ADVANCED SOFTWARE DESIGN
LECTURE 3
STRUCTURAL MODELLING
Dave Clarke
GOALS

• At the end of this lecture you will:
  • be able to do domain modelling
  • be able to use describe the structure of systems from class and deployment perspectives
  • understand and apply design principles
DOMAIN MODELLING
GOAL

• To gain a better understanding of the domain within which we are working

• To capture the key domain abstractions (classes)

• To capture real-world relationships (associations) between classes
IDENTIFYING CLASSES
IDENTIFYING CLASSES

• tangible or ‘real world’ things – book, copy, course

• roles – library member, student, director of studies

• events – arrival, leaving, request

• interactions – meetings, intersection
EXAMPLE

**Books and Journals**  The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

**Borrowing**  The system must keep track of when books and journals are borrowed and returned, enforcing the rules described above.
IDENTIFYING CLASSES

• Select key domain abstractions.

• First underline nouns and noun phrases – gives candidate classes
IDENTIFYING CLASSES

Books and Journals The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

Borrowing The system must keep track of when books and journals are borrowed and returned, enforcing the rules described above.
DISCARDING CANDIDATES

• **redundant** – encompassed by some other compass, perhaps differing by attributes: *loan* vs *short term loan*. Select most general.

• **vague** – e.g., *item*

• **an event or operation** – something done by system. If it has no state, discard it.

• **meta-language** – language used to define things. *requirements*, *system*, ...

• **outside the scope of the system** – e.g., *library*, *week*, ...

• **an attribute** – something simple, an attribute of another class. *name* of library member
REMAINING CLASSES

- book
- journal
- copy (of book)
- library member
- member of staff
RELATIONS BETWEEN CLASSES ASSOCIATIONS
WHY ASSOCIATIONS

• Clarify out understanding of the domain by describing objects in terms of how they work together

• Sanity check the coupling in our system – too make sure we are following good principles in modularising our design (later)
FINDING ASSOCIATIONS

Classes A and B are associated if:

• 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 receive a message with an object of class B as an argument

In short: if class A has to know something about some object of class B
FINDING ASSOCIATIONS

• a copy is a copy of a book
• a library member *borrows*/*returns* a copy
• a member of staff *borrows*/*returns* a copy
• a member of staff *borrows*/*returns* a journal
Representing in UML
MULTIPLICITIES

• Express more refined relationships between classes
  • an exact number
  • a range of numbers
  • an arbitrary, unspecified number
• Give guidance to implementation.
MODEL WITH MULTIPLICITIES

LibraryMember

MemberOfStaff

Copy

Journal

Book

is a copy of

borrows/returns

borrows/returns

borrows/returns

Sunday 3 November 13
ATTRIBUTES
ATTRIBUTES

• State of objects (excluding associations)

• Generally only of primitive type: ints, strings, enums, ... or library classes

• Associations are not modelled as attributes – premature decision about navigability
MODEL WITH ATTRIBUTES

```
MODEL WITH ATTRIBUTES

<table>
<thead>
<tr>
<th>0..1</th>
<th>0..1</th>
<th>0..1</th>
</tr>
</thead>
<tbody>
<tr>
<td>borrow/returns</td>
<td>borrow/returns</td>
<td>borrow/returns</td>
</tr>
</tbody>
</table>

LibraryMember

memberId : Integer

Copy

Book

title : String

1..* 1

is a copy of

Journal

title : String
series : Integer
volume : Integer
```
GENERAL GUIDELINES

- Model just domain artefacts, not software artefacts
- Getting it perfect isn’t a requirement – not cost effective
- When classes and their behaviour are modelled in more detail, resulting class model with deviate from domain model.
MODELLING CLASSES
OVERVIEW

- Make domain model more concrete – class model
- Assign responsibilities – fill in operations – methods.
- Introduce generalisations/inheritance – reuse
- Model software artefacts
ASSIGNING RESPONSIBILITIES
RESPONSIBILITIES (CRC)

• Write (on a card) the name of a class, the responsibilities of that class, and the collaborators, other classes that will help carry out each responsibility.

• Responsibilities – high level purpose of class
  • Related to use cases, but oriented towards classes.
  • Each responsibility may be realised by many methods

• Guideline: each class should not have more than three or four responsibilities – poor cohesion

• Guideline: generally class should have only a few collaborators – poor coupling
## CRC CARDS

<table>
<thead>
<tr>
<th>LibraryMember</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Responsibilities</strong></td>
</tr>
<tr>
<td>Maintain data about copies currently borrowed. Meet requests to borrow and return copies.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Responsibilities</strong></td>
</tr>
<tr>
<td>Maintain the data about a particular copy of a book. Inform corresponding Book when borrowed and returned</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Responsibilities</strong></td>
</tr>
<tr>
<td>Maintain data about one book. Know whether there are borrowable copies.</td>
</tr>
</tbody>
</table>
OPERATIONS
OPERATIONS

• Add in obvious operations (and their types).
• For each responsibility, determine operations required to realise it.
• Consider also use cases – what operations are required to support them.
• Will be refined when doing dynamic modelling.
EXAMPLE WITH OPERATIONS
GENERALISATION
GENERALISATION

• Identify classes that are generalisations of others or perhaps classes that have a common generalisation.

• Leads to opportunities for reuse.

• Often, but not always, implemented using inheritance.

• Test: *Every A is a B.*

• A dog is an animal  →  every dog is an animal ✔

• A collie is a breed  →  every collie is a breed ✗
CAPTURING COMMONALITY

**LibraryMember**
- memberId : Integer

**MemberOfStaff**
- borrows/returns

**Copy**
- borrows/returns

**Book**
- copiesOnShelf() : Integer

**Item**
- title : String
- borrow(c: Copy)
- return(c: Copy)

**Journal**
- series : Integer
- volume : Integer

is a copy of

1

1..*
GUIDELINES

• An object of a specialised 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 around.

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

• Demand no more: promise no less.
BAD GENERALISATION

• *AddressBook* is essentially a *List* of *Addresses*

• Therefore:

\[
\text{List} \quad \downarrow \\
\text{AddressBook}
\]

Why is this bad?
Better solution?
REFACTORING
REFACTORING

• Refactoring is the process of restructuring code/designs (without altering externally observable behaviour).

• Continuously improve the design to make it easier to work with – we’ll see many principles to help us here.
AGGREGATION AND COMPOSITION
AGGREGATION

• Conceptual notion of whole-part: one object is a part of another.
COMPOSITION

• Refinement of aggregation: the whole *strongly owns* the part.

• If whole is deleted, then so are all the parts.
GUIDELINE

• Don’t overuse, especially early in the design process.

• If in doubt, use a plain association.
MORE POSSIBILITIES
MORE ASSOCIATIONS

• Roles – increase readability by giving classes a role in association

• Navigability – whether instance of class can be reached by instance of other (denoted by an arrow and a cross)

• Constraints – express more detailed constraints between classes. e.g., Parent–Child, we have: \( p.child.parent = p \)

• Association classes – more later
MORE “CLASSES”

- Interfaces
- Abstract classes
- Parameterized classes
- Association classes – classes associated with each association

As in Java
ASSOCIATION CLASSES

Record information about associations

Student 1..* is taking 6 Module

GradeBook
grade : Integer
MODELLING DEPLOYMENT
DEPLOYMENT DIAGRAMS

- Express the physical deployment of software artefacts to hardware nodes – static view of run-time configuration

- Use when application spans several machines.

- Nodes correspond to
  - devices (e.g., servers, mobile devices)
  - specific execution environments (application servers, rule engines, operating system, virtual machines, database engines, web browser).

- Nodes connected by communication paths (middleware, protocol)
Node: WebServer

Component: Student Administration <<JSPs>>

Dependency: <<RMI>>

Component: ApplicationServer [OS=Solaris]

Component: EJBContainer

Components:
- Student
- Seminar
- Schedule

Deployment Spec:
- Registration
- execution: thread
- nestedTransaction: true

Persistence:
- <<infrastructure>> [vendor=Ambysoft]

Components:
- Course Management Facade <<web services>>

Dependency: <<JDBC>>

Component: DBServer [OS=Linux]

Components:
- University DB
- <<database>> [vendor=Oracle]

Dependency: <<message bus>>

Component: Mainframe [OS=MVS]

Components:
- Course Management <<legacy system>>
GUIDELINES

1. **Identify the scope of the model** – single application, group of applications?

2. **Consider fundamental technical constraints** – existing systems, robustness, who/how users connect, middleware, hardware, software, ...

3. **Identify distribution architecture** – Fat client, thin client, 3-tier?

4. **Identify nodes and their connections** – What kinds of nodes do you have? How connected?

5. **Distribute software to nodes** – indicate also critical information for whoever implements or deploys system
DESIGN PRINCIPLES
DESIGN PRINCIPLES

- Goal: to produce better designs
- Many formulations exist, we’ll look at some
  - SOLID
  - YAGNI
  - DRY
  - KISS
  - ...
SOLID
SINGLE RESPONSIBILITY

• A class should have a single responsibility.
OPEN/CLOSED PRINCIPLE

- Software entities should be open for extension but closed for modification.
LISKOV SUBSTITUTION PRINCIPLE

• Objects should be replaceable with instances of their subtypes without altering correctness.

• Related: design by contract.
INTERFACE SEGREGATION
PRINCIPLE

• Many client-specific interfaces are better than one general-purpose interface
DEPENDENCY INVERSION PRINCIPLE

• Depend upon abstractions, not upon concretions.

• Decoupling

• Program to an interface, not an abstraction.

• Dependency injection is one method of following this principle.
YAGNI
YAGNI

• “You ain’t gonna need it.”

• Implement only things that you will need.

• Do the simplest thing that could possibly work.

• From Extreme Programming.
DRY
• “Don’t repeat yourself.”

• Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

• Not well supported in existing languages – relies heavily on code generators, build systems, scripting languages to enforce.
KISS
KISS

• “Keep it simple, stupid” or

• “Keep it simple stupid.”

• Avoid unnecessary complication in systems.
OTHER PRINCIPLES

• Minimise the accessibility of classes and members
  • Abstraction, encapsulation, information hiding
• Favour composition over inheritance