From domain to application

We've looked at modelling domains – the whole application space, not just one problem

Now need to focus more on the application in hand

- Exactly what the requirements are for *this* system
- Its boundaries and capabilities
- Move towards a description more suited to building software

These are the issues of requirements and specification

Requirements and specification

The demands to be fulfilled by the contractor

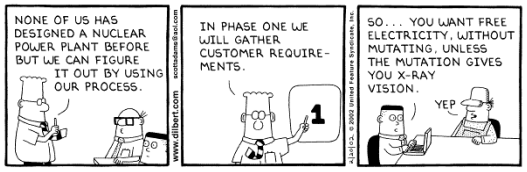
Requirements state what the user wants from the system, expressed in the problem domain

Specification refines the requirements into a precise description of what the system must (not) do

The constraints the contractor must satisfy in order to be paid

Hopefully what they want is close to what they get

Realistic Requirements…



Requirements and the domain model

Which comes first: the requirements or the domain model?

It depends

- If a *new customer* comes with a new system, they may have requirements already. You then have to build a domain model to see how these fit in
- If an *existing customer* wants a new system, you've probably already got a domain model and just need to refine it
- If there are no real requirements, you can develop domain model and requirements side by side

Usually we regard domain modelling as part of requirements gathering (*requirements analysis*)

- They focus on different things towards the same goal

Not all created equal

Functional requirements

Non-functional requirements

Resource requirements

Cost requirements

Functional or non-functional?

Functional requirements are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

Non-functional requirements are constraints on the services or functions offered by the system. They include timing constraints, constraints in the development process, standards and so on. .. Examples are reliability, response time and store occupancy.

I. Sommerville, Software Engineering, 1995.

Not all created equal

Functional requirements
- Must generate accounts using common accepted practice
- Must not send out bills for €0.00

Non-functional requirements
- Must be secure
- Must process all requests in less than two seconds

Resource requirements
- Must run on a network with bandwidth under 24Kbits/s
- Must use less than 1Gb of disc space
- Must be ready in 18 months

Cost requirements
- Must cost less than €100'000
- Must run on hardware costing less than €1000 per desk

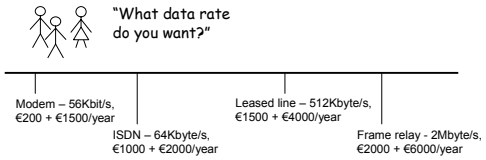
Quick quiz..

What are the requirements of an on-line hotel reservation system?

- Functional requirements?
- Non-functional requirements?
- Resource requirements?
- Cost requirements?

Quality/price trade-offs

Users will always ask for the best, but only want to pay for the worst



But what data rate do they *need*?

- Can only answer this question by using knowledge of their business activities – the domain model

Characteristics of requirements

Incomplete and contradictory

- Often due to hidden assumptions on the parts of the users – they know things you don't, and don't know you don't know
- Need to decide which are more important and which can be sacrificed without too much damage
- This is where the domain model really starts to help

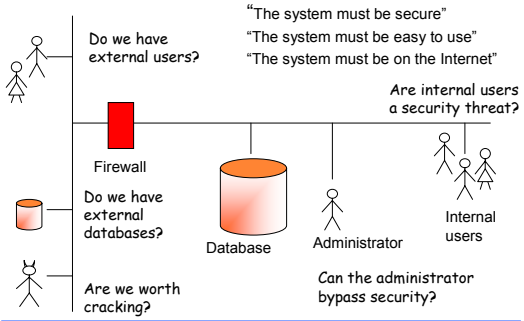
Not well decomposed

- One "requirement" may actually be a lot of little requirements ganging-up on you

Unrealistic

- Either want what's physically impossible, or don't realise how easy (or difficult) what they want actually is

Example - security



Solutions and consequences

Are internal users a security threat?

- ✓ Encrypt all data in the database and on the wire
- × Hugely complicates crash recovery
- × Standard applications won't work without modification

Do we have external users?

- ✓ Connect to the Internet and allow them to log-in
- × Crackers might attack us
- ✓ Deploy a firewall and one-time passwords
- × Complicates access for legitimate users
- ✓ Build a private IP network
- × Expensive, closed, hard to expand

Do we have external databases?

- × Firewall stops users getting out
- × Even if they get out, external services can't open connections

Requirements affect the possible solutions, which have consequences for other requirements

Lessons

There are almost always local solutions to individual requirements

But requirements interact

- Positively – solving one may address another
- Negatively – solving one may make another impossible

Price/performance

- Can usually get more performance if you pay for it

Security/convenience

- A more secure system is often harder to use

"Polluted" users

- Use the words, but in a different way – *e.g.* for marketing rather than for technical reasons

Quick quiz..

"The system must generate invoices for each customer each month"

- Functional

"The system must tolerate ambient temperatures from -200K to 200K"

- Non-functional – run on a satellite

"The system's execution platform must weigh less than 200g"

- Non-functional – run on a palmtop
- Probably has resource implications

"The system must be secure"

- Non-functional
- …and too imprecise to do much with

Representing requirements

Traditionally a requirements document

- May actually be part of the contract tender
- …or may be written by the users at your request
- …or by you at the users' request

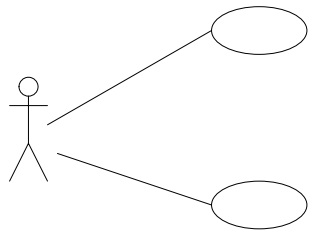
Can also use UML diagrams

- Extract the requirements from the domain model
- Remove all the extraneous bits that you gathered in order to understand the business, focus down on one part and refine it

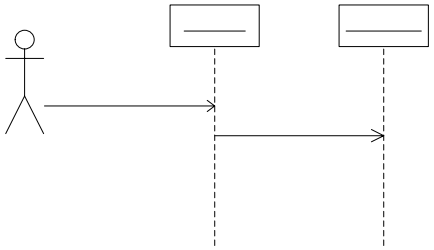
A combination of both approaches

- Managers sometimes think text is "harder" than diagrams…

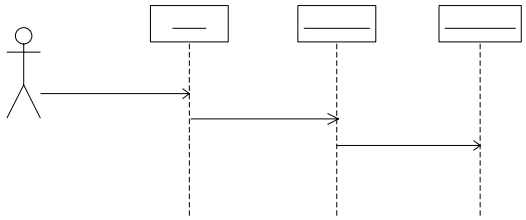
Example – lift controller use case



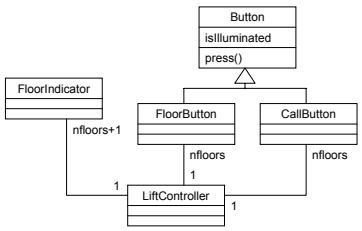
Outside Lift sequence diagram



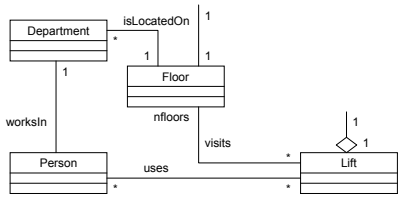
Inside Lift sequence diagram



lift controller class diagram



Lift controller – extraneous bits



Press

Lift controller requirements

"The lift will contain floor buttons and a floor indicator"
- Anything else? A light? A telephone? A door-open button?

"There will be a floor indicator on every floor, outside the lift door"
- What's on a floor indicator? Row of lights? Number? Same as inside the lift?

"Floor buttons will light up when pressed"
- How? LED? Bulbs?

"The lift will visit the floor on which a button has been pressed"
- When? Immediately? Eventually? Guaranteed? Does it matter?
- Where did this button come from?…

Review

Requirements let the users say what they want, in their terms
- Built on, alongside, or as part of a domain model
- Functional, non-functional, resources, costs, …

Requirements are often contradictory, incomplete and ambiguous
- Often too strong or too demanding for their needs
- Tacit assumptions need to be located and made concrete
- Interactions may foil local solutions

Need to refine these requirements down to a precise specification of the system to be built

Requirements to specification

The requirements say what the users *want*

The specification says what the users will *get*
- Describes of what the system will (and won't) do
- Addresses each of the identified requirements
- Doesn't address things like cost and timeliness – these are more in the domain of project management

Components
- User-visible functions of the system
- Problem-level behaviour of sub-systems and objects
- High-level descriptions of key modules, algorithms and data structures, and their interfaces

Collected together as high-level, often abstract classes in O-O approaches

What's wanted from a specification

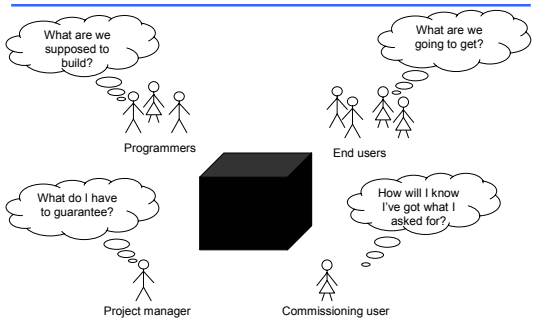
A precise, complete, unambiguous definition of the system, its key functions and constraints

- Critically dependent on the quality of the requirements – ambiguous requirements make for ambiguous specifications

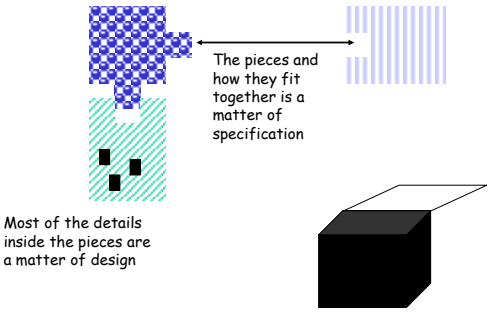
Avoid over-specification

- Do not constrain something the users don't really care about, *e.g.* internal data structures and algorithms
- Only constrain the really key algorithms and processes – and get them as clear as possible

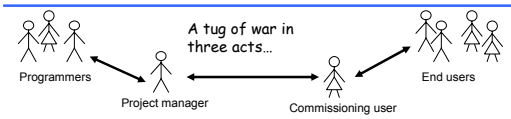
"Black-box" specifications



What's in the box



Getting a specification



It's difficult to draw hard distinctions between requirements, specification and design

- Differences in focus – what's wanted, what's technically and commercially possible, and how it can be achieved

Meet the requirements as far as possible

- *Must* be technically feasible – there is an obligation to deliver!
- *Do not commit to anything you can't deliver on time and within budget*

Agree acceptance criteria up-front

What's in a specification

Increasingly more detailed and technical

- Top-level functions
- User interface constraints
- Resource and performance constraints
- Inter-operability constraints
- Acceptance criteria
- Software Architecture
- Functional units
- Interfaces between units
- Key data structure behaviour
- Key algorithms

Approaches

- Informal specification
  - A document in natural language
- Formal specification
  - Mathematics with provable properties
- Semi-formal specification
  - Structured language with formal bits

By and large the best-developed specification techniques deal best with smaller-scale issues and not so well with the big picture

Natural language specification

A statement in English (or Irish, or Bantu) of what the software does

Pan-European projects can fail very easily through language confusions…

Tend to get *very* large

- Verbosity is a real problem, unless the writers really knew their stuff

Subject to all the usual problems with natural languages

- × Easy to be ambiguous, imprecise, contradictory and make assumptions – either accidentally or by design
- × Hard to reason about, hard to follow chains of dependencies
- × Its familiarity can be misleading
- Just weren't *designed* for talking about software…

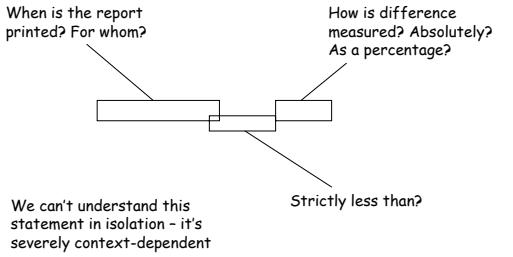
Ambiguities

"If the sales for the current month are below the target sales, then a report is to be printed, unless the difference between target sales and actual sales is less than half the difference between target sales and actual sales in the previous month or if the difference between target sales and actual sales in the current month is less than 5%"

This is almost an algorithm, but expressed in the problem domain

Adapted from Stephen Schach, *Classical and object-oriented software engineering*, Addison-Wesley (1999)

Ambiguities highlighted



Formality

Introduce some structure or formality to avoid these ambiguities

Formal specification

- When things have absolutely got to be understood and agreed precisely, *e.g.,* safety properties or safety-critical behaviour

Not enough just to bring in some squiggles

- Proof – is the maths correct? consistent? complete?
- Verification – does what we built conform to the specification?

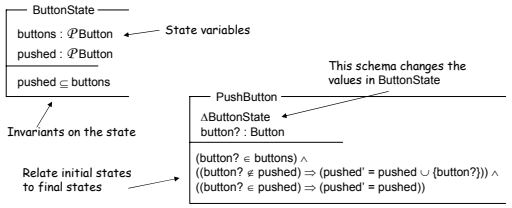
Varying formality

- Mathematically well-founded – Z, VDM, π-calculus, …
- Structured but largely informal – SDL, UML, pseudo-code…

Z

Developed mainly at Oxford University by Mike Spivey

- Essentially a structured form of set theory
- Model-based specification
- Invariants and changes



Critique

Problems are common to all formal approaches

- ✗ Completely inaccessible to non-mathematicians
- ✗ Gives to rise to huge proof obligations
- ✗ Not as structured and formal in places as it might be
- ✗ Needs extensive tool support to be really usable

However, Z has some major advantages

- ✓ Fairly easy to learn - it's only set theory with lines around it…
- ✓ The discipline of maths can lead to fewer ambiguities
- ✓ Paraphrasing the formal spec into English can lead to clearer natural-language specifications
- ✓ Proven – used successfully on a number of industrial-strength projects, especially in safety-critical systems

…and maybe that's a bit *too* formal

If we want to keep user involvement – and we almost certainly do – we need something more accessible

- Relate the specification to problem domain concepts
- Show how the specification meets the requirements
- Only get down to precise detail when absolutely necessary

Structured semi-formal specifications

- Explanatory text
- Expanded UML diagrams
- Formulæ where appropriate

The approach for the pragmatic engineer…

Refining the existing descriptions

Can happily use UML for some specification tasks

- Use case and sequence diagrams – specifying externally visible functionality
- Class diagrams – add more functionality
- State diagrams – add low-level transitions
- Activity diagrams – same sort of thing, with more detail

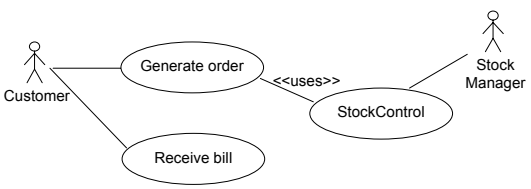
Can also use pseudo-code and class interfaces

- Show how the classes go together, but *not* how they're implemented unless the algorithm is really critical

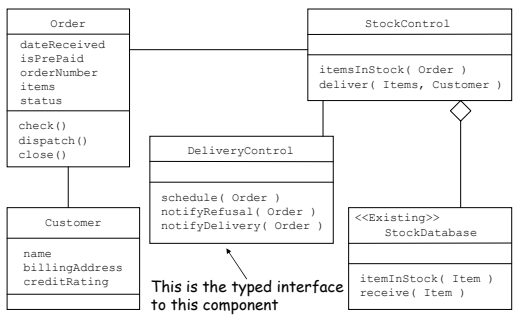
Move away from the problem and towards the solution, stopping short of complete system design

- Still basically in the problem domain
- …although with implementation concerns starting to creep in

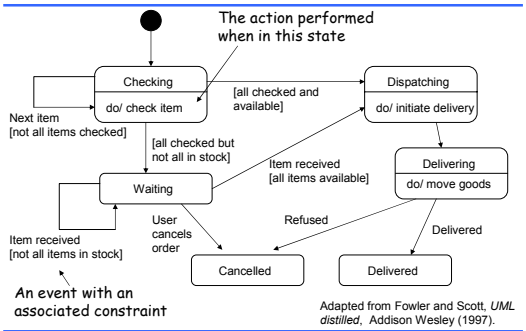
Order processing – Use Cases



Order processing - classes



Order processing - states



Adapted from Fowler and Scott, *UML distilled*, Addison Wesley (1997).

Interfaces

Specifying an interface

- Describe the functions it provides, document the constraints under which the operations work

Can work well using pseudo-code or abstract classes

```
public abstract class Order {
   private Date received;
   private int status;
   …

   /**
      Dispatch an available order (only legal
      when status == Checked)
      @return availability flag
   */
   public boolean dispatch();

   …
}
```

Java comments provide "semantics" for the interface

Some programming languages (like Eiffel) allow assertions like this to be added as part of the program rather than the comments

Software Architectures

Architecture Description Languages (ADLs) and "Styles"

- Concerned with designing and specifying (complex, distributed) system structure [Garlan and Shaw]
- ADLs are used to describe (complex, distributed) systems as a configuration of components and connectors
  - *They describe components and component interactions*
- They are more formal than textual specifications, but less detailed, and restrictive, than UML diagrams

Seeking a balance

Specification is a delicate balance

- All about defining sufficient detail without over-constraining the developers later on

The specification document is the main contractual interface between the developers and customers

- What they should expect to see when you deliver the system
- Agree on any functions, algorithms and structures that must be in the solution space
- Don't worry them with the small-scale internal solutions – there are lots equally good ones, and the users just don't care

Summary

Requirements and specification move from the general to the specific within the problem domain

- Focus on a particular application area within the general context
- What they want *versus* what they will get

Seek to remove ambiguities, omissions, contradictions and tacit assumptions in what the customers want

- Determine what's necessary and what's possible
- Refine to something definitely implementable within time and budget constraints
- Techniques addressing users, interfaces, algorithms, …

The results of analysis must be reified into software by an engineering team – and that takes planning

References

D. Garlan and M. Shaw. An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering, volume I. World Scientific Publishing, 1993.

Tutorial

UML and Rational Rose

Refine your model to capture the extra level of detail required by a specification..

Rational Rose: CASE tool for UML
Includes an editor to draw UML diagrams, code generator, ..

http://www.dsg.cs.tcd.ie/~meierr

Rational Rose is available in the ICT labs
Will be required for your course assignment