L15 - Introduction to detailed design

CS3028 - Principles of Software Engineering

Ernesto Compatangelo

Department of Computing Science

UNIVERSITY OF ABERDEEN

15.1 Reminding past issues and mapping them to current topics

Where are we now?

Software development paradigms

⇒ The Unified Process (UP) paradigm

⇒ UP phases and UP disciplines (activities) within each phase

⇒ Elaboration (second UP phase)

⇒ Elaboration Design

⇒ Architectural design & patterns

⇒ Detailed design

15.2 From architectural to detailed design

From architectural design to detailed design

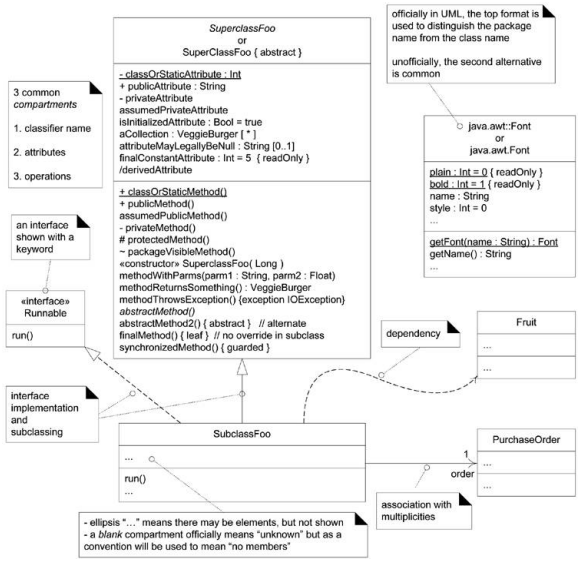
- **Architectural design** transforms a set of requirements specification (use cases, sequence diagrams, vision, domain model, glossary, non-functional requirements) into a 'box-and-line' structure of loosely coupled 'black box' components (subsystems)
- **Detailed design** fills in the details of each component (previously treated as a black box) in two stages:
  1. **Mid-Level (ML) design** – specifies software in terms of units such as packages, components or classes, their interfaces, contents, properties, relationships, and interactions
  2. **Low-Level (LL) design** – fills in the small details at the lowest abstraction level

Detailed design artifacts

- **Static models**
  - Class diagram (ML, LL)
  - (ER/Relational/other) Data structure diagram (ML)
  - Data formats (LL)
  - Operation specification (ML)
  - Dependencies, collaborations, responsibilities, interfaces (ML)
- **Dynamic models**
  - Sequence diagram (ML)
  - State machine diagram (ML)
  - Pseudocode (LL)

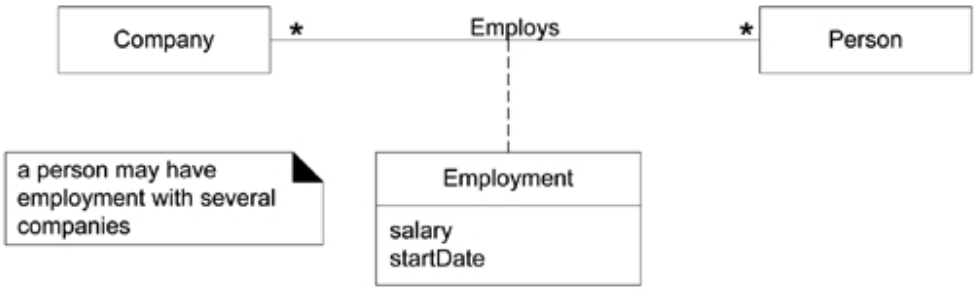
15.3 Modelling package structures: classes and sequence diagrams

UML design-level class diagram notations

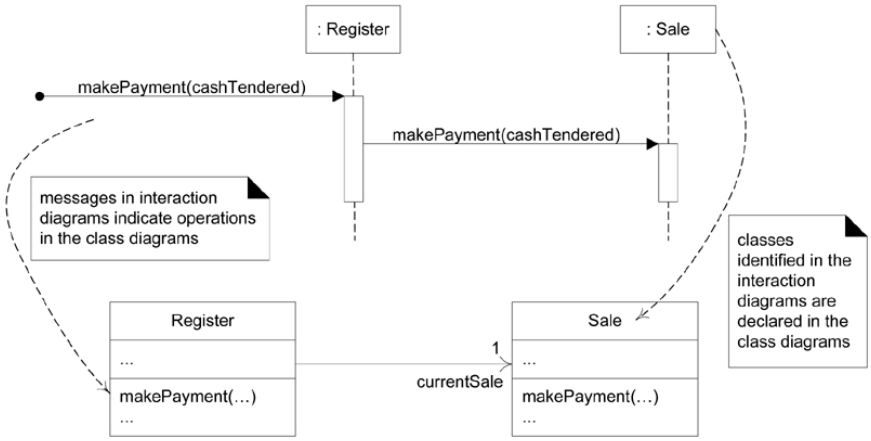


15.3.1 Association classes: an UML conundrum

- Associations are modelled as simple links if (i) they represent binary relationships and (ii) they do not have attributes of their own
- Associations are modelled as classes if (i) either they represent N-ary relationships ($N \geq 2$) (ii) or they have attributes of their own



Links between class and sequence diagrams



15.4 From subsystems to packages and components

- Software system architectures are described in terms of **subsystems** that have different names, such as layer, client, server, repository, etc.
- Each subsystem is defined at the architectural level as a *conceptual container* that is responsible for a broad cohesive area (*e.g.*, user interface, business logic, storage)
- Partitioning divides subsystems into smaller units that are 'visible' in practice (*i.e.*, real pieces of software with well-defined boundaries). Each of these units is called *package* (if source code accessible) or *component* (if only executable code accessible).

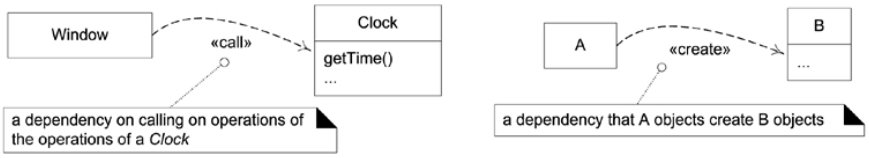
15.4.1 Packages and components

- Packages and components are two different ways of referring to the same thing from two different viewpoints (and abstraction levels)
- We use the term **component** when we consider a unit as a modular, replaceable *black box* with a *well-defined interface*. A java library (provided as a jar file) is an example of component
- We use the term **package** when we explicitly consider the internal structure and behaviour of a component in terms of classes and other components. The java group of classes within the same `package` specification and the libraries they `import` is — as expected — an example of package
- Just to make things a bit more confused, components are sometimes described in terms of a (rather simple) *internal architecture* made of other mutually linked components

15.5 Dependencies and interfaces

Dependencies inside units and between units

A dependency is a client-supplier relationship where a change to the supplier (the *independent* unit) requires a change to the client (the *dependent* unit).



There are many kinds of dependency relationships (depicted as dashed lines with arrows in UML):

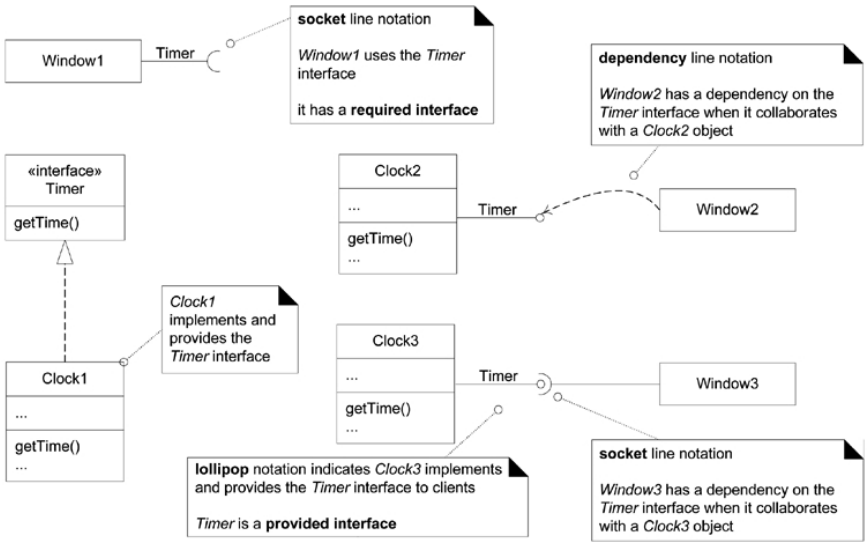
- **Access**: the supplier allows the client to access its members, limited by the visibility of each member
- **Import**: the supplier allows the client to access its public members and to make their names part of the client itself
- **Refinement**: the client improves, enhances, merges, or alters the supplier content
- **Use**, **call**, **instantiate**, **send**: a client needs to have access to the supplier member/s to function properly

15.6 Interfaces

Interfaces

- An **interface** at the detailed design level is a named collection of public attributes and *abstract* operations that defines the relationships between a component and its environment
- An **abstract operation** is an unimplemented operation which is only specified int terms of its *signature*
- A **signature** specifies the operation name, parameters, and return values but no internal *computation mechanism* (*i.e.*, algorithm)

Interface examples



15.7 Mid-level detailed design: problems and solutions

Major issues arising from mid-level detailed design

- How to assign elements (namely, objects and thus classes) to design units (subsystems, packages)
- How to assign operations to objects within each unit

The methodological solution to these problems is twofold:

- Use constructive principles for good design
- Look at exemplary solutions devised in the past to tackle similar problems, suitably generalised into mid-level detailed design patterns.

15.7.1 Heuristics for assigning objects to units

- Assign conceptual objects identified in each use case to the same unit. This is likely to be a unit in the business logic layer (or equivalent) containing *entity* classes
- Once dealing with non-conceptual units (*e.g.*, those in storage and interface layers), new objects may be needed (i) to present and (ii) to store information contained in entity classes. For each computational unit, create a corresponding unit in the other layers if needed
- Create a dedicated unit for objects used to move data among other units (some sort of controller or manager)
- Minimise the number of associations crossing subsystem boundaries (*i.e.*, minimise *coupling*)
- Make sure all objects in the same unit are functionally related (*i.e.*, maximise *cohesion*)

15.8 Coupling

Coupling (the core focus of design)

- Coupling describes the degree of interconnectedness between design elements (operations, classes, packages, units, subsystems)
- Coupling measures how interdependent these elements are: the higher the independency, the more likely a change to one element affects the others.
- Coupling is reflected (i) by the number of links an element has and (ii) by the degree of interaction the element has with other elements

Coupling minimisation (the main objective of design)

- **Interaction coupling**
  - Deals with methods and objects through message passing
  - Measures of the number of message types an element sends to other elements, and the number of parameters passed, with a view to **keep both of them to a minimum**
- **Inheritance coupling**
  - Deals with the invocation of methods and the access to attributes along the hierarchy, possibly violating encapsulation
  - Describes the degree to which a subclass actually needs the features it inherits from its base class, with a view to **keep the hierarchy depth and thus the amount of inherited features to a minimum**.

Minimising interaction coupling: Law of Demeter

An object $\mathcal{O}$ should only send messages to:

- itself
- another object contained in an attribute of $\mathcal{O}$ or in any of the superclasses of $\mathcal{O}$
- an object that is passed as a parameter to a method of $\mathcal{O}$
- an object that is created by a method of $\mathcal{O}$
- an object that is stored in a global variable.

See Wikipedia to know why the 'law' was given this name

Levels of interaction coupling (from best to worst)

- **None**: no method calls another
- **Data**: calling method passes an object variable to the called method, which uses the whole object to perform its function.
- **Stamp** calling method passes an object variable to the called method, which only uses a portion of the whole object to perform its function.
- **Control**: calling method passes a control variable, which is used by the called method to control its execution.
- **Global**: method refers to a 'global data area' outside calling object
- **Content**: method refers to the hidden part of another object (C++)

15.9 Cohesion

Cohesion (the second core focus of design)

- Cohesion is a measure of the degree to which an element (operation, class, package, unit, subsystem) contributes to a single purpose.
- Cohesion refers to how specific and 'single-minded' an element is
- In the context of object orientation, (i) operation cohesion, (ii) class cohesion, and
  (iii) gen-spec hierarchy cohesion are considered.

Operation cohesion

- Deals with cohesion within a method
  (how many different things does the method do?)
- Expresses the degree to which an operation focuses on a single functional requirement
- The design goal is that of ending with highly cohesive operations, each of which deals with a single functional requirement

Levels of operation cohesion (from best to worst)

- **Functional**: a method performs one single function
- **Sequential**: a method performs two functions, where the output from the 1st one is used as the input to the 2nd one
- **Communication**: the method combines two functions that use the same attributes
- **Procedural**: the method supports multiple weakly related functions
- **Temporal**: the method supports multiple functions related in time (same time)
- **Logical**: the method supports multiple weakly related functions whose selection is based on a control variable passed to the method
- **Coincidental**: no correlation between method functions

Class cohesion

- Deals with cohesion among the attributes and methods of a class (how many things does the class represent and how many behaviours does it have?
- Represents the degree to which a class focuses on a single requirement
- The design goal is that of ending with highly cohesive classes, each
  - containing multiple methods to perform different functions
  - having methods visible from the outside and performing a single function
  - only having methods that refer to attributes or other methods defined with the class or with any of its superclasses
  - not having any control-flow coupling between its methods

Gen-spec hierarchy cohesion & the Liskov substitution principle

- Addresses the semantic cohesion of inheritance hierarchies (is it a true **kind_of** relationship or should it rather be an **association**?)
- Can be maximised by checking that designed gen-spec hierarchies comply with the **Liskov substitution principle**:
  - If an object of type `S`
    can be used in all the places where an object of type `T` is expected,
    then `S` is a true subtype of `T`
  - In other words, any method belonging to `T`
    must be able to use instances of any subclass `S`
    without knowing whether such instances come from `S` or from `T`

15.10 Preparing for the topic ahead

Next lecture...

Delving into design:

More specifically, we will focus on:

- Detailed design issues
- Responsibility driven design for detailed design patterns