CA314 – Object Oriented Analysis & Design - 3

File name: CA314_Section_03_Ver01

Author: W.G. Tuohey

No. of pages: 25

Table of Contents

3. UML Class Models – Essentials (see ref 1, Chap. 5) 3
3.1 Discussion of Design by Contract 1 (PANEL 3.1 of ref. 1) 3
3.2 Quick recap of class basics 7
3.2.1 Classifiers and Instances 7
3.2.2 Identifying objects and classes (5.1 of reference 1) 8
3.2.3 "… Criteria … in Decomposing Systems into Modules" 11
3.3 Associations, Attributes & Operations (5.2, 5.3 of ref. 1) 12
3.3.1 Association & Dependency 12
3.3.2 Review of some basic elements of UML class diagrams 17
3.3.3 Suggested Guideline to determine class operations & attributes 17
3.4 Generalization - "***is a*** relationship" (see ref 1, sect. 5.4) 18
3.4.1 Design rules (or guidelines) for generalization 18
3.4.2 Implementing generalization, inheritance & delegation 19
3.5 Class Responsibility Collaboration cards (ref 1, sect. 5.6) 22
3.6 Refactoring (ref. 1, section 5.6.4) 24
3.7 Class model evolution during development (ref. 1, sect. 5.5) 25
3.8 Preview of "8. More on Classes" 25
3.9 Exercises 25

3. UML Class Models – Essentials (see ref 1, Chap. 5)

This section fills in some essential aspects of UML class models, with the emphasis more on OO principles than on detail of UML syntax (though there is some of this). Section 8 will supply more of this syntactic detail. As in section 2, we follow reference 1 quite closely though not exclusively.

3.1 Discussion of Design by Contract 1 (PANEL 3.1 of ref. 1)

In the state diagram of the introductory case study (Figure 2.10-1) we saw how "conditions" may impact on system behaviour. This is addressed further in "Panel 3.1" of reference 1.

Essentially, this panel is concerned with "**syntactic**" and "**semantic**" checks on interfaces, and on developing some related guidelines on good design practices. In fact, we should distinguish between (a) good design practices and (b) how such design is actually implemented in code; for (b) much depends on what is available in the chosen programming language.

First, recall that **types** in a programming language are a rough way to describe the properties which an object's attributes and operations should have. For example, in the sequence diagram of the introductory case study (Figure 2.9-2), if the message okToBorrow always returned the value false then that would be of the correct Boolean type but would never allow a copy to be borrowed. If possible, we would ***like*** to know something more precise than the type.

In other words, when trying to describe the **provided interface** of a module, we can try to tie it down roughly first using type definitions and, if that is not precise enough, we can add constraint(s) to express additional matter. By constraints we mean

- operation preconditions & postconditions

and

- class invariants.

Note: In UML an **operation** is defined as “a service that can be requested from an object to affect behaviour”. Each operation has a signature. On the other hand, a method is an “implementation of an operation”.

Preconditions & postconditions (of operations):

Example	**Commentary**
okToBorrow **pre**: numberOfBooksOnLoan ≤ maxBooksOnLoan **post**: **if** numberOfBooksOnLoan < maxBooksOnLoan **return** True Otherwise **return** False	It is a "bug" to call this operation if the precondition is False It is a "bug" if this postcondition ever turns out to be unsatisfied after the operation was called with the precondition True

Figure 3.1-1: Examples of preconditions and postconditions

class invariants: In Figure 3.1-1, the precondition does not just apply to okToBorrow but, in fact, it is always a "bug" if the precondition is False. Therefore, it is more appropriate to make it a class invariant, that is, put it as documentation of the LibraryMember class that a valid object of that class must always have numberOfBooksOnLoan ≤ maxBooksOnLoan.

The following definitions from Braude, reference 3, may help clarify the above ideas. Notice that the context for this author is not confined to operations and classes:

> Preconditions: Conditions on non-local variables that a method's code assumes. This includes parameters. However, verification of the conditions is not promised in the method.
>
> Postconditions: Value of non-local variables, including parameters, after execution of the method.
>
> Invariants: Relationships between non-local variables that a function's execution does **not** change (although values of individual variables may change). Invariants could apply to methods, blocks of code, classes, etc. If a class is in question then we speak of class invariants.

OCL and some examples: More precise notation may sometimes be needed especially if handling many such pre- and postconditions and invariants. The "**Object Constraint Language**" (OCL) is such a notation and is commonly used with UML, is part of it, in fact. We may occasionally use parts of OCL in this module but will not cover it systematically. For now, we present the following two examples:

<u>Example A: A Container class with an add message</u>

Assume that we want to assert the following:

- Precondition: An object reference passed to the add message must not be null
- Precondition: An object passed to the add message must not already be in the Container
- Postcondition: After the add message, the Container will have one more object than it had before
- Postcondition: After the add message, the Container will contain the object that was passed in as a parameter
- Invariant: The Container always contains a non-negative number of objects

The following fragment of OCL (in which the "contains" method returns true if and only if the receiver contains "o") specifies the above assertions:

```
context Container::
        add(o: Object)
                pre: (o <> null) and not contains(0)
                post: contains(o) and (size = size@pre + 1)
context Container::
        inv: size >= 0
```

<u>Explanation</u>:

OCL allows us to connect assertions to a class using the "context" keyword, which means that we can also refer to global functions and data in our specifications, for the sake of hybrid languages such as C++. Postconditions often need to refer to the value of an attribute before the method is executed - this allows us to talk about the effect that a method has on the receiver's attributes. In this case, "size@pre" refers to the value of "size" before add is executed.

The OCL fragment can be read as follows:

"When sending the add message to a Container, the parameter o must be non-null and o must not be in the container already. Once the method has completed, the container will contain o and the size of the container will be one more than it was before. The size of a Container is always greater than or equal to 0.

<u>*Example B: Photo Library (*Sommerville, "Software Engineering", Ed 8, Sec. 19.3)</u>:

Sommerville presents an example in "a notation based on the object constraint language (OCL)" so it is not quite written in the normal OCL manner (but almost so).

The example has to do with specifying a system that downloads images from a digital camera and stores them in a photograph library. The following is the specification for the **addItem** and **delete** methods in **PhotoLibrary**, where methods in the interface of **PhotoLibrary** include

public void addItem (Identifier pid; Photograph p; CatalogEntry photodesc);
public Photograph retrieve (Identifier pid);
public CatalogEntry catEntry (Identifier pid);

```
context addItem
pre: PhotoLibrary.libSize() > 0
     Photo.retrieve(pid) = null
post: PhotoLibrary.libSize() = PhotoLibrary.libSize()@pre + 1
      PhotoLibrary.retrieve(pid) = p
      PhotoLibrary.catEntry(pid) = photodesc

context delete
pre: Photo.retrieve(pid) <> null
post: PhotoLibrary.retrieve(pid) = null
      PhotoLibrary.catEntry(pid) =  PhotoLibrary.catEntry(pid)@pre
      PhotoLibrary.libSize() = PhotoLibrary.libSize()@pre - 1
```

The above specification fragment can be read as follows:

The preconditions for addItem state that

- The library must exist - assume that creating a library adds a single item to it so that the size of the library is always greater than zero. [*Note: In fact, this is really an invariant for PhotoLibrary*].
- The library must not already contain a photograph with the same identifier as that to be entered.

The post conditions for addItem state that

- The size of the library has increased by one (only)
- If you retrieve using the same identifier, then you get back the photograph that you added
- If you look up the catalogue using that identifier, you get back the catalogue entry that you made.

The precondition of delete states that

- The item must be in the library

The post conditions of delete state that

- The photograph can no longer be retrieved from the library

- The size of the library has been decreased by one
- The catalogue entry is not deleted (to allow for maintaining information about the deleted item in the catalogue).

Implementation issues: The definition of constraints during design (or specification) is good practice. However, the extent to which support is provided for checking such constraints in the implemented code is not great.

-- Many programming languages support ***compiler-time*** type checking but compiler-time constraint checking is more problematic - Eiffel is noted in the text as one language that does provide such checks.

-- ***Assertions*** are noted in the text as being provided by some languages - these are Boolean expressions (i.e. conditions) that can be checked at ***run-time*** (though possibly only in "debugging mode").

-- Some languages allow declaration of subtypes, a facility that may be useful in implementation of constraints. In Ada, for the example of Figure 3.1-1, we could declare

numberOfBooksOnLoan: INTEGER **range** 0 .. maxBooksOnLoan;

Then a CONSTRAINT_ERROR would be raised if ever numberOfBooksOnLoan had a value outside its range.

Note: Reference 1 has another panel (3.2 Persistence) on shutting down and re-starting a system without losing information. We omit this for now.

3.2 Quick recap of class basics

3.2.1 Classifiers and Instances

We are familiar with classes and objects where, in UML terminology, classes are “classifiers” and objects are “instances”. There are a number of other classifier-instance pairs, including those in Table 3.2-1.

Classifier	**Instance**
Class	Object
Association	Link
Use case	Scenario

Table 3.2-1: Examples of classifier-instance pairs

3.2.2 Identifying objects and classes (5.1 of reference 1)

What makes a class model good?:

Objectives (of a class model)	**How to meet objectives**
Build [, as quickly and cheaply as possible,] a system which meets our current requirements	Every required piece of behaviour must be able to be provided [, in a sensible way,] by objects of the classes we choose
Build a system which will be easy to maintain and adapt to future requirements	A good class model consists (as far as possible) of classes which represent *enduring* classes of domain objects, which don't depend on the particular functionality required today

Table 3.2-2: Objectives of a good class model

How to build a good class model:

- No hard and fast rules but there are useful guidelines and techniques
- Usually won't get class model right at start - expect some adjustments
- Can expect to identify the important classes of *domain* objects early on but identification of other classes may be harder.
- We present two techniques for identification of classes:

noun identification **(for data driven design)**	**CRC cards** **(for responsibility driven design)**
Extreme caricature: identify all data in the system and divide them into classes. Only after that, consider class responsibilities.	**Extreme caricature**: identify all responsibilities in the system and divide them into classes. Only after that, consider class data. (*See later in section 3.2*)

Table 3.2-3: Class identification techniques

In practice, it is best to use a mixed approach rather than just one technique.

Noun identification: *We described this already* - it involves

(1) Identify candidate classes by underlying noun and noun phrases in a requirement specification

(2) Discard candidates that are inappropriate for any reason. For example,

Discarded Candidate	Reason for discarding
library	outside scope, noun does not refer to something inside the system
loan	event, not a thing [*maybe this example is not that obvious a discard!]*. If an instance of the event or operation does not have identity, state and behaviour should discard it as a candidate class
Member of library	redundant (same as something else) - choose a class name to encompass all you mean to include
item	Too vague - cannot tell unambiguously what is meant by a noun
week	A measure, not a thing
system, rules	Not part of domain, more part of the "meta" description language

Table 3.2-4: Reasons for discarding candidate classes in noun identification

Also, quite often, a noun may refer to something simple which is an **attribute** of another class (e.g. "name" of library member on our case study)

Note: In general, a class is characterized by

IDENTITY: Notion that an object has a continuing existence

STATE: All the data an object currently encapsulates - the current values of its attributes

BEHAVIOUR: The way an object acts and reacts, in terms of its state changes and message passing.

What kinds of things are classes?:

Reference 1 notes that objects and their division into classes derive from overlapping sources:

Source	
Tangible or 'real-world' things: book, copy, course	*Much more common sources of objects and classes than following two*
Roles: library member, student, director of studies	
Events: arrival, leaving, request	*Often help to find associations*
Interactions: meeting, intersection	

Table 3.2-5: Sources of classes

Budd (*An introduction to object-oriented programming*) makes the following suggestions for discovering classes:

(a) Data managers, data, state - after the nouns in a problem description - the fundamental blocks

(b) Data sinks & sources - do not hold data for a period of time (like (a) does) - generate data or process data on demand

(c) View or observer classes - one often has base data (the model) and corresponding display (view)

(d) Facilitator or helper classes (could include common utilities such as specific data structures (and their associated attributes & operations), math 'packages', etc)

Real world objects Vs their system representation:

In our introductory case study, we adopted one approach for representing *actors* within our system design (see section 2.7). We will come back to this point when discussing *use cases*.

Reference 1 (section 5.1.4) makes two general points on the representation of real-world things within our software system:

1	Do not record information that is definitely irrelevant to the system
2	Do not lose sight of the fact that the objects *are* the system

Table 3.2-6: Guidance on the representation of real-world "things"

Point 2 of Figure 3.2-6 is important: Essentially object oriented design and programming involves the "creation of a host of helpers in the solution of a problem".

This is in contrast to an extreme top-down, functional decomposition approach. In such a system, one sometimes (often?) had a single component that "knew" about everything and embodied all the interesting behaviour. In an object oriented system we will still probably need a "main" class to provide the entry into the system. Starting the system will automatically create an instance (the only instance) of this class and this will go on create other objects of the system. However, this main object should not usually have any complex behaviour of its own. This point is also raised in section 3.2.3 (below).

3.2.3 "… Criteria … in Decomposing Systems into Modules"

It is interesting to read the classic paper "On the Criteria To Be Used in Decomposing Systems into Modules" (*D.L. Parnas*) (*full text available on-line*) which is considered to have been one of the first to look at the disadvantages of functional decomposition, the merits of information hiding, etc. Its conclusion reads:

"We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. To achieve an efficient implementation we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collections of code from various modules."

3.3 Associations, Attributes & Operations (5.2, 5.3 of ref. 1)

3.3.1 Association & Dependency

We saw that classes correspond to nouns and, similarly, *associations correspond to verbs*.

Definition of "associated" classes: Class A and class B are associated if some object of class A *has to know about* some object of class B (*or subclasses of B for which substitutivity holds - see section3.4 below*), i.e. if one of following holds:

-- An object of class A sends a message to an object of class B

-- An object of class A creates an object of class B

-- An object of class A has an attribute whose values are objects of class B or collections of objects of class B

-- An object of class A receives a message with an object of class B as argument

Note on Dependency (see section 6.4 of reference 1): Recall *that "A depends on B if a change in B may force a change in A. ... . Notice the difference between a dependency between two classes and an association between them. An association between two classes represents the fact that objects of these classes are associated. A dependency is between the classes themselves, not between the objects of those classes ... for example, a class always depends on a class from which it inherits.*" The UML terminology can be a bit confusing (dependency, relationship, association) and we will try to make it clearer in the following notes.

A general guideline on model development (see pp61, 62 of reference 1): *Throughout a development*, one should aim to develop a model which is good in both of the following aspects:

-- conceptual - reflecting real world associations between objects

-- implementation - modeling interactions to achieve required functionality

Clarification of Dependency & Association – from Braude (reference 3):

> "***Dependency***, denoted by a dotted line arrow, means that one class depends upon another in the sense that if the class at an arrow's end were to change, this would affect the dependent class. Strictly speaking, dependency includes inheritance and aggregation. However, these relationships have their own notation, and so we usually reserve dependency to indicate that a method of one class utilizes another class." [*but there are other types of dependency, see below*]
>
> "***Association***, denoted with a solid line between two classes, commonly means that objects of each class depend on objects of the other in a structural way. … Associations are useful in the early stages of building class relationships when we know a pair to be related, but are postponing the ultimate design of the relationship."

Some relevant definitions (Glossary of "OMG Unified Modeling Language Specification", Version 1.5):

association The semantic relationship between two or more classifiers that specifies connections among their instances.

behavioral feature A dynamic feature of a model element, such as an operation or method.

classifier A mechanism that describes behavioral and structural features. Classifiers include interfaces, classes, datatypes, and components.

dependency A relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).

generalization A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed. See: *inheritance.*

relationship A semantic connection among model elements. Examples of relationships include associations and generalizations

structural feature A static feature of a model element, such as an attribute.

trace A *dependency* that indicates a historical or process relationship between two elements that represent the same concept without specific rules for deriving one from the other.

usage A *dependency* in which one element (the client) requires the presence of another element (the supplier) for its correct functioning or implementation.

Note: **Permission** is a kind of dependency. It grants a model element permission to access elements in another namespace…. The predefined stereotypes of Permission are access, import, and friend. … In the case of the **friend** stereotype, the client is granted permission to reference elements in the supplier namespace, regardless of visibility.

A further note on Dependency (from "OMG Unified Modeling Language Specification", Version 1.5:

Semantics: A dependency indicates a semantic relationship between two model elements (or two sets of model elements). It ***relates the model elements themselves*** and does not require a set of instances for its meaning. It indicates a situation in which a change to the target element may require a change to the source element in the dependency.

Notation: A dependency is ***shown as a dashed arrow*** between two model elements. The model element at the tail of the arrow (the client) depends on the model element at the arrowhead (the supplier). The arrow may be labeled with an optional stereotype and an optional individual name.

Kinds of Dependency: The following Table 3.3-1 identifies kinds of Dependency that are predefined and may be indicated with keywords. … All of these are shown as dashed arrows with keywords in guillemets [i.e. <<…>>]. We will come across several of these dependency types as we progress through UML. We just note their existence for now.

Keyword	Name	Description
access	Access	The granting of permission for one package to reference the public elements owned by another package …
bind	Binding	A binding of template parameters to actual values to create a nonparameterized element. …
derive	Derivation	A computable relationship between one element and another …
import	Import	The granting of permission for one package to reference the public elements of another package, together with adding the names of the public elements of the supplier package to the client package. …
refine	Refinement	A historical or derivation connection between two elements with a mapping (not necessarily complete) between them. …
trace	Trace	A historical connection between two elements that represents the same concept at different levels of meaning. …
use	Usage	**A situation in which one element requires the presence of another element for its correct implementation or functioning**. May be stereotyped further to indicate the exact nature of the dependency, such as **calling an operation of another class**, granting permission for access, instantiating an object of another class, etc. Maps into a Usage. If the keyword is one of the stereotypes of Usage (call, create, instantiate, send), then it maps into a Usage with the given stereotype.

Table 3.3-1: Kinds of Dependency

An Example (Figure 3.3-1):

Note: The connection between a note or constraint and the element it applies to is shown by a dashed line without an arrowhead. This is not a Dependency.

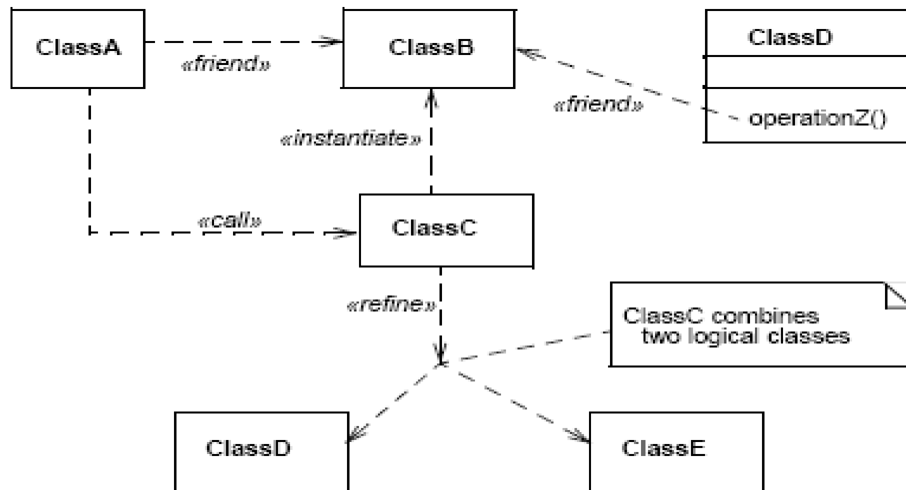

Figure 3.3-1: Various kinds of inter-class dependencies

3.3.2 Review of some basic elements of UML class diagrams

Class icons (simplest): rectangular boxes, each with its class name

Can be extended to depict the operations and attributes of a class. For example,

Book	*Name is in 1st (top) compartment*
title: String	*Attributes (2nd compartment) (with types)*
copiesOnShelf(): Integer borrow(c:Copy)	*Operations: (3rd compartment) - signature of each op. (selector, arguments, return type)*

Associations (simplest): straight lines between class icons concerned

Can be extended by adding various annotations including, in particular, label (name of association), arrows (navigability), and multiplicity.

Multiplicities: One may have an exact number, a range of numbers (2 dots between a pair of numbers), an arbitrary, unspecified number (denoted *), or a combination of these (comma separated list). The text (p63) gives the example

3, 12..15, 901..* meaning there can be, of the items concerned,

exactly 3 or else between 12 and 15 inclusive or else at least 901

Evolution during projects: Diagrammatic representations of classes and associations will evolve as a project proceeds, with more detail being added as appropriate. For example, initially one may not show navigability of an association. Similarly, class attributes and operations will emerge as responsibilities and interactions are analysed. One should avoid prematurely adding attributes that imply a particular implementation decision.

3.3.3 Suggested Guideline to determine class operations & attributes

Initially, identify the data conceptually associated with an object & what messages it seems reasonable it should understand (*If a book could talk, what questions would you expect it to answer!*). Subsequently, we have to check that we have included enough data and behaviour for the requirements at hand. For this, we have to start looking at how the objects work together to satisfy the requirements (see CRC cards, below, for example)

3.4 Generalization - "*is a* relationship" (see ref 1, sect. 5.4)

3.4.1 Design rules (or guidelines) for generalization

Recap of example: We had that LibraryMember is a generalization of MemberOfStaff, and we saw how to represent it on a UML class diagram (*arrow, with special arrow-head, directed from special to general*). Equivalently, we can say that MemberOfStaff is a specialization of LibraryMember. We observe

Needed in generalization	**Comment**
Object of class MemberOfStaff should conform to the interface given by LibraryMember	If some message is acceptable to any LibraryMember it must also be acceptable to any MemberOfStaff
A MemberOfStaff's interface may be strictly broader than LibraryMember's	MemberOfStaff may understand other, specialized messages which an arbitrary LibraryMember might not be able to accept

General design rules (informal): Guided by this example, reference 1 formulates a general design rule (guideline):

A	An object of a specialized class can be substituted for an object of a more general class in any context which expects a member of the more general class, but not the other way round

Reference 1 goes on to give the further design rule (guideline):

B	There must be no conceptual gulf between what objects of the two classes do on receipt of the same message.

This is a bit informal but what is in mind can be illustrated by the operation borrow(c:Copy) of the class LibraryMember from the case study. We expect (*and any client object should be able to assume*) that the way this operation is carried out by a MemberOfStaff object should be comparable to how it is carried out by a LibraryMember object. In the extreme, one could require that the way it is carried out by objects of both classes should be identical but this might be unduly restrictive. Instead reference 1 formulates the rules more precisely in terms of "substitutivity", as follows.

General design rules – design by contract 2: The basic idea is that an object of a subclass is supposed to be usable anywhere that an object of the superclass is usable,

or that *the subclass should fulfill* <u>*the contract*</u> *entered into by the superclass*. Hence, we may re-state rules A and B (above) as

A	(i) A subclass can have extra attributes and operations but it cannot drop any of those of its superclass. (ii) If a superclass has a class invariant (e.g. "a valid LibraryMember must always have numberOfBooksOnLoan ≤ maxBooksOnLoan"), then the subclass must have a class invariant that is "contained within" the class invariant of the superclass (or that is "at least as restrictive" as the superclass invariant).

and

B	(i) *Demand no more*: The <u>precondition</u> of a subclass operation must be no stronger than that of the corresponding superclass operation. (ii) *Promise no less*: The postcondition of a subclass operation must be at least as strong as that of the corresponding superclass operation.

Note that Rule B is concerned with where the subclass overrides any of the superclass's methods.

<u>**Example of a spurious generalization**</u>: English can be misleading re "*is a*" or generalization relationships (ref.1, section 5.4.1). We know that C is probably a generalization of B if "*every* B *is a* C". However, in English "is a" can sometimes be used misleadingly and so can suggest a generalization that is not actually there. E.G.

Misleading	**Need to be precise (pedantic?)**
"A Border Collie is a breed" (*grammatically wrong*)	"Border Collie is a breed" or "The Border Collie is a breed" (*grammatically correct*)
Conclude (falsely) that "breed is a generalization of Border Collie" (i.e. "breed" is a kind of dog!)	

3.4.2 Implementing generalization, inheritance & delegation

<u>**Inheritance**</u> (reference 1, section 5.4.2): A common way of implementing *generalization* is through *inheritance* (as can be done in Java (using "extend") and C++). At one time inheritance was considered by some people to be a really powerful mechanism ("*a silver bullet!*"), but there are drawbacks to its use that one needs to be aware of:.

<u>Potential drawbacks of inheritance</u>:

1. [*Fragile base class problem*] A subclass is dependent on its superclasses, so

using *inheritance tends to increase the degree of tight coupling* in a system. The consequences are that if a superclass is changed it will probably be necessary (a) to recompile its subclasses or even (b) to change their code. In case of (b), particularly, it will be necessary to redo tests. Also, there are implications for configuration control - keeping track of versions of source, object and executable code of the various classes.

2. [*Amount of testing of subclass*] If one is confident that a superclass is correct, *can one* reduce appreciably the amount of testing needed on a subclass? In fact, the answer may be "no" - we will return to this point later on (*time permitting*).

Recommended guideline for use of inheritance: Because of these drawbacks, it is ***recommended to use the following guideline***: Use inheritance between classes only when it models a conceptual generalization relationship (that is, essentially, a relationship reflective of the real-world to which the system applies). **Contrary to this guideline there is a** temptation to apply inheritance as a clever programming mechanism to make use of existing work, regardless of whether there is a significant or lasting connection between entities.

Reference 1 offers the example of where a class List is available and we want to create a class AddressBook in which the addresses are stored in a List. We have two solutions:

Poor solution (inheritance)	Better Solution ?
Make AddressBook inherit from List	Make AddressBook own a List (by having, for example, an attribute addresses: List)

Table 3.4-1: Contrasting solutions, one using Inheritance

Definitions of Inheritance and Delegation (Glossary of “OMG Unified Modeling Language Specification”, Version 1.5)):

delegation:

The ability of an object to issue a message to another object in response to a message. Delegation can be used as an alternative to inheritance.

Elsewhere in this specification we have we have the further explanation:

“An operation map (a set of operation-method pairs) is attached to an object. If a request type matches an operation in the map, the corresponding method is

executed. If the operation is not found, then a link points to another object that is then searched. This is the concept of *delegation*."

inheritance:

The mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior.

Implementing generalization: inheritance Versus *delegation?*: There is a lot of relevant material, on the internet and elsewhere, that is worth checking out. A useful way of thinking about it is to distinguish between *specification inheritance* (classifying concepts into type hierarchies) and *implementation inheritanc*e (just for the purpose of re-using code). The following example (from reference 2) illustrates "inheritance" and "delegation" solutions for a particular problem (in which, in fact, the "delegation" approach is preferable).

Example: Assume that Java does not provide a Set class (it does in fact) and that we need to write our own (called MySet). It's decided to re-use the java.util.Hashtable class to implement MySet. Among many other things, this class includes

boolean	**containsKey**(Object key) Tests if the specified object is a key in this hashtable.
boolean	**containsValue**(Object value) Returns true if this Hashtable maps one or more keys to this value.
Object	**put**(Object key, Object value) Maps the specified key to the specified value in this hashtable.

We focus on (a) inserting an element in MySet (operation **put**) and (b) checking if an element is in MySet (operation **containsValue**). (a) is equivalent to checking (**containsKey**) if the corresponding key exists in the table and creating an entry (**put**) if necessary. (b) is equivalent to checking (**containsKey**) if an entry is associated with the corresponding key.

The two alternative Java solutions are presented in Table 3.4-2.

Inheritance	Delegation

Class MySet **extends** Hashtable {
/* Constructor omitted */
MySet() {
}
 	
void put (Object element){
if (!containsKey(element)){
put (element, this);
}
}
boolean containsValue (Object element){
return containsKey(element);
}
/* Other methods omitted */
}
	Class MySet {
	private Hashtable table;
	MySet() {
	table=Hashtable();
	}
	void put (Object element){
	if (!containsKey(element)){
	table.put (element, this);
	}
	}
	boolean containsValue (Object element){
	return (table.containsKey(element));
	}
	/* Other methods omitted */
	}

Table 3.4-2: Example contrasting "inheritance" and "delegation

3.5 Class Responsibility Collaboration cards (ref 1, sect. 5.6)

This may be called a responsibility driven design (RDD) technique. It can help in checking how good a design is, and in refining it. There are various internet resources available including (e.g. http://www.softstar-inc.com/Methodology/CRCIntro.htm, last checked 14/10/2009). The card format is as follows:

Name of class	
Responsibilities of class	**Collaborators** of class

Responsibilities describe at a high level why the class exists. they are related to operations but more general, to begin with at least

No more than 4; else split up?

Collaborators are other classes that help to carry out responsibilities.

A design guideline: Too many responsibilities -> low cohesion

Too many collaborators -> high coupling

Examples (from case study):

LibraryMember	
Responsibilities Maintain data about copies currently borrowed Meet requests to borrow and return copies	**Collaborators** Copy

Copy	
Responsibilities Maintain data about a particular copy of a book Inform corresponding Book when borrowed and returned	**Collaborators** Book

Book	
Responsibilities Maintain data about one book Know whether there are borrowable copies	**Collaborators**

- CRC cards usage can help to identify collaborators and hence associations; also, navigability directions.

- Can use CRC cards to "walk through" the various scenarios of use cases in order to see how the class model provides the required functionality.

- Using CRC cards as above should help to identify missing bits, problems, etc

- As a result may have to modify responsibilities, collaborators, create new classes etc

Points to bear in mind:

a) Use of CRC cards may help in deciding to retain or reject classes suggested by noun identification technique.

b) As one (or, better, a team) uses CRC cards, look at the wider implications of any change. Don't fix just a local problem

c) Idea is to write modifications on CRC cards as one proceeds - eventually will have to write out cards freshly again (obviously)

d) Notice that the system *actually works by means of objects* (not classes); so need to be clear whether a CRC card always represents the same object of a class, or whether it represents several objects.

e) Use of CRC cards in a team can have benefits in terms of achieving a common understanding of the system, improving team spirit etc

f) A nice suggestion is to pile cards related by generalization together, with most abstract at the top, so that specialization is only used when relevant.

Another view on CRC cards, and on responsibility driven design [RDD]: Budd (*An introduction to object-oriented programming*) has:

1) Main aspect is delegation of responsibility

2) Responsibility entails (a) an expectation of certain behaviour according to certain rules and (b) a degree of independence or non-interference (-> want to cut links between different parts as much as possible)

3) Establish who is responsible for each action that is to be performed

4) During design, distinction between class and instance (object) is blurred

5) Importance of good class names (Pronounceable/ Capitalization or underscores to mark each new word in a name/ Make sure abbreviations are understood by all/ etc)

6) Responsibilities should be short verb phrases - they describe what is to be done, the problem to be solved)

7) If card size is insufficient it probably means class is too large & complex

8) Collaborators (a) must include classes from which class needs services, and (b) *possibly* classes to which class provides services

3.6 Refactoring (ref. 1, section 5.6.4)

This is the process of *altering the class model* of an object oriented design without altering its visible behaviour. A refactoring step might require moving an operation between classes and making all associated changes.

Something to watch for, in particular, is whether two classes have overlapping responsibilities and behaviour. If so, it might be best to put the common behaviour into a new superclass from which both could inherit. [*But bear in mind potential difficulties with inheritance noted before*]. Notice that the new superclass would not have shown up in the requirements specification - it arises from design considerations and not from the (problem) domain.

3.7 Class model evolution during development (ref. 1, sect. 5.5)

As a recap, we summarise the overall object oriented design process as

Start	**Repeat as necessary**	**Finish**
Record conceptual relationships & capabilities	Add more detail (*including of implementation*) Introduce new operations more specific associations attributes	Complete & consistent design

3.8 Preview of "8. More on Classes"

There are several points yet to cover on classes and on related UML notation. These will be taken up in section 8. For now, we just note some of the topics to be covered:

6.1 More about associations	Distinguishing types of associations (e.g. aggregation)/ Navigability/ Adding fine details/ Constraints etc etc
6.2 More about classes	Interfaces (say, to specify some operations that are visible outside a class)/ Abstract classes
6.3 Parameterized classes	Sometimes called a *template* - e.g. "List(T) might describe, given a class C to substitute for the formal parameter T, the class of lists of C objects"
6.5 Components & Packages (Also covered in section 9)	Typically, a package is a collection of related classes and dependencies between them. Convenient for purpose of work allocation, forming a component, etc.
6.6 Visibility, Protection	

3.9 Exercises

1. (p67 of reference 1): In Java and/or C++, if a subclass overrides an attribute or operation from a superclass, is it allowed to change their types? In what way?"

2. Assess the two solutions offered in Table 3.4-1 as follows:

(i) Compare effort required in initial development of the AddressBook

(ii) Compare the changes required if the implementation or interface of List changes

(iii) Compare the changes required if it decided to use a different class instead of List (Dictionary, say).

3. Compare the alternative solutions offered in Table 3.4-2.