FIT2099

Design Principles

Topics Covered
Object Oriented Paradigm
Key Design Principles
Technical Debt
Dependencies
Encapsulation
Connascence
Abstraction
Code Smells and Refactoring

Object Oriented Paradigm

Object orientation is a conceptualisation of objects that carry out the program's tasks by sending messages to each other.

Procedural Programming

- A collection of procedures
- Each procedure has its own input and output
- Easier to write than spaghetti code
- Key abstraction is the action
- Bad because
    - Need a larger-scale abstraction for big systems
    - And Action was the primary unit of organisation, not data

Object oriented programming flips this around

Object Oriented Programming

- Unit of organisation is the object
- Objects are instances of classes
- A class defines an interface: Set of messages the objects can receive and what it promises to do in response (actions)
- Better because you can **reduce dependencies as much as possible**
    - Can group things that depend on each other inside an encapsulation boundary

Key Design Principles

Avoid excessive use of literals

- If value changes in future, need to search for every occurrence and change all of it
- Can also have subtle dependencies due to value of constant

Don't Repeat Yourself (DRY)

- Use of repeated code by copying and pasting
- Makes it difficult to maintain, read and test code
- Needs to be done in many places, some repetitions may not be found

Classes should be responsible for their own properties

- Classes should be in charge of themselves
- Info should be stored as close as possible to where it is needed

Reduce dependencies as much as possible

- Or at least grouping elements that must depend on each other inside an encapsulation boundary, keep as much as possible of them private
- Classes can be grouped together inside the encapsulation boundary of a package

Minimise dependencies that cross encapsulation boundaries

- Done by using language constructs that make it impossible to create dependencies that cross boundaries
- Can declare attributes and operations to be private

Declare things in the tightest scope possible

- The tighter to scope in which something is visible, less risk that something can depend on it, the less the risk it will be a future point of failure
- E.g. Declare local variables in loops; use local variables rather than attributes; keep members to implement an abstraction together inside a class; keep classes inside packages

Fail Fast Principle

A system should fail immediately and visibly when something is wrong. This allows a developer to easily find a problem and fix it when it arises, rather than having it stay unknown and cause problems later.
See Debugging, Assertions, and Exceptions

Avoid variables with hidden meanings (Hybrid Coupling)

- Need to avoid variables with hidden meanings
- E.g. pageCount meaning number of pages except -1, meaning an error occurred

Separation of Concerns

See Abstraction

The Dependency Inversion Principle (DIP)

See Abstraction

Single Responsibility Principle

Every module or class should have responsibility for a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.
See Code Smells

Liskov Substitution Principle

- If B is a subclass of A, you should be able to treat an instance of B as an A

See Design By Contract

Command-Query Principle

See Design By Contract

Cohesion and Coupling Principles

Cohesion: The principle of being or doing one thing (as opposed to many things) well.

- Means grouping together code that contribute to a single task
- Low cohesion means that the class does a great variety of actions (broad, unfocused)
- High cohesion means that the class is focused on what it should be doing (only methods relating to the intention of the class)
- Good software has **high cohesion**

Coupling: How dependent two classes are towards each other

- Low coupled classes: changing something major in one class does not affect the other much
- High coupled classes: Difficult to change and maintain the code
    - Classes are closely knit so making a change requires an entire system revamp
- Good software has **low coupling**

See the Single Responsibility Principle

Technical Debt

- Have a piece of functionality to do
    - One is messy by quick
    - One is cleaner but slow - but it will be free of potential problems
- Doing the quick and dirty way sets us up with a **technical debt**
- Incurs interest payments, which comes in the form of extra effort to do in future development
- Inevitable in real systems

- Can be repaid by refactoring

Dependencies

Dependencies
When a piece of software relies on another one.

Indirect Dependencies
Dependencies that cannot be detected by the compiler and only understood by human readers.
Best to be documented or even better: eliminated.
E.g. A literal that means the same in multiple places in code → Solved by putting it as a named constant

Encapsulation
Encapsulation fulfils several definitions:

1. A software development technique that consists of isolating a system function or set of data and operations on the data within a module and providing precise specifications for the module
2. The concept that access to the names, meanings, and values of the methods of a class is entirely separated from access to their realisation
3. The idea that a module has an outside that is distinct from its inside, that it has an external interface and an internal implementation

Benefits of Encapsulation
- Better security of data (through access modifiers)
- Flexible code that is easier to write and maintain

Mechanisms for Encapsulation
Java was made to encapsulate. The basic unit of Java programs is the class. Java can restrict access to things in the class as:
- Within the class only (private)
- Within the package (default/unspecified)
- Only to subclasses (protected)
- No restriction (public)

Encapsulation Boundaries
An encapsulation boundary is something across which visibility can be restricted. (a class, package, even a method).
Any calls to methods not in a class crosses an encapsulation boundary; these need to be minimised.
To enforce encapsulation you can do the following
- Avoid public attributes
- Only make the methods public when necessary
- Keep the class package-private if not needed

- Use protected sparingly (consider using methods rather than attributes)
- Minimise interfaces
- Defensively copy when using getters to eliminate effects from mutability
- Don't expose implementation details, avoid returning the copy in the original data type when a better one can be used
- Be aware and avoid relying on algorithm quirks (eg, when data is returned sorted, incidentally)

Connascence

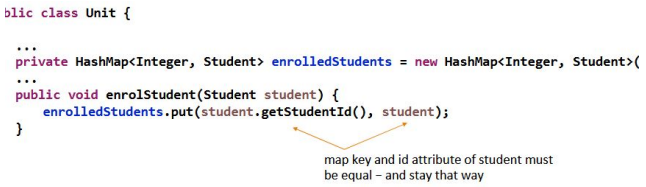
Connascence

- Two elements are connascent if at least one change can be made to A that requires a change to B in order to preserve correctness
- Static Connascence
    - Obvious from code structure
- Dynamic Connascence
    - Obvious from close inspection/execution
    - More concerning

Types of Connascence (weakest to hardest)

- Connascence of Name (CoN)
    - Need things in two places to have the same name (like method names)
- Connascence of Type (CoT)
    - When two things to have the same type
- Connascence of Position
    - Parameter position
- Connascence of meaning/convention (CoM/CoC)
    - For a Watch, 0 means no time
- Connascence of Algorithm (CoA)
    - 

- Connascence of Execution (CoE)
    - ```
      public Watch3() {
          hours = new MaxCounter(24);
          minutes = new LinkedCounter(60, hours);
          seconds = new LinkedCounter(60, minutes);
      }
      ```
- Connascence of Values (CoV)
      ```



Relevance of Connascence

- Connascence means code is
  - Harder to extend
  - More chance of bugs
  - Slower to write in the first place

Contranascence

- When two things are required to be different
- Type of connascence

Resolution

- Minimise overall amount of connascence by breaking system into encapsulated elements
- Minimise remaining connascence that crosses encapsulation boundaries
- Maximise connascence to be within encapsulation boundaries
- **Overall:** You reduce connascence using encapsulation

Abstraction

The act of considering something as a general quality or characteristic, apart from concrete realities, specific objects, or actual instances.

As a developer, this means deciding

- What information do we need in order to represent an item or object
- What we should expose to the rest of the code so they can use this part easily

We use abstraction when we bundle things together and use them

- e.g. Lines of code in a method, data in a class, classes in a package

Abstraction vs Encapsulation vs Information Hiding

- We use **encapsulation** when we bundle things together
- We use **abstraction** when we decide which things to bundle together
  - Or how things should look from the outside
  - What things to encapsulate together
- We use **information hiding** when we use an encapsulation mechanism that doesn't allow access from the outside
  - private /protected modifiers to keep implementation details private
  - Local variables to prevent access from outside the method
  - Defensive copying to prevent external code from accessing internal data structures

- Basically abstraction is a modelling term whereas encapsulation is the implementation
    - So encapsulation implements abstraction

Motivations for Abstraction

We want to design our own software in such a way as to make it easier to maintain, extend, and modify.
If we make developers lives easier, we will produce software more effectively:

- Accrue less technical debt
- Make iterative development easier
- Respond more readily to changes in requirements or environment
- Reduce cognitive load on a developer
    - Higher cognitive load leads to more mistakes, slower development

Benefits of Abstraction

- Interface protects implementer from incorrect use by the client
- Interface gives implementer flexibility to change implementation without difference from the outside
- Better error handling (either the client's fault or implementer's fault)

Separation of Concerns

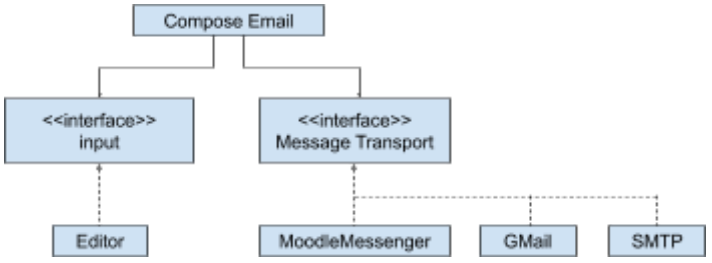
A principle for separating a program into modules with a single, well-defined responsibility.

Responsibilities should overlap as little as possible with other modules

- Shared responsibility leads to repeated code
- Unclear responsibilities make the module hard to use

The Dependency Inversion Principle (DIP)

High-level modules shouldn't depend on low-level modules. Both should depend on abstractions.



For example, consider the following which uses interfaces to abstract different implementations. Can Change GMail to HotMail and it will still work!

If there was no interface and Compose Email was using MoodleMessenger,GMail and SMTP directly. If you wanted to change GMail to HotMail, you would need to change Compose Email too.

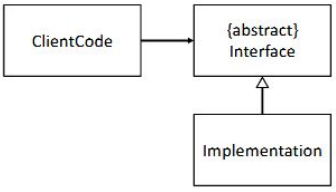
Common Mistakes

- Misunderstanding simplicity
    - Thinking reducing classes is good → Fewer classes means bigger classes → Made things simpler at class level → Larger classes are harder to maintain
    - Need to define smaller classes
        - Minimise dependencies
        - Hide any irrelevant info
- Overdoing Dependency Inversion
    - Using too many abstract classes is a bad thing
        - Adding an interface adds complexity to design
        - Only worth it if complexity is being taken elsewhere
        - Worth it if you want to develop components in isolation
- Confusing abstraction with abstract
    - Declaring a class to be abstract just means you can't instantiate it
    - Declaring an abstract method means you have to override with a concrete implementation
    - Does not guarantee that abstraction is useful or clean

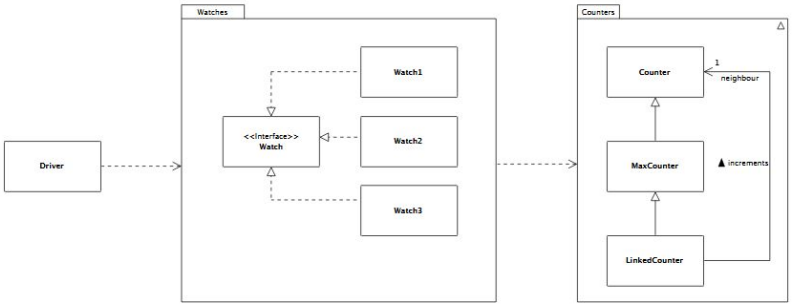
Using Abstraction in Java

Abstraction is a design principle rather than a programming technique, but most languages support it in

- **Classes:** Most important mechanism in abstraction.
    - A well-designed class should represent a single concept, expose a public interface for its responsibility, and hide implementation that doesn't fulfil that responsibility
- **Visibility Modifiers**: Deciding what to hide and expose (private, protected, public)
- **Abstract Classes:** A subclass inherits all the non-private methods declared in a base class. Client code can be passed an instance of some concrete subclass without needing to know its type. All it knows is that it does everything the abstract class says it can.
    - Useful for dependency injection
- **Hinge Points:** Applying dependency inversion to a simple relationship; client code doesn't care about implementation and vice versa, the parts can move around freely except for where they're joined.



- **Packages**
- We don't want our classes to be too large but we may need a lot of code to implement a feature. The solution is packages
    - Group related classes into a subsystem
    - Come up with a name
    - Put the package name at the top of each class
    - Move the Java files into a directory with the package name
    - *Nesting Packages*: You can't place a package in a package, but you can use dot notation to group packages together
        - Simplify Interactions
        - Ease of use for third-party programming



Abstraction Layers

Publicly accessible interface to a class, package or subsystem.

You can create an abstraction layer by restricting visibility as much as possible

- Ideally, make everything private
- If not private, then package
- Use dependency inversion to surround anything that might need to change with hinge points
    - Make interfaces public and keep all classes at default visibility

Interfaces

- Separate the publicly-accessible interface from their implementations
- Can be seen as an extreme abstract class
- An interface can be thought of as just a list of method definitions (without any body).
- If a class wants to implement an interface, it is entering into a contract, saying that it will provide an implementation for all of the methods listed in the interface

Generics

Generics allow us to define a class that may require varying types of data types. For example, without generics, we couldn't create a list of both strings and integers, but with generics, we can use <> notation to specify the types we want to use

- Can use operations without having to consider side effects/implementation details

```
For example:
    List<int, float> = new ArrayList<>();
```

Bound Type Parameters

- ```
  public class BinarySearchTree<T extends Comparable<T>> {
  ```
- In order for objects of class T to be able to be stored in our BST, class T must implement the Comparable<T> interface
    - Uses extends for both classes/interfaces

Code Smells and Refactoring

Code Smells

- Surface indication that corresponds to a deeper problem in the system
- Deeper problem is usually a design problem
- Design smells are a small bit of design that indicates a broader problem

An experienced developer develops the ability to detect bad design and implementation almost automatically when viewing code. This is through the identification of problems within the code

Refactoring

A disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour.

Refactoring can improve design, understandability and makes debugging easier.

You should refactor when:

- When adding new features
- When you need to fix a bug
- As you do a code review

Taxonomy of Code Smells

- **Long Smells**
    - Duplicated Code
  ```

        - Violates “Don’t Repeat Yourself”
    - Long Methods
        - Difficult to understand
    - Large Classes
        - Violates the Single Responsibility Principle
    - Long Parameter List
        - Should have the data it needs in the class
        - More than three is usually bad
- **Social Smells**
    - Divergent Change
        - You have a class that repeatedly changes, in a couple of ways
        - You have to change several methods together in one context
        - Hint that there might be two distinct classes
    - Shotgun Surgery
        - The opposite - when adding functionality you end up changing many classes
        - Indicative of poor encapsulation
    - Feature Envy
        - A method that spends its effort calling another class’ methods
        - Does not have the data it needs in the class where it is
- **Smells Like Python**
    - Primitive Obsession
        - Storing everything in primitives rather than classes
        - E.g. Storing in ArrayList<Strings> rather than a new class for the data type
        - Makes validation difficult
    - Data Clumps
        - Different variables that represent the same information
        - Variables should be attributes of an object
    - Switch Statements
        - If you want to change or extend it, you have to find and change them all
        - Should usually use polymorphism to solve
    - Data classes
        - Classes with data and no logic
        - Should perhaps be part of another class
- **Overengineering**
    - Speculative Generality
        - When you add methods for every special case
    - Lazy Class

        - Class doesn't do much anymore after refactoring
- **Others**
    - Branching on type information (i.e. using instanceof)
        - Introduces unwanted dependencies
        - Solved by type checking via method that returns T/F depending if it can be used
    - Message chains (e.g. x = a.getThing().getStuff(b).getAgain()
    - Middle-man

```
aFred.doThing()
…
Class Fred {
    private Ginger worker;
    public void doThing() {
        worker.doActualThing();
    }
```

    - Embedded literal numbers
    - Separate data structures for an object type
        - Chuck under one superclass and use polymorphism

Refactoring Procedure

- Make small change that eliminates or reduces a smell
- Continually test
- Repeat until the code smell is gone

Refactor Methods

- **Extract Method**
    a. Problem: You have a code fragment that can be grouped together
    b. Create a new method named after its intention
    c. Figure out visibility
    d. Copy the extracted code from the source method into the new target method
    e. Sort out issues with local variables
    f. Insert a call to the method

Before:	**After:**

`void printOwing() {`
`  printBanner();`
 	
`  //print details`
`  System.out.println("name: " + name);`
`  System.out.println("price: " + getPrice());`
`}`
	`void printOwing() {`
	`  printBanner();`
	`  printDetails(getPrice());`
	`}`
	
	`void printDetails(double price) {`
	`  System.out.println("name: " + name);`
	`  System.out.println("amount: " + price);`
	`}`

- **Move Method**
    a. Problem: A method is used more in another class than its own

b. Create a new method in the class which uses it most
c. Move code from the old method to the new
d. Replace the code in the old method with a reference to the new one

Before:
```
class AClass {
    void aMethod() {
        // some code
    }

class AnotherClass {
    // methods with many calls to
AClass.aMethod()
}
```

After:
```
class AClass {
    void aMethod() {
        AnotherClass.aMethod();
    }

class AnotherClass {
    // methods with many calls to aMethod()
    void aMethod() {
        // some code
    }
}
```

- **Too Many Temporary Variables**
    a. Problem: You store the result of an expression in a temporary variable for later use in code
    b. Extract the expression for the temporary variable into a method
    c. Replace all references to the temp with the expression
    d. The method can then be used in other methods

Before:
```
double calculateTotal() {
    double basePrice = quantity * itmPrice;
    if (basePrice > 1000) {
        return basePrice * 0.95;
    }
    else {
        return basePrice * 0.98;
    }
}
```

After:
```
double calculateTotal() {
    if (basePrice() > 1000) {
        return basePrice() * 0.95;
    }
    else {
        return basePrice() * 0.98;
    }
}

double basePrice() {
    return quantity * itmPrice;
}
```

- **Replace Magic Number With Symbolic Constant**
    a. Problem: Your code uses a number with significant meaning to it
    b. Replace the number with a constant that has a human readable meaning to it

Before:
```
double potentialEnergy(double mass, double
height) {
    return mass * height * 9.81;
}
```

After:
```
static final double GRAVITATIONAL_CONSTANT
= 9.81;

double potentialEnergy(double mass, double
height) {
    return mass * height *
GRAVITATIONAL_CONSTANT;
}
```

- **Replace Conditional With Polymorphism**
    a. Problem: A conditional performs various action depending on the case, ie a switch statement like conditional
    b. Create subclasses matching each conditional
    c. In them use a shared (abstract) method and move the code from the conditional the respective method
    d. Replace the conditional with a single method call

Before:

```
class Bird {
    //...
    double getSpeed() {
        switch (type) {
            case EUROPEAN:
                return getBaseSpeed();
            case AFRICAN:
                return getBaseSpeed() -
getLoadFactor() * numberOfCoconuts;
            case NORWEGIAN_BLUE:
                return (isNailed) ? 0 :
getBaseSpeed(voltage);
        }
    throw new RuntimeException("Should be
unreachable");
    }
}
```

After:

```
abstract class Bird {
    //...
    abstract double getSpeed();
}

class European extends Bird {
    double getSpeed() {
        return getBaseSpeed();
    }
}

class African extends Bird {
    double getSpeed() {
        return getBaseSpeed() -
getLoadFactor() * numberOfCoconuts;
    }
}

class NorwegianBlue extends Bird {
    double getSpeed() {
        return (isNailed) ? 0 :
getBaseSpeed(voltage);
    }
}

// Somewhere in client code
speed = bird.getSpeed();
```

Design Methodologies

Topics Covered
Software Design
Why Do We Design
Models
Aim in Design
Unified Modeling Language (UML)
Sequence Diagram
Communication Diagram
Software Specifications and Design
Design by Contract
Preconditions, Postconditions, and Invariants
Design Generation Methods
JavaDocs
Development Methodologies
Polymorphism

Software Design: Process of making decisions about how the software is to be implemented so as to have all the desired qualities

Why Do We Design

- All software has a design but not all software is good design
- No conscious design usually leads to bad design which leads to bad software
- Refactoring can improve design but not all bad design decisions can be refactored easily

Models

- **When do we use models?**
  - When the system (or component) is big
  - When the decisions are complex
  - When we need to reason about big complex things
  - When we need to communicate
  - When we need to record
- **What is a model?**
  - A representation of some aspect of the system we wish to model
  - Depicts that aspect in an easier-to-work-with way than the “real” system.
  - E.g. Pseudocode is a model
- **What can we model?**
  - Structure of system (static)

        - Class diagrams, component diagram, deployment diagrams
    - The behaviour of a system (dynamic)
        - Activity diagram, sequence diagram
    - Use both, with feedback between each.

Aim in Design

- A good design
    - Functionally Correct
    - Performs well
    - Maintainable
- An understanding of that design in the eyes of stakeholders
- Produce documents and designs to aid the following
    - Preliminary Domain Model
    - Sequence Diagrams
    - Revised Domain Model
    - Revised Sequence Diagram

Unified Modeling Language (UML)

Class Diagrams

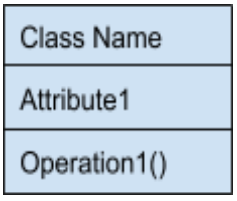
Diagram that shows the structure of all/part of an object-oriented program

- Shows which classes are present
- What their names are
- How the classes relate to each other

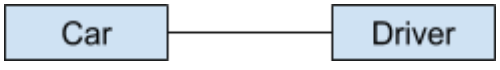
Used to explain the structure of a system
Used to model physical objects or abstract concepts

Class Diagram Notation

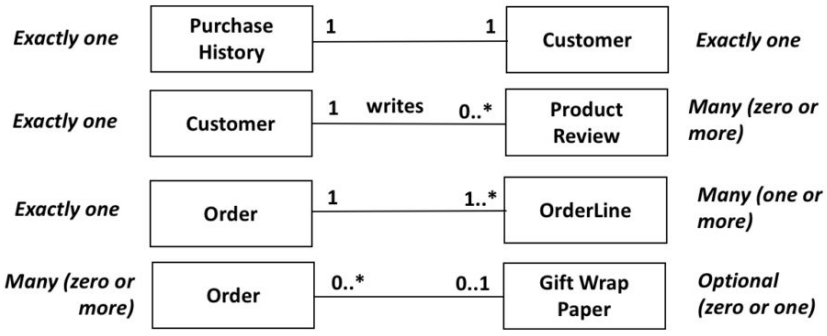
- **Classes**



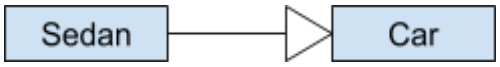
    - Operations and attributes can be omitted to emphasise other elements
- **Privacy**
    - If operation/attribute has - sign → It is private
    - If operation/attribute has + sign → It is public
- **Relationships**
    - **Association:** Relationship between classes



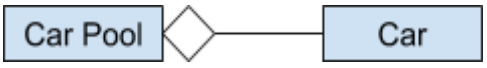
        - Allows one object to perform an action on its behalf
        - May also have an arrow between the two to indicate that only one knows about the other
    - **Multiplicities**



    - **Generalisation/Inheritance**



        - One of the classes is a specialised form of the other
    - **Shared Aggregation:** Kind of association between a group and its members



        - One class is made up of the other, but one will continue to exist once the other is gone
        - Eg. a university has departments, and they have professors. If the uni closes the departments no longer exist, but the professors do
    - **Composite Aggregation:** Aggregation between a whole and its parts

        - One object is part of the other

Association Classes

Keeps track of information about the association itself and to add attributes, operations and other features to associations

- E.g. Borrower ⟷ BorrowEntry ⟷ Book

Dependencies

A dependency is a relationship which indicates that a change in the specification of one thing may affect another thing it uses. You should use dependency when

you want to show one thing using another, but there isn't an association. This is shown as a dotted line

- Explicit: As a local variable or parameter
- Implicit: Return type of a call to an object of another class

Constraints

A specific constraint on a system to ensure a robust system. This is represented by a description of the constraint within curly brackets along with an association.

Stereotypes

A stereotype tells you something interesting about a system. It is represented by two <<arrows>>

- e.g. Java <<interface>>, <<abstract>>

Sequence Diagram

- **Interaction Diagram:** Describes how a group of objects collaborate in some behaviour.
    - Example of an interaction diagram is the sequence diagram
- **Sequence Diagram:** Captures the behaviour of a single scenario.
    - Shows the interaction by showing each *participant* with a *lifeline* the runs vertically down the page and the ordering of *messages* by reading down the page.
    - **When to use it**
    - When you want to look at the behaviour of several objects within a single use case
    - Good at showing collaboration between objects

Terminology

- **Participants:** The objects in an interaction diagram
    - Named using a name : Class format
- **Found Message:** The first message, of which doesn't have a known participant, as it should come from an unknown source
- **Message (Call):** A passing of control from one object to another
    - **Self-Call:** A passing of control from one part of an object to another part of the same object
- **Return:** Once an object has completed its task it can return some data to the object that delegated it
    - Only show the return arrow to show correspondence (using it for something more)

- **Parameter:** When an object is given a message it can also be given a set of data to use in processing
- **Activation:** A representation of processes being performed by an object
- **Lifeline:** A representation of the existence of a particular object
- **Centralised Control:** One participant does almost all the processing while the others supply data
    - Simpler, all processing is in one place, distributed control involves chasing around the objects
- **Distributed Control:** Processing is split among participants, each doing a little bit of the whole algorithm
    - Good design is localising the effects of change. Should put the data and the behaviour that uses it together in one place.
    - Can also allow for polymorphism better (e.g. handling subclasses of Products)

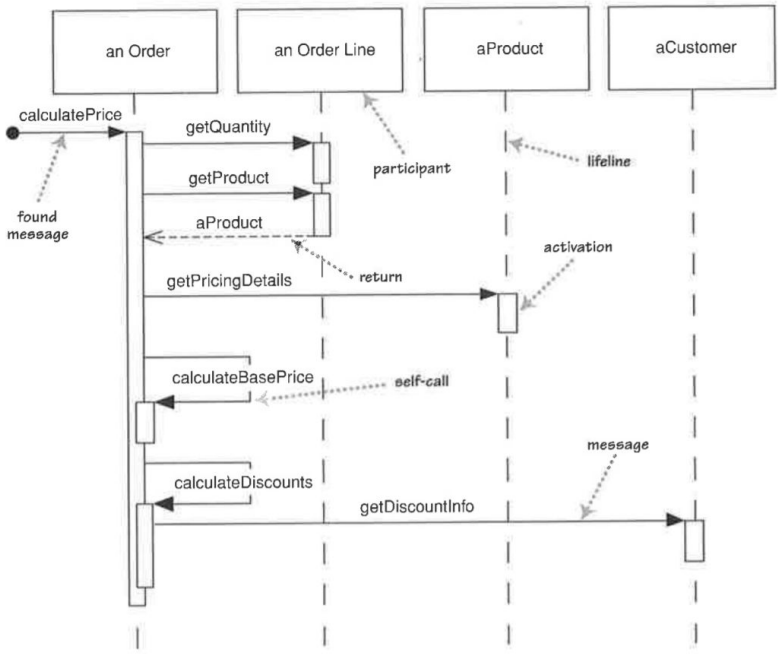


Figure 4.1 *A sequence diagram for centralized control*

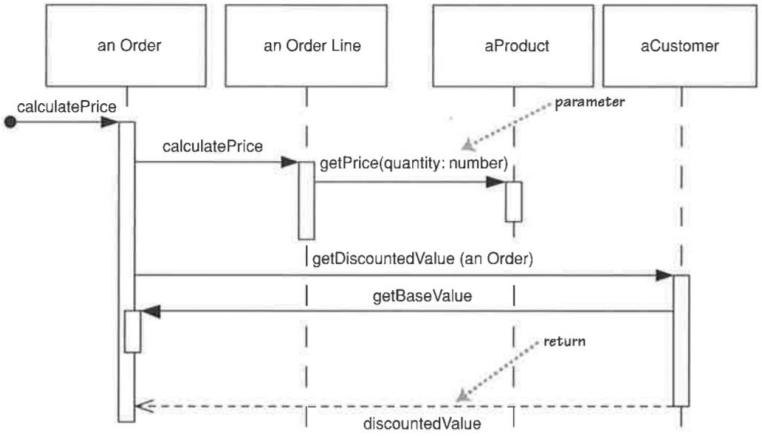


Figure 4.2 *A sequence diagram for distributed control*

Creating and Deleting Participants

Extra notation can be given, in sequence diagrams, to show when a participant is initialised (`new Object(arguments)`) and when a participant is terminated (`objectName = null;`) or self-deleted.

- **Creation:** When a new instance of an object is initialised to act as a participant overall algorithm
- **Deletion From Other Object:** When the is explicitly given a message to terminate itself
- **Self Deletion:** When, in a garbage collection environment, the participant has completed all the tasks it will be assigned, so it will automatically terminate itself

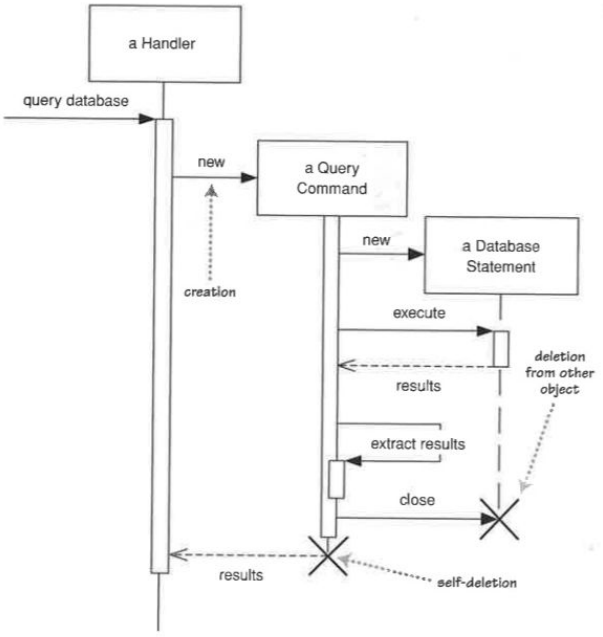


Figure 4.3 *Creation and deletion of participants*

Conditionals, Loops, and Interaction Frames

- **Frame:** Represents the scope at which a certain operator will occur
- **Operator:** An indicator describing the type of structure the frame has
- **Guard:** An expression describing the condition required for a specific fragment of a frame to be processed

Operator	Meaning
alt	Alternative multiple fragments; only the one whose condition is true will execute.
opt	Optional; the fragment executes only if the supplied condition is true.
par	Parallel; each fragment is run in parallel.
loop	Loop; the fragment may execute multiple times, and the guard indicates the basis of interaction.
region	Critical region; the fragment can have only one thread executing it at once.
neg	Negative; the fragment shows an invalid interaction
ref	Reference; refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.
sd	Sequence diagram; used to surround an entire sequence diagram, if you wish.

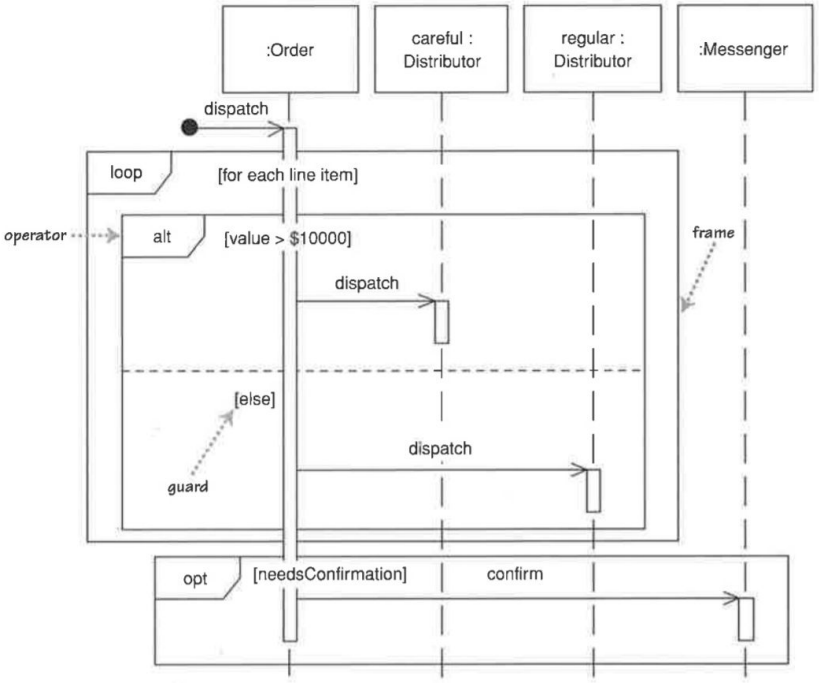
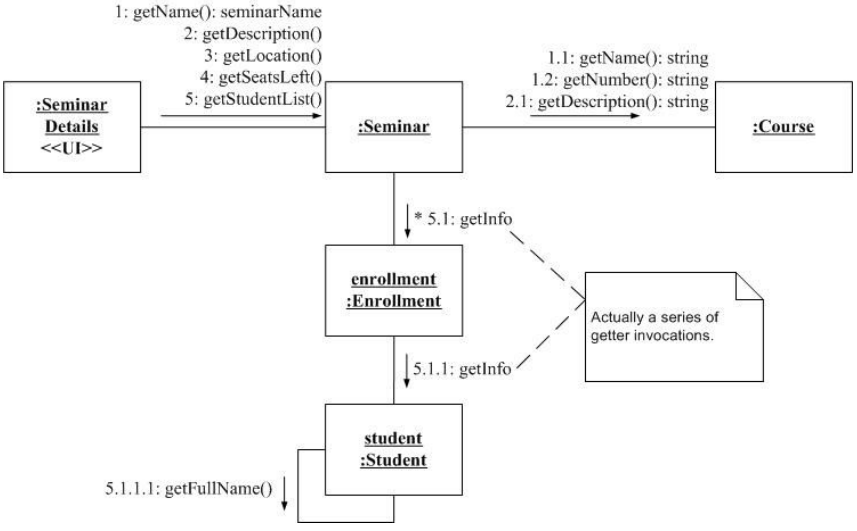


Figure 4.4 *Interaction frames*

Communication Diagram

- **Note:** Sequence diagrams are good at showing sequential logic but not good at giving the big picture view (communication diagrams are the opposite)
- Shows the message flow between objects
- Implies the basic associations between classes



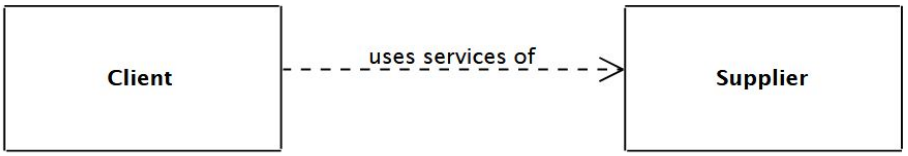
- Message notation
  - [sequenceNumber:] methodName(parameters) [: returnValue]

Software Specifications and Design

Client/Supplier Relationship

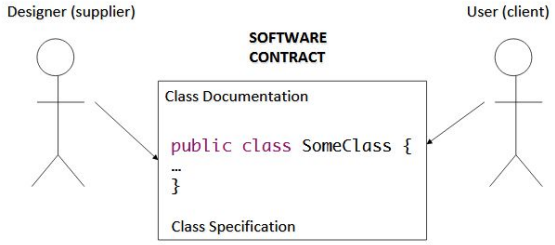
In UML, this is shown as an association or a dependency.

- Association used if Client has an attribute of type Supplier



Design by Contract

A contract is made between the class (the supplier) and the classes that are clients of that class.



- Relationship is governed by the contract of the class

Software contract provides:

- Documentation of the class for the technical user
- Possibility of enforcing the contract by using exceptions and assertions

The software designer tells the user what the class does providing a specification for the class.

A specification:

- Is ideally part of the implementation
- Should ideally be extractable from the implementation (e.g. JavaDoc)
- Is essential for supporting component reuse and maintenance
- Is more than just the API we have gotten used to seeing
    - Includes comments and contracts defined by executable specifications

The user:

- Should be able to know how to use a class by reading it's specifications
- Should not have to look at implementation details

Specification forms the public interface of the class

Exceptions and Assertions in Contracts

Contracts can be defined by using assertions and exceptions to create executable specifications

- They are verifiable by the compiler or code
- Go beyond simple commenting specifications

Also see preconditions, postconditions, and invariant

Precondition Violation

The client is at fault and an exception should be thrown to the calling routine

- Suppliers should not try to rescue the client's mistake

Postcondition Violation

Supplier is at fault, exception is raised in the called routine

- Does not always indicate a bug
    - Can be due to a transient condition that prevents the method from succeeding
    - e.g. Network outage, remote server down, disk/memory unavailable

Non-Redundancy Principle

Under no circumstances shall the **body** of a routine test for the routine's precondition

- It is the client's responsibility to check that it is meeting the preconditions of its suppliers
- It is never a good sign for any code to appear twice in a program
- A supplier cannot expect to know what it should do for all of its possible clients (some possibly not yet written)
    - A client can have code to catch and deal with an exception caused by precondition violation

See Precondition, Postconditions, and Invariants

Liskov Substitution Principle (Design by Contract Version)

- Subclasses must honour the contracts of their parents
- If this is done, we can use a subclass where its parent is expected
- See below

Principle of Sound Subcontracting

If a class is subcontracted by a supplier, the client doesn't need to know this.

This leads to the following rules for precondition and postconditions in a subclass:

- A subclass can only **weaken** the preconditions of its parents

    - Expect less from its clients
    - New preconditions should be logically 'or'-ed with those of its parents (if preconditions of parent method can be met, stop there OR try weaker preconditions)
- A subclass can only **strengthen** the postconditions of its parents
    - Guarantee more to its clients
    - New postconditions should be logically 'and'-ed with those of its parents (meet the postconditions of the parents and the extra preconditions as well)
- A subclass must preserve invariants

Command-Query Principle

Every method should either be a command or query

- A command performs actions changing the states of objects
- A query returns a value with no side effects

Results:

- Can use any query in a precondition/postcondition check with confidence that you won't change the state of the object you are trying to check

Preconditions, Postconditions, and Invariants

- **Preconditions**
    - What the client must guarantee to do
    - Usually in the form of constraints on the arguments of a method call
    - Violation of a precondition indicates a bug
- **Postconditions**
    - What the supplier must guarantee to provide
    - Promise that certain conditions will be met after a method has been called
- **Invariants**
    - Conditions that are held throughout the class before and after a method is called

Obligations and benefits

- Use of supplier methods by a client class should be governed by a precise description of the mutual benefits and obligations
-

`geometricMean(_:)`	**Obligations**	**Benefits**
Client	Supply non-negative arguments	Get geometric mean calculated
Supplier	Calculate geometric mean correctly	Simpler processing due to assumption of non-negative arguments

Redundancy

In some cases, the implementation of a routine and its postconditions are similar, but they're fulfilling different roles. Often we can, and should write the preconditions before we implement any code (part of specification, not implementation).

This leads to the non-redundancy principle; under no circumstances shall the body of a routine test for the routine's precondition

Design Generation Methods

- **Brainstorming**
  - Go for quantity
  - Without criticism
  - Welcome wild ideas
  - Combine and improve elements
  - Throw out chaff later
- **Bottom Up**
  - Start with a small problem
  - Design a solution to that
  - Do a few more
  - Put them together for a solution
- **Top Down**
  - Start with a high-level problem
  - Divide into subproblems
  - Solve these and put it together
- **Scenario Based Design**
  - Have scenarios that the design needs to support
  - Work through the scenarios
  - Modify/rework design to support scenarios more effectively
  - Repeat with additional scenarios
- **Class Responsibility Collaboration (CRC) Cards**
  - No special notation needed
  - We start with obvious cards and start playing 'what if' with scenarios.
  - If the situation calls for a new responsibility, either:
    - Add a new responsibility
    - Create a new object
  - Add collaborations as we go
  - CRC helps with encapsulation, it encourages small objects with clear responsibilities.
  - Though it doesn't generate good design, and this needs to be kept in mind.

Class Name	
Responsibilities ↓	Collaborators ↓

- **Benefits**
- CRC card process helps with encapsulation
- Encourage small objects with clear responsibilities
- But it doesn't guarantee a good design

JavaDocs

JavaDoc is a documentation generator in Java that converts the documentation given to packages, classes, methods, and other attributes into an HTML format. JavaDocs separates implementation from interface, and therefore, allows code to be used without the source code.
Note that Javadocs shouldn't, and needn't be attached to anything which is private, and it must occur outside of methods, classes and packages.

Formatting

```
/**
 * This is a Javadoc comment for the method <code>foo</code> and should explain what
 * <code>foo</code> does. Note that you can use HTML tags within your JavaDocs.
 * {@code return “code snippets can be include as so”;}
 *
 * @param FIT2099 a description of the parameter
 * @return a description of the return
 * @throw ExceptionName an explanation to why an exception is thrown
 * @see referenceToJavaDocFromOtherCode
 * @author Author Name
 * @version 1.0
 */
public String foo(String FIT2099) {
    code;
}
```

Development Methodologies

Waterfall

Requirements ⟶ Design ⟶ Implementation ⟶ Verification ⟶ Maintenance
Or, with slight refinement
Requirements ⟶ Analysis ⟶ Design

Polymorphism

- Polymorphism refers to the concept of code that uses the same interface to operate on different types of objects in a general way

Object Oriented Concepts (Java)

Topics Covered
Java Fundamentals
Classes, Interfaces, and Abstract Classes
Packages
Debugging, Assertions, and Exceptions

Java Fundamentals

Primitives

- **Byte:** 8-bit integer
- **Short:** 16-bit integer
- **int:** 32-bit integer
- **Long:** 64-bit integer
- **Float:** 32-bit floating point
- **Double:** 64-bit floating point
- **Boolean:** True or False
- **Char:** 16-bit Unicode character

Non-Primitive Types

- Classes and interfaces
- Arrays

Named Constants

Stores an unchangeable instance of an object, common to all instances of a class

```
static final type CONSTANT_NAME;
```

Arrays, Lists, and Maps

A basic array contains a fixed number of items of the same type, in a fixed order

```
Object[] arrayName = new Object[n];
arrayName[i] = value;
value = arrayName[i]
```

Lists are a class that acts like an array, but is more flexible in its ability to resize when something is added

```
List<Object> listName = new List<Object>();
listName.add(value);
listName.remove(i);
value = listName.get(i);
```

Maps are a class that acts like a list, but it uses a key rather than an index to store and get values

```
Map<Object, Object> mapName = new Map<Object, Object>();
mapName.put(key, value);
value = mapName.get(key);
```

Classes, Interfaces, and Abstract Classes

Classes

A class consists of:

- Members
    - Fields
    - Methods
- A declaration
    - Visibility
        - Public - Everything
        - Protected - Visibility set to class and subclasses and in the same package
        - Private - Only the class
    - The class it inherits from
        - Extends
    - Any interface it implements
        - Implements
- Clauses of members in the class
    - Type
    - Visibility
    - Initial values

Every object created has a unique identity, independent of the objects state.

```
visibility class ClassName extends Parent implements Interface {
    visibility Object varName = new Object(parameters);

    visibility type method(arguments) {
        code;
    }
}
```

Interfaces

Java allows us to specify a type that declares what methods any class that implements that type must have, without providing any implementation for these methods

Abstract Classes

An abstract class has at least one abstract method; that is, a method with a declaration but no implementation.
The abstract methods acts as a placeholder which could be different for certain superclasses, and normal methods which are inherited identically by every subclass.

```
Abstract class in Java:
    visibility abstract class SuperClassName {
```

```
        visibility type SuperClassName(parameters) {
            Code;
        }
        // Abstract Method
        abstract visibility type abstractMethod(parameters)
    }

Use of abstract classes in a subclass:
    visibility class SubClassName extends SuperClassName {
        visibility type SubClassName(parameters) {
            super(arguments);
            code;
        }
        // Use of abstract code
        @Override
        visibility type abstractMethod(parameters) {
            code;
        }
    }
```

Delegation

During maintenance there may be cases where we need to delete outdated portions of code, however, it can be difficult to find everything that depends on the old code, so we use delegation.
We replace as much of the old, outdated code with references to the updated code.

```
class OldClass extends SuperClass {
    private NewClass myNewClass;
    public OldClass() {
        myNewClass = new NewClass();
    }
    public void method() {
        myNewClass.method();
    }
```

The Universal Base Object

Every class in Java is a subclass of the Object class. This comes with a set of default methods which are automatically inherited.

```
Create and return a copy of the object:
    protected Object clone() throws CloneNotSupportedException
Indicate whether or not another object is equal to this one
    public boolean equals(Object obj)
Called by the garbage collector when there are no more references to an object
    protected void finalize() throws throwable
Return the runtime class of an object
    public final Class getClass()
Returns a hash code value for the object
    public int hashCode()
Returns a string representation of the code
    public String toString()
```

All of these methods are overridable by subclasses, allowing developers to design their own implementation

Packages

A package is a group of related classes. Its benefits are as follows:

- It makes it easier to find related classes
- It can prevent name clashes
- Eliminates dependencies

Debugging, Assertions, and Exceptions

Assertions

Java provides a mechanism for the programmer to assert something that should be at a certain point in the code. Note the expression should have a value, and not be a void procedure.

```
assert condition : expression
```

Exceptions

Sometimes when things go wrong the method executing cannot do its job, so there needs to be a way to report the problem so that it can be fixed.
In Java, when a method needs to signal that something's gone wrong, it does so by throwing an exception. The method that called the method can either try to catch the exception and deal with it, or throw it to the method that called it.

```
Throwing exceptions:
    thrown new Exception(message);
The method that throws the exception needs to declare that it can throw the exception
    visibility type methodName(parameters) throws ExceptionName
```

Disciplined Exception Handling Principle

There are only two ways an exception should be should be handled

- Retrying: Changing the conditions that led to the exception and to execute the routine from the start
- Failure (organised panic): Clean the environment, terminate the call, and report the failure to the caller

There are four main reasons for using exceptions and assertions

- Help in writing correct software
- Aid in documentation
- Support for testing
- Support for software fault tolerance

They are not

- An input checking mechanism

- Control structure

Note: Exceptions are usually used for precondition checking and assertions for postcondition (because assertions are usually turned off post-production)

Catching Exceptions

Rather than passing an exception on, the calling method has the option of trying to deal with it. We do this by catching the exception.

```
try {
    code;
    }
catch(ExceptionName e) {
    handle e;
    }
```

Unchecked Exception

These are subclasses of `RuntimeException` or `Error`, and they do not have to be specified in the signature of a method. They are used in situations in which it isn't reasonable for the client to be able to recover or handle the exception.

Rule of thumb for choosing checked exceptions vs unchecked exceptions

If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

Exceptions Type

- **Casting**: ClassCastException
- **Arrays**: ArrayIndexOutOfBoundsException, NullPointerException
- **Collections**: NullPointerException, ClassCastException (if you're not using autoboxing and you screw it up)
- **IO**: java.io.IOException, java.io.FileNotFoundException, java.io.EOFException
- **Serialization**: java.io.ObjectStreamException (AND ITS SUBCLASSES, which I'm too lazy to enumerate)
- **Threads**: InterruptedException, SecurityException, IllegalThreadStateException
- **Potentially common to all situations**: NullPointerException, IllegalArgumentException