XP meets UML

Do analysis and design have a place in incremental development?

Alan Cameron Wills *alan@trireme.com*

TriReme International Ltd *http: //www.trireme.com*

The speaker

- Joint author of *Objects, Components and Frameworks – the Catalysis approach*
- Mentor and presenter in software development process
  - Clients in many fields, both sides of the Atlantic
- Technical Director of Trireme International Ltd
  - Source of process and architecture mentoring to many large companies in UK, Europe and beyond
  - http://www.trireme.com

The topic

- XP is increasingly popular and successful
  - Rigorous testing, frequent refactoring and elaboration
  - Minimal documentation
  - Close links between customer and developers
- Analysis, architecture, and design used to seem quite useful – UML, use-cases, CRC, patterns, and all that
  - Clarification of goals before rushing into code
  - Definition of interfaces, conventions and standards
  - High-level descriptions of code
- Can they be used together? Can both sets of benefits be got?

Agenda

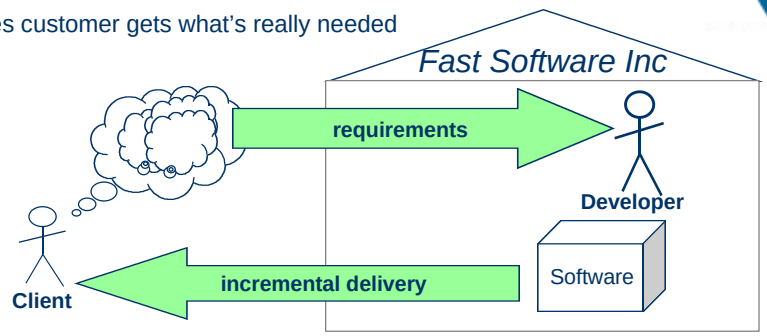
Incremental Development and Modelling
- Incremental development
- Analysis and design front-end
- Multi-loop processes

Raising Questions with Requirements Models

Testing

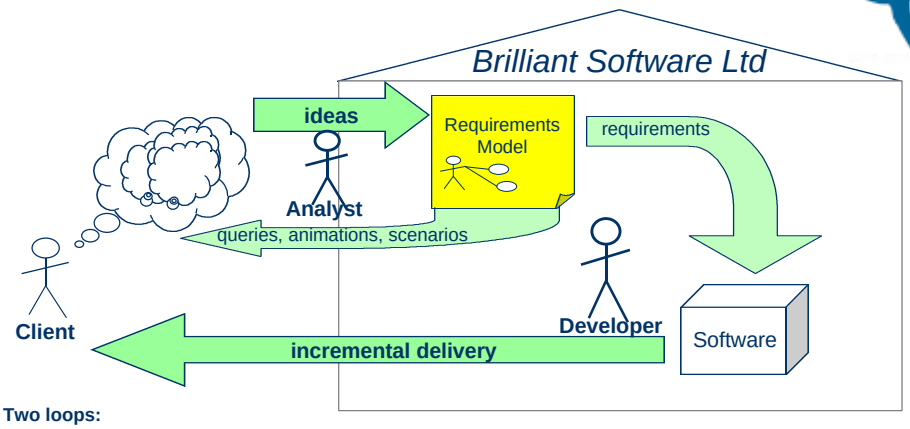
Incremental development

- Emphasis on customer feedback during development
  - Ensures customer gets what's really needed



- The loop is short, so that what's being delivered remains close to what's being developed
  - Features are delivered as they are developed
  - Ideal is to have customer working with developers
- Changes in requirements are accepted as normal
  - software developers arrange things to be able to cope with changes

Well-defined requirements and design



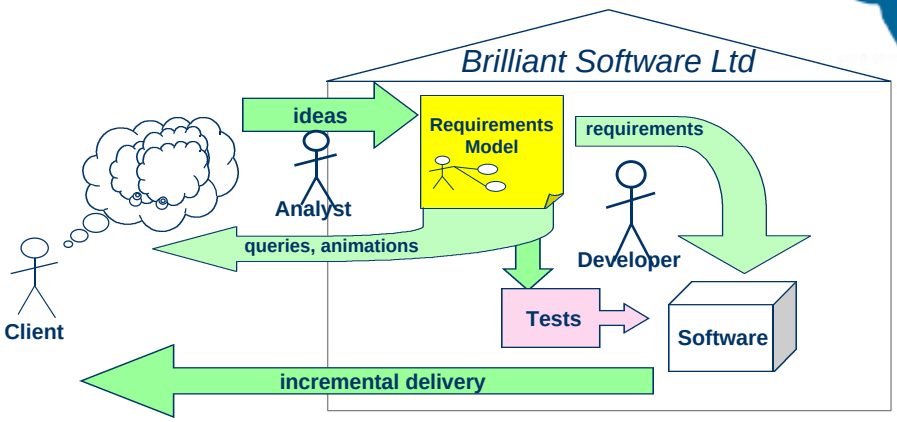
Two loops:
Incremental requirements construction:
- irons out inconsistencies and ambiguities
- sets vocabulary
- provides high-level overview
- focuses on major issues rather than details

Incremental software delivery:
- provide most concrete basis for feedback
- allows for change

Coding in UML?

- Some tools maintain a direct correspondence between UML and program code
  - eg. Together/J
  - effectively programming in a UML/Java mix
- But UML is also useful for:
  - Abstract models that don't contain all the detail
    - very useful for high-level 'domain' or 'essential' models, which define business vocabulary and rules
    - clear high-level overview of requirements or design, uncluttered by the fine detail of the code
  - Defining interfaces in frameworks and between components
    - These models are about what the components talk to each other about – not about their internal structure
- UML programming tools are not so useful for these purposes

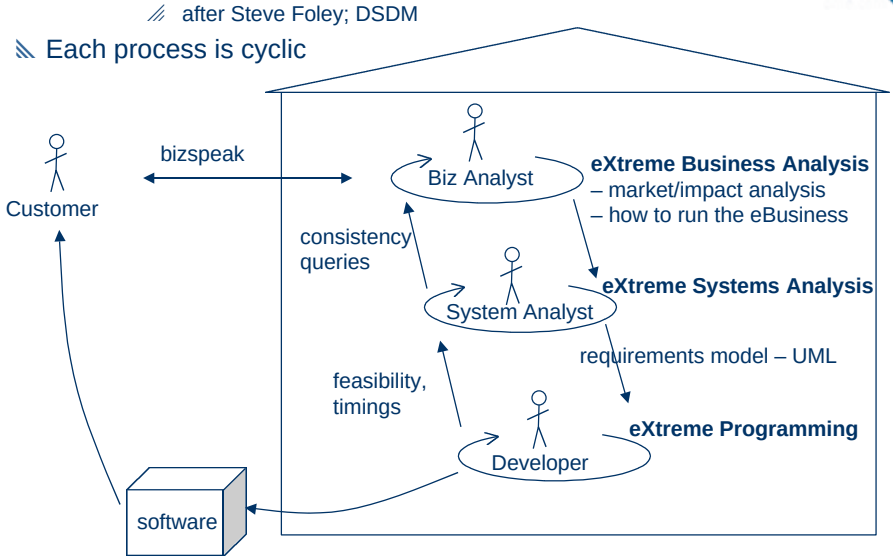
Well-defined requirements and design



Requirements model:
- **rapidly yields useful questions about the requirements**
- **gives better high-level overview than spike implementations**
- **provides basis for tests**
- **is basis for high-level OO or component design**

…tightening the loop

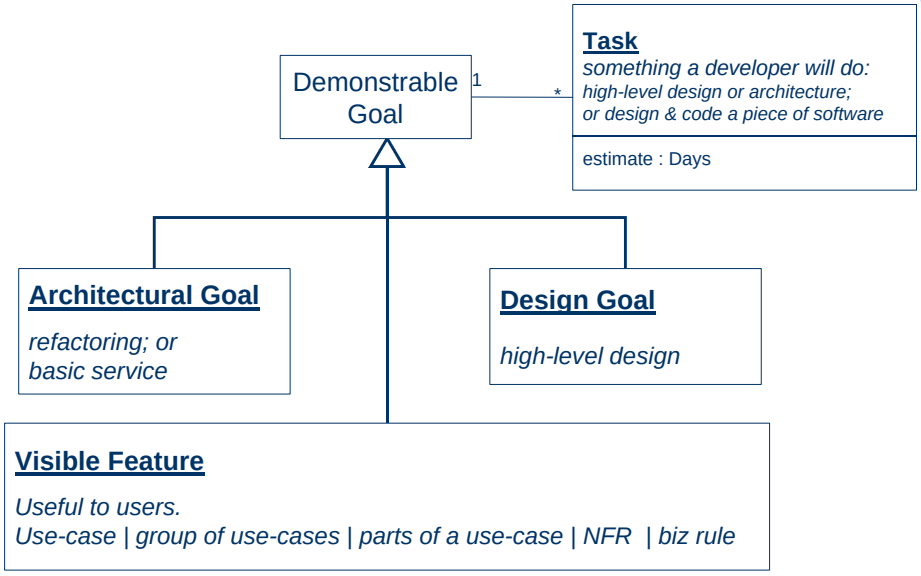
- Communicating eXtreme Processes:
  - after Steve Foley; DSDM
  - Each process is cyclic



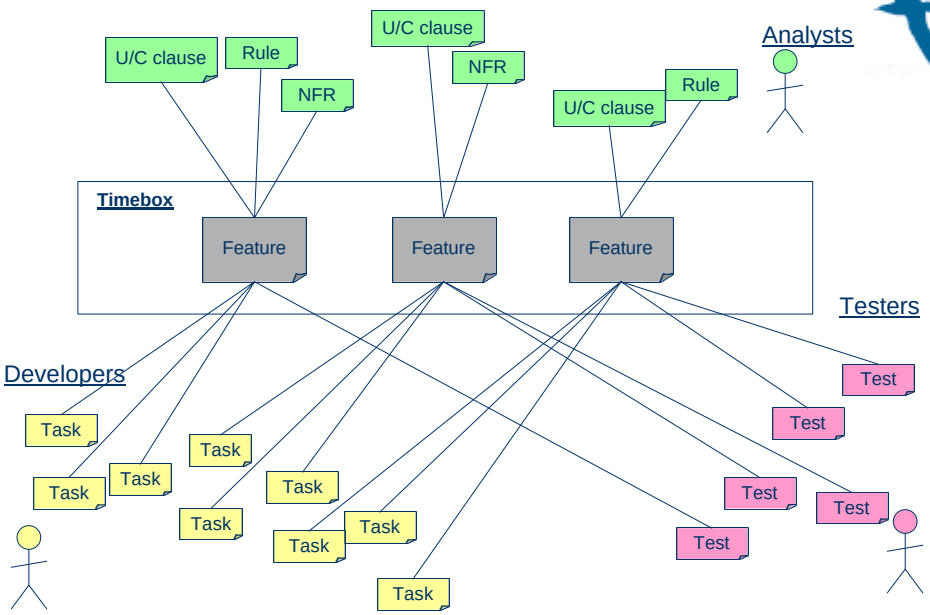
Tightening the loop

- We can put a requirements cycle in front of the development cycle
- But it only makes sense if we have ways of getting feedback from the model to the customers
- Two ways:
  - 'Animating' the model
  - **Consistency checks** <– *topic of next section*

Features, Tasks, and Demonstrable Goals



Features are central



Agenda

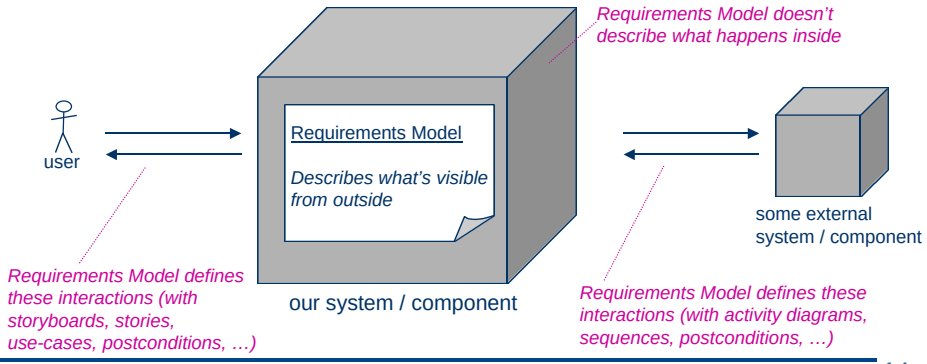
Incremental Development and Modelling

Raising Questions with Requirements Models
- Entity-action modelling
- Business rules
- Consistency checks

Testing

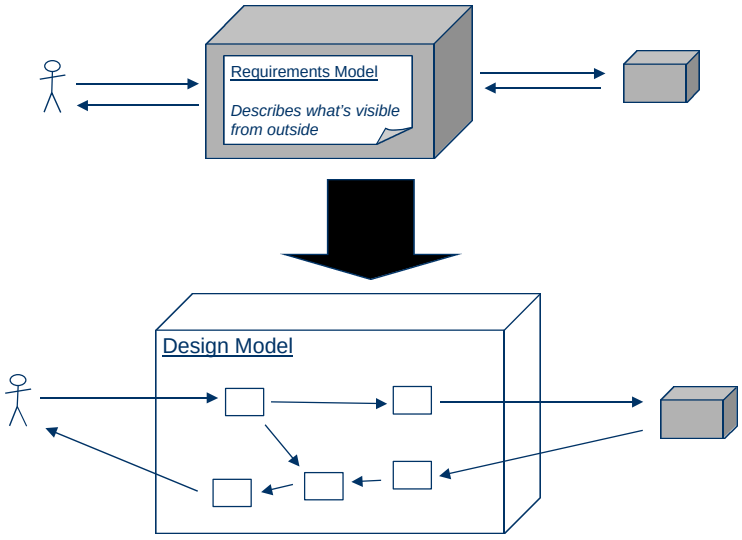
Specifications don't model implementation

- Analysts write *Requirements Models*
  - don't assign operations to classes inside the system
  - don't define collaborations inside the system
  - do define collaborations external to the system
  - do define operations or transactions visible at system interface
    - just define what they achieve, not how they work



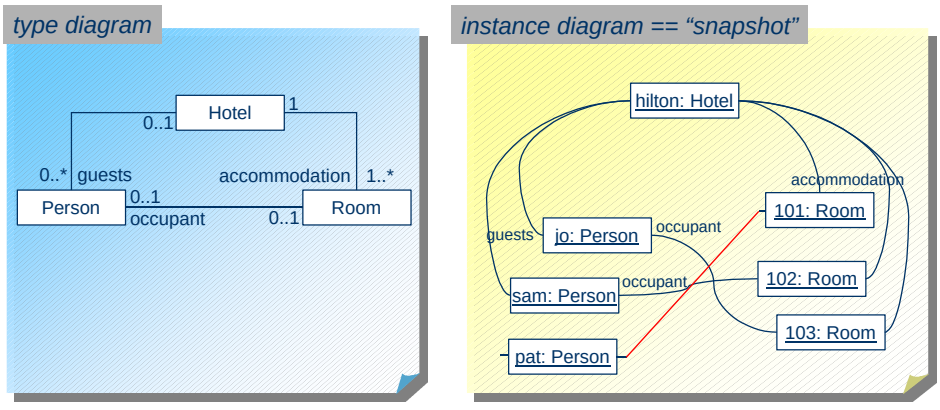
Requirements Models vs Implementation Models

- Designers determine internal structure & collaborations



Requirements Model: types and instances

- Types are sets of objects with something in common; instances are individual entities



- Snapshot must conform to type diagram

Requirements Model: use-cases

Scenario

Use-case: *Person checks into Hotel*

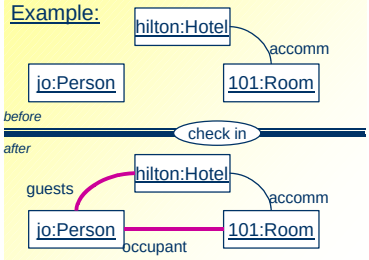
1. Person approaches Hotel entrance
2. Retinal scanner identifies Person as not being a current guest
3. System plays verbal request for credit card
4. Person swipes credit card at machine
5. System assigns an unassigned room
6. Assigned room henceforward can be opened with same credit card

Post-condition

Use-case: *Person checks into Hotel*

Outcome: person is a guest of the hotel, with access to a room that was previously inaccessible to guests.

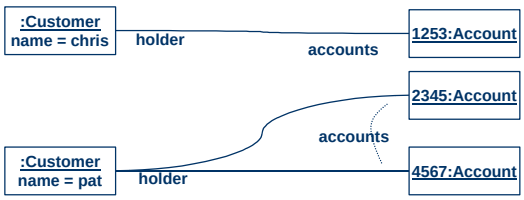
Example:



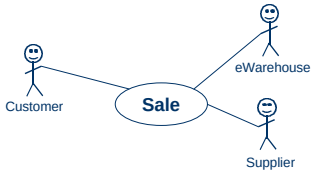
Associations in a Requirements Model

- Associations and attributes declare factual queries about the business concepts

  - I can ask for any Customer's name: the answer will be a String.
  - I can ask for any Customer's accounts: the answer will be 1 or more Accounts.
  - I can ask for any Account's holder: the answer will be 1 Customer.
- Instance diagrams used to illustrate specific situations:

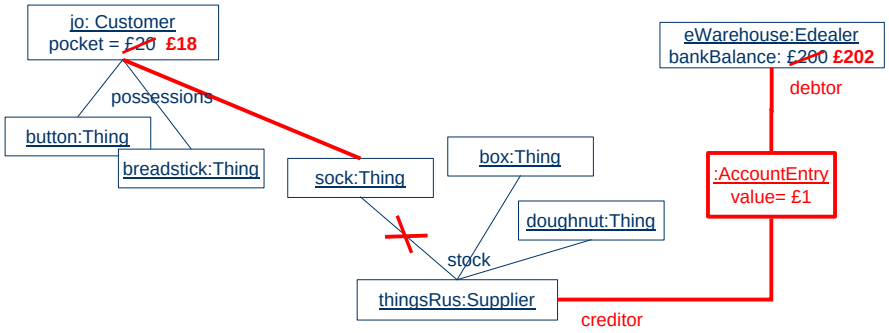


Postconditions objects, associations and attributes

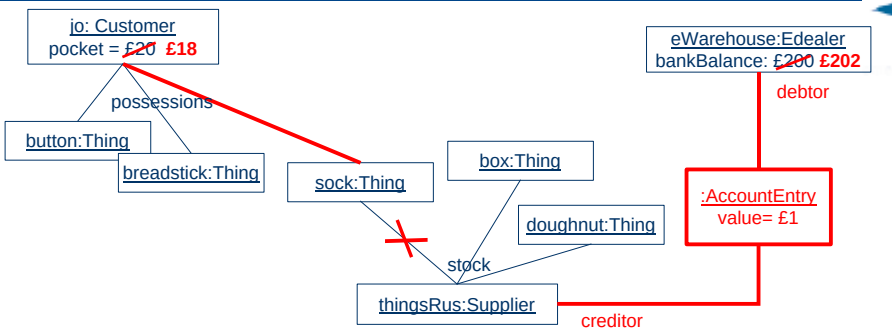


Postconditions:
- Customer has goods from Supplier
- eWarehouse has price of goods from Customer
- eWarehouse owes Supplier cost of goods

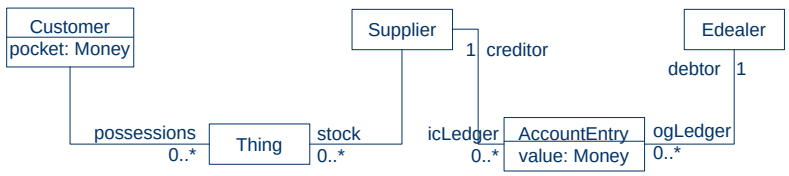
"Before/after snapshot" illustrates an occurrence:



Postconditions class diagram

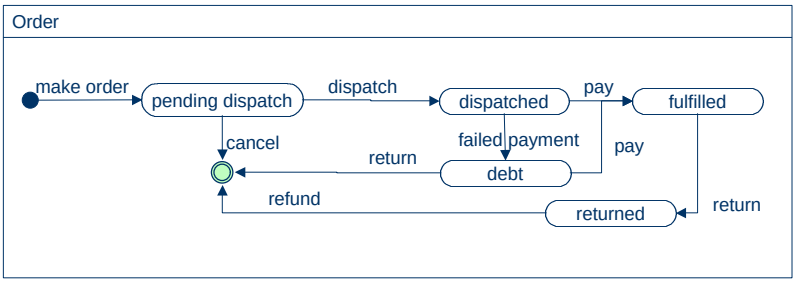


Classes needed to describe sales operation:



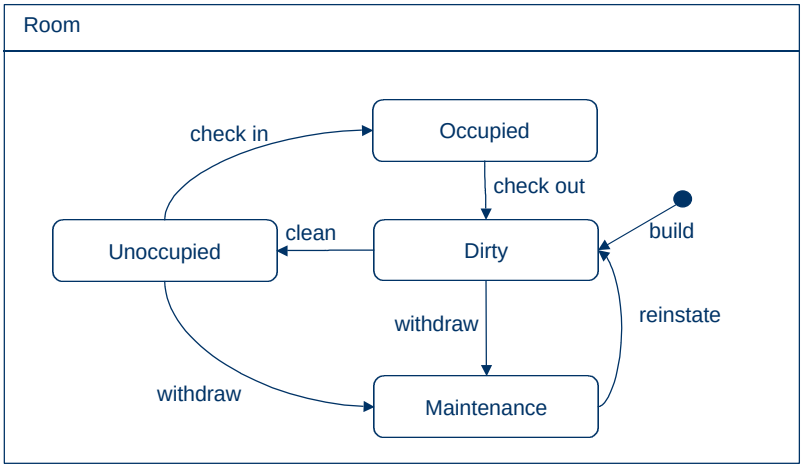
Object statecharts – > use-cases

- Consider what can happen that affects an object
  - each transition is a use-case

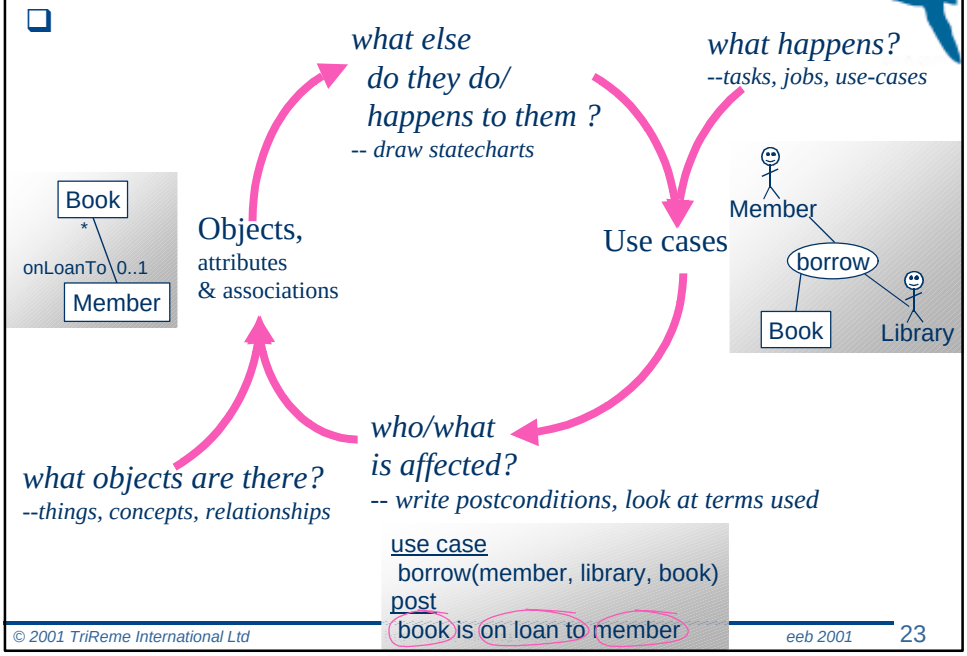


- Statechart expresses dynamic business rules

-

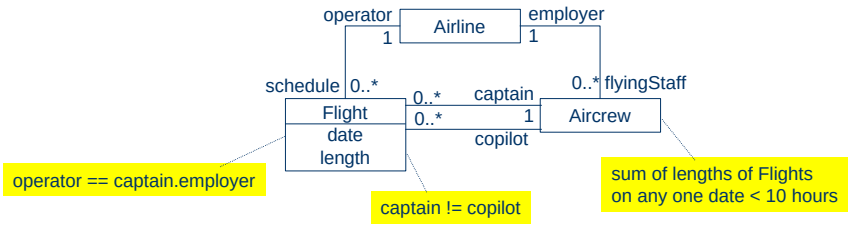


Use-cases and objects – consistency



Static business rules

- Static rules ("invariants"):
  statements always true about associations and attributes

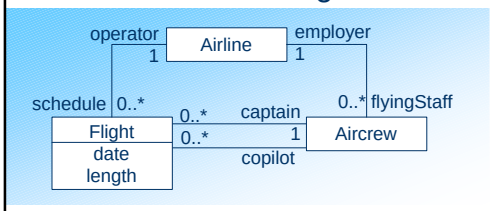


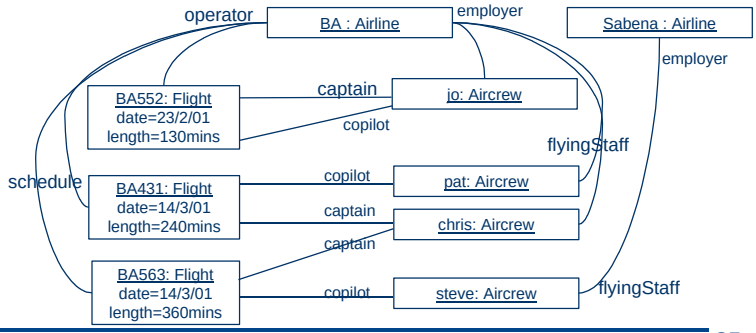
  - Look for loops in class diagrams
  - Look for instance diagrams that would conform to class diagram but be wrong within the business

- Use the vocabulary of the class diagram to express the rules

Static business rules

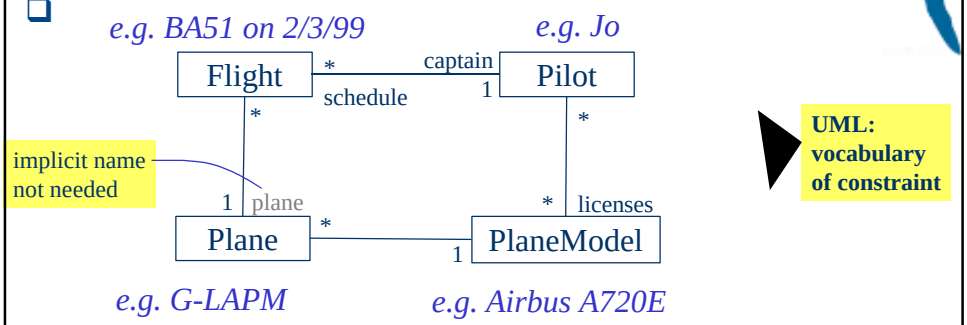
- Look for instance diagrams that conform to class diagram but would be wrong in the domain





Business rule example

-



UML: vocabulary of constraint

Every Flight must be flown by a captain who is qualified to fly the plane

constraint: as OCL expression

Flight :: captain.licenses→includes (plane.model)

starting context "for every Flight…"

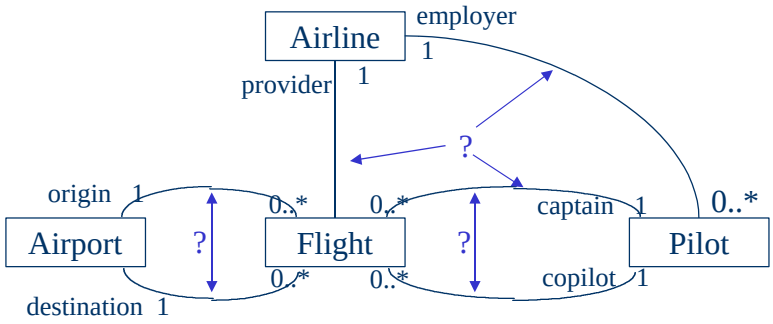
attributes of Flight

navigation to next link

set operation (licenses is a *-arity)

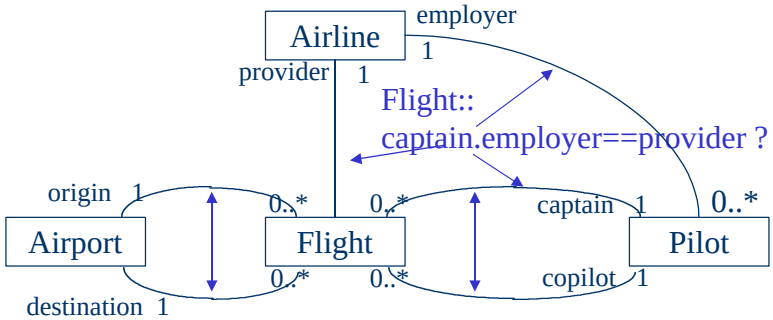
Loops => invariants

- Any loops in the type diagram suggest an invariant might be appropriate – raise questions to the customer



Loops => invariants

- Any loops in the type diagram suggest an invariant might be appropriate

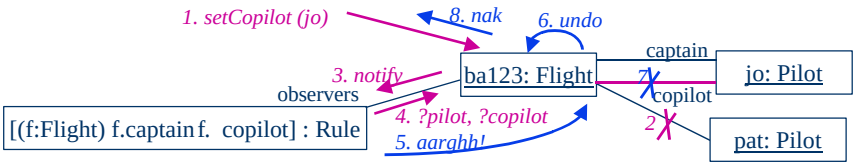


Flight::
origin<>destination ?
or do we allow circular flights?

Flight::
captain<>copilot

Rule objects

- Implementing invariants about business rules
  - How to ensure invariants are always observed?
    Basic method: just write all the code so that invariants are never broken
  - But some business rules can vary between countries & organisations – must be possible to change them independently of software



- Represent rule by an object that observes its parameters
  - Rule object re-evaluates whenever subjects (parameters) change
  - Scheme 1: subjects go ahead as though rule didn't exist
    - Rule raises an exception if undesired situation occurs
    - Subjects have to be able to roll back if exception is raised
  - Scheme 2: subjects ask permission before making change
    - can be difficult to do all of a complex transaction in 'what-if' mode

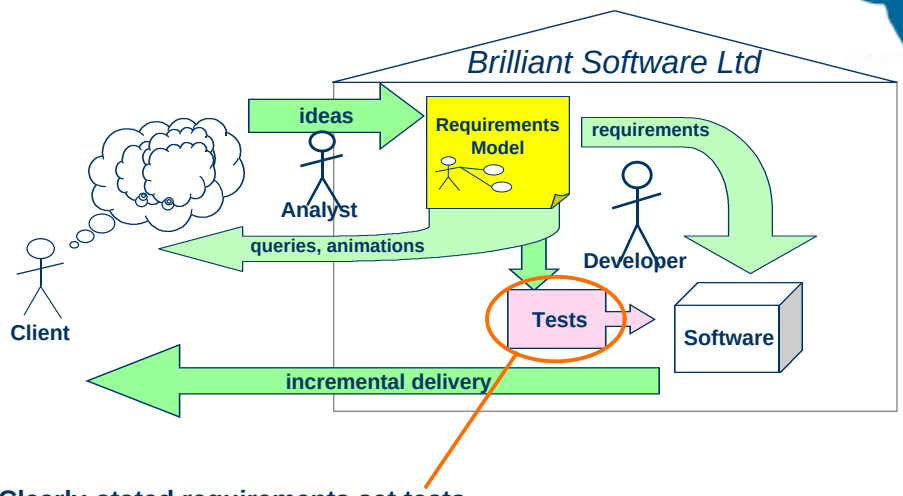
Agenda

Incremental Development and Modelling

Raising Questions with Requirements Models

Testing
- Test monitors and harnesses
- When to test rules
- What data to use

Well-defined requirements and design

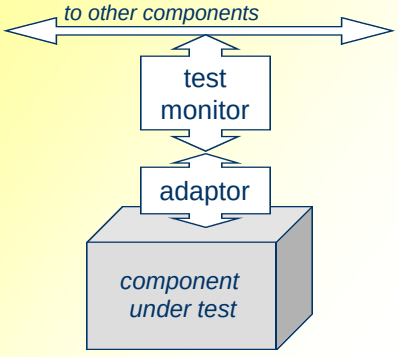


Clearly-stated requirements set tests
How to translate from requirements to tests?

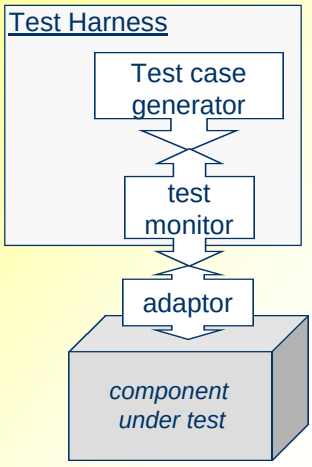
Rigorous Testing

- Incremental development relies on rigorous testing

Testing everything in and out, in the running system (like Eiffel)

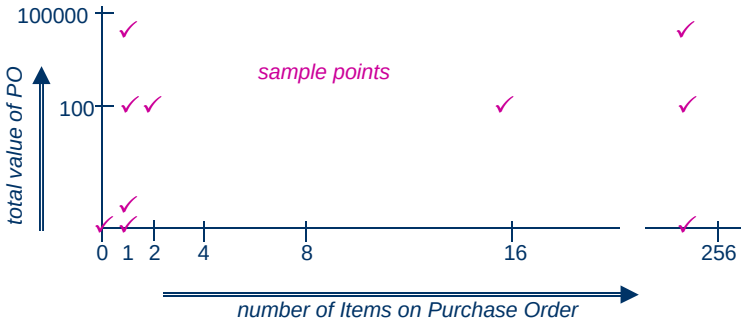


Testing Offline (like JUnit)

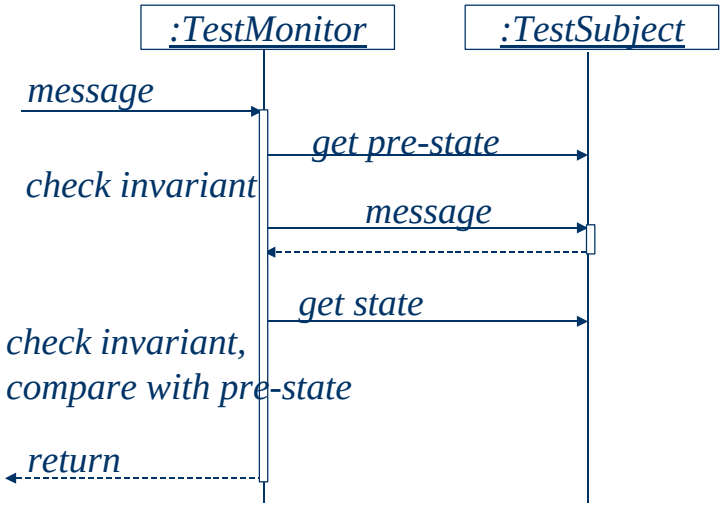


Test case generation

- White box testing
  - checks each path through code
    - no good if you can't see the code (as in CBD)
- Black box testing
  - partition state space into zones, check representative points



Test monitors: general scheme



-- or use Eiffel

Translating requirements to tests

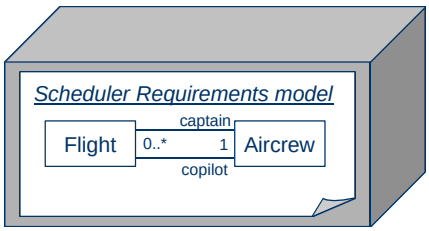
- Business rules state what should be true

```
Flight :: captain != copilot
```

- How to translate to actual tests?
  - How to interpret the attributes?
  - When to test it?
    - Just when setting captain and copilot?
    - After *every* operation?
  - What data to use to test it?
    - What sequences of operations might cause the code to break it?

Interpreting the attributes

- The model is not necessarily a picture of the internal structure
  - Component is a black box:
    We don't know anything about how Flight, captain and copilot are implemented inside the component

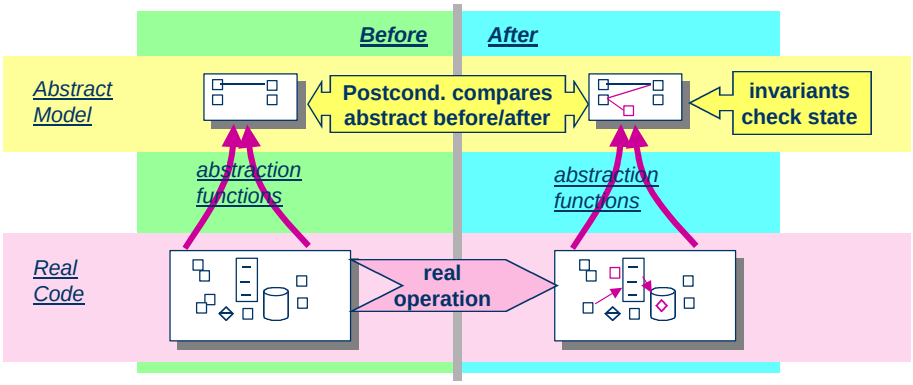


- The model says what questions the component can answer:

```
AirCrewName getCaptain (FlightId fid);
AirCrewName getCopilot (FlightId fid);
```

Retrievals: applying abstract postconditions

- Program code may be much more complex than the abstract model
- But the associations and attributes of the model can be abstracted from the code by read-only functions
  - all the information in the abstract model is there in the code
- Postconditions expressed in terms of the abstract model can be tested by applying the abstraction functions



Translating model to tests

- So this rule in the model:

```
Flight :: captain != copilot
```

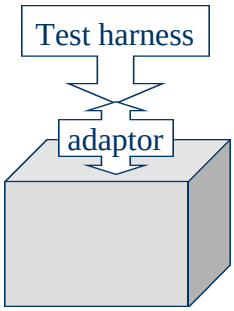
- means that for any Flight ID fid, in any scheduler component, at any time:

```
scheduler.getCaptain(fid) !=
scheduler.getCopilot(fid)
```

- The requirements model:
  - tells the test team what to test;
  - tells the design team what requirements to meet;
  - doesn't necessarily tell them how to design it

Translating model to tests

- Postconditions, invariants and statecharts: useful to express rules in requirements model
- Rules can translate to tests; but must add:
  - what data to test with
  - when to test each rule



UML for abstraction

- The high level view helps think clearly about designs

More succinct

More determined

- **UML for specification**
  - **What's required / provided**

decide test data and strategy

- **Test programs**
  - **What's required / provided**
  - **How to check it's provided**

apply tests

- **Program code**
  - **How to do it**

"reverse engineering"

- **UML pictures of program code**

Summary

Requirements modelling works well as a front end to incremental development

- better overview of requirements
- faster checks of consistency in requirements

Makes sense only if you use techniques that check consistency

- Action postconditions – > objects & associations; statecharts – > actions
- Finding static and dynamic business rules

Models at the requirements level tell the testers what to test

- but they have to decide what test data to use, and when to apply tests

- requirements models tell what to implement, though not necessarily how