ADDITIONAL MATERIAL FOR INFSEN01-2

DR ANNA V. KONONOVA

LECTURE 1. INTRODUCTION TO DESIGN PATTERNS IN OOP. MAY 2, 2017

- **Design patterns for object-oriented programming** Modulewijzer
- Books: (1) Gamma, Helm, Johnson, Vlissides, "Design patterns: elements of reusable object-oriented software;" (2) Freeman, Freeman, "Head first design patterns";
- This is a course to revisit!
- **Read all official slides** INFDEV02-4_Lec1
- Read every term on the slides and every sample
- Expected study effort is between 10 and 20 net per week
- If you do not understand it perfectly, either ask me, google, or brainstorm with the others
- every sample of code on the slides you should both understand and try out on your machine
- **Good luck!**

- **Today** we will: clarify terminology, remember basics and principles of OOP, see where design patterns come from, and where exactly they can help, review sources and consequences of design choices, understand diversity of design patterns, look at some how-tos and examples

GLOSSARY

(Making sure we speak the same language)

- **Abstraction** – gradual analysis of a system aimed at filtering out all its nonessential aspects in order to isolate key features defining/governing this system ➢ levels of abstraction; concrete
- **Algorithm** – an ordered set of actions that need to be performed to reach the set goal
- (SW) **architecture** – high level structural design of a SW system
- **Granularity** – the extent to which a system is composed of distinguishable pieces
- **Interface** (to an object) – signatures of all object's methods
- **Signature** – name of the method together with the list of objects it takes as parameters, method's return value

To be updated throughout the course?

OBJECT-ORIENTED PROGRAMMING. BASICS

- **Concept of an object**
- **Abstraction**

Fixing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level

- **Encapsulation**

Binding of data with the methods. Separation of internal and external ➢ object's internal state cannot be accessed directly

- **Polymorphism**

Provision of a single interface to entities of different types. They are interchangeable

- **Inheritance**

Basing objects/classes on other objects/classes. Co-relating classes together. Mechanism of code reuse and overriding.

- **Subtyping**

A form of type polymorphism where a subtype is a datatype that is related to another datatype. Program elements operating on supertype also operate on subtype

OBJECT-ORIENTED PROGRAMMING. PRINCIPLES

(Wake-up task: for each principle, think of a reason why it is important)

- Encapsulate what varies
- Favour composition over inheritance
- Program to interfaces not implementation
- 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
- Only talk to your friends
- Don’t call us, we call you
- A class should have only one reason to change
- more?

- Design for change

- Designing flexible and reusable OO SW:

understand the requirements ➤ find pertinent objects ➤ factor them into classes of right granularity ➤ define class interfaces and inheritance hierarchies ➤establish key relationships between them…

- Difficult to get right the first time ➤need to re-iterate
- Requires experience ➤ there's a shortcut!
- Reuse solutions! ➤ design patterns ➤ help get a design "right" faster
- There's also a catch ☹
- *Abstract factory, adapter, bridge, builder, chain of responsibility, command, composite, decorator, façade, factory method, flyweight, interpreter, iterator, mediator, memento, observer, prototype, proxy, singleton, state, strategy, template method, visitor, …*
- Purpose: creational, structural, behavioral
- Some can be used together. Some are alternatives for each other. Some result in similar designs

HOW DESIGN PATTERNS SOLVE DESIGN PROBLEMS

- Finding appropriate objects

Identifying less-obvious abstractions and objects that can capture them

- Determining object's granularity
- Specifying object's interfaces

Identifying key elements of the interface and required data. Relationship between interfaces

- Specifying object's implementations

Inheritance hierarchy. Programming to an interface, not an implementation

- Putting reuse mechanisms to work

Inheritance vs composition. Delegation. Inheritance vs parameterised types.

- Designing for change

Anticipating changes in SW requirements maximises reuse. Lower the risk of a major redesign and retesting

REUSE MECHANISMS

- **Inheritance vs composition**

(white-box reuse) Class inheritance: **+** defined statically, straightforward to use, easy to modify; **-** cant change inherited implementation at run-time, exposes a subclass to details of its parent's implementation (breaks encapsulation!)

(black-box reuse) Object composition: **+** defined dynamically at run-time through objects acquiring references to objects, objects are replaceable, object's implementation is in terms of interfaces bringing less implementation dependencies, each class is focused on one task; **-** more objects, system depends on their interrelationships

- **Delegation**

Object receiving a request involves its delegate into handling a request, passing itself ➢ composing behaviours at run-time, changes ways object composition; complicates things

- **Inheritance vs parameterised types**

Not very OO. Allows one to define a type without specifying all other types it uses. Unspecified types are supplied as parameters at the point of use.

COMMON CAUSES OF REDESIGN

- Creating an object by specifying a class explicitly
- Dependence on specific operations
- Dependence on hardware or software platforms
- Dependence on object representations or implementations
- Algorithmic dependencies
- Tight coupling
- Extending functionality by subclassing
- Inability to alter classes conveniently

All these can be addressed by using correct design patterns

DESIGN PATTERNS. HOW-TOS

How-to select

- Easy: consult a book

➢ understand pattern's intent, purpose and tradeoffs, understand how patterns interrelate, analyse possible causes of redesign, encapsulate the varying concept

How-to use

- Fully understand the pattern(s) you are planning to use
- Choose meaningful names and define classes
- Define application-specific names for operations in the pattern
- Implement operations to carry out the responsibilities and collaborations in the pattern

MORE REASONS TO USE DESIGN PATTERNS

- In case you still need convincing…
- Design patterns provide common vocabulary among SW designers. Systems seem less complex and can be discussed in higher levels of abstraction
- Design patterns help reorganise a design to reduce the potential amount of refactoring or point to targets for refactoring

➢ A thorough requirement analysis can highlight those requirements that are likely to change during the lifetime of the SW. A good design should be robust to such changes

{ prototyping ➢ more requirements ➢ expansion ➢ more reuse ➢ consolidation }

DESIGN PATTERN CATALOGUE

- Creational patterns

Abstract the instantiation process; make a system independent of how its objects are created, composed and represented. Example: *Abstract factory*

- Structural patterns

Concerned with how classes and objects are composed to form larger structures. Examples: *Decorator, Adaptor*

- Behavioural patterns

Concerned with algorithms and the assignment of responsibilities between objects. Describe not just patterns of objects or classes but also the patterns of communication between them. Examples: *Visitor, Iterator*

WAKE UP BREAK. NATURAL EVOLUTION OF REQUIREMENTS

- Duck pond simulator game. A variety of ducks species swims around and makes quacking noises. ➢ superclass Duck with methods quack(), swim() and display() and two subclasses: MallardDuck and RedheadDuck both of which redefine display() method to show an appropriate duck.
- Now we need ducks to fly. How would you redesign the classes?
- Most straightforward is to add a new method to the superclass, right?
- However, unbeknownst to you, your colleague has added a new type of the duck, a rubber duck.. Which seems to be able to fly now. Can you think of any change for your design?
- Inheritance springs to mind, right? Override fly() method of rubber duck to do nothing
- However, rumour has it that soon the simulator will also have wooden decoy ducks
- Solution! Introduce Flyable() interface with fly() method which flying ducks can implement. And Quackable() interface too, with quack() method
- But what if ducks suddenly change the way they fly? All subclasses will need a change!
- Design pattern? Not yet! Just an OO principle (which?)

WAKE UP BREAK. NATURAL EVOLUTION OF REQUIREMENTS (CONT'ED)

- "Encapsulate the change" principle. How to use it here?
- Duck superclass looks stable. The change comes from flying and quacking. And now?
- Separate them in two separate classes! Actually two separate sets of classes: flying behaviour and quacking behaviours. This kind of modification will later also accommodate squeaking and being mute. Take care to stay flexible!
- We can now even assign behaviours run-time, can we?
- Looks like another OO principle is coming to play here as well. Which one?
- "Program to interface, not an implementation".
- An interface for each behaviour and each implementation of behaviour implements one of those interfaces ➢ behaviour classes!
- Now the superclass uses a behaviour represented by an interface. An actual implementation is not locked into sub-classes
- Draw our new design
- Draw the big picture. Is it clear?
- The above actually happens to be a design pattern! Strategy

CONCLUSION

- **Today** we have clarified terminology, remembered basics and principles of OOP, saw where design patterns come from, and where exactly they can help, reviewed sources and consequences of design choices, understood diversity of design patterns, looked at some how-tos and examples

✓ Lesson 1: Introduction

⏳ Lesson 2: Visitor pattern

⏳ Lesson 3: Iterator pattern

⏳ Lesson 4: Adapter pattern

⏳ Lesson 5: Abstract classes

⏳ Lesson 6: Abstract factory pattern

⏳ Lesson 7: Decorator pattern

⏳ Lesson 8: Revision

- **Thanks for your attention!**

- Back-up slides

BACK-UP SLIDES