SIT771 – Lecture 4

Modelling and object/class relationships

DEAKIN UNIVERSITY

Further reading

- Paul Deitel and Harvey Deitel (2018). Visual C# how to Program (6th ed). Pearson. Ebook on Deakin Library – Chapter 4.

- Bernhard Rumpe (2016). Modeling with UML: Language, concepts, methods. Switzerland Springer. Ebook on Deakin Library – Chapter 2.

Outline

In this lecture...

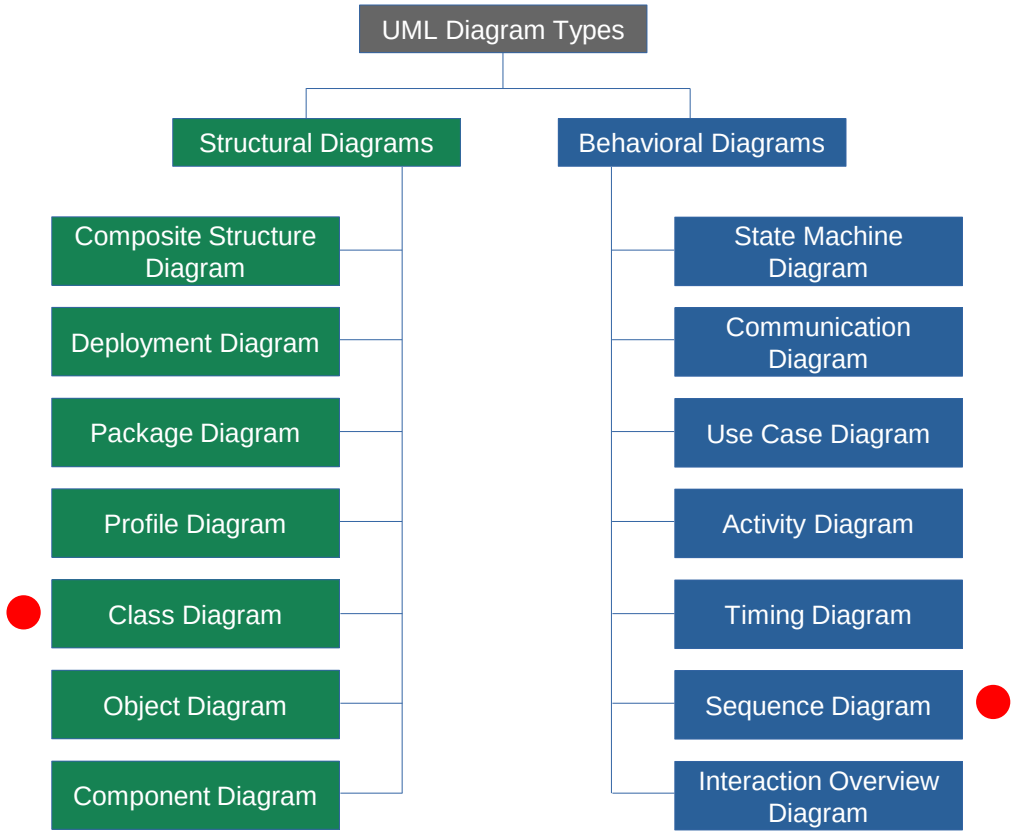
- Modeling
- Class relationships
  - Association
  - Aggregation
  - Composition
- Design
  - Main concepts
  - Procedure
  - Evaluation

MODELING

Unified modeling language

Referred to as UML

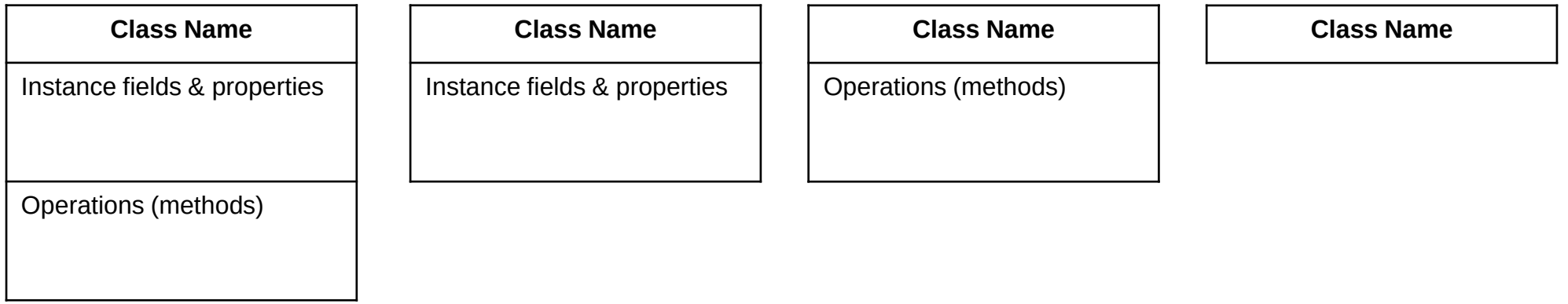
- Unified Modeling Language…
  - Is the de facto modelling tool for OOP
  - Models both...
    - *Behavioral* aspects of a system
    - *Structural* aspects of a system
  - Bridges between analysts, designers, coders, testers, etc.
  - Has a set of integrated diagrams to model and document software artifacts



Unified modeling language

UML – Class diagrams

- Are static structural diagrams that show…
  - the system's classes (names required)
  - the class attributes (optional)
  - the class methods (optional)
  - the relationships among classes (optional)



Unified modeling language

UML – Class diagrams (cont.)

- Attributes (instance fields & properties)
  - Syntax: visibility name : type multiplicity = default_value {property}
  - Visibility, indicates the visibility (access modifier) of the attribute using special symbols:
    - '+' for public; or
    - '–' for private; or
    - '#' for protected;
  - Multiplicity (optional), indicating how many elements the attribute refers to (default=1)
  - Default_value (optional), an equals symbol (=) followed by the attribute's default value
  - Property (optional), surrounded by braces ('{' and '}'), indicates any additional properties about the attribute, e.g., readOnly
  - Static attributes are underlined

Unified modeling language

UML – Class diagrams (cont.)

- Operations (methods)
  - Syntax: visibility name(parameters) : return_type
  - Visibility, is the same as for attributes
  - Name, is the name of the operation
  - Parameters are optional, the parameters to the operation use a similar syntax to attributes
  - Return_type, indicates the data type; blank for no return value (void)
  - Static operations are underlined

BankAccount
- customer : string - balance : double = 0.0
+ Deposit(amount : double) + Withdraw(amount : double)

Customer
- name : string - address : string [1..2]
+ GetName() : string + SetName(name : string) + IsBirthday() : boolean

CLASS RELATIONSHIPS

Justification and types

Why?

- In real scenarios, there are relationships among classes, e.g.,
  - Cats are a specific type of pets
  - Dogs are another type of pets
  - Both cats and dogs have tails
  - Humans feed pets, and pets please humans

What?

- Thus, in software system design, there are at least four types of class relationships
  - Association
  - Aggregation
  - Composition
  - Inheritance (does not result in any object relationship) [*next week*]

Association

Definition

- Is a semantically weak relationship between otherwise unrelated classes.
  - The objects have their own lifetime and there is no owner.
  - The objects of associated classes can be created and destroyed independently.

- E.g., the relationship between a **doctor** and a **patient** where...
  - Each doctor and patient object has its own life cycle and there is no “owner” or parent.
  - A doctor can be associated with multiple patients.
  - One patient can visit multiple doctors.

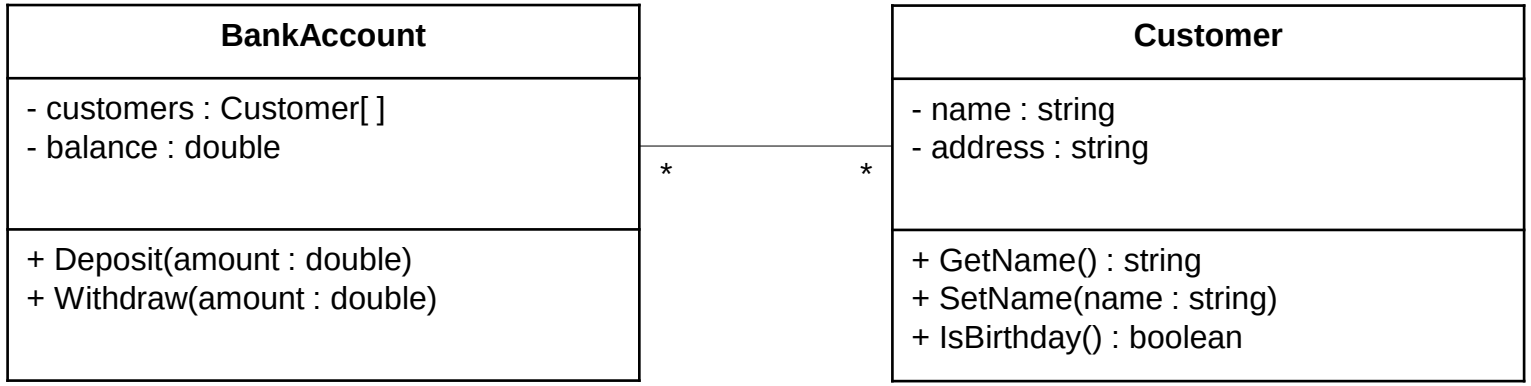
```
18  public class Doctor
19  {
20      private Patient[] _patients;
21      // ... other members of the Doctor class
22  }
23
24  public class Patient
25  {
26      private int _id;
27      private string _name;
28      private int _age;
29      // ... other members of the Patient class
30  }
```

Association

UML representation

- In UML, an association relationship is represented by a solid line. An association relationship can be represented with its **cardinality**…
  - One-to-one
  - One-to-many
  - Many-to-many

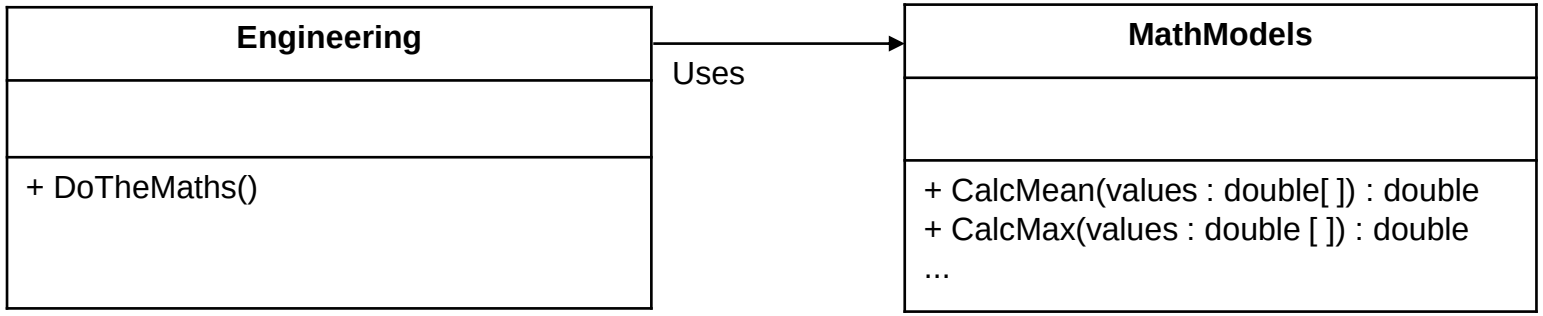
- Cardinality indicators
  - 0..1, zero or one
  - 0..* (*), zero or more
  - 1..* (+), one or more
  - 0..n, zero to n (n>1)
  - 1..n, one to n (n>1)
  - 1, only one – n, only n (n>1)



Association

Example

- Temporary object/method use:
  - The **MathModels** objects are used within the **Engineering** class to calculate specific measures...
  - Arrowheads can be used to show the direction of navigation.




```
173    public class MathModels
174    {
175        public double CalcMean(double[] values)
176        {
177            double _mean = 0;
178            //calculate mean ... and put it into _mean
179            return _mean;
180        }
181
182        public double CalcMax(double[] values)
183        {
184            double _max = 0;
185            //find max ... and put it into _max
186            return _max;
187        }
188    }
189
190    public class Engineering
191    {
192        public static void DoTheMaths()
193        {
194            double[] values = new double[10];
195            MathModels _maths = new MathModels();
196            double _max = _maths.CalcMax(values);
197        }
198    }
```

Association

Method calls and links

- Methods can communicate via arguments
- The links are usually uni-directional…
  - One method can invoke the services of another method, but not vice-versa.
  - The direction does not prevent data from traveling in both directions, e.g., through **parameters** and **return** values. The parameter modifiers are:
    - **out**: call by reference, no initialization needed
    - **ref**: call by reference, initialization needed
    - **in**: the called method cannot change its value

```
222  public class TestArgs
223  {
224      private static void caller()
225      {
226          int x = 10;
227          called_1(out x);
228          called_2(ref x);
229          called_3(in x);
230      }
231
232      private static void called_1(out int returnOnlyVar)
233      {
234          //Console.WriteLine(returnOnlyVar); Error: Use of unassigned
235          //parameter 'returnOnlyVar'
236          returnOnlyVar = 200;
237      }
238
239      private static void called_2(ref int bidirectionalVar)
240      {
241          Console.WriteLine(bidirectionalVar);
242          bidirectionalVar = 200;
243          Console.WriteLine(bidirectionalVar);
244      }
245
246      private static void called_3(in int passOnlylVar)
247      {
248          Console.WriteLine(passOnlylVar);
249          //passOnlylVar = 200; ; Error: Cannot assign to variable 'in int'
250          //because it is a readonly variable
251      }
252  }
```

Association

Association classes

- A class that is part of an association relationship between two other classes.
  - They provide additional information about the relationship.
  - They are identical to other classes and can contain operations, attributes, as well as other associations.

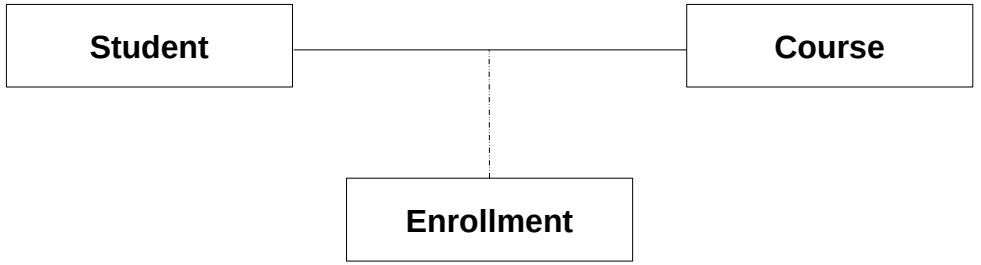
- E.g., the relationship between **student**, **course**, and **enrollment** classes...
  - The objects of class student can enroll in a course object.
  - An enrollment object further defines the relationship by providing section, grade, and semester information related to the association relationship.

```
198     class Course...
202
203     class Student...
207
208     //The association class bringing together students and courses
209     class Enrollment
210     {
211         private Student _student;
212         private Course _course;
213
214         public Enrollment(Student student, Course course)
215         {
216             this._student = student;
217             this._course = course;
218         }
219     }
220 }
```

Association

Association classes – UML representation

- In UML, an association class is connected to an association by a dotted line.
- E.g., the relationship between **student**, **course**, and **enrollment** classes...
  - The objects of class student can enroll in a course object.
  - An enrollment objects further defines the relationship by providing section, grade, and semester information related to the association relationship.



Aggregation

Definition

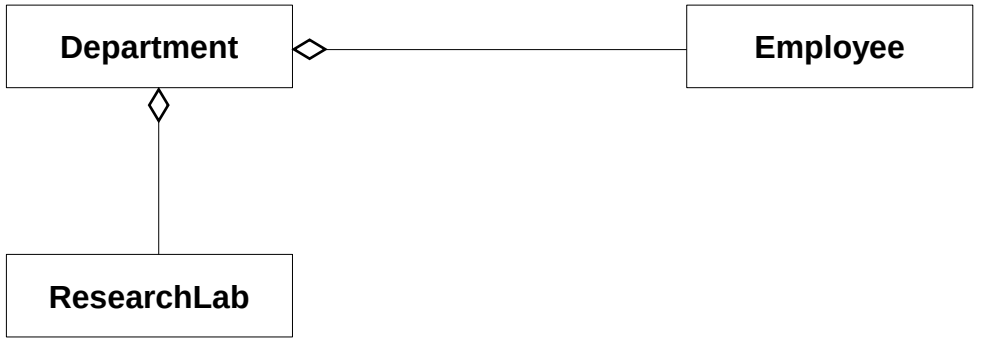
- Is a specialized form of association between classes where…
  - A class object has its own life cycle, but there exists an ownership.
  - A whole/part or parent/child relationship exists which may or may not denote physical containment.
  - The whole or parent (i.e., the owner) can exist without the part or child and vice versa.
- E.g., the relationship between classes **employee** and **department** in an organization...
  - An employee object may belong to one or more departments in an organization.
  - If an employee's department is deleted, the employee object will not be destroyed; it can live on.
  - A department may "own" an employee, but the employee does not own the department, i.e., the relation cannot be reciprocal.

Image source: Deakin University, Wikipedia

Aggregation

UML representation

- In UML, aggregation relations are shown using a line and a “hollow” diamond.
- E.g., for the example of classes **employee** and **department**, adding a class **research lab** too...



Composition

Definition

- Is a specialized form (a strong type) of aggregation where...
  - If the object from the parent class is destroyed, the child object will cease to exist too.
  - A whole/part or parent/child relationship exists.
  - The life cycle of the part or child is controlled by that of the parent that owns it.

- E.g., the relationship between the classes **house** and **room**…
  - A house object may be composed of one or more rooms.
  - If the house object is destroyed, then all rooms that are part of that house will be destroyed too.

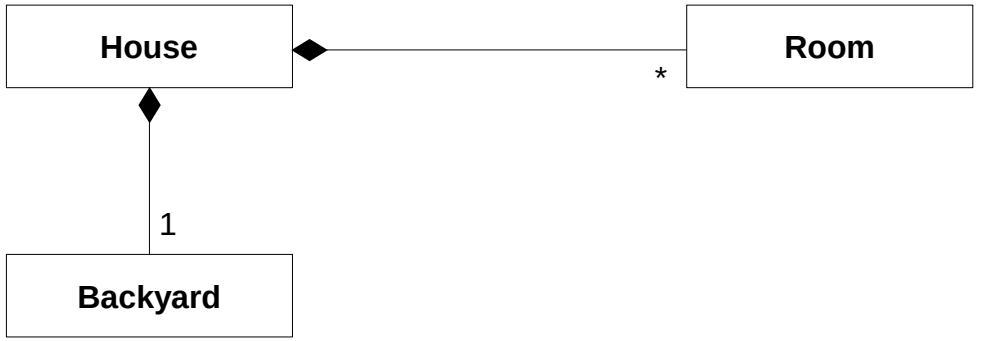
```
32    public class Room...
36
37    public class House
38    {
39        private Room _room;
40
41        public House()
42        {
43            _room = new Room();
44        }
45    }
```

Image source: https://lovehomedesigns.com/

Composition

UML representation

- In UML, composition relations are shown using a line and a “solid” diamond.
- E.g., for the example of **house** and **room** classes, adding a class **backyard** too...



DESIGN

Main concepts

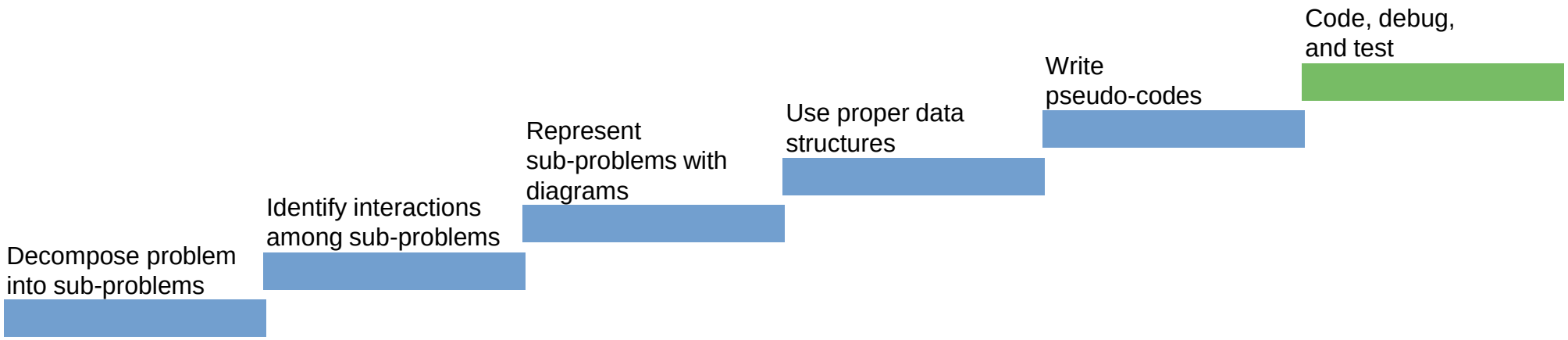
In software engineering...

- Design...
  - Has a focus on identifying main primitive components and functionalities and their relationships within a software solution.
  - Is not coding!
  - Has the following general concepts…
    - **Abstraction**, generalize and reduce information content for specific phenomena.
    - **Refinement**, elaborate more and improve the first design.
    - **Modularity**, divide software into components called modules.
    - **Information hiding**, hide information from other modules or classes where not needed.
    - **Refactoring**, reconstruct the design to i) reduce complexity, and ii) simplify design without affecting the behavior or its functions.
    - **Pattern**, repeat a design as a solution to a common recurring problem.

Procedure

Take design in steps...

- Software design is a step-by-step process that requires attention to each step before the actual coding of the software begins. Coding without a proper design is premature, any success may not be repeatable, and changes will result in tedious code rewrites.



Evaluation

Good design features

- A good design should be...
  - **Extensible**, new components/modules can be easily added to it.
  - **Adaptable**, it can adapt to changes easily.
  - **Reusable**, the design (or parts of it) can be used again with little or no modifications for similar problems.
  - **Iterative**, the design is done in iterations where in each iteration, the design becomes more accurate.

- Also, a good design should…
  - not suffer from tunnel vision (should see alternatives)
  - not reinvent the wheel
  - minimize the distance between the software and the real world
  - accommodate change as it arrives

Evaluation

Good design vs. bad design

- Code smells ~ design flaws in software
- Find where **refactoring** can help, e.g., ...
  - simple renaming
  - changing inheritance structure
  - moving responsibilities or features between classes
  - converting members (e.g., field to property)
  - decomposing God Classes
  - decomposing complex methods

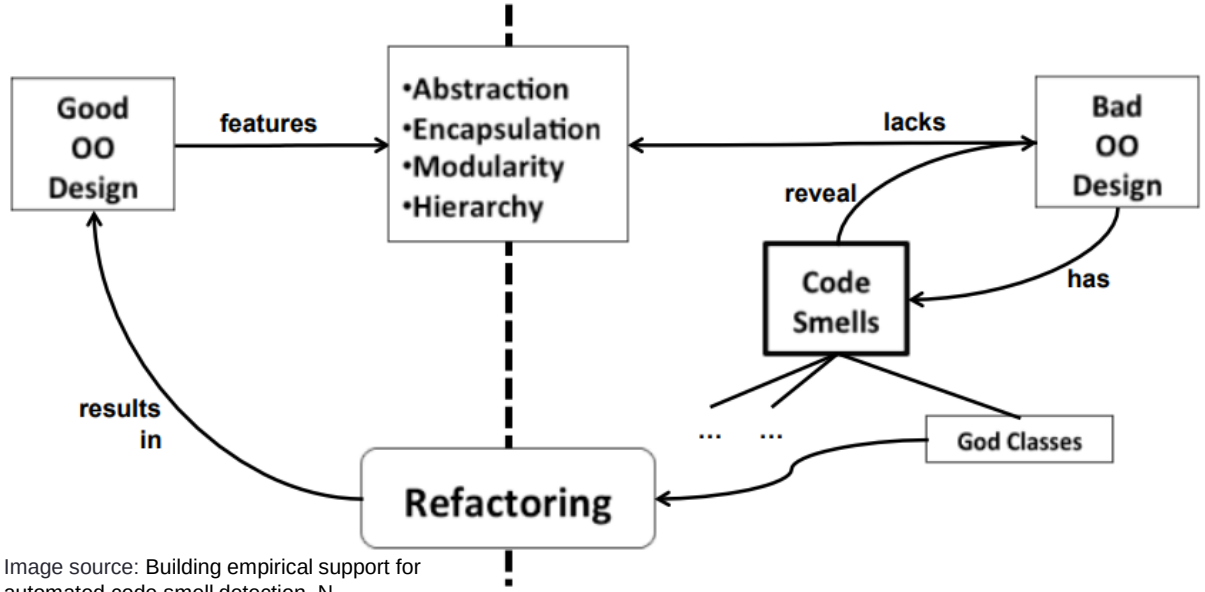


Image source: Building empirical support for automated code smell detection, N. Zazworka and M. Shaw, 2020.

Evaluation

From the class interaction's perspective...

- A good design focuses on **cohesion** and **coupling** concepts
  - Low coupling, better design
  - High cohesion, better design

- Some questions to ask…
  - Can the design (elements) be actually built and implemented?
  - Will the design elements add up to the entire desired system?
  - Are there any risks in implementing the design elements?
  - Read more in "Evaluating the software design of a complex system of systems" by Blanchette et al., 2010 (https://resources.sei.cmu.edu/asset_files/TechnicalReport/2010_005_001_15128.pdf).

Evaluation

From the outcome's (code's) perspective...

- There are several measures that could be used to evaluate code
  - Does it meet the requirements?
  - Does it behave as expected for a variety of inputs: both expected and unexpected?
  - Is it fast enough? When is it slow? How frequent is the slow case?
  - What are the performance bottlenecks?
  - Other quality measures: include...
    - Is the code easy to maintain?
    - Is the code safe/secure?
    - Is the code readable?
    - Is the code formatted well?

Epilogue

THE HARDEST PART OF DESIGN IS KEEPING FEATURES OUT...

DONALD A. NORMAN