MANCHESTER
1824

The University
of Manchester

COMP23420 Software Engineering Semester 2

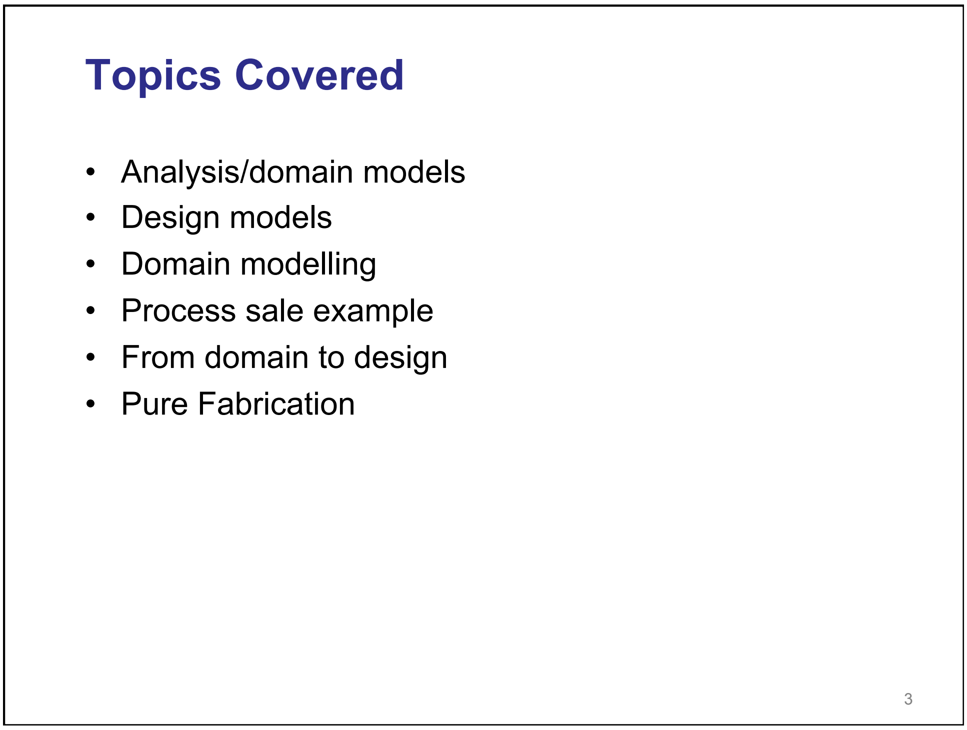
Week 3: From Analysis to Design

Dr Liping Zhao

School of Computer Science

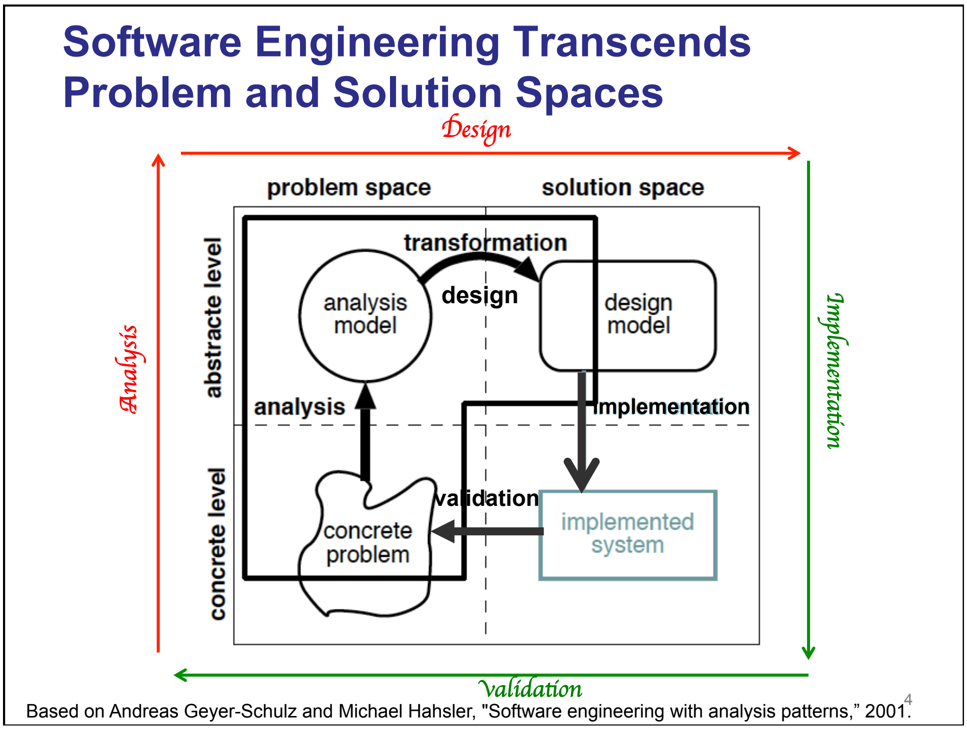
Today’s Objectives

- To describe how to construct domain models and to provide guidelines for the construction
- To describe how to transform domain models into design models and to provide guidelines for the transformation



Topics Covered

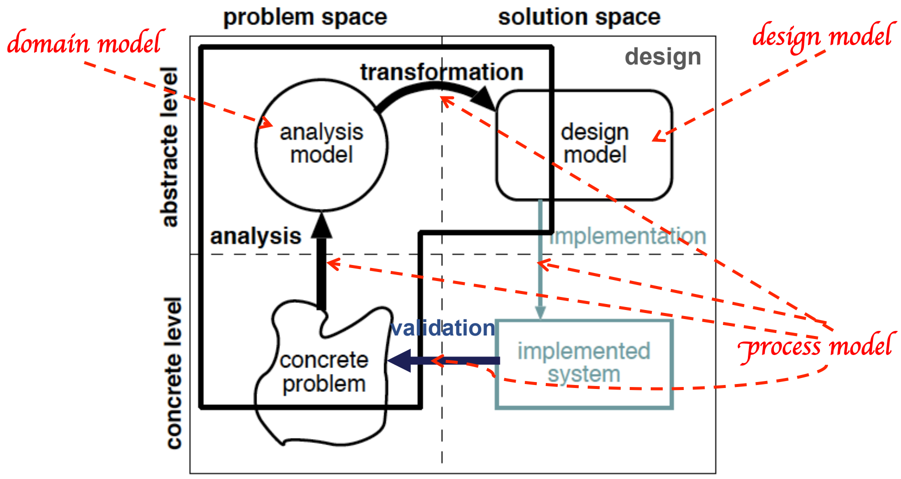
- Analysis/domain models
- Design models
- Domain modelling
- Process sale example
- From domain to design
- Pure Fabrication



Software Engineering Transcends Problem and Solution Spaces

Based on Andreas Geyer-Schulz and Michael Hahsler, "Software engineering with analysis patterns," 2001.

SE involves three main kinds of model



Models are vital building blocks in SE

Three main kinds of model:

- Process models
  - is an abstract representation of a software process
- Domain models (aka analysis models)
  - is an abstract representation of a software development problem
- Design models (aka software models)
  - is an abstract representation of a software solution

Questions:
- How to construct domain models?
- How to convert domain models into design models?

How to construct a domain model?

Guidelines:

1. Identify ***domain objects***
2. Represent them as "classes" (aka ***domain classes***) in a UML Class Diagram and show ***key attributes*** only (not details of classes)
3. Identify ***relationships*** (***association, aggregation*** or ***inheritance***) between domain classes
4. Label relationships with ***names*** and ***multiplicities***

- Keep the model simple – don't try to include everything (no "right answer") – and make it as informative as possible
- Use stakeholder's terminology - "mapmaker principle"

How to identify domain objects?

Most domain objects fall into five categories:

1. Tangible things

- Person, airplane, bus, book, till, computer, house

2. Roles

- Student, doctor, cashier, web browser

3. Incidents

- Film, concert, system crash, phone call

4. Interactions

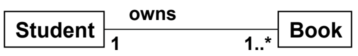
- Transactions or contracts between two or more domain objects
- Purchase (related to buyer, seller and thing purchased)
- Marriage (related to two people)
- Rental (related to landlord, tenant and agent)

5. Specifications

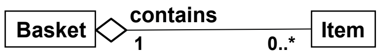
- Bill, bank statement, order confirmation, software specification

How to identify relationships?

- Domain objects (aka domain classes) are usually related by "*association*"
  - Customer and cashier; student and book



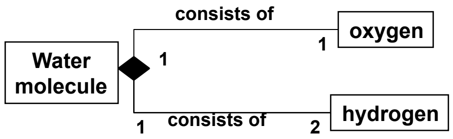
- *Aggregation* has two different meanings: *containment* or *composition*
- It is also very common for one domain object to *contain* one or more other domain objects



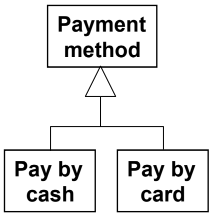
  - Use open diamond for "contains" (You must use the notation correctly in the exam)

How to identify relationships?

- *Composition* should be used with care



- *Inheritance* may be useful for representing alternatives



How to represent multiplicities?

- *Multiplicities* are important – often generate questions to stakeholders
- Multiplicities model the situation at any one time (e.g., for the sale there is a 1-1 relationship between customer, cashier, and till)
- Should be read both ways – a basket contains one or more items (or could be 0..*) and an item cannot be in more than one basket (or could be 0..1)
- See Larman Chapter 9 For a more realistic version of this example.

The Role of Domain Models

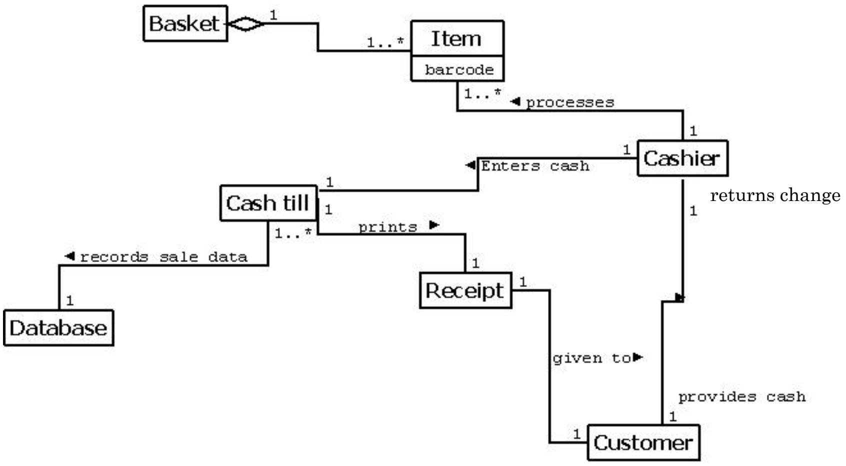
- Domain class diagrams show “real world” domain classes, not software classes, but…
- Domain classes often *inspire* software classes
- But: the mapping is not 1to1 – some domain classes will not lead to design classes,
- You may discover more domain classes in the process of design….
- which means you may need to go back to the stakeholders to clarify requirements…
- which means you may need to revise the use cases

Example: Process sale, cash only, use case (brief format)

A **customer** brings a **basket** containing **items** to the **cash desk**. The **cashier** enters details of the items into the **cash till** using the **barcodes** on the items, and the cash till calculates the total **amount due**. The customer offers an amount of **money** at least as much as required, and the cash till indicates the amount of **change** required. The cashier gives the change and a **receipt** printed by the cash till to the customer.

The details of the transaction are recorded in a **database**.

Domain model for processing sale



Note the relationships, multiplicities & labels

From Domain to Design

- Domain classes provide a useful starting point for design
- Domain models can be refined with analysis patterns (beyond the scope of this course)
- Most designs will need to be refined with GRASP principles (next 2 lectures) and design patterns (COMP33411)
- Since design classes are more detailed, we usually include less of them on one diagram.
- A whiteboard is good for discussing design choices and you can take pictures of it for documentation.

How to convert a domain model into design model?

Guidelines:

1. Map domain classes to design classes - *which domain classes carry over depends on the application*

   > e.g. Basket is needed for an online store but not a physical one.

2. *Human actors* should be included in the design iff you need to store data about them (e.g. Cashier, not Customer).
3. *External software systems*, databases etc. are not included. Instead, use a ***Pure Fabrication*** class such as a database connector.
4. *1-many and many-many associations* often lead to collections in the design

Use Pure Fabrication in Design

- A ***Pure Fabrication*** is a *design class* which does not correspond to anything in the domain, e.g.
  - Collections
  - Interfaces to external systems, e.g. database connectors
  - Factories – classes whose sole job is to create objects of other classes.
  - UI components
  - Indirections to, and abstractions of, other classes.
- Over time, the PFs usually outnumber the domain-inspired classes.

Design Class Notation

- *Types:* UML standard is `name:type`, e.g.

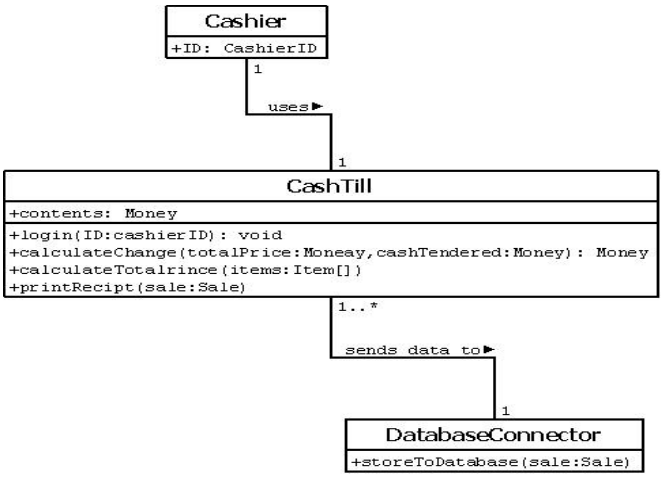
  `CashierID:String` but use Java-style

  `String CashierID` if you prefer
- *Operations*: `getName(CashierID: String): String`
- *Visibility:* + (public) – (private) # (protected)
- *Static things:*

  `loadFromFile(Filename: String): List<Cashier>`

NB: Displaying types, visibilities etc. is optional – avoid premature design decisions.

Cash Till Design



Cashier

+ID: CashierID

1

uses ▶

1

CashTill

+contents: Money

+login(ID:cashierID): void
+calculateChange(totalPrice:Moneay,cashTendered:Money): Money
+calculateTotalrince(items:Item[])
+printRecipt(sale:Sale)

1..*

sends data to ▶

1

DatabaseConnector

+storeToDatabase(sale:Sale)

Points to Note

- We’re modelling the software for a physical cash till.
- The diagram is still relatively simple – for comprehensive documentation, use javadoc.
- We have modelled the cashier logging in (not in the UC) because that’s the only reason for having the Cashier class.
- The Money domain class is clearly needed – that doesn’t mean the domain model was wrong.
- We also seem to need a Sale (or SaleDetails) class.
- We have added the PF database connector.

Key Points

- Models are critical constructs of software engineering
- Three main mode types are process, domain and design models
- A structural model represents the static structure of something – in UML this is done mainly with a class diagram.
- Domain class diagrams represent domain objects, whose instances are real-world objects relevant to the system being built.
- Design class diagrams model design classes, aka software classes, whose instances are software objects in the system being built.

Key Points

- A domain model provides a good starting point for a design.
- Doman classes often inspire design classes, but it is definitely not a 1-1 mapping and it can even happen the other way round.
- Design classes are more detailed than domain classes, but it is important not to clutter them.
- An initial design model is likely to generate questions about requirements, not just design decisions
- Pure Fabrication is a design class used to model a software class that has no counterpart in the real world

Questions raised

- (Scope) what other information do we need to store about a cashier, e.g. hours worked to calculate pay?
- (Functional requirement) – what does the calculateTotalPrice operation need do to (special offers, bogof etc)?
- (Non-functional requirement) do the sale details need to be recorded in the database once at the end, or incrementally as the sale goes on?
- (Implementation decision) should we have one database connector for all the tills, or one per till, or something in between?

Key References

- Craig Larman, Applying UML and Patterns, Prentice Hall
- Ian Summerville, Software Engineering, 6th Ed. Addison-Wesley, 2000.