Introduction to Java

5. Object Model and Java Classes

Introduction

- The heart of Java programming is the object model
- All code in Java is written in class definitions or associated stuctures
- In this module we will examine
  - The structure of a class including attribute and methods
  - How objects are instantiated from class definitions
  - The design of Java methods
  - The different use cases for instance methods and variables as opposed to static methods and use cases

The Object Model

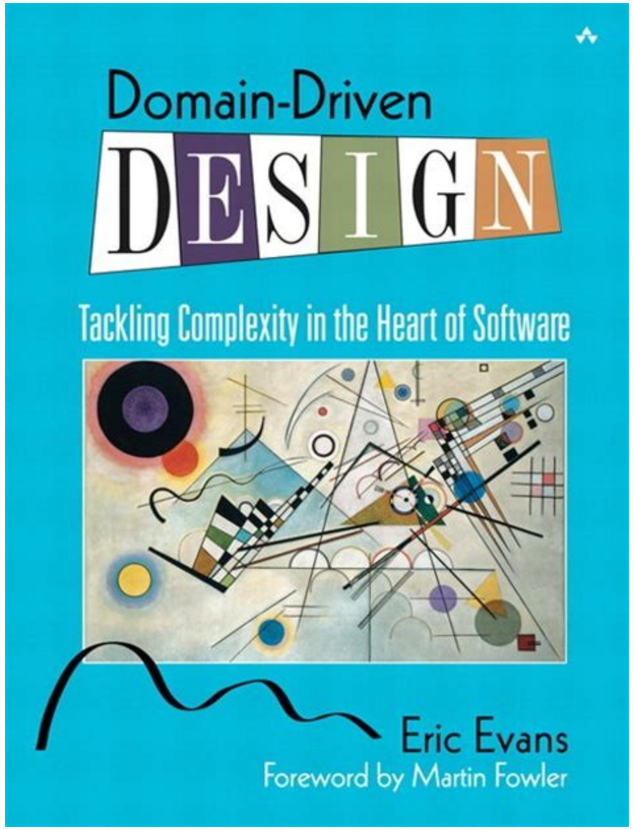
- There is no formal definition of the object model
- However in OO it is generally agreed that
  - Objects have a type and unique identity
  - Object contain a set of data relevant to their type which are called attributes
  - Objects contain a set of instance methods that execute the messages other objects send them
  - The attributes and methods an object has are defined by its type
- A class definition consists of:
  - A set of instance variables representing the attributes of the object
  - An optional set of static variables representing attributes of all the objects of that type collectively
    - *For example, the number of objects in existence of a type is a property of the collection or class, not any individual object in that class*
  - A set of executable functions called instance methods that define the behaviour of objects of that type
  - An optional set of static methods that refer to some functionality of the class as a whole
    - *For example, incrementing the number of objects or a type in existence*

Designing Classes

- Writing classes is not the same as writing a sort method
- Remember that OOP is *iconic* which means that our classes look like what they automate.
  - Or at least what they are motivating influences our choice of classes
- Classes have two *layers*
  - An *interface* through which other classes interact with our class
    - The interface is made up of the public methods for that class (more on that in a bit)
  - An *implementation* which is where our code and data exist
    - The implementation cannot be accessed by anything outside the class
    - As a result, we are free to re-architect, re-design and rewrite our implementation code
    - As long as we keep the interface stable, changing the implementation will not break anything
- The actual classes and their public interfaces are defined during the design process
  - Usually comes out of a high level architecture
  - Defines the roles and responsibilities of the different classes that need to be written
  - As well as the public interfaces the classes will need to expose

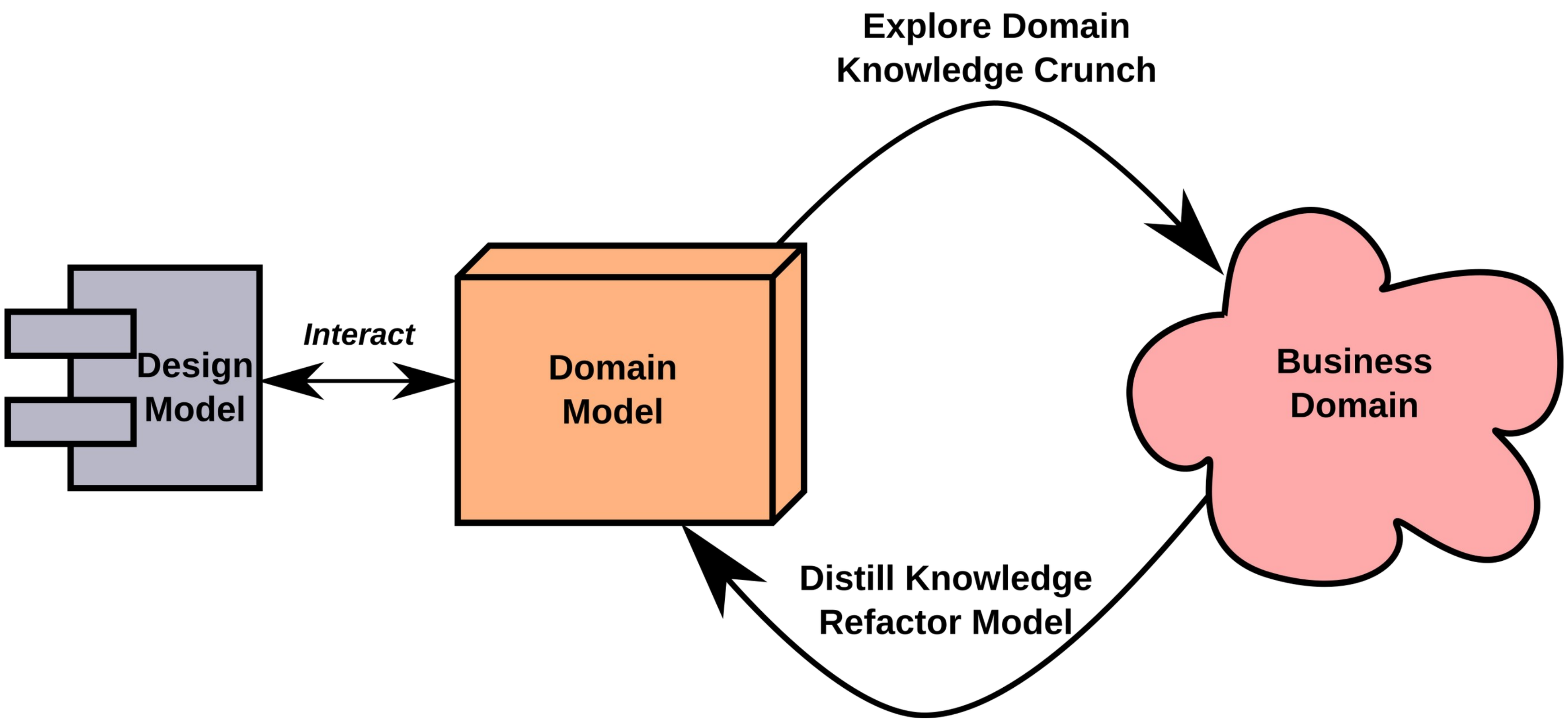
Domain Driven Design

- Developed by Eric Evans to deal with the problem of complexity in OO designs
  - Especially when dealing with complex domains
- The approach is to take deep dives into the domain
  - Refine your understanding of the objects and their relationships in the domain
  - Distill this into a domain model
  - Use the domain model to guide the building of a design model
  - Repeat the process iteratively
  - Called knowledge crunching
  - Consistent with modern design thinking research
- Proven to be highly effective
  - Motivated by the problem of designing micro-service architectures

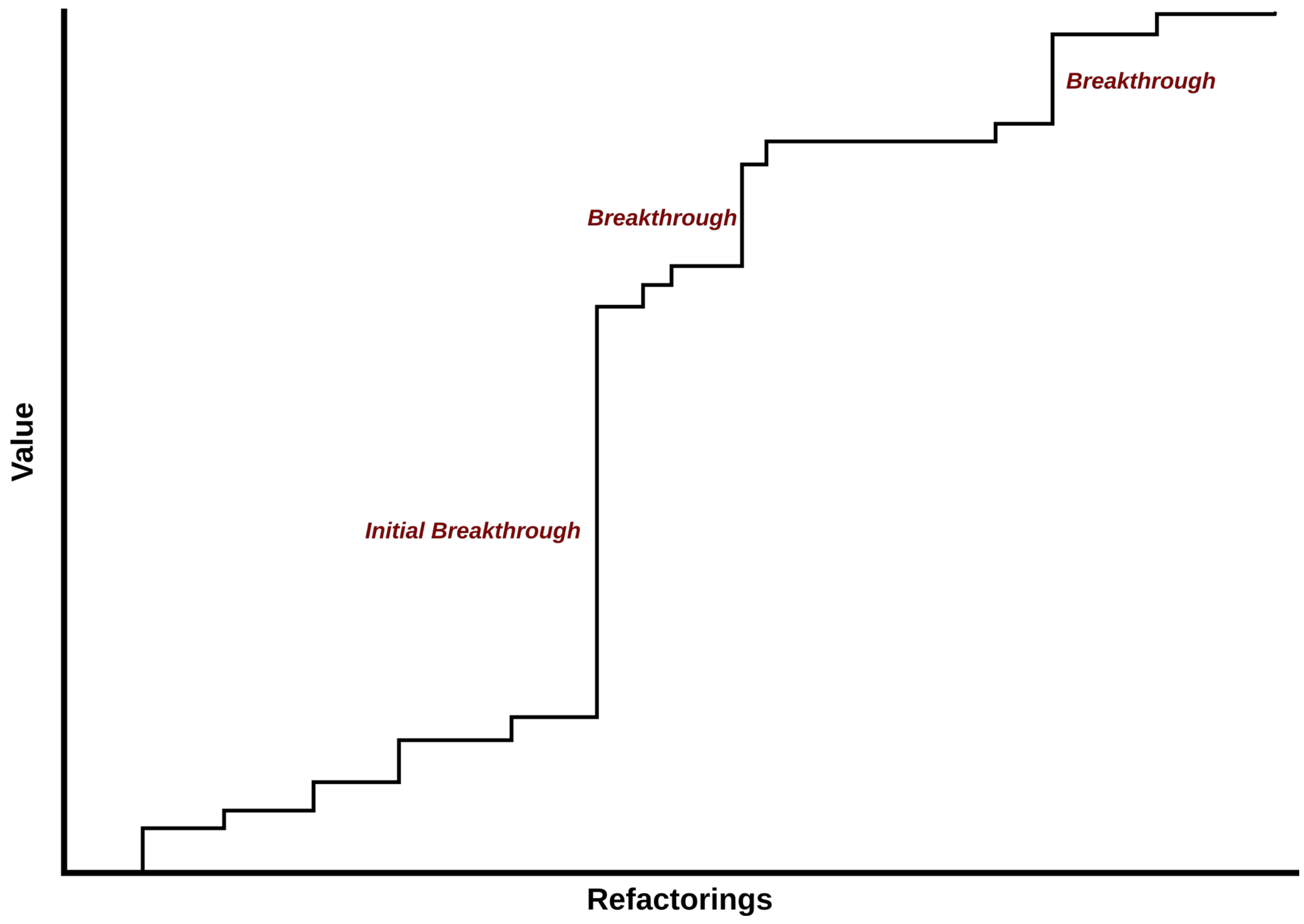


Domain Drive Design

The Knowledge Crunching Iterative Process



Domain Drive Design



Visibility

- There are three visibility modifiers used for both variables and methods
  - **Public** – visible to any object that can see the containing object
  - **Package** – visible to any object *in the same package* as the containing object
  - **Private** – visible only to methods in the containing object (data hiding)
- That means that there are actually two interfaces to each class
  - The *public* interface which are all the methods marked *public*
  - The *package* interface which are all the methods without a *public* or *private* modifier
  - Note that the *public* interface is a subset of the *package* interface

Demo

Data Visibility

Lab 5-1

Data Visibility

Methods

- Methods in Java have two parts
  - The return value
  - The method signature which is made up of the method name plus the argument list
  - For example, all of the following can be used in a class definition because they have different signatures
    - *String convert(int k)*
    - *String convert(int k, int u)*
    - *String covert(boolean b)*
  - However, this would be an error because it is the same signature as a prior method, the return value does not make up part of the signature
    - *float convert(int k)*
  - This is called method overloading or method polymorphism
  - The idea was that similar operations should have similar names for readability
  - Called method polymorphism because the same method has different forms depending on its parameter list

Method Invocation

- When we send a message to an object,
  - We use the notation obj.method() and expect some sort of return value
  - When we call a method in the same class, we typically use the form this.method()
    - The *this* is optional but it is considered good form so that the code is more understandable
  - The same is true for data
    - If we referred to the Student.name instance variable from method in the Student class
    - The form this.name would be used
    - The this is optional but once again, it is considered good form to use it
- All data should be private
  - It is accessed through special methods called *getters* and *setters*
  - Not providing a *setter* for an instance variable makes it read only
  - Not providing a *getter* restricts access to the data to just methods in the class definition
  - The *getter* methods are typically *public*
  - *Setters* also function as validators to check to see if a value being set is legal

Demo

Working with Methods

Lab 5- 2

Method Invocation

Static Members

- Static Variables and data are syntactically the same as instance variables
  - They are just prefixed by the keyword *static*
- Rules for static methods
  - They can only refer to static data or other static methods
  - They may not refer to any non-static data or non-static methods
    - Remember that static mean initialized when the JVM starts
    - Instance variables and methods cannot be referenced until an object is created
- Constants are define using *public static final* variables
  - By convention, constants are always in upper case

Demo

Static Methods and Data

Lab 5- 3

Static Methods and Data

Java™