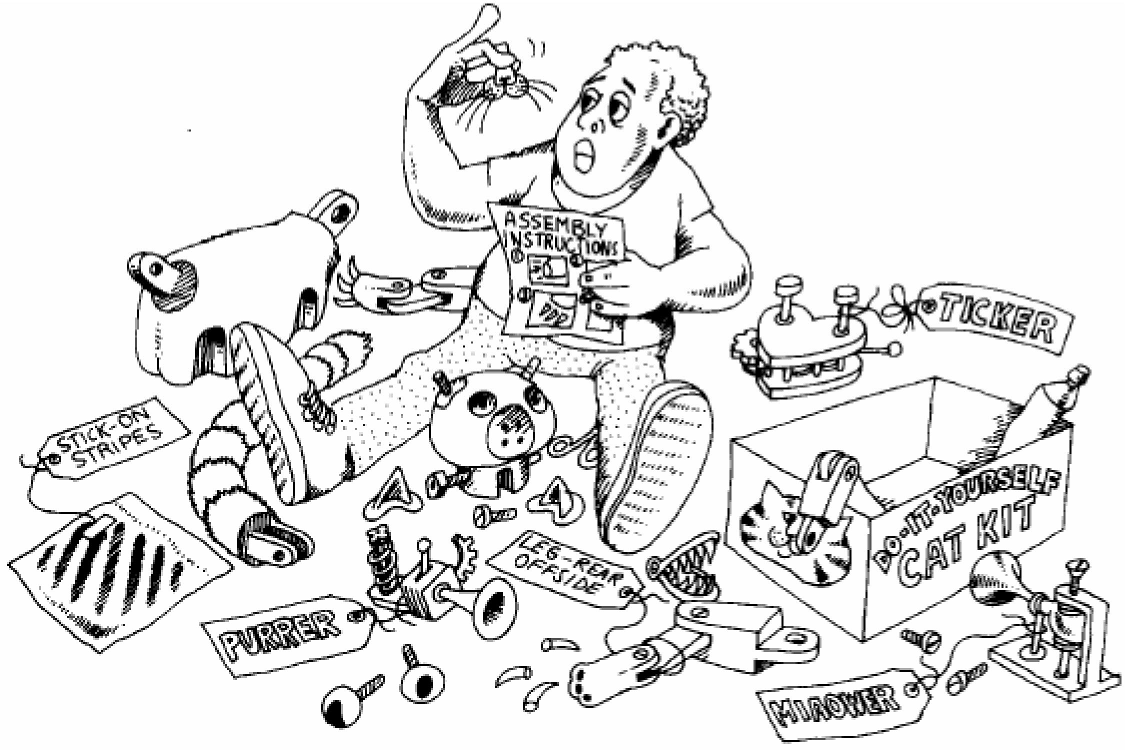
Encapsulation hides the details of the implementation of an object.

The Object Model [Booch 1994]

- Modularity:
  - "Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."
  - The unit of modularity in the object oriented world is the class
  - Key concepts:
    - Modules are cohesive (performing a single type of tasks): Makes modules more reusable
    - Modules are loosely coupled (highly independent): Makes modules more robust and maintainable

The Object Model [Booch 1994]

- Modularity



Modularity packages abstractions into discrete units.

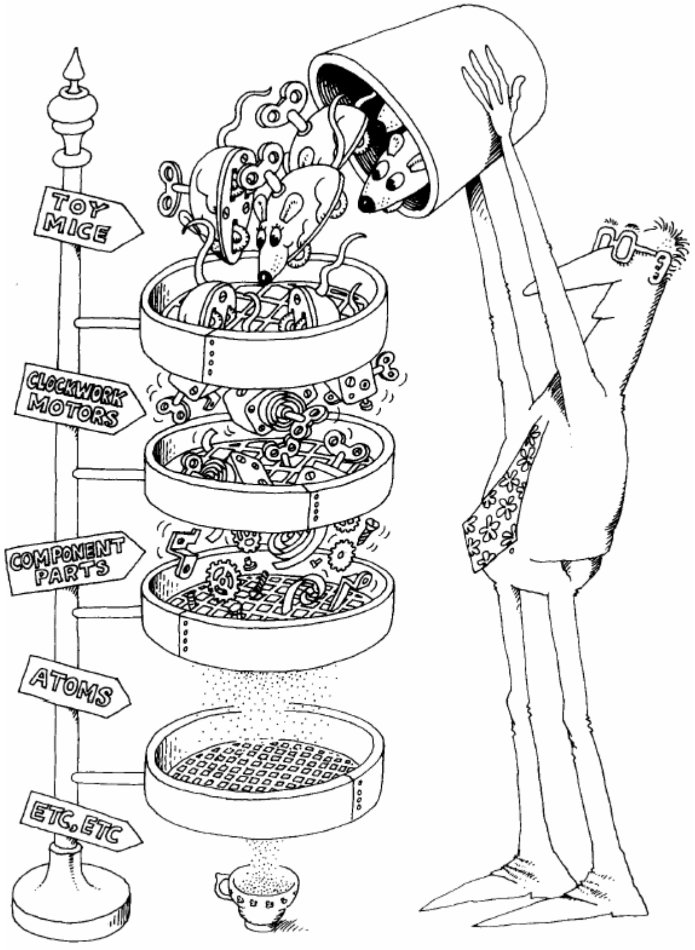
The Object Model [Booch 1994]

- Hierarchy:
  - "Hierarchy is a ranking or ordering of abstractions."

- Types of hierarchies:
  - Class
  - Aggregation
  - Containment
  - Inheritance
  - Partition
  - Specialization

The Object Model [Booch 1994]

- Hierarchy

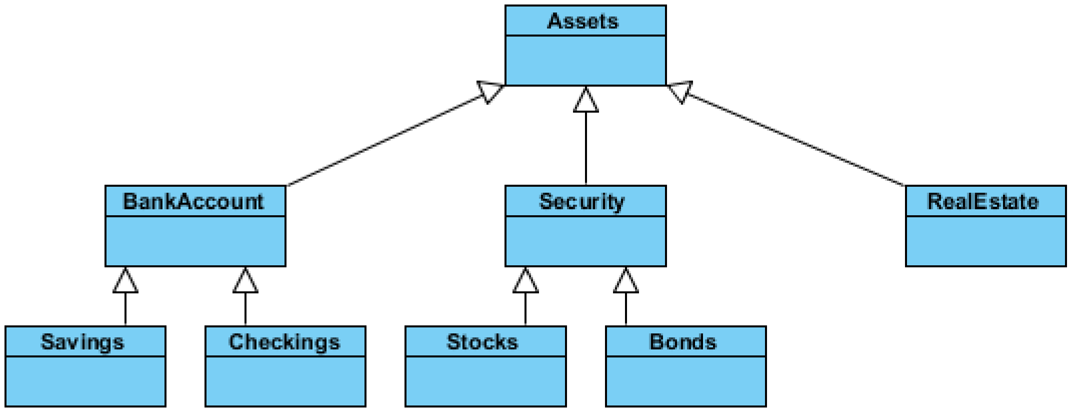


Abstractions form a hierarchy.

The Object Model [Booch 1994]

- Hierarchy:
  - "Hierarchy is a ranking or ordering of abstractions."

- Classes at the same level of the hierarchy should be at the same level of abstraction



OO Basics: Other Object Oriented Concepts

Other Object Oriented Concepts

- Data Abstraction
  - The technique of creating new data types that are well suited to an application; allows the creation of user defined data types, having the properties of build in data types and a set of permitted operators
  - Abstract data type: A structure that contains both, data and the actions to be performed with that data
  - Class is an implementation of an abstract data type

Other Object Oriented Concepts

- Inheritance
  - New data types (classes) can be defined as extensions to previously defined types; parent class = super class; child class = sub class; subclass inherits properties from super class
  - If multiple classes have common attributes and methods, these attributes and methods can be moved to a common class (parent class); this allows reuse since the implementation is not repeated

Other Object Oriented Concepts

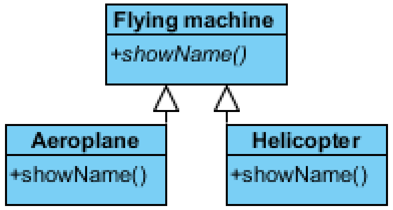
- Code Reuse: Inheritance vs Aggregation
  - Inheritance:
    - Inheritance defines implementation of a class in terms of another's; reuse by sub-classing often called **white box reuse**
  - Aggregation:
    - Aggregation obtains new functionality by assembling objects; requires that objects have well-defined interfaces; reuse by aggregating is often called **black box reuse**

- Favour aggregation over inheritance

Polymorphism

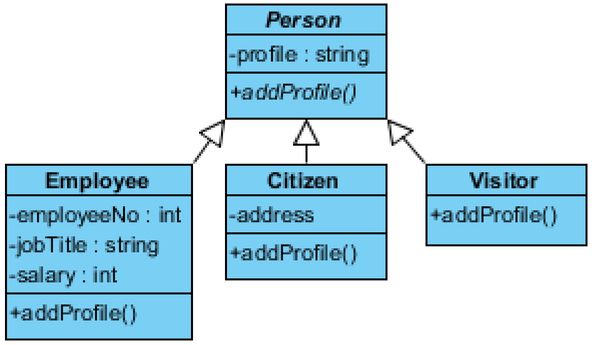
- Polymorphism
  - Polymorphism through method overloading (design time p.)
    - Create more than one function with same name but different signatures
  - Polymorphism through method overriding (run time p.)
    - Create a function in the derived class with the same name and signature
  - Polymorphism through sub classing (run time p.)
    - A pointer of a base class is able to reference, instantiate and destroy objects of a derived class
    - A pointer may reference objects of many classes at runtime, but the compiler cannot predict which they will be at compile time

```
FlyingMachine* flyer
flyer = new Aeroplane;
flyer = new Helicopter;
```



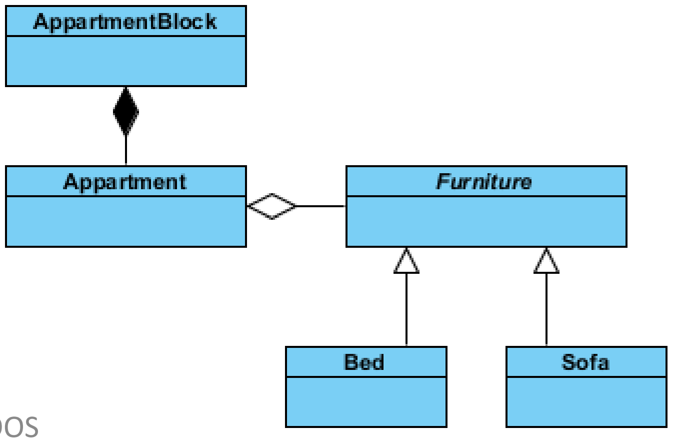
Other Object Oriented Concepts

- Abstract vs Concrete Classes
  - Abstract class
    - A class that cannot be directly instantiated
    - An abstract class is written with the expectation that its concrete subclasses will add to its structure and behaviour
  - Concrete class
    - A class that can be directly instantiated
    - A class whose implementation is complete and thus may have instances



Other Object Oriented Concepts

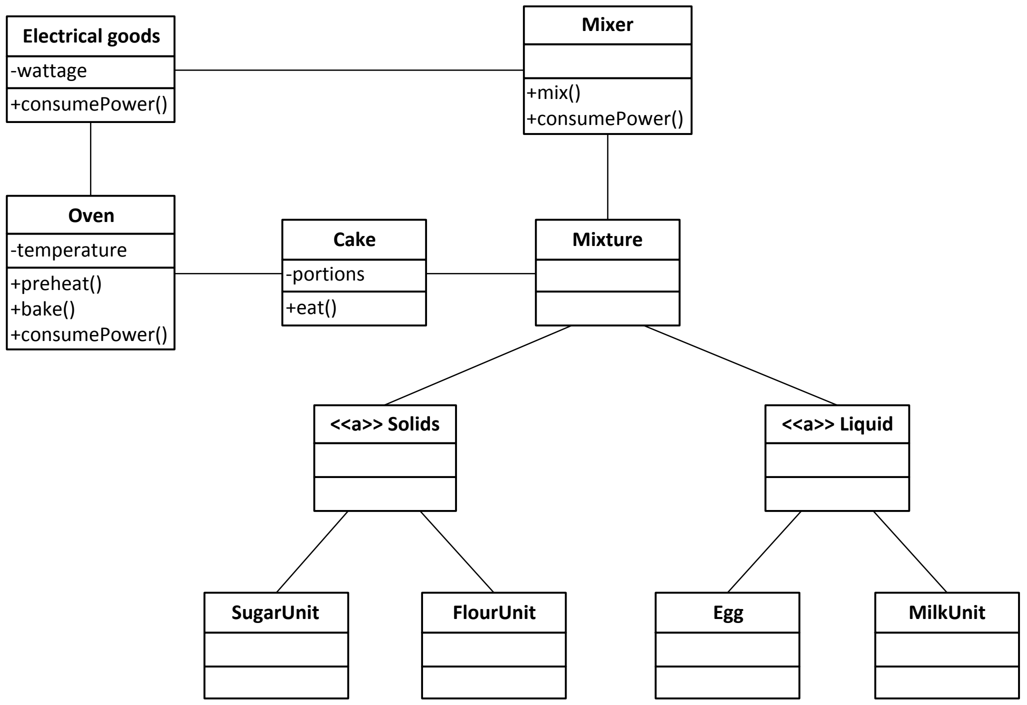
- Composition vs Aggregation
  - Composition
    - Implies that the internal objects are not seen from the outside
  - Aggregation
    - Aggregated objects might be directly accessed
    - They can be seen to have a separate existence



Other Object Oriented Concepts

- Interface
  - An interface describes the behavior or capabilities of a class without committing to a particular implementation of that class
  - Interfaces are implemented using abstract classes
  - An interface is a contract of related services and a set of conditions that must be true for the contract to be faithfully executed
  - Interfaces formalize polymorphism, they allow us to define polymorphism in a declarative way unrelated to implementation

A small challenge …



Break

- See you back in 10 minutes

OO Design Principles

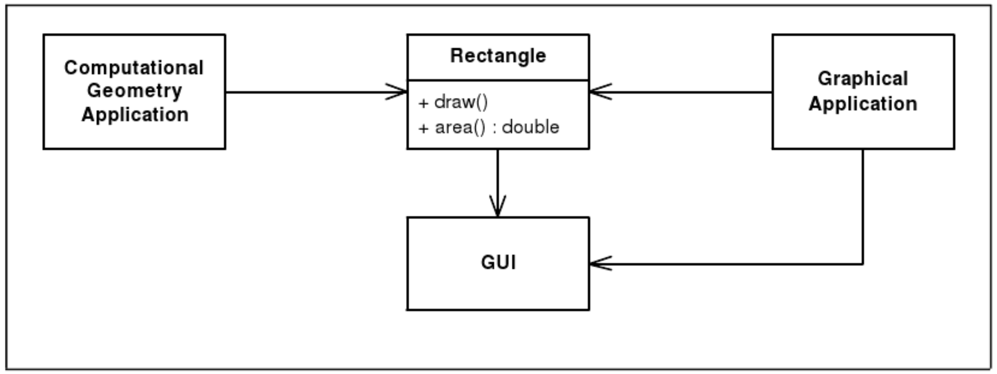
Design Principles (SOLID)

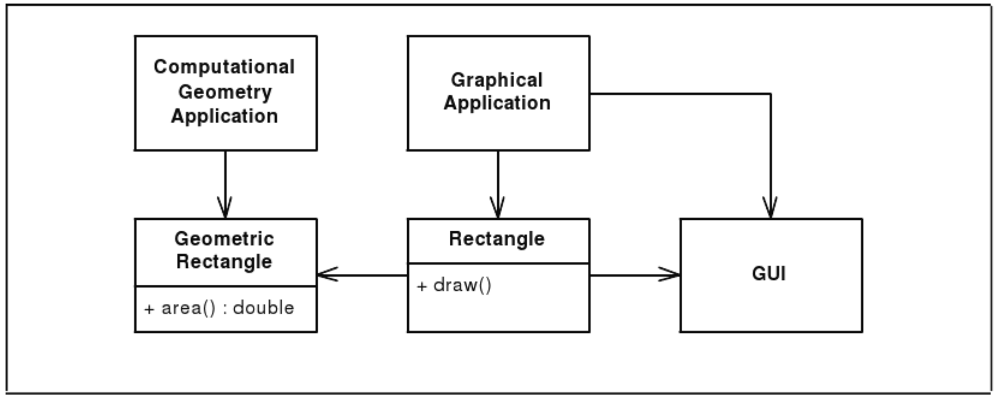
- Software solves real life business problems and real life business processes evolve and change - always.
- A smartly designed software can adjust changes easily; it can be extended, and it is re-usable.

- SOLID Principles (by Uncle Bob) [http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod]
  - **S** = Single Responsibility Principle
  - **O** = Open-Closed Principle
  - **L** = Liscov Substitution Principle
  - **I** = Interface Segregation Principle
  - **D** = Dependency Inversion Principle

Single Responsibility Principle

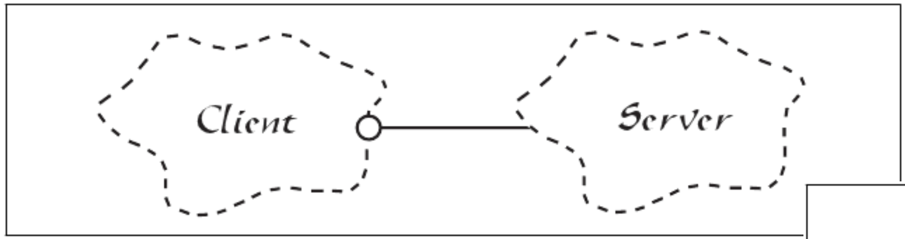
- A class should have one and only one responsibility

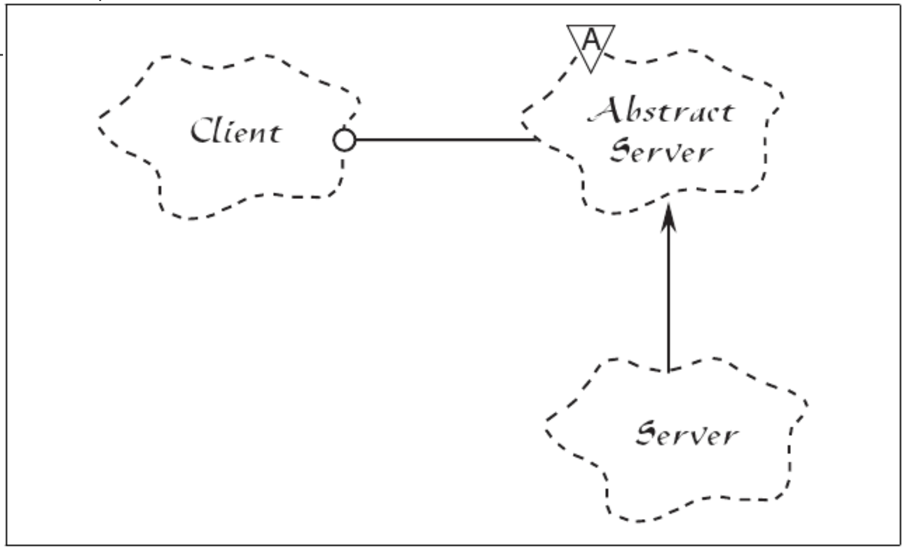




Open-Closed Principle

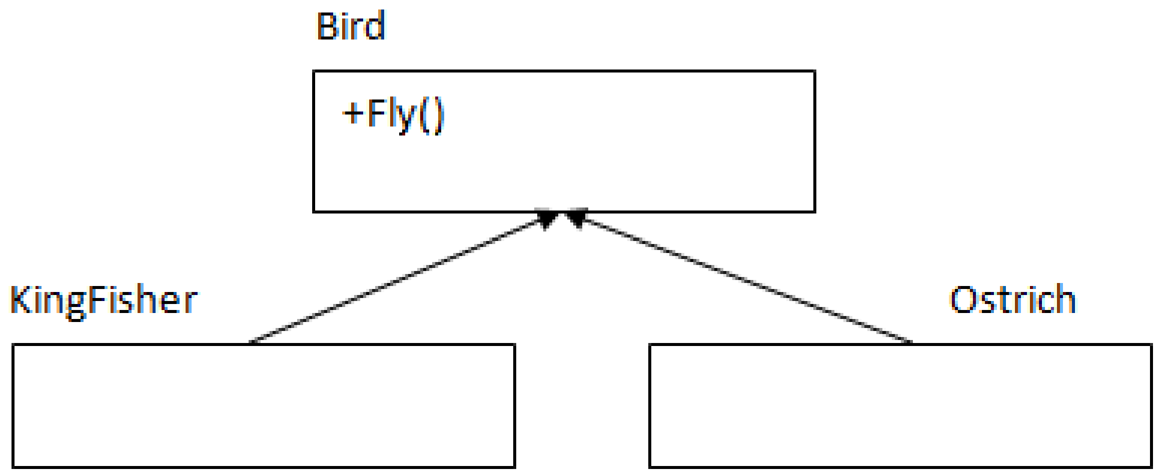
- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.





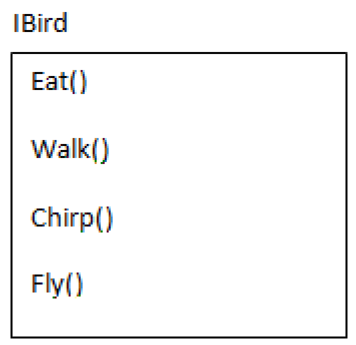
Liskov's Substitution Principle

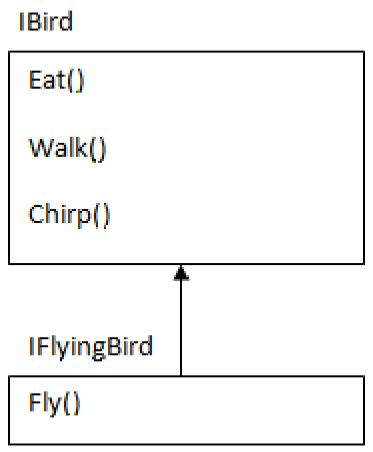
- Subtypes must be substitutable for their base types



Interface Segregation Principle

- Clients should not be forced to depend upon interfaces that they do not use





Dependency Inversion Principle

- High level modules should not depend upon low level modules. Rather, both should depend upon abstractions

- For more on SOLID see: [http://www.codeproject.com/Articles/93369/How-I-explained-OOD-to-my-wife]

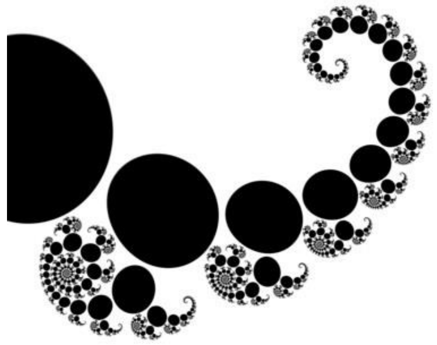
OO Design Patterns

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Design Patterns

- What is a Design Pattern?
  - A pattern describes a problem which occurs over and over again and then describes the core of the solution to that problem in such a way that it can use the solution over and over again without ever doing it the same way again.
  - A pattern provides an abstract description of a design problem and how a general arrangement of elements solves it
  - A design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities



Design Patterns

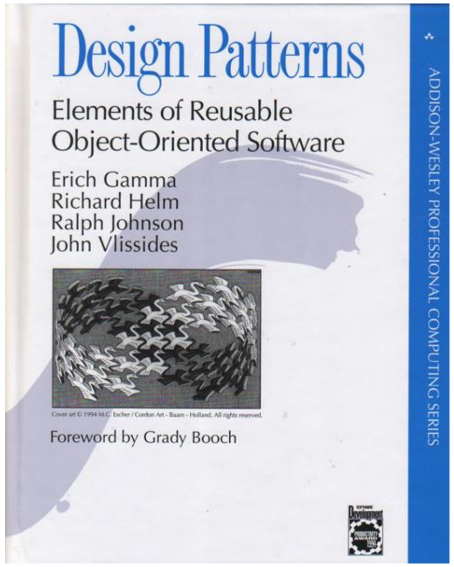
- A pattern has four essential elements:
  - Name: A handle that we can use to describe a design problem, its solution and the consequences in one or two words; having a vocabulary for patterns lets us talk about them with our colleagues and in our documentation
  - Problem: Describes when to apply a pattern; sometimes the problem includes a list of conditions that must be met before it makes sense to apply a pattern
  - Solution: Describes the elements that make up the design and their relationships
  - Consequences: Describe the results and trade-offs (time and space) of applying the pattern; critical for evaluating design alternatives

Design Patterns

- Design patterns are organised in two ways:
  - Purpose: Reflects what the pattern does
    - **Creational:** Concern the process of object creation
    - **Structural:** Deal with the composition of classes and objects
    - **Behavioural:** Characterise the way in which classes and objects interact and distribute responsibility
  - Scope: Specifies whether the pattern applies to classes or objects
    - **Class:** These patterns deal with relationships between classes and sub-classes (which are fixed at compile time)
    - **Object:** Deal with object relationships (which can be changed at runtime)

Design Patterns

- Design pattern organisation



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter
				Template Method
	Object	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Proxy	Flyweight
				Observer
				State
				Strategy
				Visitor

- See also on YouTube: [http://www.youtube.com/playlist?list=PLF206E906175C7E07]

Design Patterns

- Creational Patterns Intents:
  - Factory Method: Creates an instance of several derived classes
  - Abstract Factory: Creates an instance of several families of classes
  - Builder: Separates object construction from its representation
  - Prototype: A fully initialized instance to be copied or cloned
  - Singleton: A class of which only a single instance can exist

Design Patterns

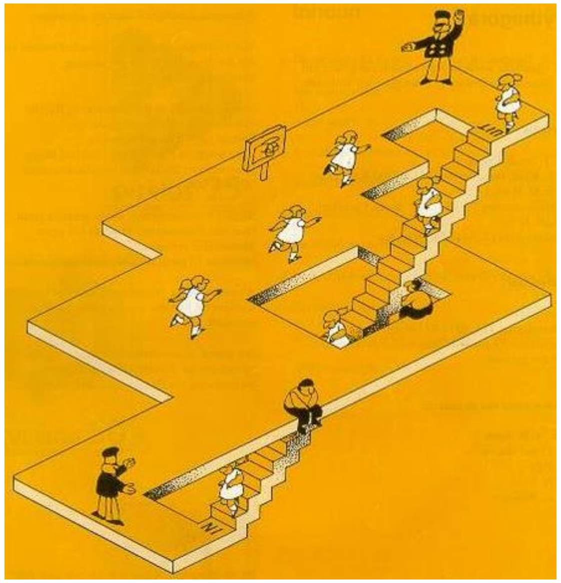
- Structural Pattern Intents
  - Adapter: Match interfaces of different classes
  - Bridge: Separates an object's interface from its implementation
  - Composite: A tree structure of simple and composite objects
  - Decorator: Add responsibilities to objects dynamically
  - Facade: A single class that represents an entire subsystem
  - Flyweight: A fine-grained instance used for efficient sharing
  - Private Class Data: Restricts accessor/mutator access
  - Proxy: An object representing another object

Design Patterns

- Behavioural Design Pattern Intents
  - Chain of responsibility: A way of passing a request between a chain of objects
  - Command: Encapsulate a command request as an object
  - Interpreter: A way to include language elements in a program
  - Iterator: Sequentially access the elements of a collection
  - Mediator: Defines simplified communication between classes
  - Memento: Capture and restore an object's internal state
  - Null Object: Designed to act as a default value of an object
  - Observer: A way of notifying change to a number of classes
  - State: Alter an object's behaviour when its state changes
  - ...

Design Patterns

- How design patterns help solving design problems
  - The hard part about OOAD is decomposing a system into objects; design patterns help to identify less obvious abstractions and the objects that can capture them
  - It is difficult to specify object interfaces; design patterns help you define interfaces by identifying their key elements and the kinds of data that get send across an interface
  - ...

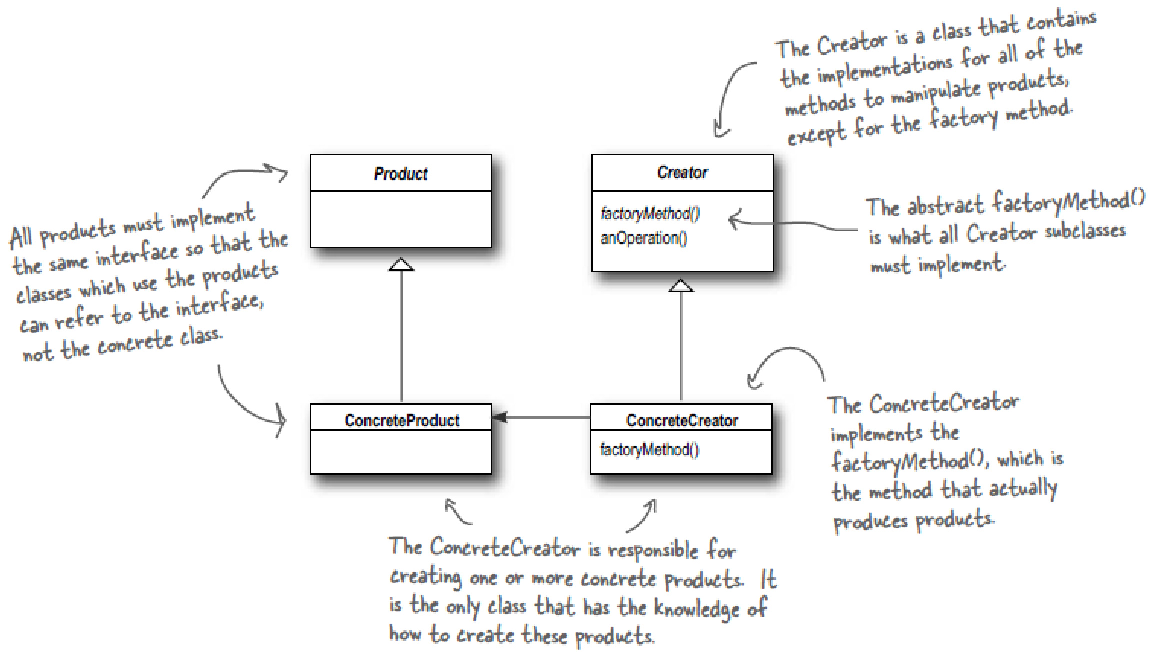


Design Patterns

- How do you select a design pattern?
  - Consider how design patterns solve design problems
  - Scan the "Intent" sections (see Gamma et al 1995 or Wikipedia)
  - Study how patterns interrelate (see Gamma et al 1995)
  - Study patterns of like purpose
  - Examine a cause of re-design
  - Consider what should be variable in your design

Design Pattern Example

- Creational Pattern: Factory Method [Freeman et al 2004]
  - Defines an interface for creating an object, but lets subclasses decide which class to instantiate



The Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method.
Product
Creator
factoryMethod()
anOperation()
All products must implement the same interface so that the classes which use the products can refer to the interface, not the concrete class.
The abstract factoryMethod() is what all Creator subclasses must implement.
ConcreteProduct
ConcreteCreator
factoryMethod()
The ConcreteCreator implements the factoryMethod(), which is the method that actually produces products.
The ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products.

```
#include <iostream>
using namespace std;
// creator
class Pizza{
    public:
        virtual int getPrice()const=0;
};

class HamAndMushroomPizza:public Pizza{
    public:
        virtual int getPrice()const{return 850;}
};

class HawaiianPizza:public Pizza{
    public:
        virtual int getPrice()const{return 1150;}
};
// concrete creator
class PizzaFactory {
    public:
        enum PizzaType{HamMushroom,Hawaiian};
        static Pizza* createPizza(PizzaType pizzaType){
            switch (pizzaType){
                case HamMushroom:return new HamAndMushroomPizza();
                case Hawaiian:return new HawaiianPizza();
            }
        }
};

void pizza_information(PizzaFactory::PizzaType pizzatype){
    Pizza* pizza = PizzaFactory::createPizza(pizzatype);
    cout<<"Price of "<<pizzatype<<" is "<< pizza->getPrice()<<endl;
    delete pizza;
}

int main (){
    pizza_information(PizzaFactory::HamMushroom);
    pizza_information(PizzaFactory::Hawaiian);
}
```

[Wikipedia 2013]

Some final words from Stroustrup ...

- Be clear about what you are building
- Good software development is a long-term activity
- No "cookbook" methods that can replace intelligence, experience, and good taste in design and programming
- Design and programming are iterative processes
- Different phases of a project (e.g. design, programming, testing) cannot be strictly separated
- Programming and design cannot be considered without also considering their management

Questions / Comments

References

- Barclay and Savage (2004) Object-Oriented Design with UML and Java
- Freeman et al (2004) Head First Design Patterns
- Gamma et al (1995) Design Patterns: Elements of Reusable Object-Oriented Software
- Raymond (2003) The Art of Unix Programming [url]
- Wikipedia (2013) C++ Programming/Code/Design Patterns: Factory Design Pattern [url]