Object-Oriented Specification + Design

The Software Development Process

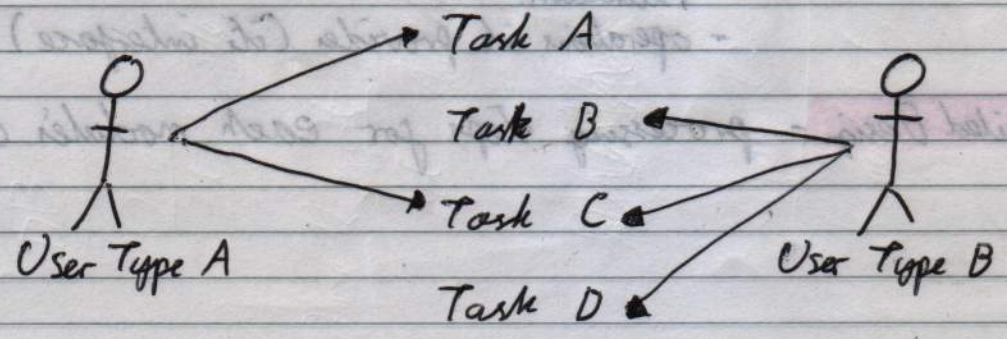
- The process has several stages:
  - Feasibility analysis
  - Requirements analysis
  - Specification
  - Design
  - Implementation
- These stages may be repeated (as in iterative development) or carried out once to completion (eg. waterfall model).

• Feasibility Analysis

- Evaluates technical options for implementation
- Attempts to find an option (or options) that are feasible and cost effective.

• Requirements Analysis

- Records requirements that stakeholders (inc. customers) have for the system.
- Records constraints imposed on the system
  - May include work practises and pre-existing systems.
- May be split into functional and non-functional.
- There may be conflicts and ambiguities - these must be resolved before a specification is started.
- Requirements may be modelled as use cases for the various users. Eg:



- **Specification**

  - Creates precise models of the system, using graphical / mathematical notation to represent its state and behaviour in a platform-independent manner.

  - UML class diagrams are ideal.

  - Avoids details of implementation.

  - Spec. forms a contract between developers and stakeholders, clearly defining what is to be developed.

- **Design**

  - Based on spec., this stage defines an architecture and structure for the system, dividing it into modules and sub-systems.

  - Includes:

    - **Architecture Design** - defines the global architecture as a set of major subsystems and the dependencies between them.
      Eg. GUI → Core → Data

    - **Subsystem Design** - decomposes the global subsystems into smaller subsystems which each handle a set of responsibilities, which can be sub-divided into modules.

      Modules are usually an entity (or closely-related group) and the operations upon them.

    - **Module Design** - defines each module:
      - the data it encapsulates
      - properties (invariants and constraints) it must maintain.
      - operations it provides (its interface)

    - **Detailed Design** - processing steps for each module's operations

- Implementation

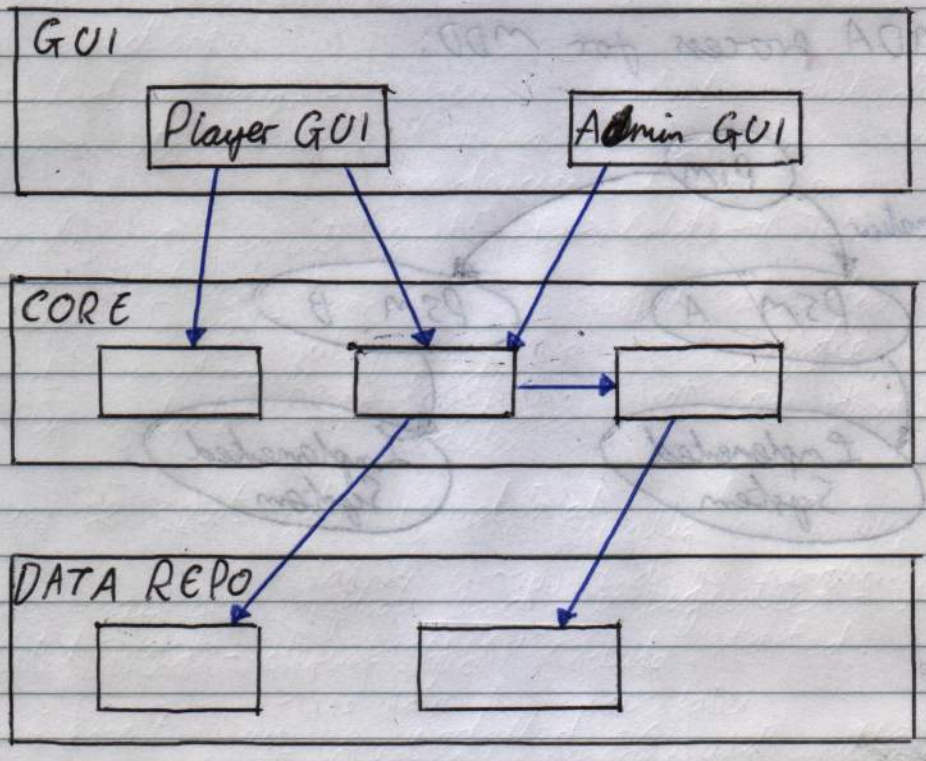
  - Produces an executable version by translating modules into code.

  - Complex data structures may be used for efficiency.

Architecture Diagrams

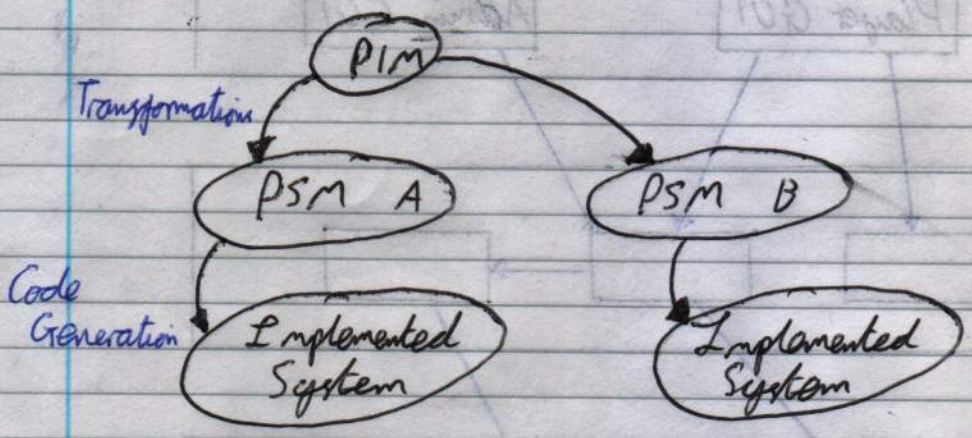
- Notation used to document system architecture.
- These consist of:
  - Rectangles - shows subsystems/modules
    - nesting shows ownership
  - Arrows - A → B shows that A depends on B.

Eg.



Model Driven Development / Architecture

- Tech evolves rapidly, so MDD focusses on models, not code, to assist in system migration and change.
- MDA focusses developer effort at higher levels of abstraction, in creation of platform-independent models (PIMs).
- From PIMs, versions of a system can be generated semi-automatically for certain technologies using Platform-specific models (PSMs).
- Allows companies to retain key elements, like business logic, in a form that is independent of changes in tech.
- MDA can be seen as a trend towards greater abstraction in programming languages.
- "Programming" takes place at a diagramatic level
- MDA process for MDD:



UML: Unified Modelling Language.

- Language for precisely expressing requirements, specification models and designs.

- Platform-independent.

- UML uses many modelling notations:
  - Use case diagrams
  - Class diagrams
  - Object diagrams
  - State machines
  - Object Constraint Language (OCL)
  - Collaboration diagrams
  - Sequence diagrams
  - Activity diagrams
  - Deployment diagrams.

= Studied in this course.

- Use Case Diagrams

- The use case model describes:

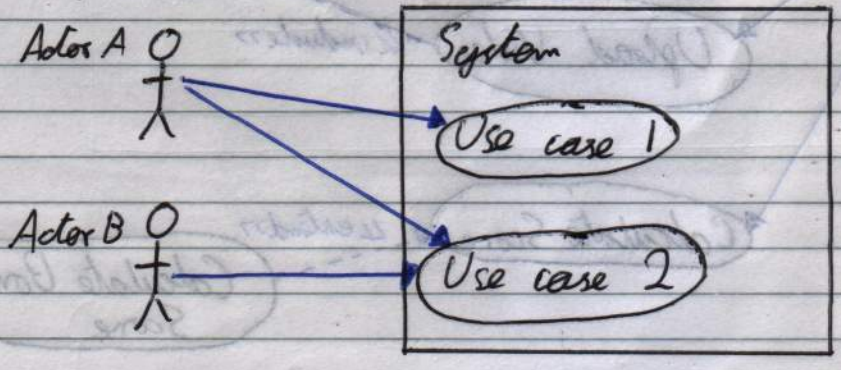
  - The system to be built

  - The actors - any roles played by people/entities that must interact with the system.

  - The use cases - families of usage scenarios, grouped into cases of functionality.

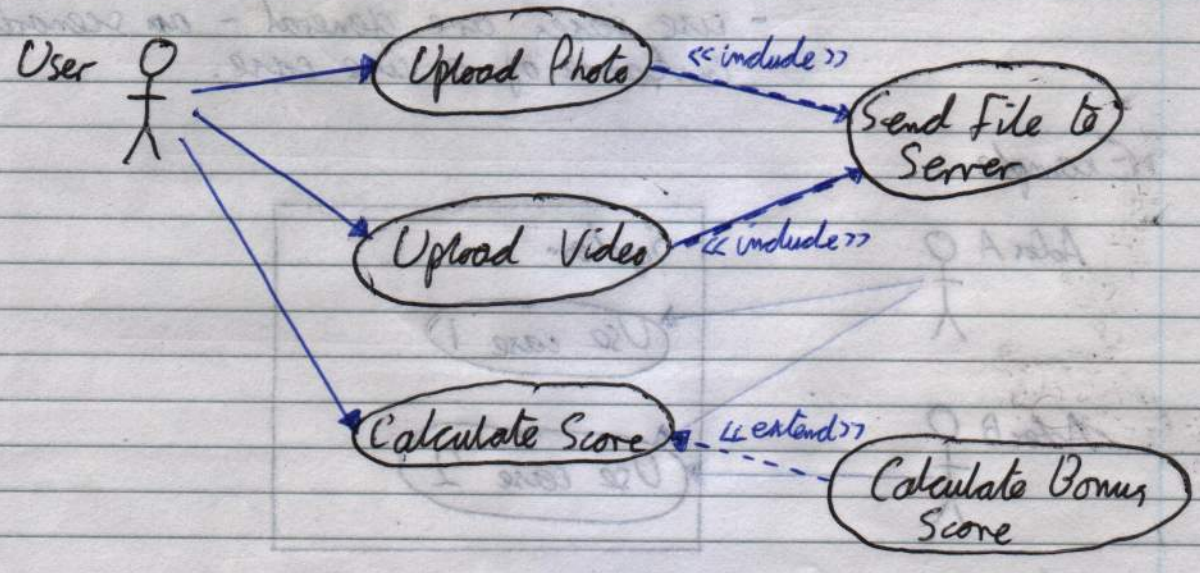
    - use cases are general - a scenario is an instance of a use case.

- Example:



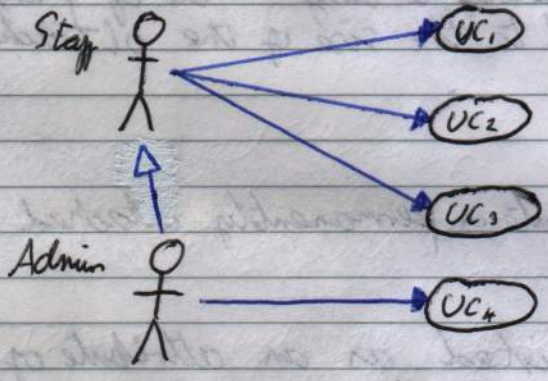
- Use cases may have **textual descriptions**:
  - Summary: description of purpose
  - Start of use case
  - End of use case

    } A trigger, eg. "use case starts/ends when ... occurs.
  - Interaction between use case and actors
  - Exchanges of information between system and actors
  - Chronology and origin of information - when the system requires information and when it records it.
  - Optional situations - points where system or actor may
- Use case UC_1 **includes** use case UC_2 if doing UC_1 always includes doing UC_2.
  - This is useful if a given use case is a common sub-task of two or more use cases.
- Use case UC_1 **extends** use case UC_2 if UC_1 provides additional functionality to carry out UC_2 in some cases.
- Example:



- Use Case Relationships

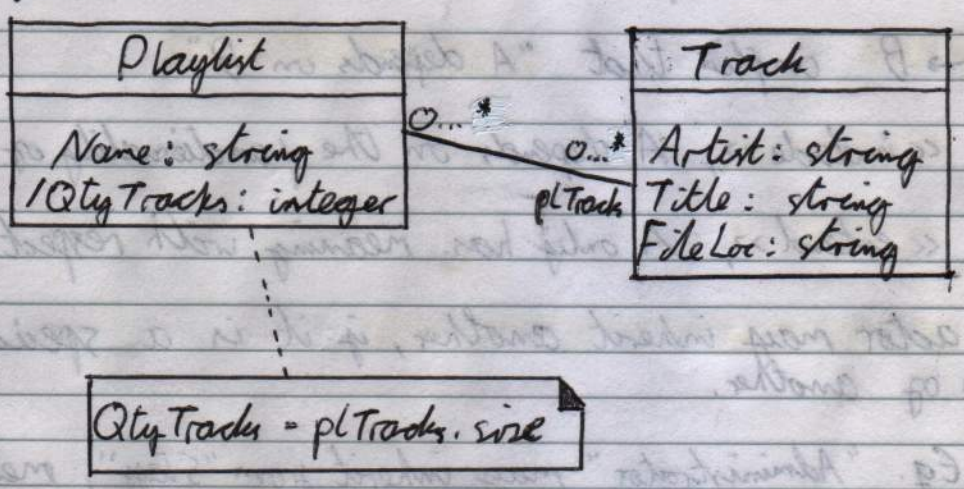
  - $A \rightarrow B$ implies that "A depends on B"
  - For «include», A depends on the functionality of B
  - For «extend», A only has meaning with respect to B
  - An actor may inherit another, if it is a specialised form of another.
    - Eg. "Administrator" may inherit from "Staff", meaning it has access to all of "staff"'s use cases, plus any of its own.



- Class Diagrams

  - Represent classes and the relationships between them.
  - Can be used at conceptual, spec. and design stages of development.
  - Four kinds of element:
    - Classes - rectangles with the name at the top
    - Attributes - inside box of class, written as name: type
    - Relationships - lines between classes, named as "x-y" with class names.
      - each end can have multiplicity indication and a role name
    - Constraints - in dog-eared boxes attached by dashed lines
      - derived attributes are prefixed with /

Eg:



- Each playlist can have any number of tracks
- Each track can belong to any number of playlists
- QtyTracks is equal to the size of the plTracks.

• Attributes

- An object attribute is permanently attached and cannot be removed.

- If $a:T$ is listed as an attribute of C, and o is an instance of C, then $o.a$ is of type T.

- Attributes can be given a default value:
  - $att:T = x$
  - This value is set for all new objects of the class, unless overridden in a constructor

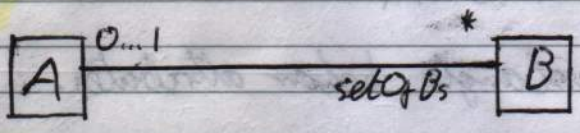
• Associations

- If C_1 and C_2 are associated, and there is a role r at the C_2 end, then for each obj of C_1, $obj.r$ is a set of C_2 objects.
  - If the multiplicity at the C_2 end is 1, $obj.r$ is a single C_2 object.

- At each point in time, there will be a set of instances of each class. An association is also a set of pair of associated objects.

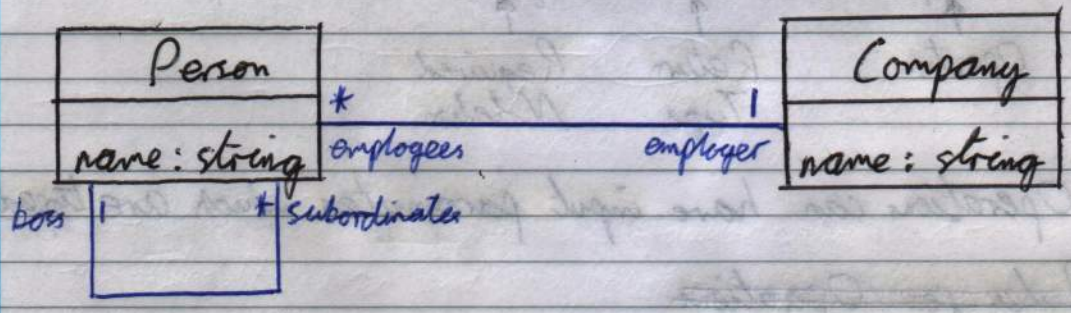
  - Eg. Playlist_Track = { $p1 \mapsto t1$, $p1 \mapsto t3$, $p1 \mapsto t4$, $p2 \mapsto t1$, $p2 \mapsto t2$ }

- Common Mistakes / Points to Remember
  - Multiplicities go on the end nearest the class whose number of instances is being restrained
  - Role goes on the opposite end to the class of which it is a property.



- Each A can have any number of Bs
- Each B can have 0 or 1 As
- A.setOfBs = B1, B2, B3....

- A common association type is a many-one association.



- Associations can be derived, in which case role names are prefixed with /
- Associations can have a navigation direction, indicating the direction in which the system will navigate.

A → B	Navigation from A to B
A ↔ B	Bi-directional navigation
A — B	Not specified
A →× B	Cannot navigate from A to B
A —▷ B	IMPORTANT: A inherits from B

- If class A can call operations of class B via navigable association from A to B, then A is a client of B and B is a supplier of A.

- **Operations**
  - Classes may have operations/methods that specify the behaviour of a class.
  - Listed in a rectangle below attributes.
  - (Attributes, roles and methods are called features of a class).
  - Operations can be a query or an update:
    - Queries do not modify internal state
    - Updates do.
  - Query operations are written as:

    isComplete(): Boolean {query}

    ↑ Operation ↑ Return Type ↑ Required Notation

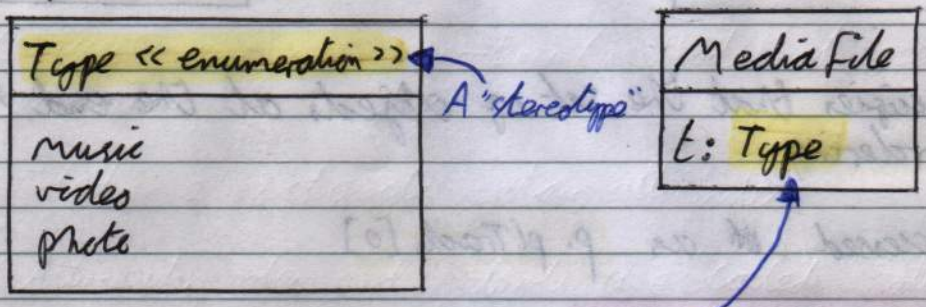
  - Operations can have input parameters, which are typed.

~~- Rules for Operations~~

- The effect of an operation is specified by a postcondition rule.
- An operation can be declared / defined / abstract:
  - **Declared**: written in a class box.
  - **Defined**: declared, or declared in a superclass
  - **Abstract**: same meaning as Java; written in italics.

• Enumerations.

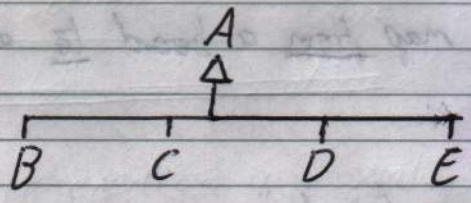
- A simple enumerated type, such as {on, off}.
- Represented as:



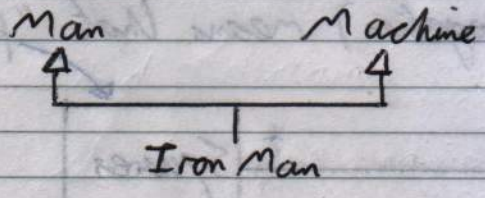
- Can be used as a type

• Inheritance

- Same meaning as Java.
- A ⇦ B means A is a superclass of B
- Common Mistake: inherited features do not need to be copied.
- Multiple subclassing is okay:



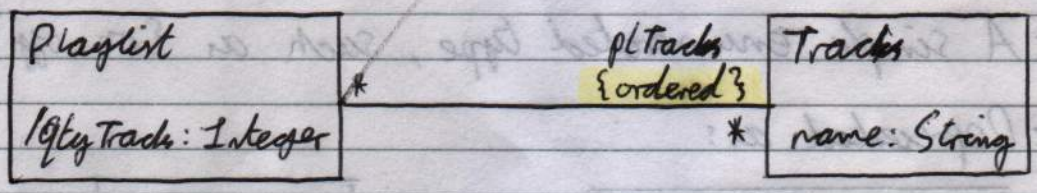
- Multiple inheritance is okay in UML, but it is not allowed in some languages.



• Abstract Classes / Operations

- Classes and operations can be abstract - they are written in *italics*.
- A class with abstract operations must be abstract.

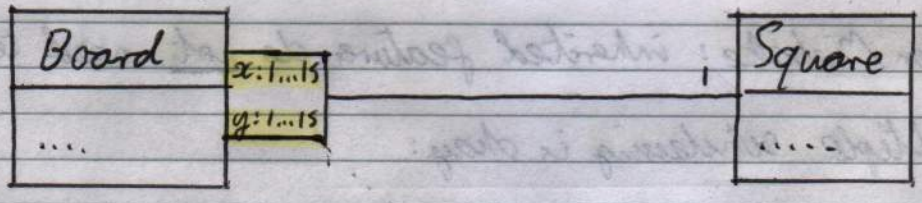
• Ordered Associations



- Specifies that the set of objects at the end has an ordering
- Accessed as p.plTracks[0]

• Qualified Associations

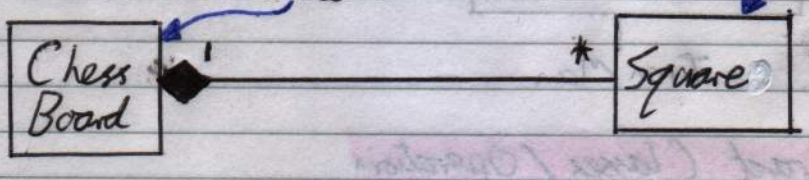
- From an object at one end, a set of values maps to the set of objects at the other end.
- Eg. a game may have: chessBoard.boardPieces[x, y]
- Without qualifier values, the entire set is returned.



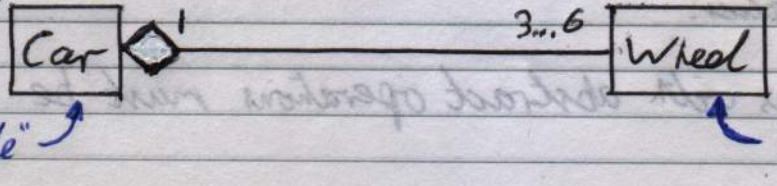
"The values x and y map from a board to a square"

• Aggregation and Composition

- Association: "a playlist has tracks"
- Inheritance: "a track is a file"
- Aggregation: "a wheel is part of a car"
- Composition (strong aggregation) means that "parts" cannot exist without the "whole".



- Aggregation can also be "simple":



• Object Diagrams

- A variant of class diagrams in which instance specifications are drawn, representing detailed specifications of objects.
- Drawn similar to class diagrams, but labelled as

  var : Class

  and attribute values given with equalities.
- Lines between objects represent an instance of an association.

Object Constraint Language & Dynamic Modelling

- OCL can express properties of diagram elements and inter-relation between elements.
- Used in class diagrams to:
  - Express invariants of classes - properties relating attributes and role names, which should be true for all objects of that class at all times when an operation is not being executed.
  - Express operation preconditions - properties of an object responding to an operation, and properties of operation inputs, that must be true when execution starts.
  - Express operation postconditions - as above, but concerning operation output, that must be true when execution finishes.
  - Properties that relate states of several objects, particularly objects of classes that have associations between them.

Built-In Constraints (keywords)

Written in italics →

Name and type underlined →

keyword	Constrains	Meaning
query	operations	Does not modify object state.
abstract	classes	Has no instances of its own.
	operations	Has no implementation in this class.
static	attribute or operation	Same as Java.
identity	attributes	Any two objects of this class have a different value for this attribute.

Specifying Constraints

- Class Invariants

- propertyA = propertyB
- propertyA ≠ propertyB
- The keywords **and** and **or** work as expected.

- Preconditions

- **pre:** moving = false on startMove()

- Postconditions

- **post:** squarePiece = {} => result = false
- **post:** squarePiece ≠ {} => result = true

} on isOccupied() : Boolean

- General format is "conditions => result = value"
- **result** is a special keyword.

OCL Operations & Types

- **self** - denotes object being constrained
- **Type Real** - mathematical concept of real numbers, not computer approximations such as FP numbers.
  - (In) equality: $r_1 = r_2$ or $r_1 \neq r_2$ or $r_1 /= r_2$
  - Arithmetic: $r_1 * r_2$ where $* \in \{+, -, \times, /\}$
  - Comparison: $r_1 * r_2$ where $* \in \{<, >, <=, >=\}$
  - Functions: r_1.abs, r_1.sqr, r_1.sqrt
  - Functions (returning Integers): r_1.floor and r_1.round

- Type Integer - a subset of Real; all above operators and functions work, plus...
  - div : integer division, such that 7 div 2 is 3
  - mod : modulus, such that 7 mod 2 is 1
- Type String - literal strings appear between "double quotes".
  - $s_1 + s_2$ is the concatenation of s_1 and s_2
  - $s_1 = s_2$ is true iff they have exactly the same strings in the same order
  - s_1.size, s_1.toLower and s_1.toUpper behave as expected.
  - Comparators (<, >, <=, >=) compare lexicographical order.
- Type Boolean
  - Conjunction: and , &
  - Disjunction: or
- Enumerations and class types can be used in OCL expressions within the model in which they are defined.
- = operator can be used on objects, as can /=

Collections in OCL

- Represent multiple data items grouped together.
- Two types:
  - Sets : unordered, no duplicates
  - Sequences: ordered, duplicates allowed
- Operations can be applied:
  - $s_1 = s_2$: For sets, this is true if they have exactly the same elements. For sequences, the ordering must match as well.
  - s.size : Number of elements in the set, or length of sequence.

- $x : s$: x occurs in s
  $x \in s$

- $x \notin s$: x does not occur in s
  $x \notin s$

- $s_1 \subset s_2$: s_1 is a subset of s_2

- $s_1 \not\subset s_2$: s_1 is not a subset of s_2

- s.min : min/max for non-empty collections of elements supporting
  s.max the < and > operators.

- s.sum : sum of collection of elements ~~containing~~ supporting the + operator.

- s→select(P) : Set: set of all elements that satisfy P
  Sequence: sub-sequence of all elements that satisfy P, retaining order.

- Extra Set Operations

- $s_1 \cup s_2$: union of s_1 and s_2

- $s_1 \cap s_2$: intersection of s_1 and s_2

- $s_1 - s_2$: difference of s_1 and s_2; s_1 with all elements of s_2 removed.

- Extra Sequence Operations

- $s_1 \frown s_2$: all s_1 elements, followed by all s_2 elements.

- s[i] : the ith element of s, with numbering starting at 1

- s.asSet : set of elements in s, with ordering and duplicates discarded.

- Literals

- A literal set is denoted as {x, y, z} or Set{x, y, z}

- A literal sequence is denoted as Sequence{x, y, z}

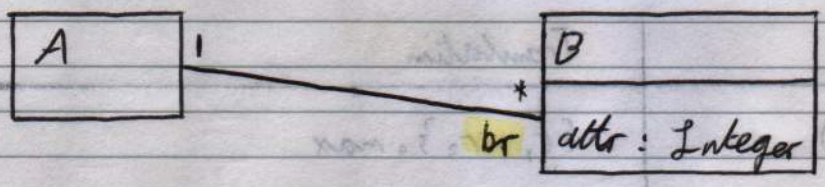
- OCL Notation

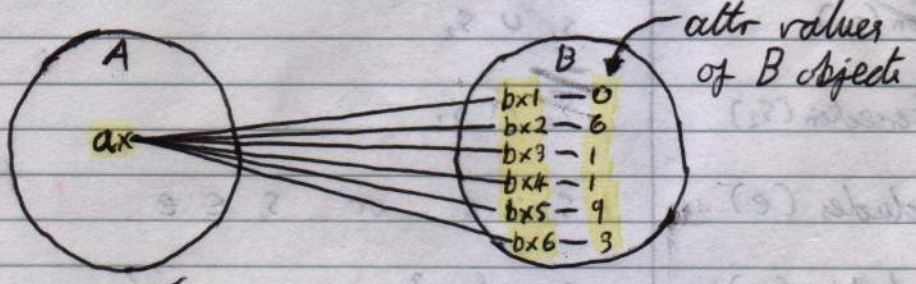
OCL	Translation
r_1.max(r_2)	$\{r_1, r_2\}$.max
S_1 → union(S_2)	$S_1 \cup S_2$
S_1 → intersection(S_2)	$S_1 \cap S_2$
S → includes(e)	$S_1 : e$ or $S_1 \in e$
S → including(e)	$S_1 \cup \{e\}$
S → excludes(e)	$S_1 \not: S_2$ or $S_1 \notin e$
S → excluding(e)	$S_1 - \{e\}$
set → isEmpty()	set = {}
seq → at(i)	seq[i]
C.allInstances()	C - the set of all current instances of the class C.

Navigation Expressions

- Two variants:
  - **Object references**: rolename or obj.rolename
    - Where obj is an object or collection of objects, obj.rolename is the collection of all objects related by association to obj (or any element of obj if it is a sequence).
  - **Value references**: obj.attr
    - Where obj is an object or collection of objects, obj.attr is the collection of attr values for obj or each element in obj.
    - If obj. is a sequence, then obj.attr will be a sequence, possibly with duplicates.

- Example





Sequence.

ax.br = ~~Set~~ {bx1, bx2, bx3, bx4, bx5, bx6}

ax.br.attr = Sequence {0,6,1,1,9,3}

- Set or Sequence?

The following rules determine whether the result of a navigation expression is a set or a sequence:

- obj.role

  - If obj is single and role is ordered
  - If obj is a sequence and role is ordered
  - If obj is a sequence and role is single-valued.

  → Sequence.

  - If obj and role are sequences:

    result = obj[1].role ⌢ obj[2].role ⌢ ... obj[obj.size].role (a Sequence)

  - If obj is a set or role is a set → Set

- obj.attr

  - If obj is a sequence → Sequence
  - If obj is a set → Set

Example Uses.

- As an invariant:

letterMoves.x.size = 1 or letterMoves.y.size = 1

means that for each move, all x-coords or y-coords must be the same.

- As an invariant:

s: history[1].letterMoves.x and s: history[1].letterMoves.y.

means that the first move must include square (8,8).

- Operations

addEmployees(c: Company, p: Person)
post: c.employees = c.employees@pre ∪ {p} &
p.employer = c

The @pre tag denotes the value of employees at the start of the operation.

- Query Operations in Navigation Expressions

wordsFormed.getScore().sum

computes the sum of wd.getScore() values for each wd in wordsFormed.

Query operations behave the same way as attributes if they have a single-value result, or as *-multiplicity roles if they have a collection result.

Quantifiers

- s → forAll(P) "P is true for every element of s"
- s → exists(P) "P is true for at least one element of s"

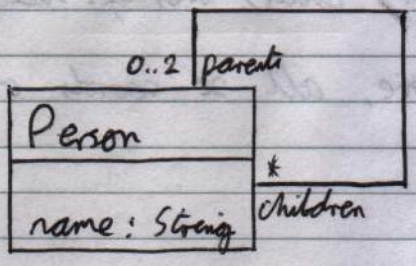
P usually involves features of the class of the objects of s, or navigation expressions starting from this class.

Eg. gameBag.gameLetters → forAll(score ≤ 10) means that every letter in the bag has a max score of 10.

Recursion

The function **closure** provides recursion when applied to a **self association**.

Eg.



For a Person p...

p.children.closure =
p.children ∪
p.children.children ∪
p.children.children.children ∪ ....

Identity Attributes

If the attribute attr has the constraint {identity} beside its declaration, then no two objects of that class may hold the same value in that attribute.

They are used as **unique identities**.

If the class C has identity attribute key, then:

$$a:C \;\&\; b:C \;\&\; a.key = b.key \Rightarrow a = b.$$

They are a useful way to define equality for objects, and in particular classes such as integer can be used this way.

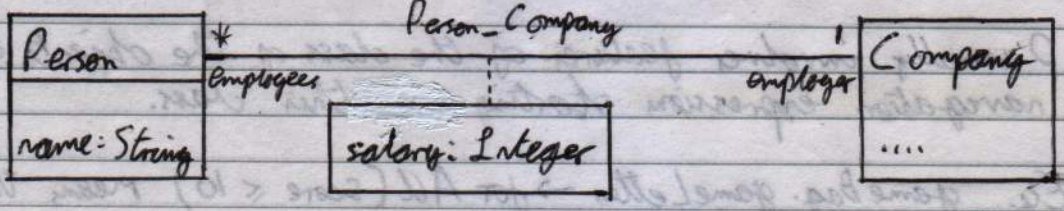
For the ID attribute key:T of class C, the set...

$$C \rightarrow select(key = val)$$

will always have 0 or 1 elements for each val in T. In UML-RSDS this set/object can be written as C[val].

Association Classes

- Associations with their own sets of attributes, operations and roles.
- Denoted by separate association and class symbols, joined by a dotted line:



- Association classes have implicit single-value roles back to the associated classes.

- They have a uniqueness property:

$$j1 : Person\_Company \;\&\; j2 : Person\_Company \;\&\; j1.person = j2.person \;\&\; j1.company = j2.company \Rightarrow j1 = j2.$$

Implicit Associations

- An assoc. may be implicit if it represents a relationship which will not be explicitly recorded in implementation data.

- In code, the fact that two objects are associated may not be stored in memory. Instead, it will be determined by computing some predicate or condition associated with the objects.

  - In UML, this predicate can be attached to an association.

- This approach is necessary if recording an entire association is impractical (eg. large multiplicities).

Interfaces

- Class rectangles with the stereotype «interface»

- Cannot inherit from a class, but can be an assoc. endpoint.

- No private features.

- Everything else is exactly like Java.

Creating a Platform-Independent Specification Model

Steps:

- Identify functional/non-functional requirements
- Formalise functional requirements into use cases
- Identify entities and relationships that the system must be aware of / operate upon.
- Formalise data model as entities and associations in a specification class diagram.

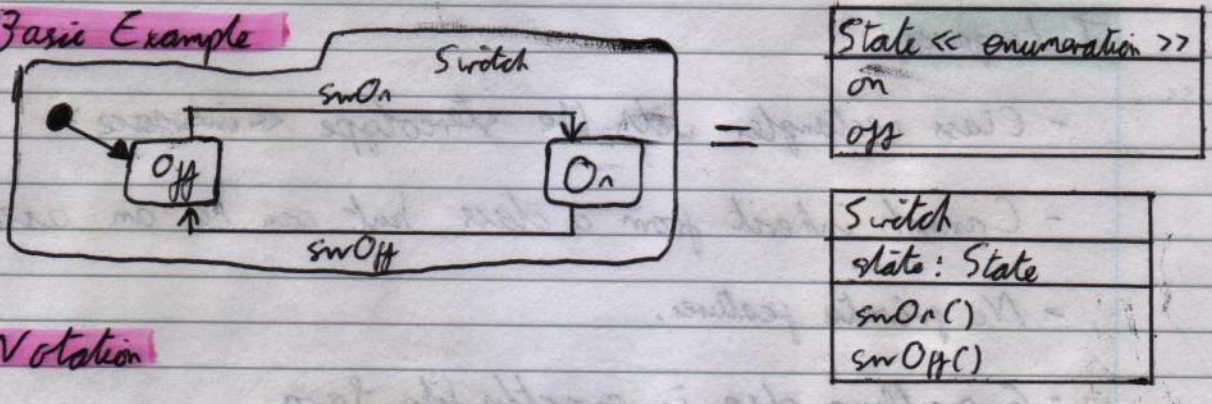
UML Dynamic Modelling Notations

- State Machines, Collaboration Diagrams and Sequence Diagrams

State Machines.

- These describe dynamic behaviour of objects, graphically represent the history of objects and show patterns of inter-communication.

Basic Example



Notation

- **States** - rounded-corner boxes with a name.
- **Transitions** - an arrow from source to target, labelled with the name of the event.
- **Initial state** - an "entry point" pseudostate is defined as a filled black circle.
- **Termination** - shown by a bullseye ⊙ symbol.

Types of State Machine

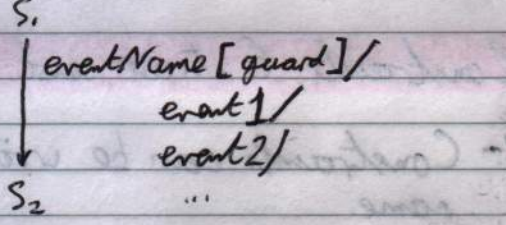
1. Protocol State Machines - describe allowed life histories of objects of a class. Transition events are object operations and may have pre/post conditions.
2. Behaviour State Machines - describe method execution. Transitions do not have postconditions but can have actions.

Semantics

- For a protocol state machine attached to a class:
  - Each object of the class begins its life history in the default state of the state machine.
  - If object obj is in state s and event α occurs on obj, then if there is a transition labelled with α and a source of s then this transition occurs and the object moves to the target state of the transition.

Transition Guards and Actions

- A guard is a condition using features of an object that must be true for a transition to take place.
- An action specifies the event(s) generated when a transition occurs.
- The typical format is:



- Actions can include updates as feature = val or feature := val of local attributes or role names, or invocations of operations on the object or objects that can be navigated to from it.

Rules for State Machines

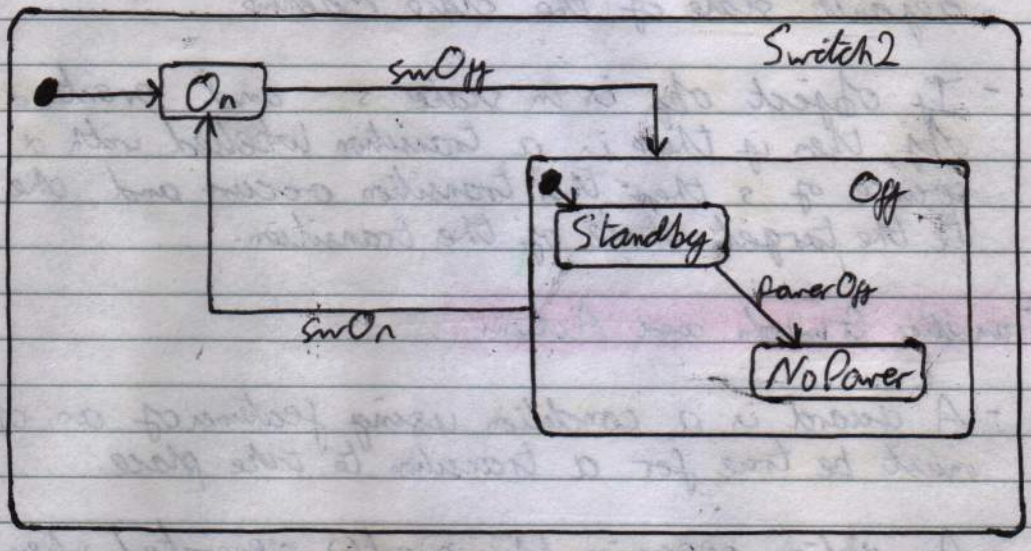
For each state s and event α, there can be at most one unguarded transition for α from s.

If there are multiple transitions for α from s, all must be guarded by disjoint guards.

For simple state machines, an object is in exactly one state at all times between creation and destruction.

Composite States

Yep. They exist:



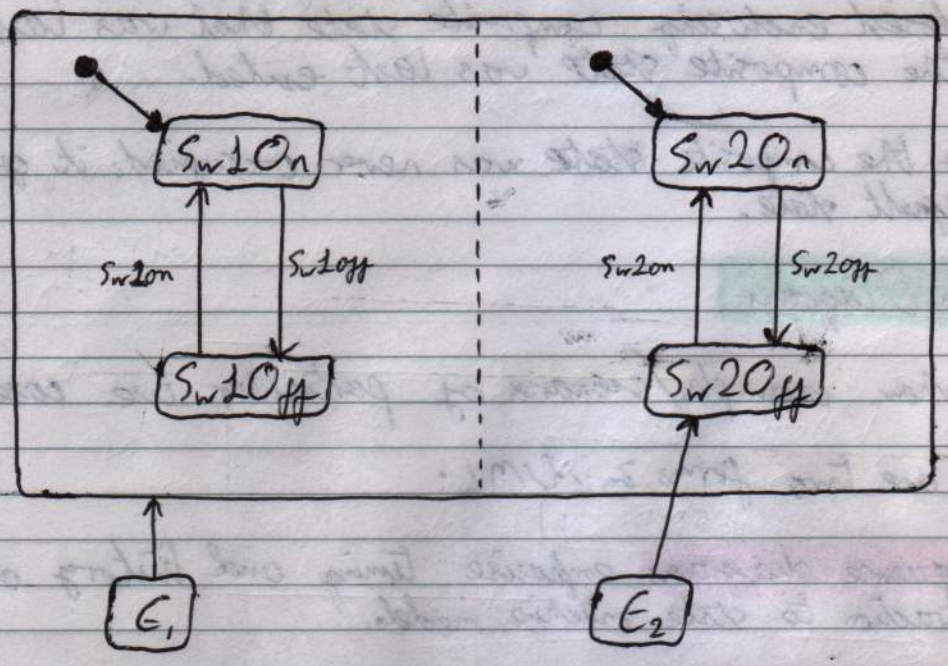
If an object is in the basic state s, it is also considered to be in every state that contains s.

Constraints, Entries and Exits

- Constraints can be written in square brackets after a state name.
  - Objects must satisfy these constraints while in the state, including any substates.
- If every transition into the state s has the same sequence of final actions, they can be removed and abbreviated to entry/act1/act2/... on the state s.
- Likewise for transitions out of the state s, if they have the same initial actions they can be abbreviated to exit/act1/act2/...

Concurrent States

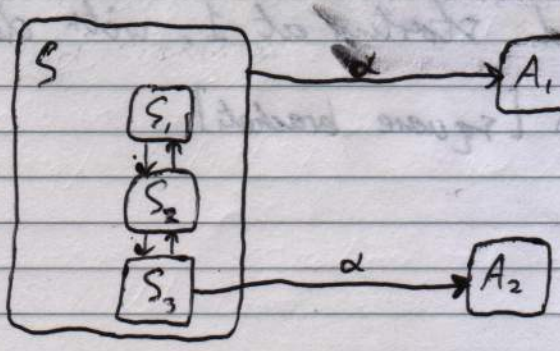
- When concurrent composite states are entered, each subcomponent is entered.
  - Usually from the default state (E_1), but it is possible to specify direct entry to specific states (E_2).



- Transitions may exit from multiple sources; these sources must be from different components of a concurrent state.
- Transitions can enter multiple targets, from different components of a concurrent state.
- In both of these cases, the multi-transitions have a vertical bar joining entry and exit lines.

Transition Priority

- Substate transitions have a higher priority for transitions of the same name.



$S_1 \xrightarrow{\alpha} A_1$

$S_2 \xrightarrow{\alpha} A_1$

$S_3 \xrightarrow{\alpha} A_2$

History States

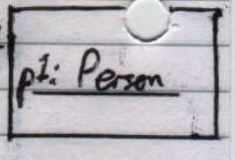
- (H)
- These states "remember" what substate of a composite state was last occupied.
- Transition to this state will actually go to the substate of the closest enclosing composite state that was last occupied when the composite state was last exited.
  - If the composite state was never occupied, it goes to the default state.

Interaction Diagrams

- These show examples/scenarios of particular use cases.
- There are two forms in UML:
  - Sequence diagrams emphasise timing and history and relate interaction to state machine models.
  - Collaboration diagrams show interaction as messages between objects.

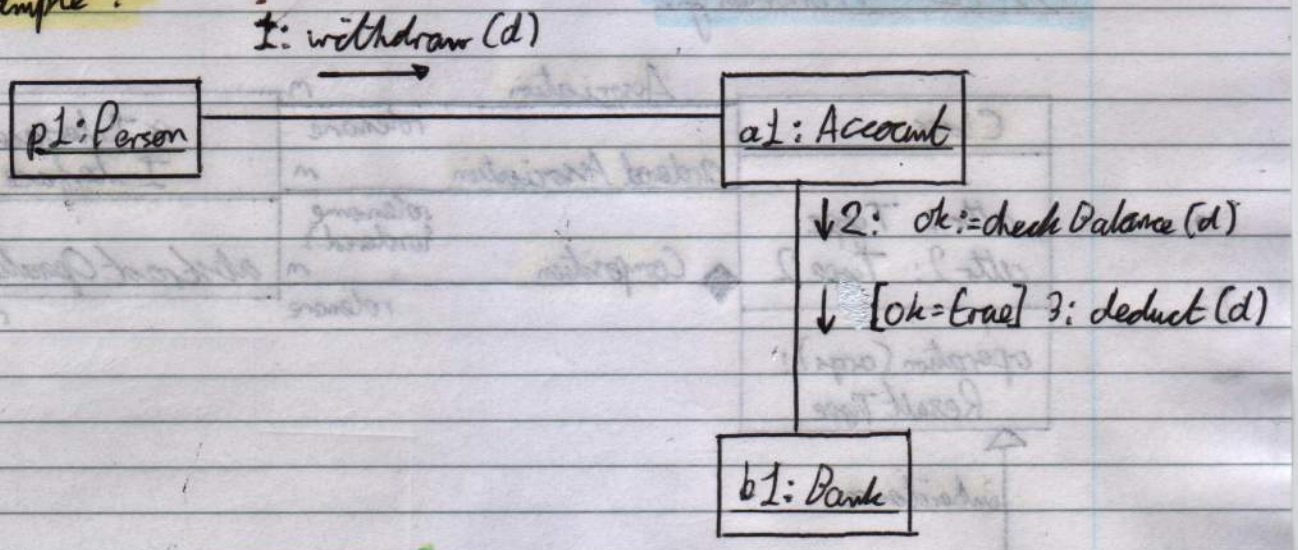
Collaboration Diagrams

- Objects are shown as boxes with name and type, eg



- Objects created during execution are noted as {new}
- Links can have arrows for nav. direction.
  - If a message is sent from a to b, a must have access to b, eg. by association.
- Messages are numbered starting at 1, with decimals used for nesting.
- Conditions are given in [square brackets].

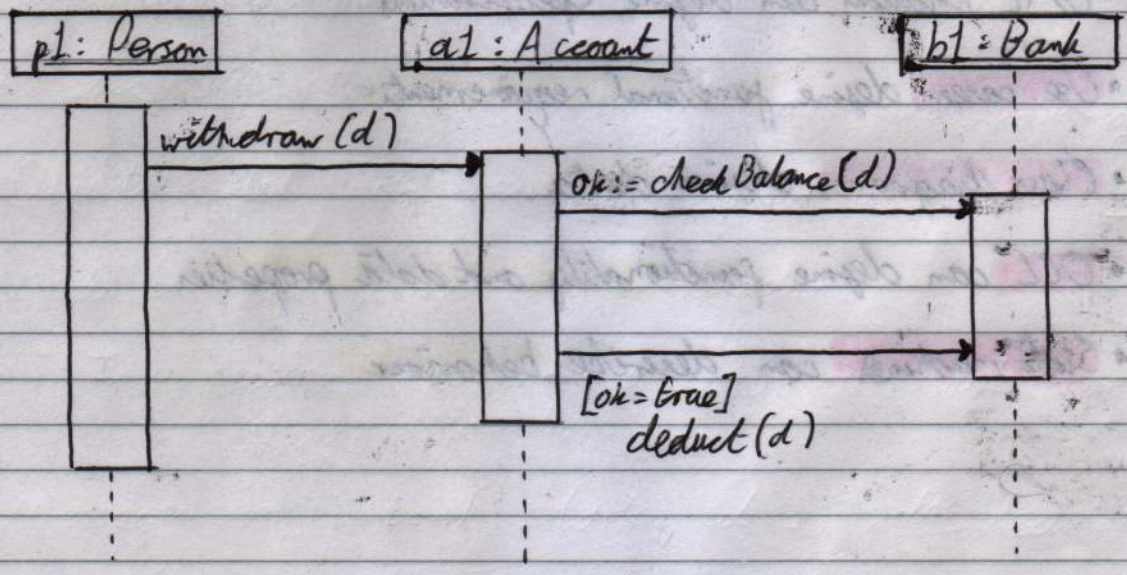
Example:



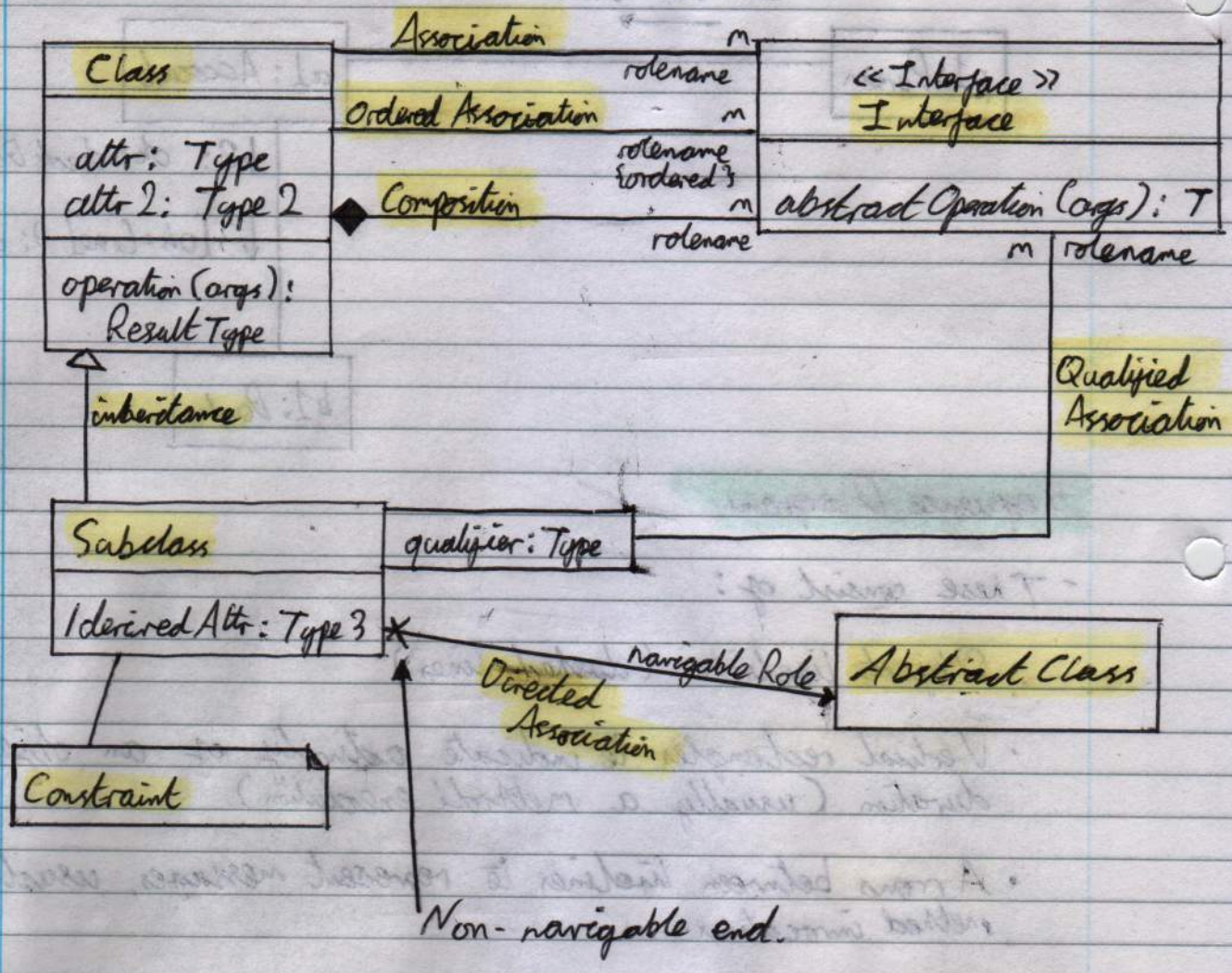
Sequence Diagrams

- These consist of:
  - Object lifelines (dashed lines)
  - Vertical rectangles to indicate activity of an object and duration (usually a method execution).
  - Arrows between lifelines to represent messages, usually method invocations.

Example:



UML Summary



- UML notations can define specifications
- Use cases define functional requirements
- Class diagrams define data
- OCL can define functionality and data properties
- State machines can describe behaviours

Platform-Independent Design

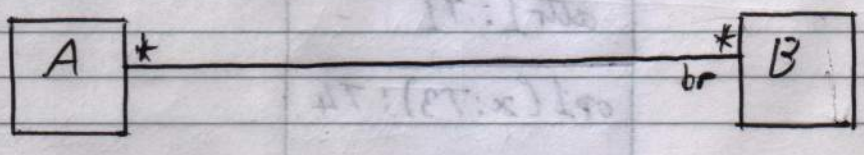
- Concerned with constructing components, organising their interations to achieve system requirements.
  - Identifying subsystems and modules which have well-defined purpose and functional cohesion.
  - Identifying module dependencies and specifying interfaces and responsibilities.
  - Includes model transformations and design patterns.

Model Transformations

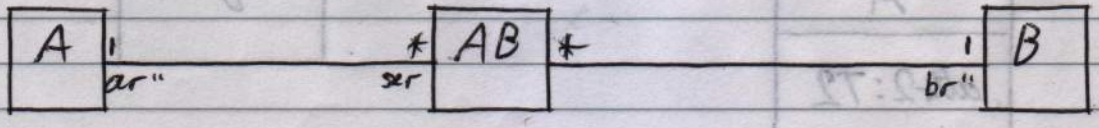
- Used to improve structure or progress towards implementation
- In MOA terms, these are PIM → PIM or PIM to PSM mappings
- Transformations must preserve constraints.

Removing Many-Many Associations

- Replaces a *-* assoc. with two 1-* assocs and an intermediate class.
- Useful for implementation as a relational database.
- If a:A and b:B are related by A-B, we make an AB object, x, such that x.ar" = a and x.br" = b, and vice-versa.

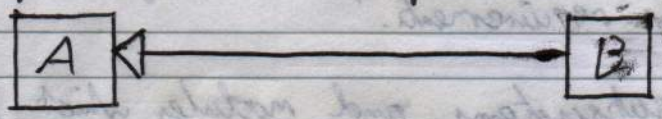


becomes...

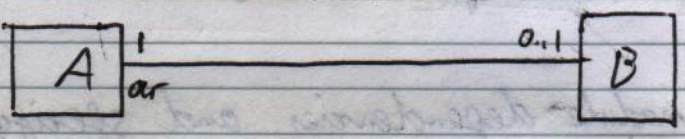


Replacing Inheritance with Association

- Useful to remove multiple inheritance from PIMs.



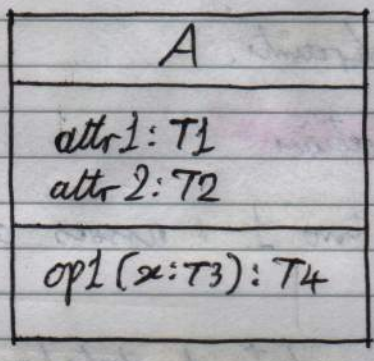
becomes...

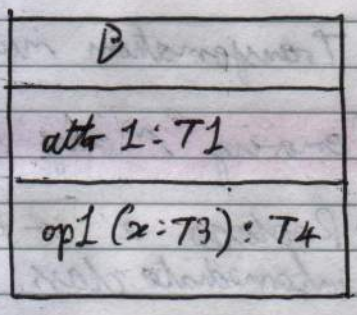


- Features f of A used in B must be refered to as b.ar.f.

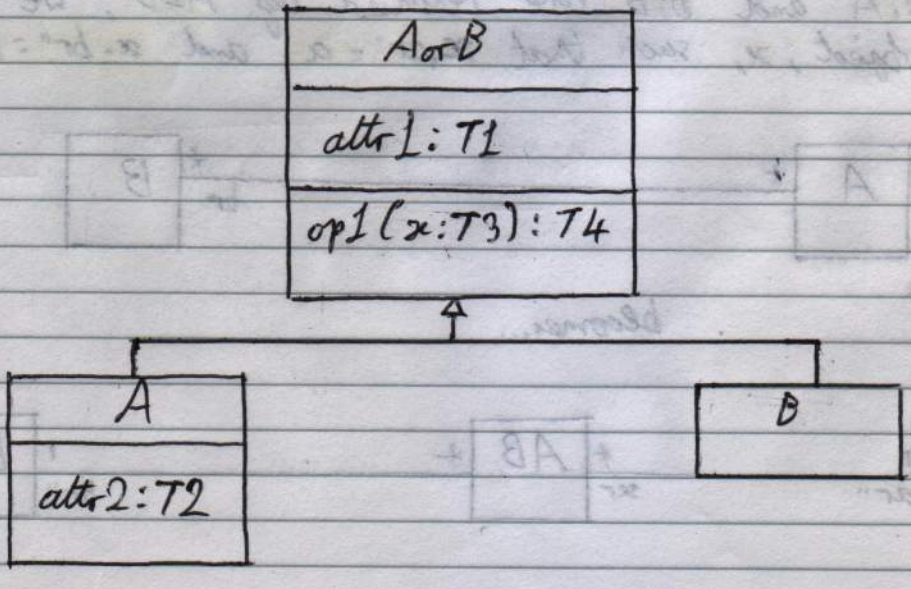
Introducing a Super-Class

- Merges common features from 2 or more classes.
- The new class is usually abstract.





becomes...



Design Patterns

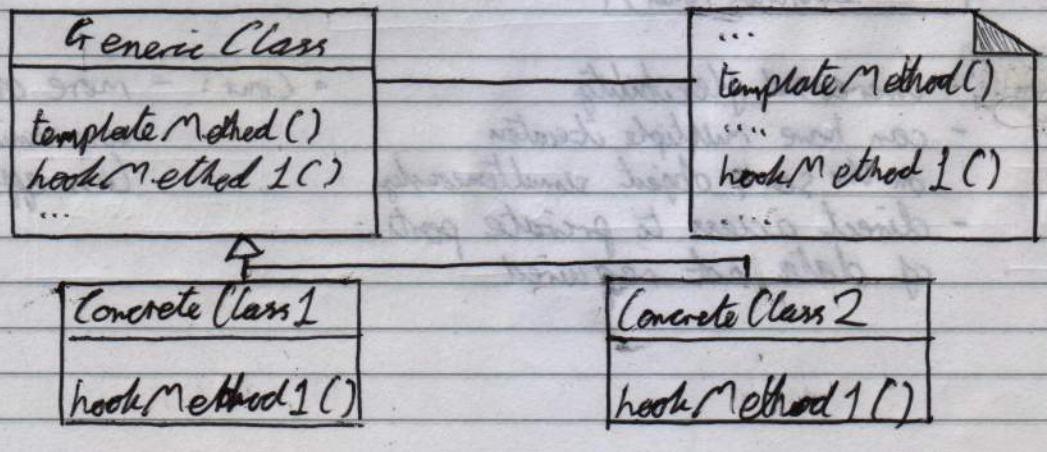
- Characteristic structures of software used to solve particular design problems.
- Mostly independent of programming languages, so can be used in platform-independent design.

Types of Design Pattern

- Creational - organise creation of objects and structures.
- Behavioural - organise distribution of behaviour amongst objects.
- Structural - organise the structure of classes and relationships

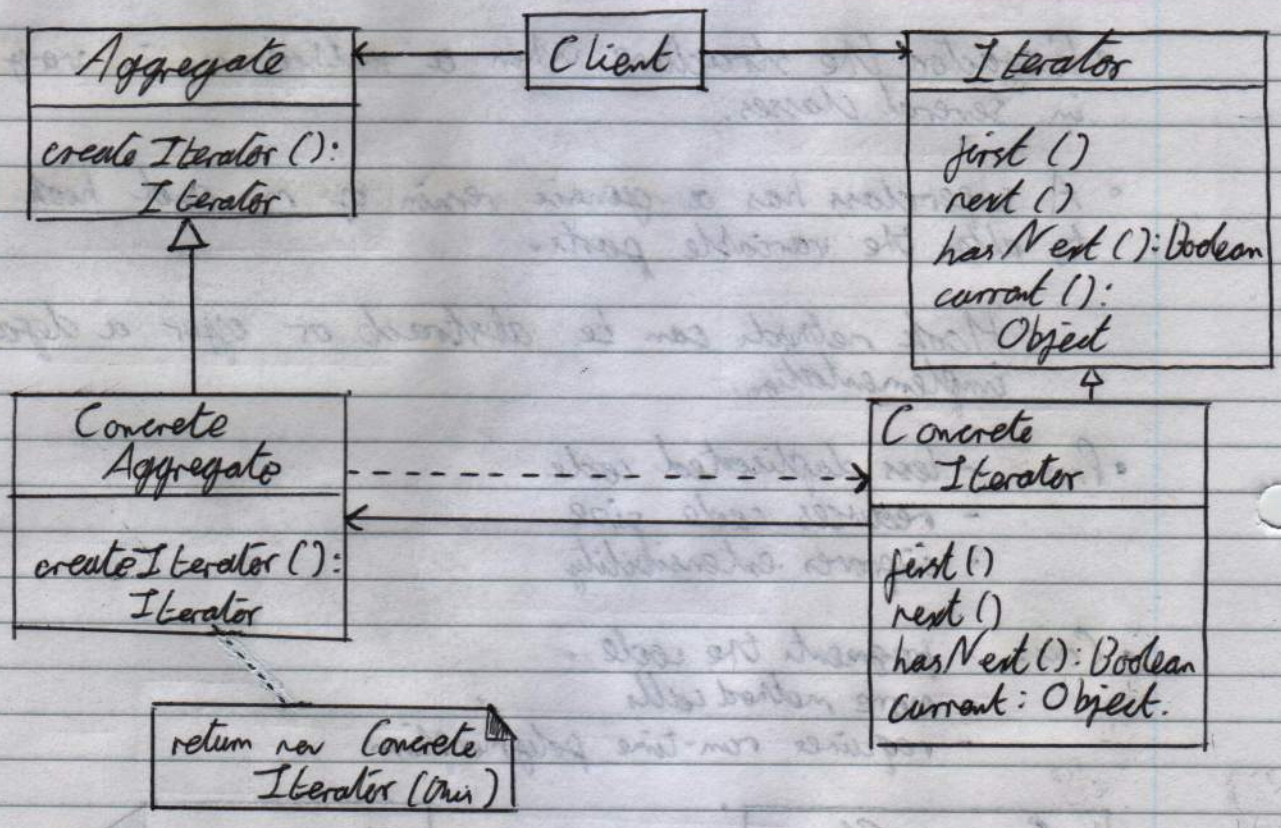
Template Method Pattern B

- Refactor the structure when a method m is very similar in several classes.
- A superclass has a generic version of m and hook methods to alter the variable parts.
  - Hook methods can be abstract or offer a default implementation.
- Pros:
  - less duplicated code
  - reduces code size
  - improves extensibility
- Cons:
  - fragments the code
  - more method calls
  - requires run-time polymorphism



Iterator Pattern . B

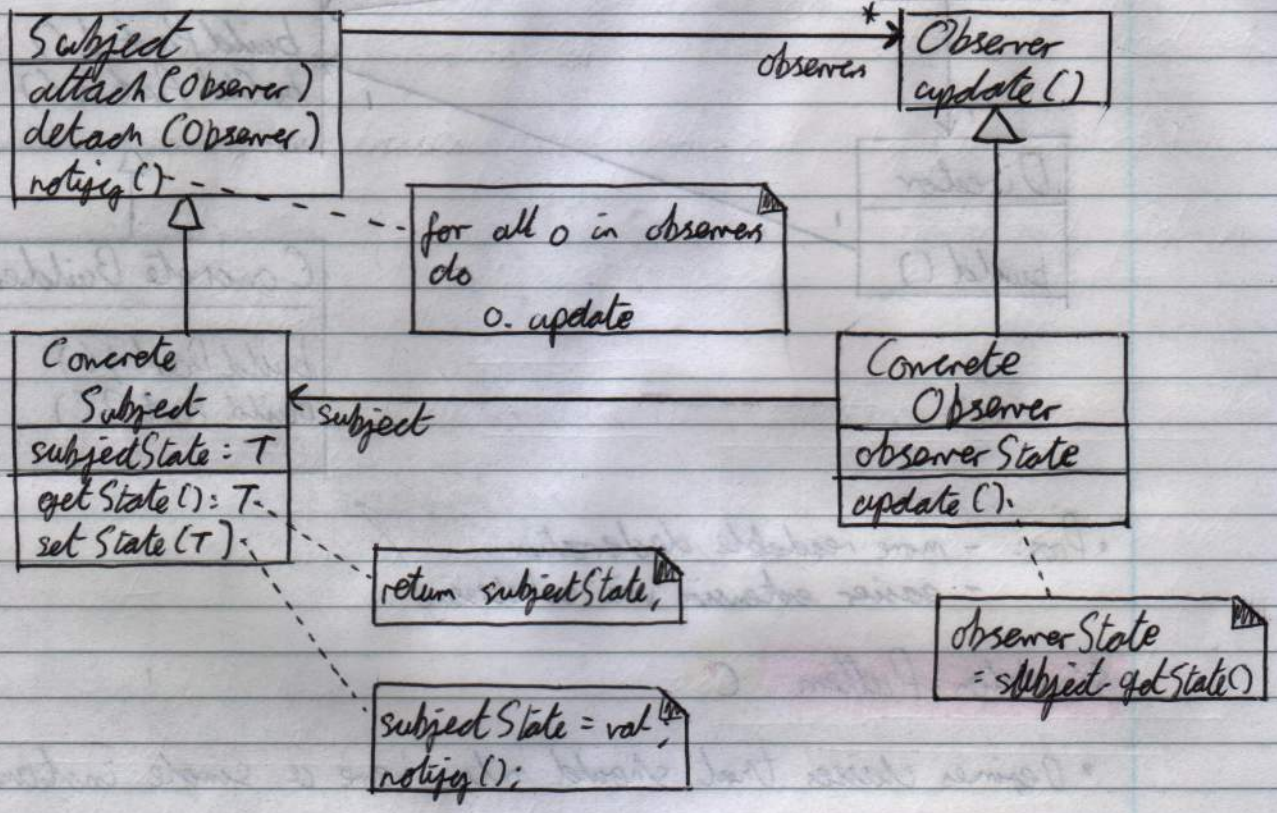
- Supports access to elements of an aggregate data structure.
- Applicable:
  - multiple traversal algorithms over an aggregate are required (eg. pre/post order).
  - a uniform traversal interface is required over different aggregates
  - aggregate classes and traversal algorithms must be able to vary independently.
- The iterator object acts as a pointer into a structure.



- Pros:
  - increased flexibility
  - can have multiple iterators on the same object simultaneously
  - direct access to private parts of data not required
- Cons:
  - more object/object communication, so less efficient.

Observer Pattern B

- Defines a one-to-many relationship between objects so that when one object (the data/subject) changes state, all dependent objects are notified.
- Applicable when an action in one entity requires action to be taken in another.



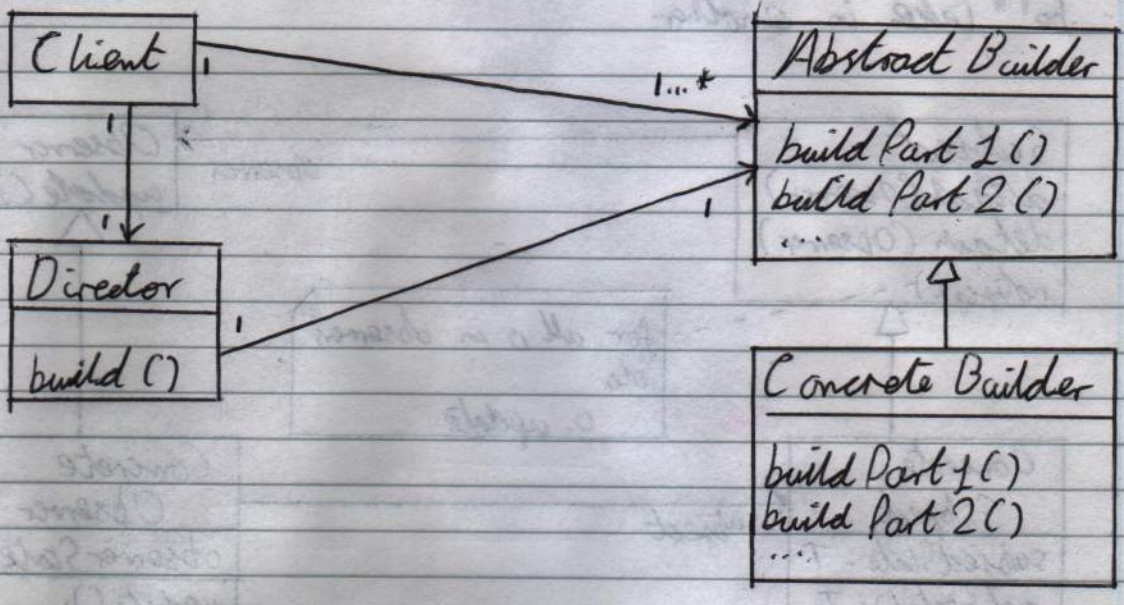
- Referential integrity must be maintained, ie.

$$o : s.observers \equiv o.subject = s$$

- Heavily used in MVC Structures
- Pros: - modularity, so components may vary independently
  - extensibility, so more observers can be easily defined and added
  - customizability
- Cons: - cost of communication between objects
  - need to maintain referential integrity.

Builder Pattern C

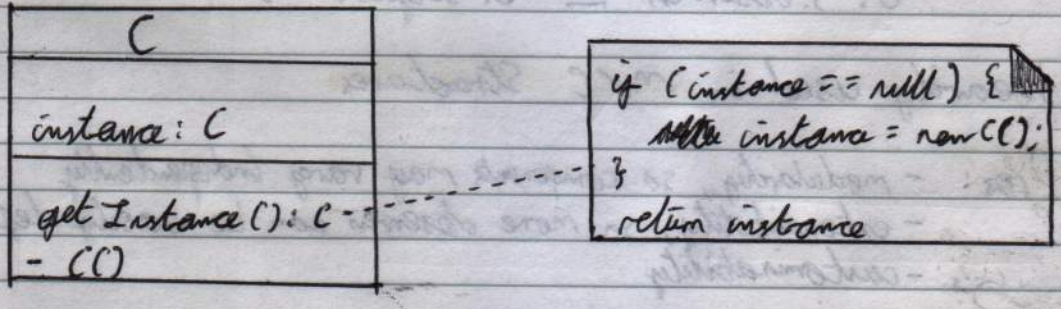
- Allows the declarative creation of complex objects.
- Details of creation are managed in a separate class with subclasses for each variant of the object to be built.



- Pros: - more readable declaration
  - easier extension via subclassing

Singleton Pattern C

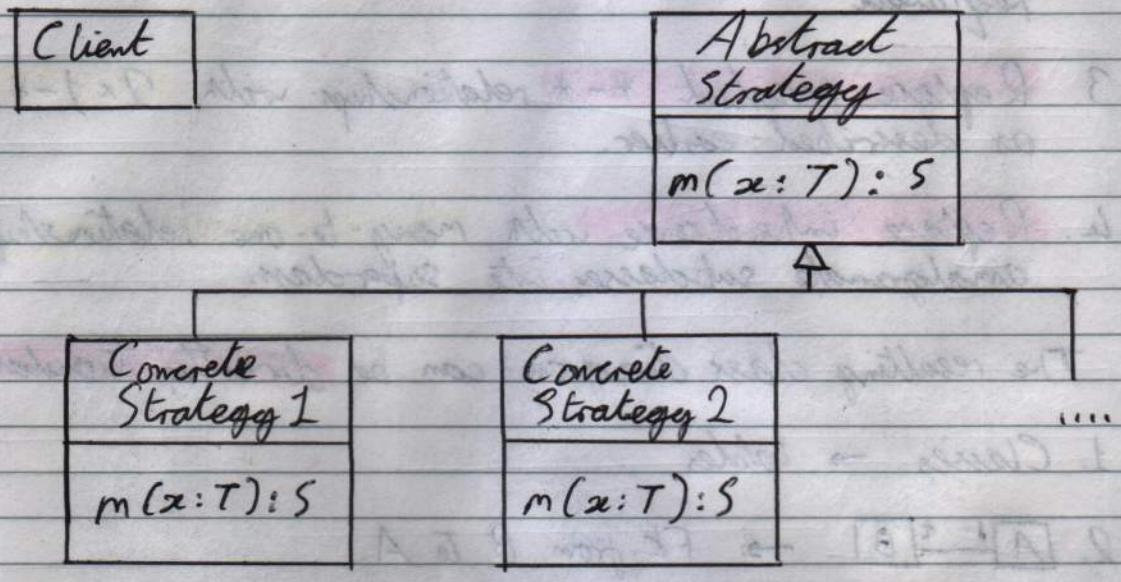
- Defines classes that should only have a single instance.
- Applicable when a unique instance of a class must be accessible to clients.



- Pros: - provides controlled access to singleton
  - reduced name space - no global var
  - permits subclassing singleton
  - can be adapted to give any number $N \geq 1$ of instances
- Cons: - requires a platform that allows private constructors.

Strategy Pattern B

- Defines a common interface for different versions of an algorithm, allowing the client to select which version to call operations on.



- Pros: - simplifies code
  - easy extension
  - allows dynamic selection of algorithm variant

User Interface Design

- Platform-independent GUI design can be carried out using generic terms such as "frame", "button", etc.
- This really didn't deserve its own section.
- There's nothing else to say that isn't related to Scrabble.

Data Repository Design

- This identifies what data needs to be stored persistently, and what structure will be used.
- Several options exist: - relational DB like MySQL.
  - object-oriented DB like ObjectStore
  - structured text such as XML or JSON.

Transforming Class Diagram → Relational DB Schema.

1. Identify which entities, attributes and associations are to be made persistent.
2. Introduce primary keys as new identity attributes where required.
3. Replace persistent *-* relationships with 2 x 1-* relationships, as described earlier.
4. Replace inheritance with many-to-one relationships or amalgamate subclasses into superclass.

The resulting class diagram can be directly translated:

1. Classes → tables
2. [A] 1 —— * [B] → Fk from B to A.

Transforming a PIM to a PSM

- PIMs and PSMs can usually be expressed in UML notations, so the transformation from PIM → PSM can be expressed as transformation on a UML model.

PSM Class Diagrams

- Association navigation direction needs to be stated.
- Feature visibility must be explicit.
  - Visibility notations go before feature names.

    - private
    # protected
    + public

Java-Specific Considerations

- Method overloading is okay, as long as the difference is in arguments, not first return type.
- Static methods cannot be overridden in a subclass

- Trying to restrict visibility of a method in a subclass is invalid.
- If a class contains an abstract method, it must itself be abstract.
- Interfaces cannot contain constructors or instance-scope attributes.
- For aggregations, deleting a "whole" must delete all "parts".
- No multiple inheritance.
- Data types must be translated:

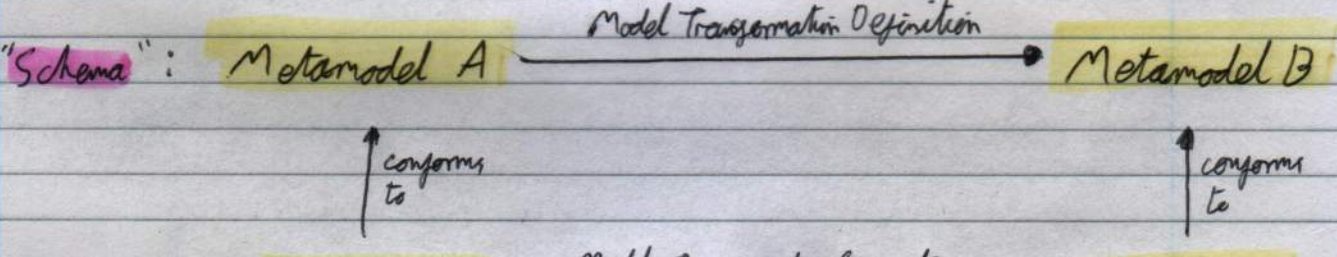
Integer	→ int
Real	→ double
Boolean	→ boolean
String	→ String

Model Transformations (Again)

- These are an essential part of MDD.
- There are many model transformation (MT) languages
- We focus on UML-RSDS: a UML-based MT lang. that creates executable Java code to carry out transformations

MT Concepts

- A source language/metamodel is mapped to a target language.
  - They can be the same.
- An MT takes an input source model and produces the corresponding target model.
- An MT is defined by transformation rules.



- A transformation can be:
  - code generation/refinement
  - migration
  - refactoring/restructuring

Transformation in UML-RSDS

- The source and target metamodels are drawn as class diagrams
- Transformations are defined as use cases
- Simple example: Precond: blank

  Postcond: "Hello World"->display()

- Another example: this squares numbers in A.y and copies the answer into B.x

Operates on: A
Precond : true
Postcond : B -> exists(b | b.y = x * x)

in.txt:	out.txt:
a1 : A	a1 : A
a1.x = 3	a1.x = 3
a2 : A	a2 : A
a2.x = -5	a2.x = -5
	b1 : B
	b1.y = 9
	b2 : B
	b2.y = 25

- Constraints are defined by $P \Rightarrow Q$ on entity E.
  - 1st text box: E
  - 2nd " " : P
  - 3rd " " : Q

- Rule mappings from source class Sc to target class Tc usually have the form:

$$Tc \rightarrow exists(t \mid t.f1 = e1 \ \& \ \ldots \ \& \ t.fn = en)$$

  - where f1 ... fn are features of Tc
    and e1 ... en are equations using the features of Sc.

- More complex example:

Operates on : Patient
Precond : p : Patient & id /= p.id & name = p.name
Postcond : (id + " and " + p.id + " may be duplicates") -> display()

- This means "for every Patient, and any other Patient p, if they have different IDs and the same name then they may be duplicates."

Agile Development

- Traditional development is "plan-based"
- Agile processes aim to be lightweight: the primary goal is to deliver a product to the customer that meets their needs in the shortest amount of time.

Concepts

- Traditional development is too slow: code is out of date by the time it is delivered.
- Agile addresses this by delivering parts of the system as working code as soon as possible.
- Delivery cycles are short so developers can adapt to changing requirements.

Principles

- Responding to change is more important than following a plan.
- Working software is more important than comprehensive documentation
- Individuals/interactions are " " " processes and tools.
- Customer collaboration is " " " contract negotiation
- Development cycles are iterative, and build small parts of systems with CT and CI.

- Self-selecting teams: these often produce the best results.
- Agile is suitable for smaller organisations/projects.
- XP is suitable for small projects maintained by a single person/small team.

Agile vs. Plan-Based

Agile	Plan-Based
• Small/medium scale	• Large scale (10+ people)
• In-house project/co-located team	• Distributed/outsourced team
• Experienced/self-directed developers	• Varying experience levels.
• Volatile requirements	• Fixed requirements
• Close interaction w/ customer	• Distant customer/stakeholders
• Rapid value/high-responsiveness required	• High reliability/correctness required.

Development Techniques

• Sprints

- Regular, re-occuring iterations to deliver project work that contributes to specific user requirements.
- Each iteration involves a set of use cases (aka. "stories"). UCs are classified as high/med/low priority by the customer, and by risk/effort by the developer.
    - High priority/High risk UCs should be done first.
- Project velocity is a value derived from the amount of developer-time available per iteration.
- Taking all of these together allows developers to create a release plan.

• Release Plan

- Consists of a set of sprints.
- The UCs in each sprint are chosen based on priority, dependencies and developer availability.

- **Refactoring**
  - Small changes to rationalise, generalise or improve system structure.
    - Eg. moving common features to a superclass.
  - Refactoring should not change functionality.
  - Should be carried out regularly, whenever a need is identified.

Methods

- **Extreme Programming (XP)**
  - Five phases:
    - Exploration - determine feasibility, understand requirements, develop exploratory prototypes.
    - Planning - agree dates and UCs for first release.
    - Iterations - implement/test UCs in iterations
    - Productionising - prepare documentation, training, etc. and deploy system.
    - Maintenance - fix and enhance deployed system
  - Sprints should be 1/2 weeks long.
- **Scrum**
  - Four phases:
    - Planning: establish vision, set expectations, develop prototypes.
    - Staging: prioritise and plan for first iteration
    - Development: implement UCs in a series of sprints and refine iteration plan.
    - Release: prepare docs, training, etc. and deploy system

- Iterations should be 1 week - 1 month long.

- Key Events:

  - Sprint planning: performed by Scrum team before a sprint to agree UCs to be worked on.

  - Daily Scrum: short (eg. 15 mins) meeting to organise activities, review progress & deal with issues. key questions for developers:

    - What did I achieve yesterday?
    - What will I achieve today?
    - What is blocking me?

  - Sprint review: at the end of a sprint

  - Sprint retrospective: after sprint review, before sprint planning. Analyse achievements of the sprint, ideas for improvement.

- Terms:

  - "Story": a UC
  - "Product Backlog": prioritised list of stories remaining in project
  - "Sprint Backlog": " " " " " " sprint
  - "Scrum Board": to-do / in progress / completed
  - "Burndown Chart": graph of remaining work vs. time.

Agile MDD

- Combines agile and MDD
- Specification is expressed in an executable model (a PIM or PSM), which can be delivered as a running system and demonstrated to customers.
- Incremental development now uses exec. models in place of code.
- Executable UML (xUML) or UML-RSDS.
  - These can ensure correctness.

• In Practise

- Build a "baseline" class diagram and refine use cases.
- Developers chose required use cases (from the current sprint) to work on and write unit tests for.
  - All share the same system model.
- Developers work on their tasks: can only enhance/extend, not remove or modify.
  - Rationalisations / refactorings need team agreement.
- Do regular full builds or CI.
- Deliver when all tests are passed.
- Good communication is required.

- Alternatively, each developer takes responsibility for one module.
- Developers agree on interfaces of modules and shared data.

Other Approaches

- TDD
- Pair Programming