`

FIT2099
Object Oriented Design & Implementation

Design Principles

Topics Covered
Object Oriented Paradigm
Refactor
Connascence
Encapsulation
Abstraction
Code Smells and Refactoring
Other Principles

Object Oriented Paradigm
Object orientation is a conceptualisation of objects that carry out the programs tasks.

Procedural Programming
- A collection of procedures
- Each procedure has its own input and output
- Easier to write than spaghetti code

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

Connascence

Dependency control
- Controlling the extent of dependencies
- Controlling nature of dependencies

Why dependencies?
- Dependencies are unavoidable
- If code unit A depends on code unit B:
  - Bugs in B may manifest in A
  - Changes to B may require changes to A

- Dependencies should be:
    - Only present where necessary
    - Explicit
    - Easy to understand

Connascence

- they are "born together"
- there is at least one change that could be made to A that would necessitate a change to B.
- More connascence means:
    - Harder to extend.
    - More chance of bugs
    - Slower to write in the first place
- When to concerns
    - Locality matters
    - Explicitness matters

Types of Connascence

- Static
    - Obvious from code structure
    - Can be automatically identified by IDE/analysis tools
- Dynamic
    - Only obvious from close inspection/execution
    - Can't be (easily) identified by IDE
- Connascence of name (CoN)
    - Defining and calling a method
- Connascence of type (CoT)
    - Type of parameter
- Connascence of position
    - Ordering of parameters
- Connascence of meaning/convention (CoM/CoC)
    - Special meaning (e.g. reset to 0)
- Connascence of algorithm (CoA)
    - Encryption and decryption
- Connascence of execution (CoE)
    - Ordering of execution
- Connascence of timing (CoT)
    - Parallel computing
    - Interacting with hardware – especially real-time computing
    - Distributed computing
- Connascence of values (CoV)
    - Student and student Id
- Connascence of identity (CoI)

`

- Two objects have variables pointing to a same object

Harder to find, know more things, easier to screw up, result in nastier bugs.

Contranascence

- When two things are required to be different
- This is a form of connascence

What to do with connascence

- Minimise overall amount of connascence by breaking system into encapsulated elements.
- Minimise remaining connascence that crosses encapsulation boundaries (guideline 3 will help with this)
- Maximise connascence within encapsulation boundaries

Encapsulation

Encapsulation fulfills 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

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
- 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)

Abstraction
Abstraction is 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 we need in order to represent an item or object
- What we should expose to use this part easily

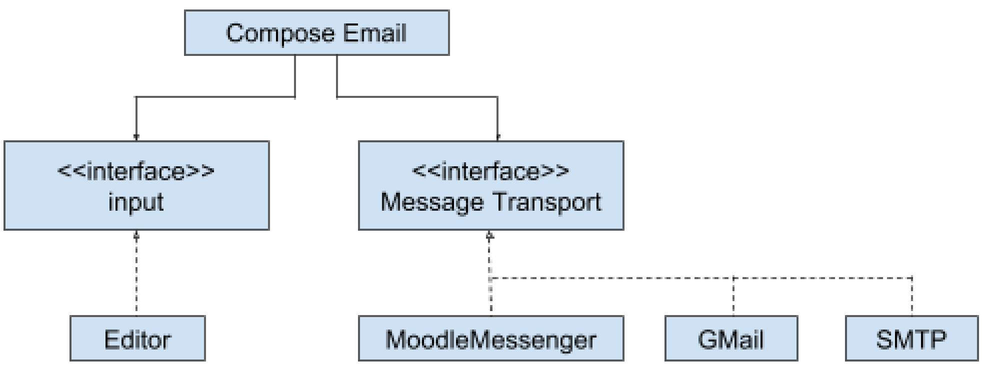
We use abstraction when we bundle things together and use them, eg lines of code in a method, data in a class, classes in a package.

Abstraction in OO Design
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

Abstraction vs Encapsulation
- We use encapsulation when we bundle things together
- We use abstraction when we decide what to bundle together
    - Or how things should look from the outside
- We use information hiding when we use encapsulation that doesn't allow access from the outside

Separation of Concerns
A principle for separating a program into sections, or modules with their own concern or responsibilities. Concerns should be well defined and have little overlap with others.

The Dependency Inversion Principle (DIP)
High level modules shouldn't depend on low level modules, but rather depend on abstractions.
For example, consider the following which uses interfaces to abstract different implementations



Common mistakes

- misunderstanding simplicity
    - Classes are systems
- overdoing dependency inversion
    - Add complexity
- confusing abstraction with abstract

Using Abstraction in Java

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

- **Classes:** The class is the most important mechanism in abstraction; A well designed class should represent a single concept, expose a public interface for its responsibility, hide implementation that doesn't fulfill that responsibility, and ensure that its attributes are in a valid condition rather than relying on client code
    - *Visibility Modifiers*: Deciding what to hide and expose is a big part of applying abstraction
    - *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 subclass can
    - *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 packages7
        - 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 package: use this a.b.c package naming notation to group packages together
        - simplify interactions
        - ease of use for third-party programming
        - *Nesting Packages*: You can't place a package in a package, but you can use dot notation to group packages together

Abstraction Layers

An abstraction layer is the 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 hinges to avoid public
- Common problem: making too much public

Interfaces

Used exclusively in Java

- Separate 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 examples, 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

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

Code Smells and Refactoring

Code Smells

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 i.e, its a surface indication that corresponds to a deeper problem in the system.

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
    - Don't Repeat yourself

`

- Long Methods
  - Difficult to understand
- Large Classes
  - Violates 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 can often change several methods together
  - Shotgun Surgery
    - The opposite - when adding functionality you end up changing many classes
    - Indicative of poor encapsulation
  - Feature Envy
    - A method spends its effort calling another class
- **Smells Like Python**
  - Primitive Obsession
    - Storing everything in primitives rather than classes
  - Data Clumps
    - Groups of variables that pop up repeatedly to represent the same information
    - Should likely be attributes of an object
  - Switch statements
    - Same cascade of switch conditions appears in multiple places
  - Data classes
    - Classes with data and no logic
- **You do it**
  - Message chains
    - A long line of getThis
  - Middle man
    - A class is doing too much simple delegation.
- **Overengineering odours**
  - Speculative Generality
    - When you add methods for every special case
  - Lazy Class
    - Class doesn't do much anymore after refactoring

`

Refactoring With Fowler

- **Temporary Variables**
    - Fowler doesn't like them:
        - Only useful in their own routine
        - Easy to lose track of
    - Replace them with queries
        - Accessible to any method
        - Exchanges cleaner design

Refactor Methods

- **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	After

<pre>Before:
class AClass {
    void aMethod() {
        // some code
    }
 	
class AnotherClass {
    // methods with many calls to
AClass.aMethod()
}</pre>
	<pre>After:
	class AClass {
	    void aMethod() {
	        AnotherClass.aMethod;
	    }
	
	class AnotherClass {
	    // methods with many calls to aMethod()
	    void aMethod() {
	        // some code
	    }
	}</pre>

- **Extract Method**
    a. Problem: You have a code fragment that can be grouped together
    b. Create a new method named after it's 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 method

Before	After

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

- **Replace Temp With Query**

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:	After:

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

- **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:	After:

<pre>Before:
double potentialEnergy(double mass, double
height) {
    return mass * height * 9.81;
}</pre>
	<pre>After:
	static final double GRAVITATIONAL_CONSTANT
	= 9.81;
	
	double potentialEnergy(double mass, double
	height) {
	    return mass * height *
	GRAVITATIONAL_CONSTANT;
	}</pre>

- **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:	After:

<pre>Before:
class Bird {
    //...
    double getSpeed() {
        switch (type) {
            case EUROPEAN:
                return getBaseSpeed();
            case AFRICAN:
                return getBaseSpeed() -
getLoadFactor() * numberOfCoconuts;
            case NORWEGIAN_BLUE:</pre>
	<pre>After:
	abstract class Bird {
	    //...
	    abstract double getSpeed();
	}
	
	class European extends Bird {
	    double getSpeed() {
	        return getBaseSpeed();
	    }
	}</pre>

`return (isNailed) ? 0 :`
`getBaseSpeed(voltage);`
`}`
`throw new RuntimeException("Should be unreachable");`
`}`
`}`
	`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();`

Other Principles

Cohesion and coupling

Cohesion refers to what the class (or module) can do. Low cohesion would mean that the class does a great variety of actions - it is broad, unfocused on what it should do. (Has a set of features that don't belong together).

High cohesion means that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class. (Has a set of features that all contribute to the type abstraction implemented by the class)

As for coupling, it refers to how related or dependent two classes/modules are toward each other. For low coupled classes, changing something major in one class should not affect the other. High coupling would make it difficult to change and maintain your code; since classes are closely knit together, making a change would require an entire system revamp.

Good software design has high cohesion and low coupling.

Separation of Concerns

A principle for separating a program into sections, or modules with their own concern or responsibilities. Concerns should be well defined and have little overlap with others.
See Abstraction

The Dependency Inversion Principle (DIP)

High level modules shouldn't depend on low level modules, but rather depend on abstractions.
See Abstraction

`

Single Responsibility Principle

Every module or class should have responsibility over 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

- For a class S to be a true subtype of T, then S must conform to T
- A class S conforms to class T only if an object of class S can be provided in any contract where an object of class T is expected and correctness is still preserved

See Design By Contract

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

See Design By Contract

Fail Fast

The fail fast principle says that 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

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

Avoid excessive use of literals

- if the value of the constant needs to change in future, you have to hunt for every place it occurs in the code and change it in all of them

`

Don’t Repeat Yourself

- When the same code appears in multiple places, it must be maintained in multiple places.
- If a bug is discovered in that code, every piece of duplicate or very similar code must be checked and fixed.
- If the requirements change and the code needs to be modified, it needs to be tracked down and changed everywhere.

Classes should be responsible for their own properties

- Information should be stored as close as possible to where it is needed

Reduce dependencies as much as possible

Group elements that must depend on each other together inside an encapsulation boundary

Minimize dependencies that cross encapsulation boundaries

- As soon as something is visible, other programmers can write code that depends on it. If you then change it, you break other systems.

Declare things in the tightest possible scope

- the less risk there is that something can depend on it, the less the risk that it will be a future point of failure

Avoid variables with hidden meanings

- The variable is stretched over two jobs, meaning that the variable is the wrong type for one of the jobs

Design Methodologies

Topics Covered
Model
Unified Modeling Language (UML)
Sequence Diagrams
Software Specifications and Design
Design Generation Methodologies
JavaDocs
Design by Contract and Conditions
When and Why Design
Development Methodologies

Model

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.

What can we model?

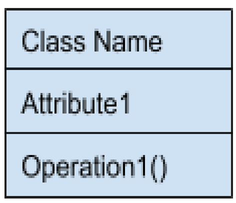
- Structure of system (static)
- Behaviour of system (dynamic)
- Use both, with feedback between each.

Unified Modeling Language (UML)

A class is a set of objects that share attributes and/or operations. There is a correspondence between class diagrams in UML and ER diagrams. Classes are analogous to entities, the only difference being the lack of operations in entities.

Class Diagram Notation

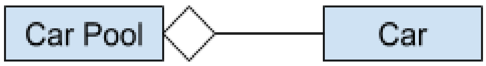
- **Classes**



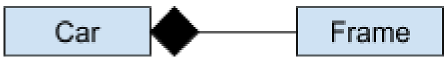
    - Operations and attributes can be omitted to emphasise other elements
- **Relationships**
    - **Association**

        - 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
    - **Generalisation/Inheritance**

        - One of the classes is a specialised form of the other
    - **Shared aggregation**



        - 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
        - Eg Forest is composed of trees, but a tree can exist without a forest
    - **Composite aggregation**



        - 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

Dependencies

A dependency is a relationship which indicates that a change in 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

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 an association.

Stereotypes

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

Sequence Diagram

- **Interaction Diagram:** describes how a group of objects interact with each other. The sequence diagram captures the behaviour of a single scenario.
- **Sequence Diagram:** 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.

Terminology

- **Participants:** The objects in an interaction diagram
- **Found Message:** The first message, of which doesn't have an 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
- **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
- **Distributed Control:** Processing is split among participants, each doing a little bit of the whole algorithm

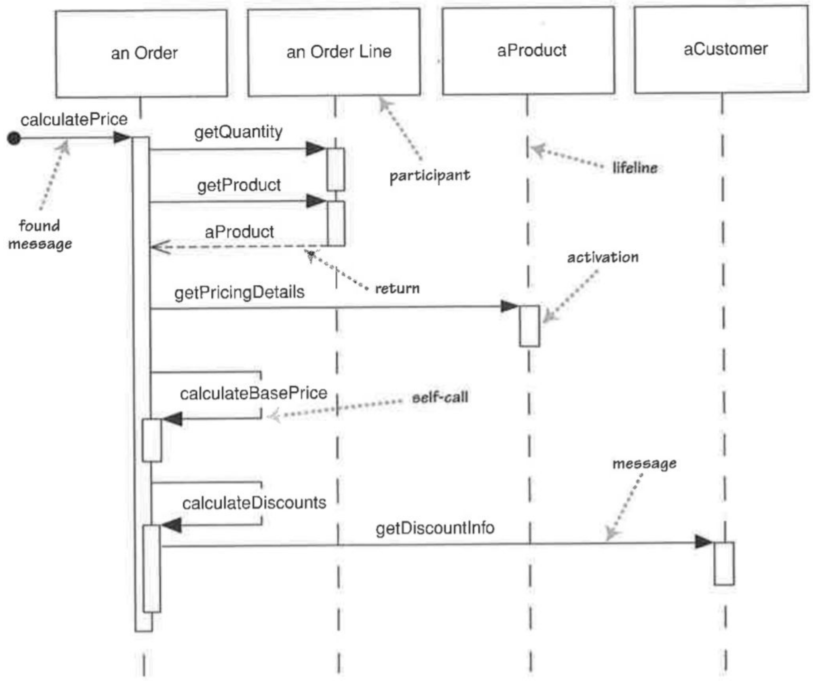


Figure 4.1 *A sequence diagram for centralized control*

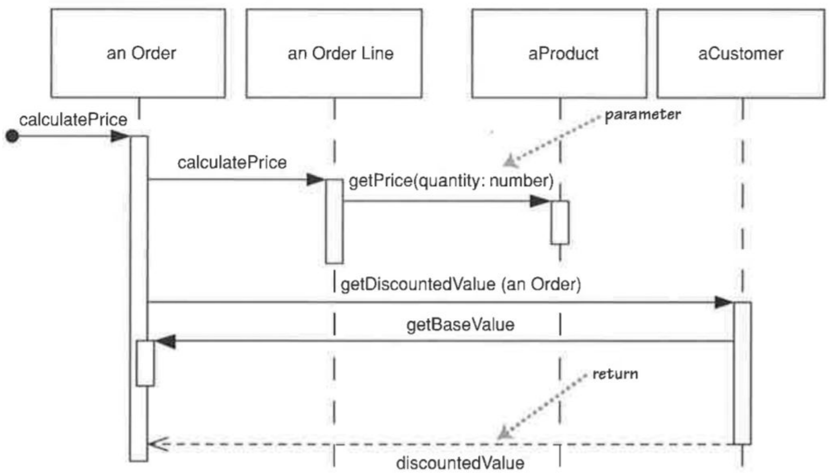


Figure 4.2 *A sequence diagram for distributed control*

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

`

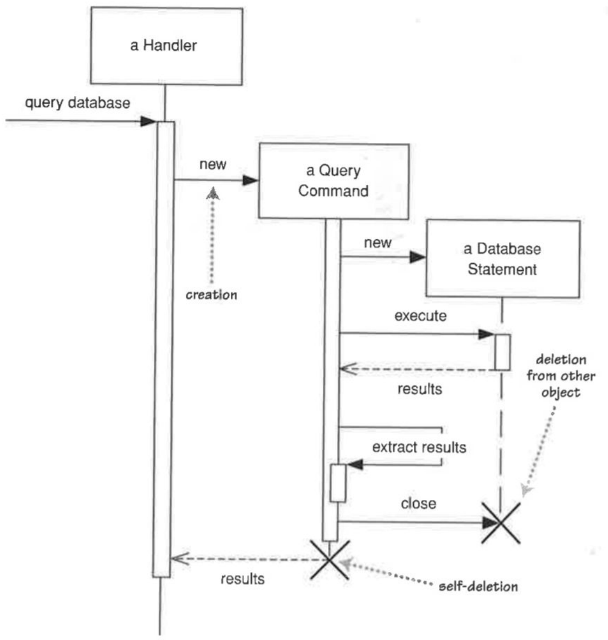


Figure 4.3 *Creation and deletion of participants*

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.

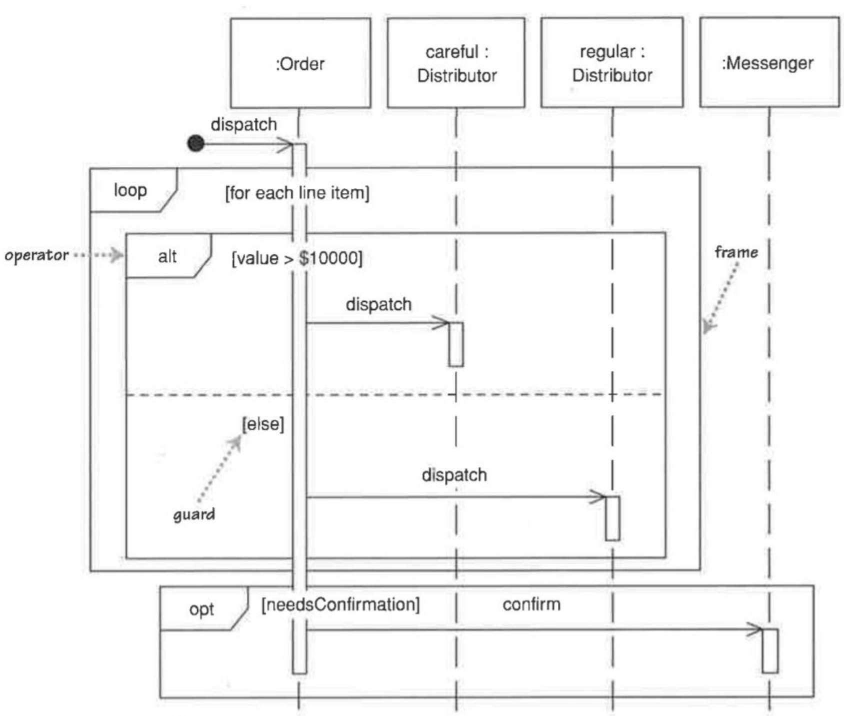


Figure 4.4 *Interaction frames*

Software Specifications and Design

Client/Supplier Relationship

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

`

Design Generation Methodologies

- **Start**
    - Understanding the problem domain
    - Draw models of the problem domain
    - Can be evolved
- **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 sub problems
    - Solve these and put it together
- **Scenario Based**
    - Have scenarios that the design need to support
    - Work through the scenarios
    - Improve 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 ↓

`

JavaDocs
JavaDoc is a documentation generator in Java that converts the documentation given to packages, classes, methods, and other attributes into a HTML format. JavaDocs seperates 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. Not 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
 */
public String foo(String FIT2099) {
    code;
}
```

Design by Contract

A class designer establishes a software contract between him/herself and the users of the class they design. We can make this impersonal, and think of this as a contract between the class (supplier) and the classes that use that class (client).
The software contract provides:
- the documentation of the class for the technical user
- the 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.
- What the methods of the class need to operate correctly
- What the class will guarantee to be true if it is used correctly

A specification:
- Is ideally part of the implementation
- Should ideally be extractable from the implementation
- Is essential for supporting component reuse and maintenance
- Is more than just the API we have gotten used to seeing

The user:
- Should be able to know how to use a class by reading its specifications
- Should not have to look at implementation details

The specification forms the public interface of the class
- Comments
- Method signatures (name and typed arguments)
- Preconditions, postconditions, and invariants

`

Make it easier to do **Fail Fast**

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 simply commenting specifications

Also see preconditions, postconditions, and invariant

Precondition Violation

The client is at fault and an exception should be thrown to the caller. Suppliers should not try to rescue the clients mistake

Postcondition Violation

The supplier/designer is at fault and an exception is raised in the called routine. Could be caused by a bug or transient condition.

Liskov Substitution Principle

- For a class S to be a true subtype of T, then S must conform to T
- A class S conforms to class T only if an object of class S can be provided in any contract where an object of class T is expected and correctness is still preserved

This leads to the design by contract paradigm with the following rule

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

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 of its clients
  - New preconditions should be logically 'or'-ed with those of its parents
- 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
- 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

`

Exceptions provide a mechanism for commands to report failure without becoming mixed with queries
Useful when doing Design By Contract

- Use any query in a precondition or postcondition check with confidence that you won't change the state of the object

When and Why Design

Typical times for conscious design

- Project inception
  - Architectural design
- Before implementing something complex or risky
- Refactoring existing code

Aim in Design

- A good design
  - Functionally correct
  - Performs well enough
  - Usable
  - Reliable
  - 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

Development Methodologies

Waterfall

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

Technical Debt

You have a piece of functionality that you need to add to your system You see two ways to do it, one is messy but quick - though it will cause problems in the future, the other is cleaner but slow - though it will be free of potential problems. This dilemma is inevitable in real systems. It happens because sometimes we need to get features out the door, though it can be repaid by refactoring

`

Object Oriented Concepts (Java)

Topics Covered
Java Fundamentals
Classes, interfaces, and abstract classes
Generics
Packages
Superclasses
Dependencies
Debugging, Assertions, and Exceptions
Preconditions, Postconditions, and Invariants

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

Variable Declaration

```
object varName = new Object(arguments);
primitive varName = primitiveValue;
```

Named Constants

Stores a variable which can only be set once
The static modifier indicates that the class member it qualifies is shared by all instances of a class. The final modifier indicates that the value of the field it qualifies can never change. A static final field is thus used to represent a constant: something common to all instances of the class that never changes

```
static final type CONSTANT_NAME;
```

Control Structures

- **If, else if, else**

```
if (condition) {
    code;
}
else if (condition) {
    code;
}
else {
    code;
}
```

- **While**

`

```
while (condition) {
    code;
}
```

- **For**

```
for (initialisation, condition, action) {
    code
}
```

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);
```

The java Numbers Classes

The wrapper classes “wrap” primitive data types (e.g. int, double, etc.) so that they can be used as objects.
Reasons:

- use the primitive when an object is expected (e.g. with a generic data structure).
- access constants (e.g. Integer.MAX_VALUE)
- Access class methods defined by the wrapper class (e.g. Integer.parseInt(String s)).

Classes, Interfaces, and Abstract Classes

Classes

A class consists of:

- Members
    - Fields (store the state of the objects)
    - Methods (specify the behaviour of the objects)
- A declaration

`

- Visibility
  - Public
  - Protected
  - Private
- The class it inherits from
  - Extends
- Any interface it implements
  - Implements

- Clauses of members in the class
  - Type
  - Visibility
  - Initial values
  - Definition for methods

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 subclasses, 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;
        }
    }
```

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 some object is equal to this one
    public boolean equals(Object obj)
Called by the garbage collected when there are no more references to an object
    protected void finalize() thrown 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 superclasses, allowing developers to design their own implementation

Packages

In Java, the convention is to use the reversed domain name of the organisation as the prefix for the package name.
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

Dependencies

Dependencies

a broad software engineering term used to refer when a piece of software relies on another one.

- Dependencies should be reduced as must as possible and elements that must depend on each other should be grouped within an encapsulation boundary. Related to this, dependencies should be minimised for those which cross boundaries and things should be declared within the tightest possible scope.

Indirect Dependencies

Some dependencies are obvious, others are invisible to the compiler. They exist because the meaning of various elements in the code to humans; only a human

`

can know they exist. These are the most dangerous dependencies as they're not able to be found by software and can be overlooked during maintenance.

Debugging, Assertions, and Exceptions

Fail Fast

The fail fast principle says that 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. The resultant program will be more robust in the end

Assertions

Java provides a mechanism for the programmer to assert something that should be at a certain point in the code.

```
assert condition : expression
```

where:

- Expression1 is a boolean expression.
- Expression2 is an expression that has a value. (It cannot be a call to method that is declared void.)

Exceptions

Why not assertion:

- Checking assertions adds computational overhead at run time, so it is more efficient to run deployed code without them
- They are tools to aid in development. The messages are aimed at developers, not end users.

Using return values is problematic

- Programmers often don't do the right thing and check the return value for error signals.
- It is in general considered bad design to give special meanings to certain parts of the range of a value.

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 thrown the exception
    visibility type methodName(parameters) throws ExceptionName
```

Disciplined Exception Handling Principle

There are only two ways an exception should be handled

- Retrying

`

- To change the conditions that caused the bug
- Failure
  - 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

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;
    }
```

When catching an exception, common to do:

- error recovery, for example:
  - wait and try again (transient problems)
  - try a different way to achieve the method's goal
- prompting the user to make a decision
- propagating the error up to a higher-level handler
  - often after doing some cleaning up

Unchecked Exception

These are superclasses 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.

Preconditions, Postconditions, and Invariants

- **Preconditions**
  - What the client must guarantee to do
  - Usually in the form of constraints on the arguments of a method
  - Violation of a precondition indicates a bug
- **Postconditions**
  - What the supplier must guarantee to provide
  - Violation of a post condition is usually due to a bug
- **Invariants**

`

- Conditions that are held throughout the class

Redundancy
In some cases, the implementation of a routine and its postconditions are similar, but they are fulfilling very different roles. Often we can, and should write the preconditions before we implement any code.
This leads to the non-redundancy principle; under no circumstances shall the body of a routine test for the routine's precondition.

Delegation
It will be hard to remove all references to some codes when we want to delete them.
We want to keep the some codes around so as not to break code that depends on them, but not its separate (and out-dated) implementation.

```
package edu.monash.fit2099.watches;

public class Watch1 extends Watch {

   private Watch2 myWatch2;

   public Watch1() {
      myWatch2 = new Watch2();
   }

   public void tick() {
      myWatch2.tick();// delegation
   }

   @Override
   public void display() {
      myWatch2.display(); // delegation
   }
}
```

Documenting
Code is often worked on by many people and has a very long lifespan. These people need information about how the code works, and why it is written the way it is. It is thus vitally important to document your code properly. Documentation in the class is essential when we need to maintain or extend the class.
Other programmers using your class, however, should not be required to look at the source code. It is thus vitally important to be able to produce separate documentation for the interface of the class: those class members that are visible from outside the class.

`

Original Document

https://docs.google.com/document/d/1abtAVskH0AukXyr9-0PyRkoc1H72AfAPqS9bCURogFA/edit?usp=sharing

- Resolved Formatting
- Contains most information added to this document