Applied Object Oriented Design

Introduction

Mikael Svahnberg*

November 14, 2024

1 About Me: Mikael Svahnberg

- Assistant Professor, PhD in Software Engineering
- mailto:Mikael.Svahnberg@bth.se
- https://sites.google.com/site/mikaelsvahnberg/
- Interests:
  - Software Architectures, Software Architecture Evaluation, Software Architecture Evolution, Requirements Engineering, Large Scale Requirements Engineering, Market-Driven Requirements Engineering, Software Product Lines, Software Reuse, Empirical Research Methodology, Software Engineering Decision Support, Static Code Analysis, Software Architecture Reconstruction

*Mikael.Svahnberg@bth.se

2 Discuss: Course Charter: PA1482 Discussion

The following learning outcomes are examinded in the course: **Knowledge and understanding** On completion of the course, the student will be able to:

- In depth explain basic concepts and techniques in object oriented programming and design.
- Understand and in depth explain design and program code for a simple object oriented system.

Competence and skills On completion of the course, the student will be able to:

- Structure a problem solution for a smaller system witht he help of classes and present it in a class diagram.
- Apply design patterns to create a maintainable object oriented design.
- Implement a small object oriented system according to a specific design.
- Use the standard libraries for the programming language.

Judgement and approach On completion of the course, the student will be able to:

- Analyse and discuss a design and program code for potential improvements.

3 Course Structure

Lectures

- *Lecture Plan* on Canvas
- Contains reading instructions

Assignments

0. Early Design Sketch
   - Not graded
   - Early ideas on how to design the system, to get feedback
   - **Used as three week Roll Call**
1. Software Design
   - Graded (see rubric on Canvas)
   - Design of a system, using design patterns.
2. Implementation
   - Graded (see rubric on Canvas)
   - Implement the designed system

- Working Software? Adherence to design?

Assignment Submissions

- Submit your work in groups of 3-5 students.
- Deadlines on Canvas
- Resubmissions:
  - Within two weeks after course ends.
  - Same dates as the re-exams

Written Exam

4 Literature

- Barnes, Kolling "Objects First with Java", Sixth Edition, Pearson 2016. ISBN (Global Edition): 978-1-292-15904-1
- Freeman & Robson "Head First Design Patterns", 2nd Edition, O'Reilly 2020. ISBN: 9781492078005

5 Tools

- You will need a Java Development Kit
  - e.g. OpenJDK `https://openjdk.org/` .
  - May be possible to install as a package inside VSCode.
- You may also wish to install a C/C++ compiler
  - e.g. gcc `https://gcc.gnu.org/`
- Make sure that the Java version you install is compatible with your IDE

  - e.g., BlueJ requires Java 11+.

- For the lazy, this is as good an opportunity as any to getting to know a `make` tool.
  - `https://www.gnu.org/software/make/`
  - `https://cmake.org/`

Examples of Development Environments:

- VS Code (Probably already installed)
- BlueJ `https://www.bluej.org/`
- VS Codium `https://vscodium.com/`
- IntelliJ `https://www.jetbrains.com/idea/`
- Emacs `https://www.gnu.org/software/emacs/`
- ...
- Vim `https://www.vim.org/`

Examples of UML Modelling tools

- PlantUML `http://plantuml.com/`
- IntelliJ `https://www.jetbrains.com/help/idea/class-diagram.html`
- StarUML: `http://staruml.io/`
- VisualParadigm `https://www.visual-paradigm.com/`
- ...

6 Outline

Remainder of this presentation introduces

- Object Oriented Programming
- Object Oriented Analysis and Design
- Design Patterns and Design Principles
- Getting Started [with Java and C++]

7 Object Oriented Programming

1. Background: Data Representation and Manipulation

   - Computer Programs are essentially about *Data*, e.g.
     - Text
     - Bank Account Information
     - Demographics
     - Warehouse Inventory
     - Medical Journals
     - Collections of Live or Historical Measurements from some device
     - Images, Video, ...
     - Maps
     - ...
   - Some data only exist within the realm of a computer program
   - Other (most) data represent something in the real world

   Challenges

   - *Represent* the data accurately
   - *Manipulate* the data consistently
   - *Store* data sufficiently durably

2. Representing Data

   - Simple datatypes: *int, float, char*
   - Compound datatype: *person, bank account, text document, position*

   Challenge

   - We usually do not only store *one* datum; we have collections of data, e.g.
     - *temperature readings every minute for the past 24 hours*
     - *persons currently employed by this company*
     - *patients in Sweden*

3. Manipulate Data

   - Program Structuring ensure that all code that manipulates a certain type of data is found in the same place.
     - All code that manipulates X is in this directory.
     - All code that manpulates X is in this file.
     - All code that manipulates X have the moniker `X` in their name.
     - ...
     - *Only* code with an `X` in their name is allowed to operate on X.
     - code with an `X` in their name is only allowed to operate on a single instance of X.
     - ...

- The data representation may make it easier or harder to manipulate the data *consistently*.
  - All parts of X start with `X` , .e.g. `PersonName`, `PersonPhoneNumber`, `PersonAddress`,
  - Position `i` in all collections represent part of the same datum, e.g. `PersonName[1]` has the corresponding `PersonPhoneNumber[1]`.

Or, we may take a different approach.

4. Objects: Representing Compound Data Types

```
struct Person {
  char name[50];
  char phoneNumber[20];
  struct Address address;
};

struct Person p1;

let p1 = {
    name: "",
    phoneNumber: "",
    address: {},
};
```

- In both these examples, we have created an *Object* `p1`.
- `p1` contains three datum; name, phoneNumber, and address.
- We can create another Object, `p2` which may contain a different name, phoneNumber, and address.
- In fact, we can create a whole collection of `Person` objects.

5. Objects: Encapsulating Behaviour

- This is a nifty extension to the built-in data types, we can now create our own types.

However:

- we still have to keep track of where we put the code to manipulate these data.
- all data is still available to anyone, we cannot protect access.

Encapsulation

- Cleaner public interfaces of object/class/package/subsystem/system/service
- Protect data from unsanctioned access
- Focus each object/class on *their* task, remain unaware of other objects and tasks.
- *Easy To Change*

Example: How to get from *Victoria Station* to *Paddington Station*?

- Know every road and roadblock along the way?
- Know the bus-routes that will take you there?
- Know which tube-line to board?
- Know how to wave down a taxi?
- *Do not* know how to drive a car in London traffic.
- *Do not* know whether there is enough fuel in the taxi.
- *Do not* know how other travellers will get there, their names, or anything else about them.

6. Classes describe Objects

   - Java and C++ are *typed languages*
   - Every item of data must have a type
   - At the very basic level it is used to allocate sufficient memory for the data item.
   - The language enforces type consistency, if you'll let it.
     - (A `Person` can only be accessed as a person and not e.g. as a `Car`).
   - Developer-defined data type **Class**
   - describes the structure of the data type
   - gathers methods (functions) that operate on the data.
   - Encapsulates methods and parts of the data type to create a cleaner interface.
   - *Blueprint for creating objects*
   - *Description for what a developer may do with an object*
   - The programming language helps developers to "stick to the script"

7. One Class, many Objects

```
public class Person {
  private String name;
  private String phoneNumber;
  private Address address;

  // Constructors and methods
  // ...
}

// ...
Person p1 = new Person("Ada", "1", "Newstead Abbey");
Person byron = new Person("George Gordon", "0", "Newstead Abbey");
Person[] students = new Person[25];
System.out.println(new Person("Coleridge", "2", "Coleridge Cottage"));
```

- Same class is used to create many objects
- Each object contains the same attributes (variables), but with their own values.
- Each object contains the same set of methods (functions).
- Each object contains all the necessary data to describe *one and only one* instance of that type.
- You *may* have a variable that reference an object.
- The name of this variable is unknown to the object itself.
- The variable name can (and often will) change as you pass an object around.

8. Summary

- **We use objects to represent the real world.**
  - **Reduced cognitive gap (Real World Object ⇔ Object in Computer Program)**
- Classes describe Objects
- Classes, or Types, are necessary in programming languages such as Java or C++.
  - Encapsulates a compound data type
  - Encapsulates associated behaviour
- Object Oriented Programming with Classes is *one* language design choice
  - Other languages (e.g. JavaScript) do not *need* classes but they help readability and maintainability.
  - Other languages (e.g. Clojure) do not encapsulate behaviour together with data
    * focus instead on describing data types and their relations
    * made possible because data is immutable && code structure is enforced in other ways.

8 Object Oriented Analysis and Design

1. Growing Systems

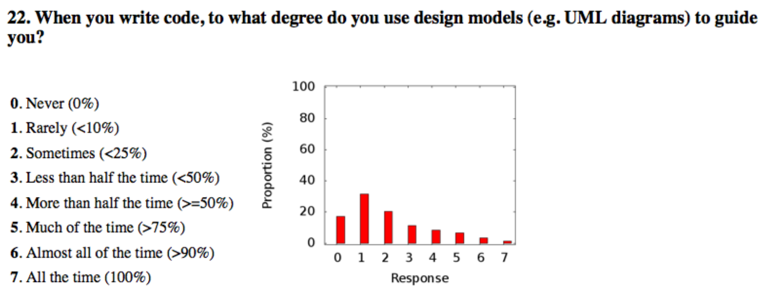
- OOP is a good start, but as systems grow we run into challenges.
- Which Real World Objects should be represented as Program Objects?
- What are the responsibilities for each object?
- Which Objects need to collaborate to solve some task?
- How to get an overview of all the classes?
- What hinders maintainability? What facilitates maintainability?
- How to make best use of the entire toolchain, including the programming language itself?

- Object Oriented Analysis
  - Problem domain and requirements
  - *Objects* in the problem domain
- Object Oriented Design
  - Logical Software Objects (with attributes and methods, plus collaborations)
- OO Construction/Implementation

2. Discuss: Why Bother About Modelling — Discussion

   T. Gorschek, E. Tempero, L. Angelis, *On the use of software design models in software development practice: An empirical investigation*, in Journal of Systems and Software 95(2014):176–193.

   - TL;DR: Nearly 4000 industry practitioners were asked "Do you model?".
   - Answers ranged from "no" to "hell no!".
   - … **There is, of course, more to this story.**



3. Why Bother About Modelling

   - In the freetext answers a different story emerges:
     - They do use sketches, informal models, casual diagrams, etc, but not formal UML.
   - Common explanations:
     - "Only for very complex designs, sometimes"
     - "Only use initially then start coding (diagrams not kept/updated)"
     - "Enables visualisation of the big picture/high level"
     - "Other types of models but not UML"
     - "Use models to communicate and coordinate with other developers"

   $\sum$ Models are not used as researchers expect.

   - Instead they are used for **conceptual analysis and exploration, problem solving, visualisation, and communication.**

4. RUP/UML

   - Unified Modelling Language (**UML**)

  - A traceable chain of different models from requirements to code/test.
    * Each model is transformed to a [more detailed] model that is closer to the end-product.
    * Do this fully, and you have *Model-Driven Development* (Which we won't do in this course)
  - UML is *one* set of models; there are many others, especially for different domains.
- Rational Unified Process (**RUP**)
  - the process used to, whith the help of UML diagrams step by step increase the understanding of which system to build.

Process:

(a) **Understand Requirements** Use Case Diagrams / Use Cases

(b) **Understand Problem Domain** Conceptual Model / Domain Model

(c) **Translate from *user actions* to *system calls*.** System Sequence Diagram

(d) **Describe *Object Interactions*.** Sequence Diagrams / Interaction Diagrams

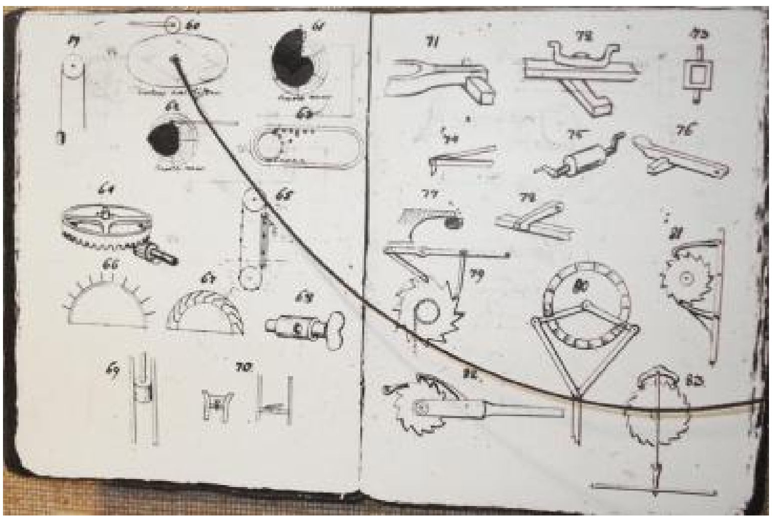
(e) **Establish an overview over what to build** Class Diagram

(f) **Gradually build the system** Goto (4)

In this course, we focus on *Interacting Objects* and the corresponding *Class Diagram.*

9 Design Patterns and Design Principles

1. Design Patterns



(Christopher Polhem, *Mekaniskt Alfabet*, ca 1697)

- Design patterns are *reusable solutions to known problems*
  - ... with known consequences
- There is nothing that *requires* you to use design patterns; they are a convenience.
- Design patterns focus primarily on structure (class view), and interaction (sequence diagrams)

2. Design Principles

- Fundamendal Object Oriented "rules" for how to create a flexible and maintainable design.
  - "They're more like guidelines, really..."
- As opposed to Design Patterns, you use these *all the time* and *implicitly.*

Responsibility Driven Design
  - Principles for assigning responsibility for *knowing* or *doing* to the right object.
  - Overall goal: Easy To Change (ETC)

3. When & Where

- Design Principles are used whether you have made an explicit object oriented design or not.
  - They can be seen in the design, but will be equally important when you jump straight into programming.
- Design Patterns are *more often* designed first and implemented second.
- Design Patterns are especially important in "classic" object oriented languages.
  - Less important (but still useful) in duck-typed languages such as JavaScript or Clojure.
  - (Clojure in particular takes a completely different path to achieve the same flexibility.)

10 Getting Started without an IDE

1. Compiled Languages

- Java and C++ are *compiled* languages
  - A tool translates your source code into something machine readable *before* you run it.
  - ... as opposed to e.g. Python, PHP, or JavaScript, where a tool *interprets* your source code at runtime.
- This adds a few steps to the toolchain, i.e. `write \rightarrow compile \rightarrow maybe-more-compilation \rightarrow run`
- If you use a modern IDE you probably do not have to worry too much about this.

Benefits

- Efficiency (performance *and* e.g. memory efficiency)
- Compiler will syntax-check *all* your code before you deploy
  - *s/runtime errors/compiation errors* – Fix your code before you deploy
  - Fosters a more stringent approach to programming
- Source code is not accessible to end-users
- Smaller size of shipped program
- Access to low-level APIs on your computer, e.g. Operating System, CPU, network, disk, memory, etc.

Challenges

- Compiled code *may* be platform dependent (C++ is, Java isn't)
- More complex toolchain
- Difficult to edit a running program on the fly
- Some programming language constructs are difficult to achieve, e.g. homoiconicity.
  - (But not impossible; Clojure accomplishes this)

2. Basic Steps Java

   (a) Write Java Source Code
   (b) **Compile** to machine independent `bytecode`
   (c) **Interpret** `bytecode`

   Optional: Package the bytecode files into a `JAR` file.

   C/C++

   (a) Write Source Code
   (b) **Compile** to `runnable binary` or `relocatable machine code`
   (c) **Link** relocatable machine code to a `runnable binary`
   (d) **Run** the `runnable binary`

3. Getting Started with Java

   (a) Appendix E in Barnes and Kölling
   (b) Make sure you have a JDK/JRE installed
   (c) Start a terminal
   (d) use `javac` to compile a `.java` file
       - `javac Start.java`
   (e) use `java` to run a `.class` file
       - `java Start`

   Basic rules:

- Each class is defined in a file *with the same name as the class*
  - Class names are in principle case sensitive (lower/upper case letters)
  - In practice, some filesystems are not; this may create problems.
  - ⇒ Use proper and unique names, and make sure the file and class are spelled the same way.
- The "root" Class, where you want the program to start *must* have a `main()` function:

```
public class Start {

  public static void main(String [] args) {
  }

}
```

- It is good practice to keep this function *very* small; a simple printout and an object creation or two.

0. Getting Started with C++

   (a) Make sure you have a C++ compiler installed.
   (b) Start a terminal
   (c) Use your compiler (e.g. `g++`) to compile your `.cc` and `.hh` files.
       - `g++ start.cc -o start`
   (d) Run the resulting program as usual:
       - `./start`

   Basic Rules:

   - The compiler doesn't care, but:
     - keep *class declaration* in a `.hh` - file
     - keep *class definition* in a `.cc` file with the same name as the `.hh` file.
   - *somewhere* in the compiled program there must be a `main()` function.

```
int main() {
  // ...
  return 0;
}
```

   - It is good practice to keep this function *very* small; a simple printout and an object creation or two.
   - It is also good practice to keep this function in an easily recognisable file, e.g. `main.cc` or `start.cc` .

1. Build Tools (multi-file project)

- `javac` will follow class dependencies until it encounters a class that does not need to be compiled (source code unchanged).
- C++ compiler will just do one file at the time.
- Can use wildcards `javac *.java` to re-build everything.

Build tools save time

- Your IDE can help you (e.g. a project in Visual Studio)
- Be a Good Friend (TM), create a `makefile`.
  - `https://makefiletutorial.com/`

```
VARIABLE = value

target: dependency
  Command to build target
```

2. Generic makefile for Java

```
JAVAC=javac
sources = $(wildcard *.java)
classes = $(sources:.java=.class)

all: myProgram

clean:
  $(RM) *.class

run: myProgram
  java Start

myProgram: $(classes)

%.class: %.java
  $(JAVAC) $<
```

3. Generic makefile for C++

```
CC = g++
CFLAGS = -g -Wall
INCLUDE = -I.
TARGET=myProgram
EXT = .cc
SRCS = $(wildcard *$(EXT))
OBJS = $(SRCS:$(EXT)=.o)


all: $(TARGET)

run: $(TARGET)
```

```
  ./$(TARGET)

$(TARGET): $(OBJS)
  $(CC) -o $@ $^ $(CFLAGS) $(INCLUDE) $(LDFLAGS) $(LIBS)

%.o : %$(EXT)
  $(CC) -c $< -o $@ $(CFLAGS) $(INCLUDE)

clean:
  $(RM) $(OBJS) $(TARGET)
```

- This will get you started, but the build file will need to be extended.
- Note that changes to `.hh` files will not be considered with this. Options:
  - Explicitly define `DEPS= file1.hh file2.hh` (bad idea)
  - Make sure you touch the right `.cc` - file (better idea, but may miss places where the file is included)
  - Plan your classes and APIs beforehand to minimise changes (even better idea)
  - Advanced `makefile-fu` to fix this once and for all with `g++ -M` .

4. Summary
   - For now, `BlueJ` is a good abstraction to learn Object Orientation.
   - Grok the command line interface
     - Increase your understanding of what is really going on
     - Will help you later in your career
     - Will enable you to work with build systems
     - Easier to share your project with troglodytes who refuse to use a modern IDE.
   - In an IDE, much of this is hidden.
   - Read the error messages!

11 Next Lecture: Programming Fundamentals

- Barnes & Kölling Chapter 1, Objects and Classes
- Barnes & Kölling Chapter 2, Understanding Class Definitions
- Objects, Classes, Methods, and Attributes
- Built in data types
- Fundamental Input and Output
- Conditional Execution: `if`
- Design Philosophy: *Program objects represent real world entities*
- Design Principle: *Low Coupling*