BSc Business Computing Systems,

BSc Computer Science (all strands), MSci (all strands),

2nd Year Team Project (IN2018),

2017-2018

Briefing on System Design

Dr Vladimir Stankovic

Centre for Software Reliability

Room: A304j, Ext.: 3079

E-mail: Vladimir.Stankovic.1@city.ac.uk

www.city.ac.uk/people/academics/vladimir-stankovic

Technical material

- All technical material needed for System Design (and Requirements Specification) has been covered in previous modules
  - IN1005, IN1010 and IN2013

- Suitable coverage/support is provided in this module too (in part due to the IN2013 module feedback)!
  - You would not want us to repeat the material
  - NB: Let me know, if you want me to ask the other lecturers to give you Observer access on the currently running modules, e.g. IN1010.

BAPERS: System Design

- System Architecture
- *Detailed* Design
  - **static** view on the system:
    - detailed class-diagram (in terms of software artefacts);
    - database design (ER diagram + database schema);
    - GUI visual design.
  - **dynamic** view on the system
    - representative set of SQL statements
      - including the ones for generation of 2 reports!
    - (optionally) activity diagrams for GUI navigation
      - but, other approaches to GUI navigation are allowed.

BAPERS: System Design (cont.)

- System Architecture
  - package the classes to achieve ***loose coupling*** between the packages (i.e. minimise the associations that cross the package boundaries)
  - consider clean division between different domains:
    - problem domain classes,
    - ‘visual’ classes (GUI), and
    - database connectivity
  - may use design patterns, e.g. ‘façade’, MVC, ‘chain of responsibilities’, etc.

Design Class Diagram

- Organise classes based on:
  - Common behaviour;
  - Common attributes (name, address, phone, …)
- Introduce explicitly **interfaces** between domains, especially between problem and database domains
- Organise classes into packages based upon:
  - Common actor interacting with the classes
  - Common business process
  - Class similarity and function (library, common control)
  - May include third party libraries as packages
- Add separate GUI and DB Connectivity domains

Design Class Diagram (cont.)

- From the Student's Brief document:

"

Fully refined and correct Design class diagram(s) showing Entity, Boundary (i.e. GUI) and Control classes, associations (including roles and navigability), cardinalities, methods (i.e. operations) and attributes. A complete set of operations should be specified including: parameter lists, return types, visibility, exceptions, set and get operations, constructors and destructors. Also a complete set of attributes including types and default values must be provided.

…

"

What are design classes?

(IN2013 – Lecture 3)

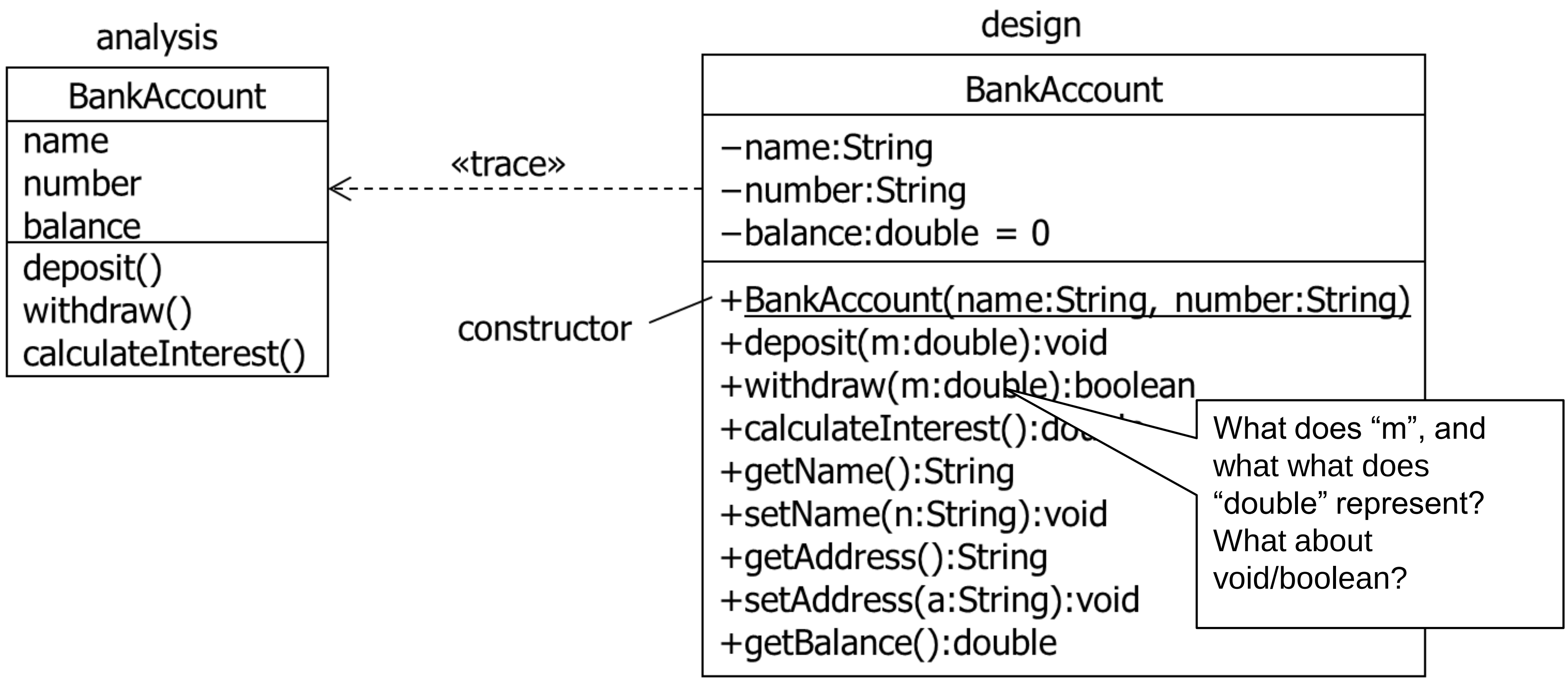
- Design classes are classes which specifications have been completed to such a degree that they ***can be implemented***
  - A design class ***specifies*** an actual piece of code
- Design classes arise from analysis classes:
  - Recall: analysis classes arise from a consideration of the problem domain *only*
    - *You are not required to explicitly produce Analysis class diagram in the document – **what we ask for is Design class diagram**! BUT you must go through the process of identifying (Analysis) classes!*
  - A refinement of analysis classes to include **implementation details**
  - One analysis class may become **many design classes** (e.g. in design you may add “controllers”)
  - All attributes are completely specified including **type, visibility** and **default values**
  - Analysis operations become **fully specified operations** (methods) with scope, a return type, list of parameters and their types, visibility, exceptions; you need *set* and *get* operations, constructors and destructors.

What are design classes? (cont.)

(IN2013 – Lecture 3)

- Design classes arise from the solution domain
  - Utility classes – String, Date, List etc.
  - Middleware classes – database access, communications etc.
  - GUI classes – Applet, Button etc.

Anatomy of a design class



Need to produce Design Class Diagram, but

- Finding classes is a necessary step
- Recall IN1005/IN2013 material
  - Perform noun/verb analysis on documents, for example:
    - Nouns are candidate classes
    - Verbs are candidate *responsibilities*
    - Another possibility is use of Class Responsibility Collaboration (CRC) analysis
  - Beware of spurious classes:
    - Look for synonyms - different words that mean the same
    - Look for homonyms - the same word meaning different things
  - Look for "hidden" classes!
    - Classes that don't appear as nouns
    - or as cards in CRC analysis

Need to produce Design Class Diagram, but

- **Robustness analysis** is useful
- Recall IN2013 material (week 2)
- Walk through the flow of each use case and identify 3 kinds of classes:
  - **Boundary** classes – actors use these to communicate with the system
  - **Entity classes** – these come from the domain model and often represent persistent data
  - **Control classes** – represent the application logic and glue together the interface/boundary and entity classes

- Robustness analysis gives you:
  - A first guess at what the right (analysis) classes *might* be
  - A check that your use case flow can actually be realized
  - Ideas about the user interface

Sources of design classes

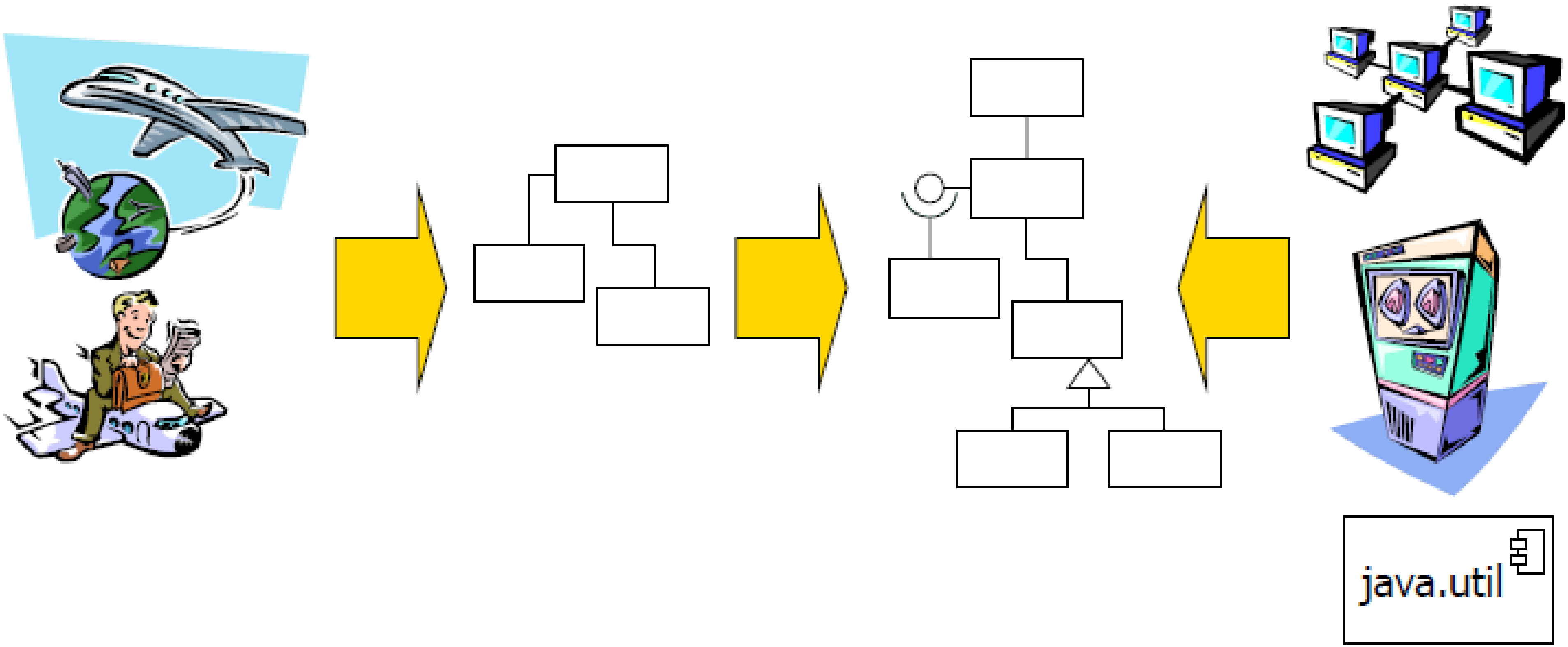
(IN2013 – Lecture 3)

Problem domain

Analysis classes

Design classes

Solution domain



Design classes - continued

(IN2013 – Lecture 3)

- Design classes come from:
  - A refinement of analysis classes (i.e. the business domain)
  - The solution domain
- Design classes must be well-formed:
  - Complete and sufficient
  - Primitive operations
  - High cohesion
  - Low coupling
- Don’t overuse inheritance
  - Use inheritance for "is kind of"
  - Use aggregation for "is role played by"
  - Multiple inheritance should be used sparingly (mixins)
    - This is mainly an option in case the programming language planned for implementation supports multiple inheritance.
    - Some languages, e.g. Java, do not support multiple inheritance between classes.
  - Use interfaces rather than inheritance to define contracts

Design classes – continued

(IN2013 – Lecture 3)

- Aggregation
  - Whole-part relationship
  - Parts are independent of the whole
  - Parts may be shared between wholes
  - The whole is incomplete in some way without the parts
- Composition
  - A strong form of aggregation
  - Parts are entirely dependent on the whole
  - Parts may not be shared
  - The whole is incomplete without the parts
- One-to-many, many-to-many, bi-directional associations and association classes are refined in design
  - State Roles of an association (see the SB document, Section 6.2)

Completeness, sufficiency and primitiveness

- Completeness:
  - Users of the class will make assumptions from the class name about the set of operations that it should make available
  - For example, a BankAccount class that provides a `withdraw()` operation will be expected to also provide a `deposit()` operation!
- Sufficiency:
  - A class should never surprise a user – it should contain exactly the expected set of features, no more and no less
- Primitiveness:
  - Operations should be designed to offer a ***single primitive, atomic service***
  - A class should ***never offer multiple ways*** of doing the same thing:
    - This is confusing to users of the class, leads to ***maintenance burdens*** and can create consistency problems
  - For example, a BankAccount class has a primitive operation to make a *single deposit*. It should ***not*** have an operation that makes *two or more deposits* as we can achieve the same effect by repeated application of the *primitive* operation

The public members of a class define a "contract" between the class and its clients

High cohesion, low coupling

- High cohesion:
  - Each class should have a set of operations that support the ***intent of the class***, no more and no less (remember CRC cards)
  - Each class should model a single abstract concept
  - If a class needs to have ***many responsibilities***, then some of these should be implemented by “helper” classes. The class then ***delegates*** to its helpers
- Low coupling:
  - A particular class should be associated with just enough other classes to allow it to realise its responsibilities (CRC cards may be useful here)
  - Only associate classes if there is a true ***semantic link*** between them
  - Never form an association just to ***reuse a fragment of code*** in another class!
  - Use aggregation rather than inheritance (use inheritance for cases “is kind of”, not “role played by” cases)

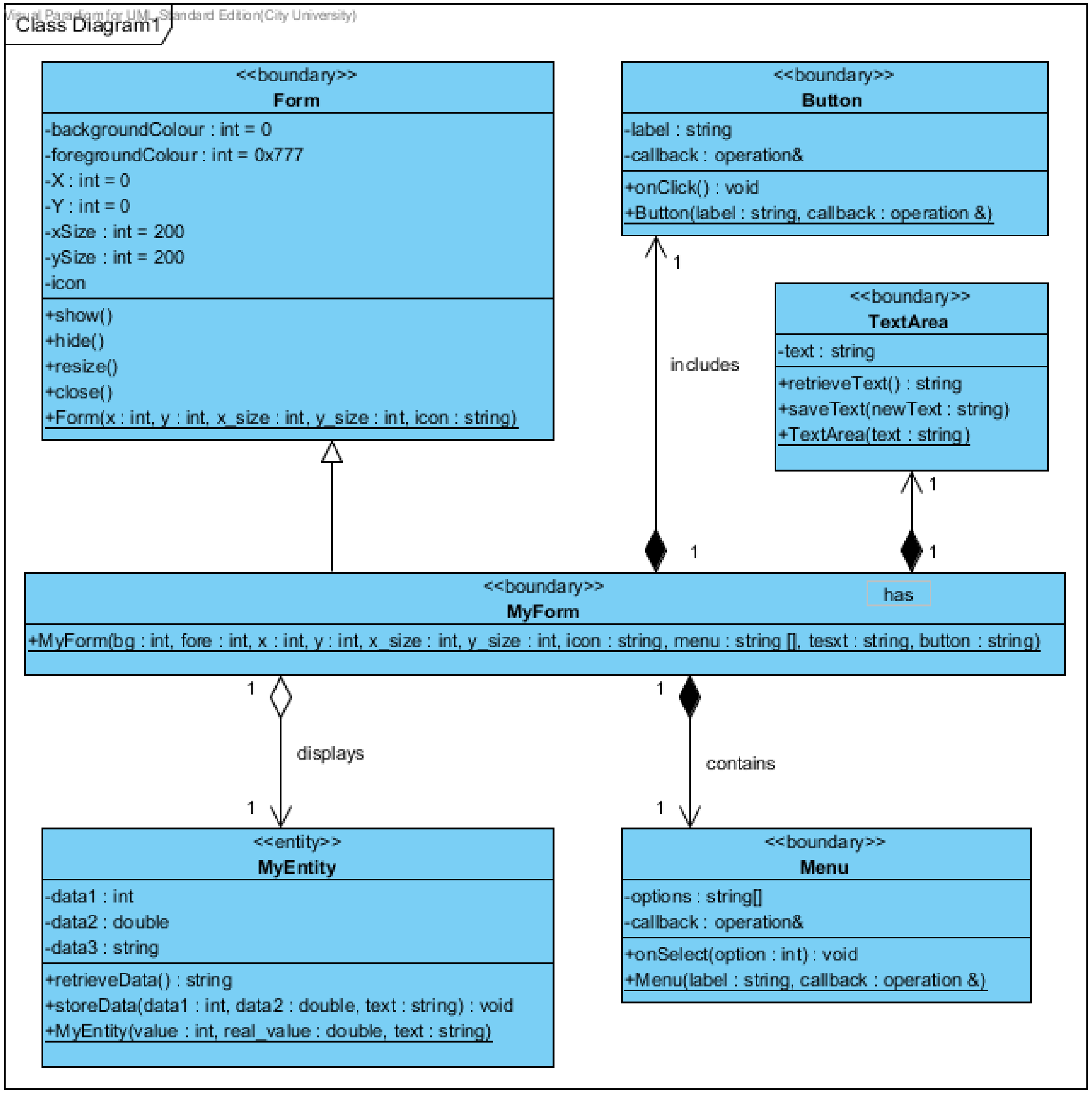
Class Attributes and Methods

- Map **actions** of use-cases to **methods** of a class.
- Map use-case **data** to either:
  - class attributes, or
  - associations (simple associations, inheritance, whole-part)
  - remove from classes the attributes, which represent relationships (relationships will eventually be removed in implementation, but must be used in design):
    - Associations must be refined into whole-part (aggregation, composition) relationships, if applicable;
    - Add **navigability** to aggregations and compositions (if/how the instances of the associated classes can access each other).
- Syntax:
  - Class names should be unique (Package name is part of a class name).
  - Operations names need not be unique.
  - Pay attention to abstract (virtual) methods and their being redefined in sub-classes.
  - Attributes names need not be unique but their type should be specified.

Modelling GUI in the class diagram

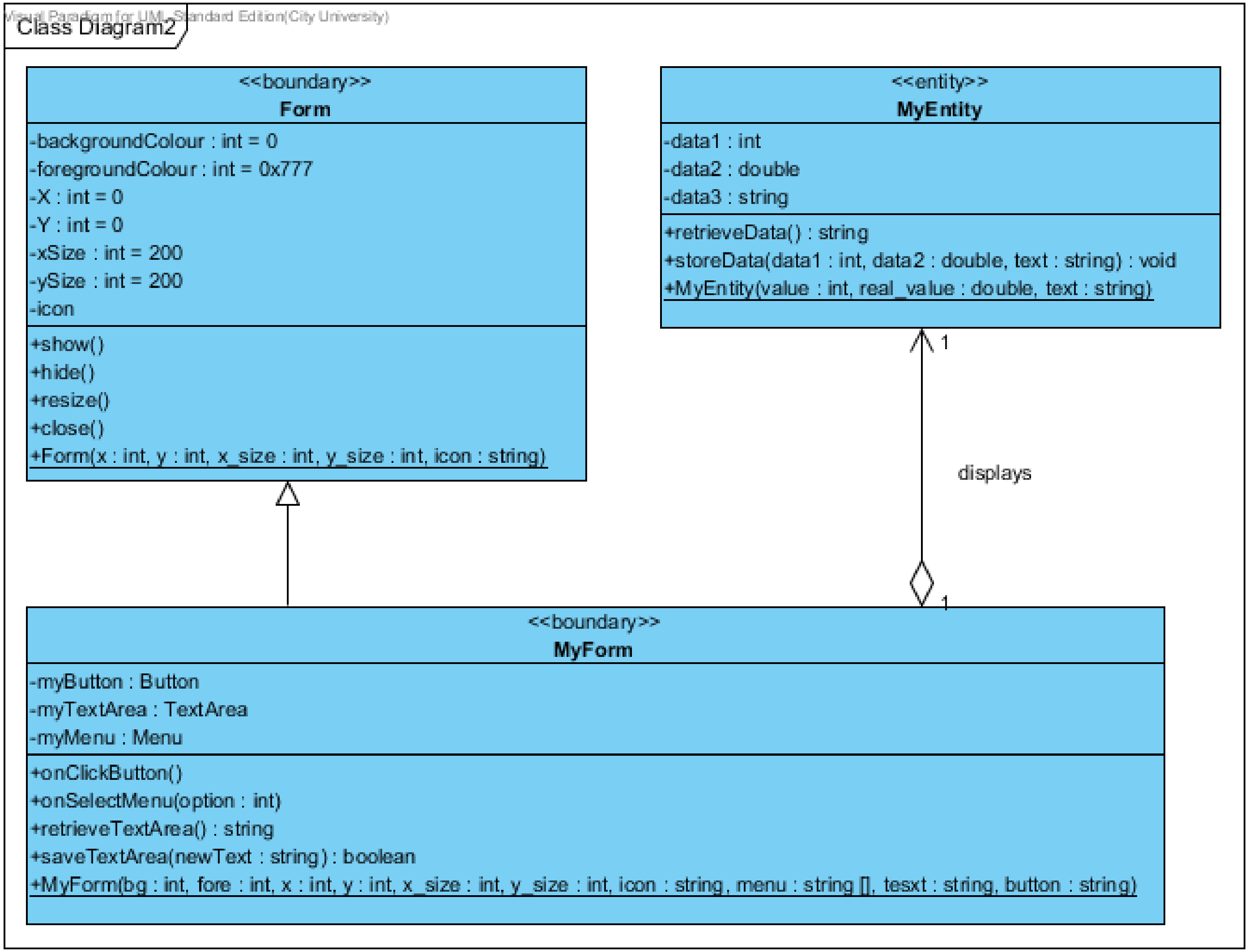
- Do not use ‘Mouse’, ‘Screen’, ‘Keyboard’ classes: this is a too low level of modelling, not useful (especially when RAD/IDE tools are used)!
- GUI must be modelled as a set of ‘visual classes’ (Forms, Frames, etc.) in a GUI package.
  - All forms ***inherit*** from a ‘basic’ form: can be displayed, closed, refreshed, etc.
  - GUI will be implemented using a RAD tool / IDE (Visual Studio, NetBeans, Eclipse and alike) and only minimum details are required as class design:
    - a form is a ***composition*** of visual controls (Buttons, TextBoxes, grids, etc.) which are provided by the RAD tools (e.g. `java.awt` and `javax.swing`, or MFC, VCL or .NET library under Windows)
- Specific forms can have their own attributes and methods. List their specific ‘`clicks()`’ and their ‘`enters()`’ as separate methods

Modelling of GUI classes: An example



- Useful and possibly correct.
- But it ***does not scale to cases with many forms***. Your class diagram may become difficult to read and use.
- I expect you to develop in detail ***ONE form*** using the fragment as a guide, and then use a simplification for the rest of the GUI forms.
  - An example of an acceptable simplification is given on the next slide.
- Feel free to ‘invent’ your own simplifications.

An ‘acceptable’ simplification

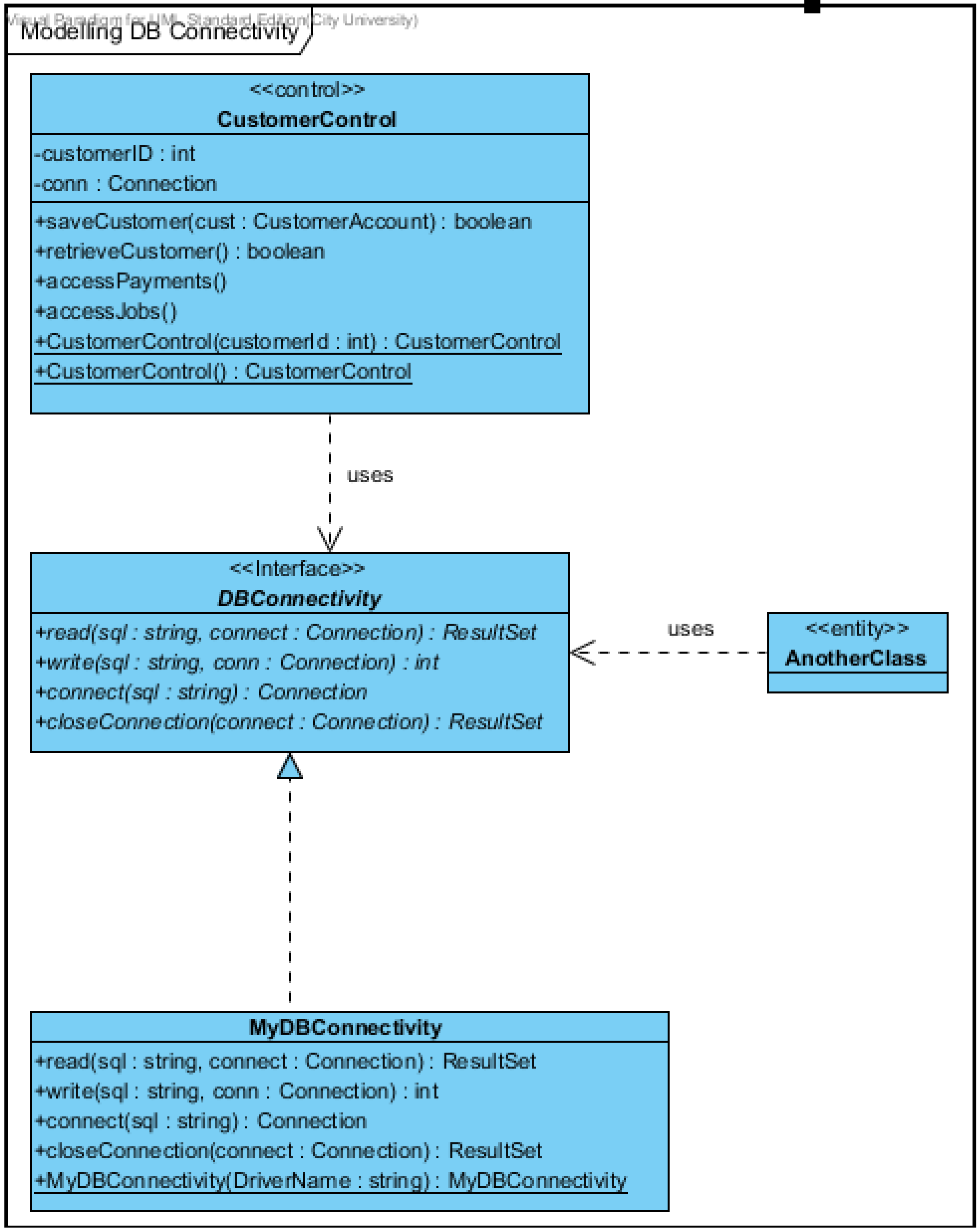


- All forms inherit from a generic form
- The visual controls used in the individual forms (Buttons, TextAreas, etc,) are shown as ***attributes*** of the forms of appropriate types (Button, TextArea, etc)
- The main operations of the visual controls including events, such `onClickButton()` are shown as methods of the form (‘*delegation of responsibilities*’).

Modelling the Interaction with DB

- Must show how the classes in the **problem domain** interact with those in the **database domain**, using interfaces and classes, collectively known as ***DB connectivity***:
  - connection class
  - transaction class (isolation level)
  - Statement class (execute (SQL) query)
  - ResultSet, etc.
- As a minimum show the ***interface(s)*** which the problem domain classes will use to communicate with a database. This/these interface(s) will be **implemented** by the DB connectivity class(es):
  - **No need to go into deep implementation specific details** (JDBC, ADO/OLE DB, ODBC or native connectivity to a particular database).
  - Documenting a **particular design decision**, e.g. having a limited pool of connections created at start-up vs. establishing a new connection before a transaction is started, **may require more details to be provided**.
    - Detailed solutions along these lines are very welcome but ***NOT*** mandatory.

A simplified example



- DB access is done via an interface, (e.g. `DBConnectivity`), which is implemented (realised) by an implementation class.
- Objects that need access to DB will do that by accessing an ***abstract data type*** defined by the interface. This is modelled as dependence between the entity/control classes and the interface, the weakest relationship between classifiers.
- This trick ***can be implemented*** by ***casting*** the instances of the implementation class to the data type defined by the interface.
    - "Using an Interface as a Type"
        - https://docs.oracle.com/javase/tutorial/java/IandI/interfaceAsType.html

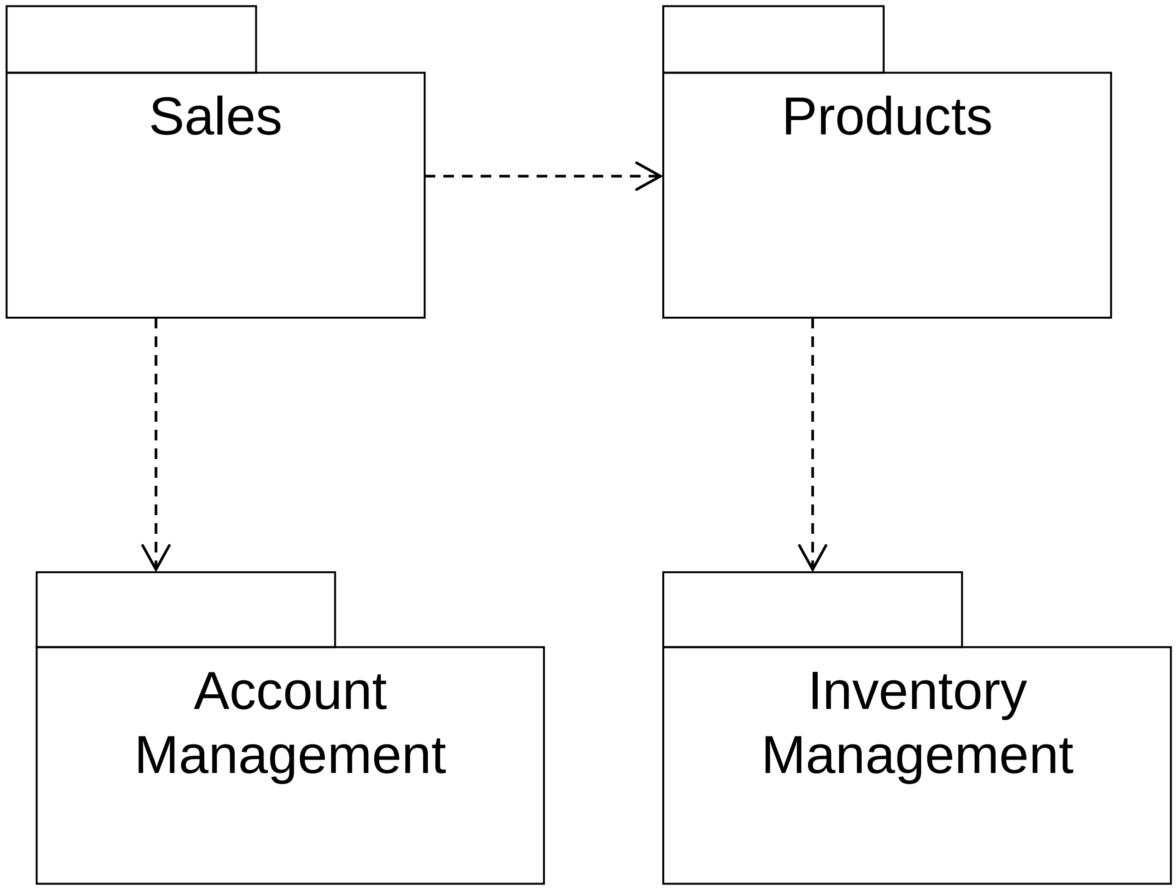
Packages in class diagram

- Packages are needed to:
  - manage large number of modelling elements, e.g. classes;
  - provide optional functionality;
  - minimise effects of change.
- Packages should achieve:
  - **Tight coupling between the classes within the package**. E.g. have a ‘GUI’ package with all the forms and a ‘Database Connect’ package with all the interfaces and (implementation) classes needed for interacting with the database.
  - **Weak coupling between packages.**
- Packages can be nested recursively

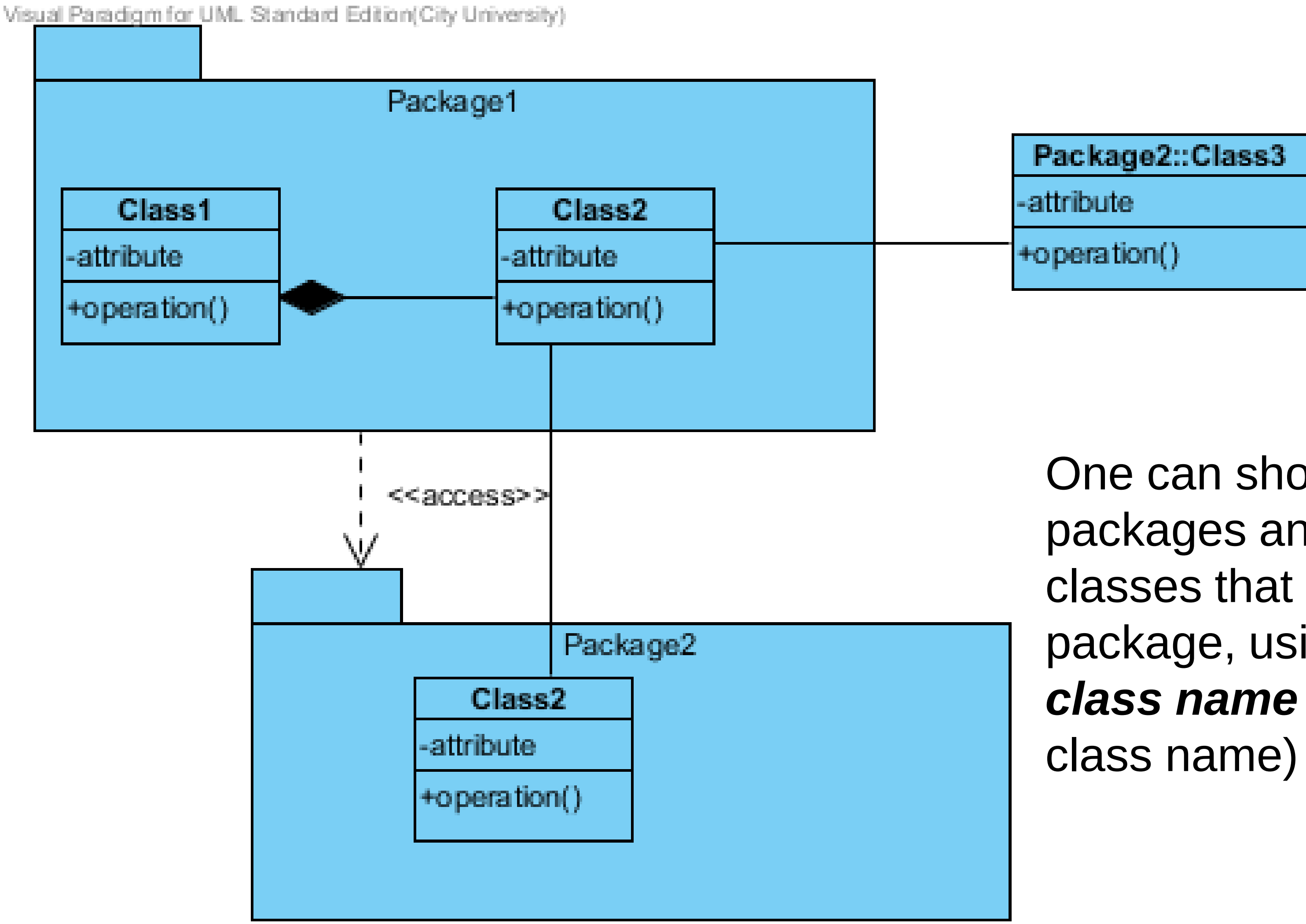
Dealing with complexity

- You need to use packages with class diagram.
  - Start with a ***package diagram*** and show the dependencies between packages
  - There may be associations between classes that are defined in different packages.
    - Make sure that you model the associations that cross the package boundaries!
    - Use the fully qualified class names (`package::class`) when necessary.

Package diagrams



Associations between



One can show a combination of packages and classes and even classes that do not belong to a package, using a ***fully qualified class name*** (e.g. a package and class name)

Summary of Design Class Diagram

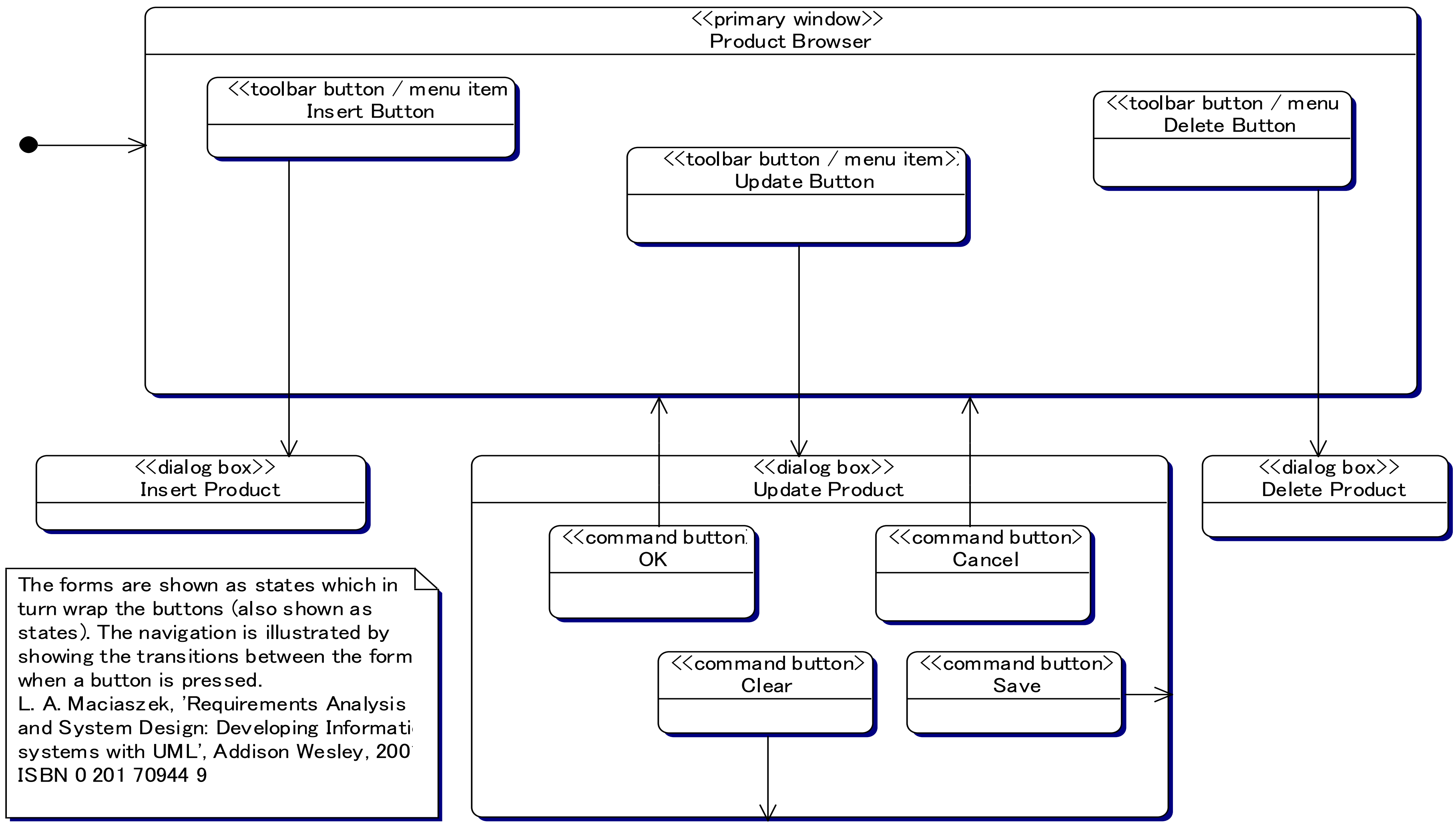
- Revise in detail relevant lectures of the IN2013 module (e.g. Lecture 3) and IN1005.
- Again, remember that the case study that you will use in the Team Project is more complex than the tutorials and coursework you have dealt with in previous modules, e.g. IN2013.
- Hence, make sure you start early, and work closely with the other members of your team.

User Interface (GUI) <u>Visual Design</u>

- Screen layouts must be provided for **ALL** forms/menus/etc. to be used in BAPERS. Must give a reference to the class diagram - **map** the “screens” to the corresponding classes
  - The “screens” are the visual appearances of the corresponding classes (forms) which must be included in the class diagram.
- Document how the user **navigates** through various forms (e.g. pressing button X takes the user to form Y).
  - Draw hierarchy or map (e.g. a tree) showing how the user can navigate via the forms, menus etc., OR a UML Activity diagram (see relevant first year IN1005 notes for details, Session 8 week 9, I think)
  - This is **in addition** to the associations between the ‘visual’, i.e. UML `<<boundary>>`, classes shown in the class diagram.
- Need to explicitly **map** these “screens” (i.e. forms/windows) to the boundary classes shown in the class diagram
- Keep the visual appearance clean, consistent (colour and layout), attractive and easy to use.

GUI navigation (UML Activity Diagram)

(Note: the following diagram is not drawn with MagicDraw or Visual Paradigm, so notations may differ)



Database Design

- Database design should result in the following being included in the design document:
  - ER diagram
  - *Relational* DB schema (3rd normal form) should be documented:
    - CREATE TABLE statements for ALL tables in the database;
    - state clearly the ‘SQL dialect’ used (Oracle, or MySQL, or ?)
    - the primary and foreign key(s) clearly identified
    - be aware of mistakes introduced by conversion tools;
    - OO database is not to be used!
  - A representative set of operational SQL statements used for:
    - data manipulation
    - achieving persistence of the main problem domain entity classes
    - generating some of the required reports.

Database Design (cont.):
Operational set of SQL statements

- A representative set of the following types of SQL statements (**2 statements per type**), with meaningful values:
  - SELECT - DELETE
  - UPDATE - INSERT
- *In addition*, the full set of SQL statements must be provided which, if executed (in a transaction), will produce a required report. This must be provided for ***2 of the non-trivial reports specified*****.**
- Design Hints & Tips for producing SQL statements that generate reports:
  - The queries used are likely to be SELECT statements, but you may need to store some intermediate results in *temporary tables* and/or database *views* (in which case all related statements, including respective CREATE statements, must be provided).
  - You may also use *stored procedures (SP)* for reports, in which case you will need to specify the type of the target SQL server (e.g. Oracle, MS SQL, etc.).

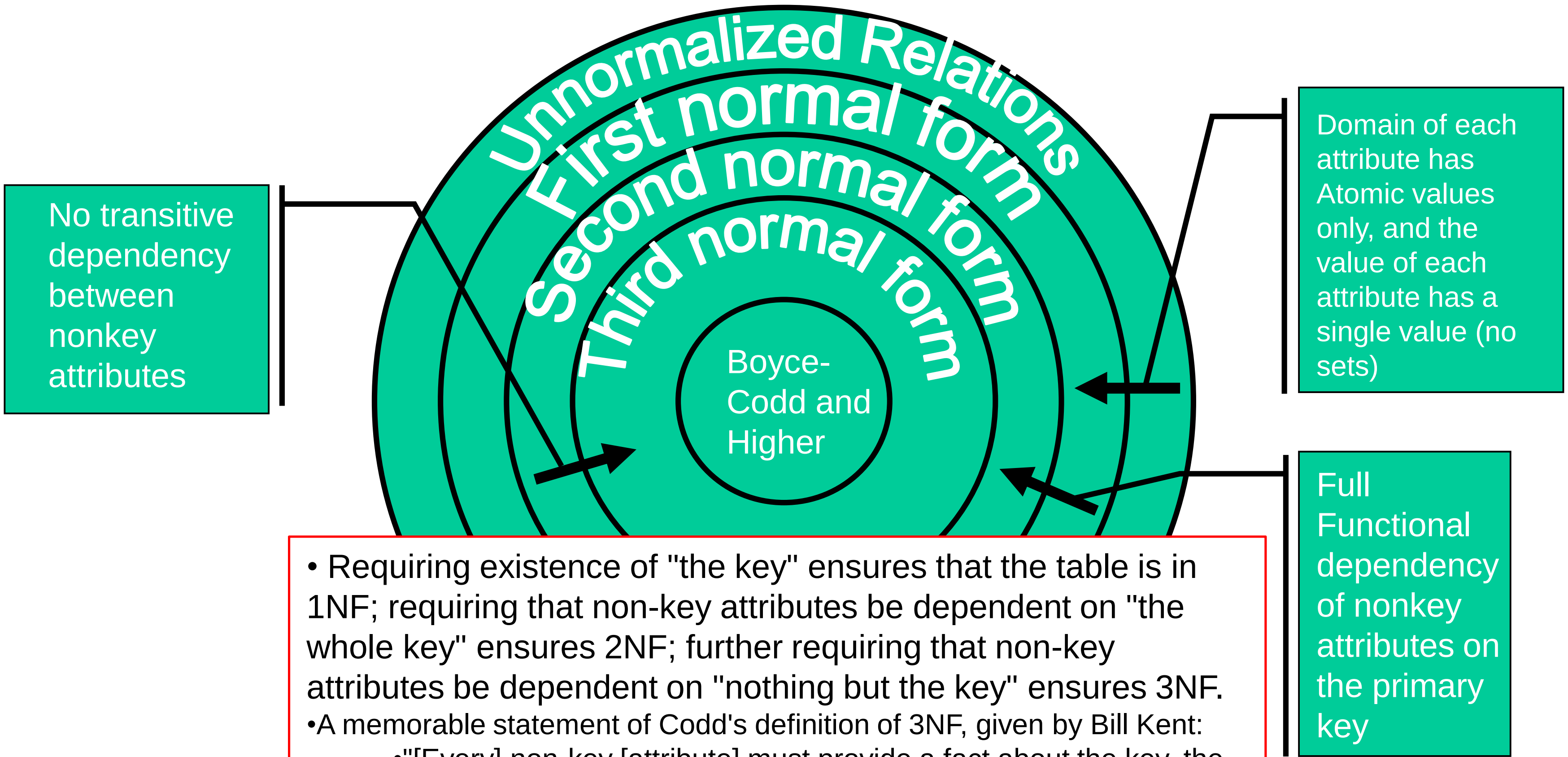
Database Normalisation
(mentioned in IN1010)

- The process of organizing the attributes and tables of a relational database to minimize data redundancy.
- Normalization theory is based on the observation that relations with certain properties are more effective in inserting, updating and deleting data than other sets of relations containing the same data
- Normalization is a multi-step process beginning with an “unnormalized” relation

Database Normalisation (cont)

- **First normal form (1NF)** is a property of a relation in a relational database. A relation is in first normal form if the domain of each attribute contains only atomic values, and the value of each attribute contains only a single value from that domain
  - https://www.youtube.com/watch?v=K7vzLrGCV50&list=PLQ9AAKW8HuJ5m0rmHKL88ZyjOIKejvrj0&index=1
- A table is in **2NF** iff it is in 1NF and no non-prime attribute is dependent on any proper subset of any candidate key of the table.
  - A non-prime attribute of a table is an attribute that is not a part of any candidate key of the table.
  - I.e. a table is in 2NF if and only if it is in 1NF and every non-prime attribute of the table is dependent on the whole of every candidate key.
  - https://www.youtube.com/watch?v=A9sezRxNhWY&list=PLQ9AAKW8HuJ5m0rmHKL88ZyjOIKejvrj0&index=2
- **3rd NF:** reduces the duplication of data and ensures referential integrity by ensuring that the entity is in 2nd NF, and all the attributes in a table are determined only by the candidate keys of that table and not by any non-prime attributes.
  - https://www.youtube.com/watch?v=GP_RcibUicQ&index=3&list=PLQ9AAKW8HuJ5m0rmHKL88ZyjOIKejvrj0

Database Normalisation (a reminder)



- Requiring existence of "the key" ensures that the table is in 1NF; requiring that non-key attributes be dependent on "the whole key" ensures 2NF; further requiring that non-key attributes be dependent on "nothing but the key" ensures 3NF.
- A memorable statement of Codd's definition of 3NF, given by Bill Kent:
  - "[Every] non-key [attribute] must provide a fact about the key, the whole key, and nothing but the key…"

- A relation is in Third Normal Form if there is no transitive functional dependency between nonkey attributes
  - When a nonkey attribute can be determined with 1 or more nonkey attributes, there is a *transitive functional dependency*
- *The image adapted from "Physical Database Design" lecture of the module* ***257. Database Management,*** *from the School of Information Management and Systems, University of California, Berkeley.*

Database design – relevant material

- Relevant material (from previous modules)
  - Recall the relevant, considerable material from IN1010
    - Whole of Semester 1; Lectures on MySQL in Semester 2
- But, you ought to conduct self-directed studies and actively seek solutions!
    - This holds for other sections of the Design document too, and the rest of the project!
- A reminder: 150 hours on a 15-credit module, including contact hours and self-directed studies
  - This is a double, 30-credit module – thus, 300 hours including the contact hours. There are about 30 contact hours, so if we consider 12-week semester, each student needs to spend additional 22.5 hours every week on the module!

Implementation Constraints

- Factors to consider in your design
  - Underlying DBMS
  - Available libraries and code generation
  - Lack of support for inheritance in the language intended for the implementation.
- Dealing with constraints
  - In the application design you assume using a true OO programming language and an RDBMS (an SQL server). The deviations from these will be documented later, in the implementation report
  - Make the parts of your design clear and explain how they fit together. You are likely to need to provide **extensive descriptions** (use Notes), in addition to (UML) diagrams.

Hints and Tips

- Think whether you can actually implement a design element *before* you commit yourself.
- Remember and update your diagrams as you go. You will need *many* iterations before the work is completed to a satisfactory level! Use tools!
  - A concrete database server
  - An IDE for creating forms (opting for a high-level programming language like Java, C#, etc.)
  - This will allow you to be more effective in the implementation phase.
- Connectivity with DB may still allow for alternative implementations, do not commit yourself to a particular DB connectivity, e.g. JDBC.
- Abstract out your relational database schema. Document SQL queries (a **representative** set of SELECT, INSERT, UPDATE, DELETE statements + report-generating statements).
- Concentrate on user interface mechanics.

Marking Scheme (1)

- **Presentation 10%**
  - Presentation criteria (details are provided in the Student's Brief):
    - Appropriate title, page numbering, version control (2)
    - Introduction, and Purpose & Scope (2)
    - Use of language appropriate to audience, and Spelling & Grammar (2)
    - Clear layout and structure (2)
    - Appropriate use of graphics and diagrams (2)

Marking Scheme (2)

- **Content 90%**
  - Specific contents:
    - Requirements Specification of the system to be (see slides from the previous Briefing):
      - Description of existing system (5)
      - Use case diagram(s) (15)
      - Use case descriptions/specifications for 10 key UCs (15)
      - Prioritisation of use cases (5)
    - **System Design:**
      - **Fully refined and correct class diagrams showing entity, control and boundary classes and interfaces … (20)**
      - **ER-diagram, relational DB schema, SQL statements (incl. report-generating ones) (20)**
      - **GUI design / layout (screenshots) (10)**

Submission deadline and method

- The deadline for the “Requirements Specification and System Design” document is:

  5pm, Sunday, 04th March 2018

- You need to submit the coursework on Moodle.
  - Only one member of the team is to submit the document