1 Introduction

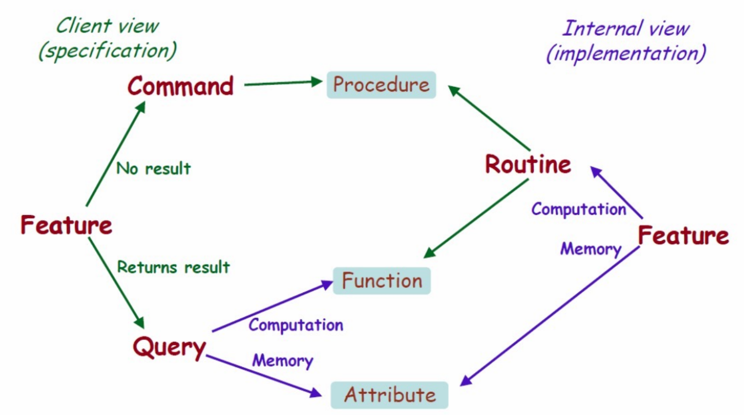
1.1 Basic definitions

- Class: A class is a description of a possible set of run-time objects. It only exists in the software text.
- Object:
  - An object is a software machine that allows program elements to access and modify a collection of data.
  - It's an actual representation (instance) of a class (generating class).
  - Objects only exist at runt-time and are visible in the program text through names.
  - The type of any object you use in your program must be declared either
    * In feature declarations
    * In the `local` clauses of routines (when only the routine needs to know about it)
- Feature: An operation that may be applied to all the objects of a class.

```
set_name(a_name: STRING)
--Set name to a_name
do
        name := a_name
end
name: STRING
```

  - Command:
    * No return value
    * Does modify the state of the objects
    * On the syntactical level, it is an instruction.
    * May or may not have arguments
  - Query:
    * Returns a value
    * Does not modify the state of any object
    * The syntax equivalent is the expression.
    * May or may not have arguments
    * Functions get their results through computation
    * Attributes are values directly stored in memory

    For queries, there is the uniform access principle, which states that it doesn't matter to the client whether a query is implemented as a function or attribute. Features should be accessible to clients the same way whether implemented by storage or by computation.
  - Creation Procedure: Commands to initiate objects, can be several. There is also a `default_create`, which is inherited by all classes, and does nothing by default.



- Feature Calls
  - Unqualified calls: Feature calls which apply to the current object, e.g. `f(args)`. Has `Current` as an implicit target.
  - Qualified calls: Feature calls which apply to a certain object (has an explicit target), causing this object to become the current object, e.g. `x.f(args)`
- Class clauses
  - Indexing
  - Inheritance
  - Creation
  - Feature
  - Invariant
- Specimen: A syntactic element, such as a class name or an instruction, but no delimiters. The type of a specimen is its construct. See Describing syntax in the book.
- Abstract syntax tree: Shows the syntax structure with all its specimens, but obviously without any delimiters, A tree has nodes, each one of the following kind:
  - Root: Node with no incoming branch.
  - Leaf: Node without outgoing branches
  - Internal node: Neither of the former
- Basic elements of a program text:
  - Terminals
    - Identifiers: Names chosen by the programmer
    - Keywords
    - Special symbols, such as a period
- Describing a program

- Semantic rules: Define the effect of programming, satisfying the syntax rules
- Syntax rules: Define how to make up specimens out of tokens satisfying the lexical rules
- Lexical rules: Define how to make up tokens out of characters

- Syntax: The way you write a program; characters grouped into words, grouped into bigger structures.
- Semantics: The effect you expect from this program at runtime
- Identifier: Name chosen by the programmer to represent certain program elements, such as classes, features or runtime values. If it denotes a runtime value, it is called an identity or variable if it can change its value. During execution, an entity may become attached to an object.
- Creating an object consists in:
  - If a class has no `create` clause, use basic form: `create x`
  - If the class has a `create` clause listing one or more procedures, use `create x.make()`, where `make()` is one of the creation procedures, and `(...)` stands for the arguments

1.2 Variables

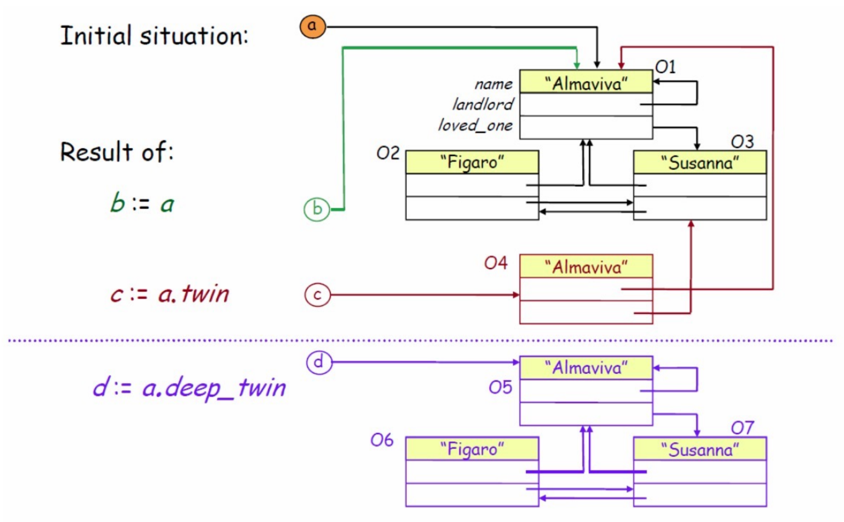
- Types
  - Reference types:
    - Entities with a reference value (contains the address, which can be the reference or location of the object).
    - They don't exist when we declare them (they are initially `void`), like so: `s: STATION`
    - We need to explicitly create them with a create instruction (`create s`)
  - Expanded types:
    - Entities with an object as a value (points directly to the object)
    - They exist by just declaring them (they are never `void`)
    - Entities of expanded types are compared by value!
    - To declare an expanded type:

      ```
      expanded class COUPLE
      feature --Access
              man, woman: HUMAN
              years_together: INTEGER
      end
      ```

      All entities of type `COUPLE` will automatically become expanded, i.e. `pitt_and_jolie: COUPLE`
    - Default values:
      - `false` for `BOOLEAN`

- · 0 for numeric types (INTEGER, NATURAL, REAL)
  - · "null" for CHARACTER
- – A type is one of:
  - ∗ A non-generic class
  - ∗ A generic derivation, i.e. the name of a class followed by a list of types, the actual generic parameters, in brackets

- Setters: It is possible to make assignments such as `x.att:=val`, which is shorthand for `x.set_att(val)`
- Effect of an assignment
  - Reference types: Reference assignment
  - Expanded types: Value copy
- Variable copy
  - Shallow object duplication (creates a new object): `b:=a.twin`
  - Deep object duplication (creates a new object): `b:=a.deep_twin`
  - Shallow field-by-field copy (does not create an object): `b.copy(a)`



1.3 Interface

- A client of a software mechanism is a system of any kind - such as a software element or a human - that uses it. For its client, the mechanism is a supplier
- The client is interested in a set of services that a software module provides, not its internal representation (what not how)
- Interface: The description of techniques enabling clients to use these mechanisms. For example: GUIs (Graphical User Interface), command line interfaces (shell, bash,. . .), or APIs
- An object can be an instance of a class, if the class is the generating class of the object

1.4 Information Hiding

- For its clients, an attribute may be:
  - Secret
  - Read-only
  - Read, but partially write restricted (only certain things are allowed to be written)
  - Writing one or more classes in curly brackets after the keyword `feature` exports these features only to these classes and its descendants. If no class is listed, the features are exported to `ANY`.

  Information hiding only applies to use by clients using dot or infix notation. Unqualified calls are not subject to information hiding.

1.5 Control structures

- Sequence or compound: consists of instructions listed in a certain order. Its execution consists of executing these instructions in the same order

- Loop: consists of a sequence of instructions to be executed repeatedly.

```
from
        --initialization Instructions
invariant
        --Loop invariant: Must hold true, satisfied after
            initialization, after the from clause
until
        --Exit condition; the condition under which the loop stops
loop
        --Instructions to be executed in each of the successive
            iterations
variant
        --Non-negative, decreases after every iteration
end
```

  - Loop invariant
    * Satisfied **after** initialization, after the *from* clause
    * Preserved by every loop iteration executed with the exit condition not satisfied. So in the end, the loop invariant <u>and</u> the exit condition hold!
  - Loop variant
    * Non-negative (i.e. ≥ 0) integer expression, right after initialization
    * Decreases while remaining non-negative for every iteration of the body with exit condition not satisfied.
    * A loop with a correct variant cannot be infinite

- Conditional: consists of a condition and two sequences of instructions. Its execution consists of executing one or the other of these sequences depending on whether the condition evaluates to `true` or `false`.

```
if condition then
        -- Do something
elseif condition_2 then
        -- Do something different
else
        -- Do something else
end
```

1.6 Contracts

- Contracts are made of assertions, each containing an assertion tag and a condition (a Boolean expression), i.e. `not_too_small: i>=1`
- Precondition
  - Property that a feature imposes on every client
  - If there is no `require` clause, the precondition always evaluates to `true`, as if it was written like this:

    ```
    require
            always_OK: true
    ```

- Postcondition
  - Property that a feature guarantees on termination to every client
  - A feature with no `ensure` clause always satisfies its postcondition, as if it was written like this:

    ```
    ensure
            always_OK: true
    ```

  - Can make use of keyword `old` (checks the result against the previous value of it)
- Class invariant
  - The invariant expresses consistency requirements between queries of a class

1.7 Miscellaneous

- Semistrict operators
  - Lets us define the order of expression evaluation:
    * **and then** is the semistrict version of **and**. Use it if a condition only makes sense when another is true.
    * **or else** is the semistrict version of **or**. Use it if a condition only makes sense when another is false
    * **implies** is always semistrict!

2 Describing syntax

2.1 BNF

Backus-Naur-Form (BNF): A metasyntax used to express context-free grammars. A formal way to describe formal languages. It consists of the following parts

- **Delimiters**: Fixed tokens of the languages vocabulary, such as keywords and special symbols
- **Constructs**: They represent structures of the language, for instance *Conditional*. A particular instance of a construct or known as a specimen of the construct. There are two kinds of constructs:
  - **Nonterminal construct**: They are defined by a production
  - **Terminal construct**: Terminal constructs such as *Identifier* or *Integer* are not defined by this grammar, they are described at the lexical level
- **Productions**: They are associated with a particular construct and specify their specimens

Each production defines the syntax of specimens of a particular construct, in terms of other constructs and delimiters. An example for a production (not BNF-E)

$$A = B|C[D]\{E\text{“;}''\}^*$$

Depending on the right side of a production, they can be separated into three kinds:

- **Concatenation**: This production lists zero or more constructs, some may enclosed in brackets and said to be **optional**
- **Choice**: Listing one or more constructs, separated by vertical bars. A choice specifies that every specimen of the construct on the left consists of exactly one specimen of one of the constructs on the right
- **Repetition**: A construct, enclosed in curly brackets, followed by a star. This indicates zero or more occurrences of the construct, Example: $A = \{B\}^*$. The star might be replaced by a plus, indicating one or more repetition

2.2 BNF-E

- Every non-terminal must appear on the left side of exactly one production, called its defining production.
- Every production must be of one kind: either concatenation, choice or repetition
- There is also a major change in the repetition production. Instead of

$$A = [B\{\text{terminal } B\}^*]$$

one may write

$$A = \{B \text{ terminal } \dots\}^*$$

The same is also true for the plus instead of a star.

2.3 Regular Grammar

The regular grammar is generally used to describe the terminal construct, which could be done using BNF, but can be achieved more easily using a regular grammar. The rules are quite similar, although different:

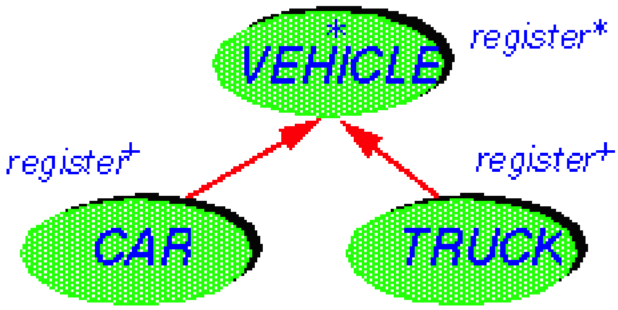
- The use of **choice** is no problem, possibly with character intervals
- There are also **concatenations**, although they do not assume breaks (spaces, new lines, ...) between elements. But you may define them explicitly using a lexical construct.
- **Repetitions** have a different, simpler form; A^* or A^+, following the same rules
- No **recursion** is allowed whatsoever. As a result you may write any language in a single regular expression
- Unlike BNF-E, you may mix different kinds of productions

3 Inheritance and Genericity

3.1 Inheritance

- Principle: Describe a new class as extension or specialization of an existing class (or several with multiple inheritance)
- If `B` inherits from `A`:
  - As modules: All the services of `A` are available in `B` (possibly with a different implementation)
  - As types: Whenever an instance of `A` is required, an instance of `B` will be acceptable
- Deferred classes:
  - Deferred classes can have deferred features
  - A class with at least one deferred feature must be declared as deferred
  - A deferred feature does not have an implementation yet
  - Deferred clauses cannot be instantiated and hence cannot contain a create clause
- Effective classes:
  - Effective classes do not have deferred features
  - Effective routines have an implementation of their feature body
- Terminology

- A class is a **parent** to another, if the other inherits directly from it, i.e. class `A` is a parent to class `B`, if class `B` inherits from class `A`
- The **descendants** of a class are the class itself and (recursively) the descendants of its heirs (parents). **Proper descendant** excludes the class itself.



- **Features of classes**
  - They can be **inherited** if it is a feature of one of the parents of the class. They can also be **immediate** if it is declared in the class. In this case, the class is said to introduce the feature
  - Fully implemented features are called **effective**, otherwise one may call them **deferred**
- Contracts
  - The **invariant** of a class automatically includes the invariant clause from all its parents, and-ed
  - If no pre-/post condition is explicitly stated, features inherit the contracts from their parents
  - One can weaken the precondition with the keyword `require else`, resulting in a precondition `orig_pre` and `new_pre`
  - One can strengthen the postcondition with the keyword `ensure then`, resulting in a postcondition `orig_post` and `new_post`

3.1.1 Multiple Inheritance

- **Name clash**: If `C` inherits both from `A` and `B`, which both have a feature `f`, then we have a name clash. To resolve it, we can redefine one feature as `A_f`. A name clash must be resolved, unless:
  - It is under repeated inheritance (the feature of `f` in `A` and `B` comes from a common ancestor, for instance `ANY`)
  - If at most one of the features `f` is effective, and all others are deferred. In that case (only one feature is effective), the features are said to be **merged**. Merging also works when one or more features are renamed. The merging happens after renaming!

- If more than one feature are effective, merging can still help. We can **undefine** effective features, so that they are deferred again. Syntax: `undefine a,b,c end`. It is even possible to first rename a feature, undefine it and then merge it.
- They all have the same signature

- Repeated Inheritance
  - Features, not renamed along any of the inheritance paths, will be shared
  - Features, inherited under different names will be replicated
  - A potential ambiguity arises because of polymorphism and dynamic binding when class `C` inherits from `B` and `A`, but `A` redefines now copy (a feature of the common ancestor `ANY`). In this case, a simple rename will not be enough, we have to select (`select copy, f end`) the features from one parent, and rename the ones from the other.

3.2 Genericity

- Terminology:
  - A **formal generic parameter** is the parameter in the class, e.g. `LIST[G]` with `G` as formal generic parameter.
  - An **actual generic parameter** is the actual type passed as parameter in a type, e.g. `LIST[INTEGER]` with `INTEGER` being the actual generic parameter.
  - One can obtain a **generic derivation** of a generic class by passing a type
- Static vs. Dynamic types:
  - Static types are the types that we use while writing, to declare types for entities (arguments, locals, return values)
  - Dynamic types are created at run-time. Whenever an object is created, it gets assigned to be of some type.
- Types
  - **Unconstrained** genericity: Any generic type is allowed. Example: `LIST[G]`, which is the same as `LIST[G->ANY]`
  - **Constrained** genericity: Only descendants are allowed as generic type. Example: `LIST[G->NUMERIC]`

3.3 Static Typing

- **Type-safe call** (during execution). A feature call `x.f` such that the object attached to `x` has a feature corresponding to `f`
- **Static type checker** is a program-processing tool (such as a compiler) that guarantees for any program that it accepts, that any call in any execution will be type-safe

- A programming language is called **statically typed language** if it is possible to write a static type checker

3.4 Polymorphism

- **Polymorphism** is the existence of the following possibilities:
  - An **attachment** (assignment or argument passing) is **polymorphic** if its target variable and source expression have different types.
  - An entity or expression is polymorphic if it may at runtime, as a result of polymorphic attachments, become attached to objects of different types.
  - A **container data structure** is polymorphic if it may contain references to objects of different types.
- The **static type** of an entity is the type used in its declaration in the class text. Similarly, the **dynamic type** of an entity is the type of the object, it is attached to. The type system ensures that the dynamic type of an entity will always conform to its static type.
- **Conformance**
  - A reference type `U` **conforms** to a reference type `T` if either:
    * They have no generic parameters, and `U` is a descendant of `T`
    * They both are generic derivations with the same number of actual generic parameters, the base class of `U` is a descendant of the base class of `T`, and every actual parameter of `U` (recursively) conforms to the corresponding actual parameter of `T`
  - An expanded type only conforms to itself
- **Object test**. Test the dynamic type of an object, e.g. `if {r:TYPE} obj then ... end`
  - `{r:TYPE}` is the object-test local, and only available in the `then`-part, not in the `else`-clause.
  - `obj` is the object to be tested
- **Assignment attempt**. Earlier mechanism for the object test. `a?=b` assigns `b` to `a` if and only if `b` is attached to an object whose type conforms to the type of `a`. Otherwise, `a` will be void

3.5 Dynamic Binding

- **Dynamic binding** as a semantic rule is the property that any execution of a feature call will use the version of the feature best adapted to the type of the target object.

4 Recursion

- **Definition**: A definition is recursive if it involves one or more instances of the concept itself. Recursion is the use of a recursive definition.
- Recursion can be either direct (routine `r` calls `r`) or indirect (routine `r_1` calls `r_2` .. calls `r_n` calls `r_1`)
- To be useful, a recursive definition should ensure that:

  R1: There is at least one non-recursive branch

  R2: Every recursive branch occurs in a context that differs from the original

  R3: For every recursive branch, the change of context R2 brings it closer to at least one of the non-recursive cases R1
- Recursive calls cause (without further optimization) a run-time penalty: the stack of preserved calls needs to be maintained. Various optimizations are possible:
  - Recursive schemes can sometimes be replaced by a loop (**recursive elimination**)
  - **Tail recursion** (last instruction of a routine is a recursive call) can usually be eliminated
- **Recursive variant**: Every recursive routine should use a recursion variant, an integer quantity associated with any call, such that
  - The variant is always ≥ 0
  - If a routine execution starts with variant value v, the value v' for any call satisfies $0 \leq v' \leq v$

5 Data Structures

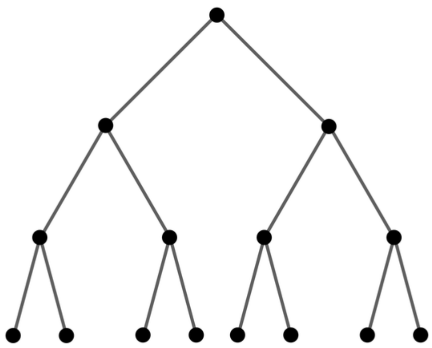
5.1 Trees

5.1.1 Binary Trees

- A binary tree `G`, for an arbitrary data type `G`, is a finite set of items called nodes, each containing a value of type `G`, such that the nodes, if any, are divided into three disjoint parts:
  - A single node, called the root of the binary tree
  - (Recursively) two binary trees over `G`, called the left and right subtree
- **Theorem:** For any node of a binary tree, there is a single downward path connecting the root to the node through successive applications of `left` and `right` links.
- **Traversals**
  - **In-order**: traverse left sub-tree, visit root, traverse right sub-tree

- **Pre-order**: visit root, traverse left sub-tree, traverse right sub-tree
- **Post-order**: traverse left sub-tree, traverse right sub-tree, visit root

- **Binary search tree**: This is a tree over a sorted set `G` if for every node `n`:
  - For every node `x` of the left sub-tree of `n`: `x.item` $\leq$ `n.item`
  - For every node `x` of the right sub-tree of `n`: `x.item` $\geq$ `n.item`
- In a binary search tree, average behavior for insertion, deletion and search is $O(\log(n))$, only worst case is $O(n)$



5.2 Container data structures

- Containers contain other objects, and store them in numerous ways. They differ in various properties, like the available operations, the speed of these operations and storage requirements. Some fundamental operations on a container:

Insertion	Add an item
Removal	Remove an occurence (if any) of an item
Wipeout	Remove all occurences of an item
Search	Find out if a given item is present
Iteration (or trasversal)	Apply a given operation to every item

- The `EiffelBase` classes use standard names for basic operations:
  - Queries:
    * `is_empty: BOOLEAN`
    * `has(v:G): BOOLEAN`
    * `count: INTEGER`
    * `item: G`
  - Commands
    * `make`
    * `put(v:G)`
    * `remove(v:G)`

    * `wipe_out`
    * `start, finish`
    * `forth, back`

- The **cursor** is present in many containers. It ranges from 0 to $count + 1$, and `before` and `after` hold if the cursor is not on an item. In an empty list, the cursor is at position 0

- **Alias notation**. A feature may be declared as follows: `item(i: INTEGER)alias "[]": G assign put`. It is then possible to do `a[i]` for `a.item(i)` and `a.item(i):= x` or `a[i] := x` for `a.put(x,i)`

5.2.1 Lists

- A list is a sequence of elements of a certain type. List is a general concept and has various implementations, including `LINKED_LIST`, `TWO_WAY_LIST`, `ARRAYED_LIST`, etc.

- Lists have a **cursor**. The current cursor position can be obtained by the query `index`. The element at this position is generally obtained by `item` and there are many other queries about the cursor position: `after`, `before`, `off`, `is_first`, `is_last`

- There are also various commands for **cursor movement**: `start`, `finish`, `forth`, `back`, `go_i_th`

- **Adding and removing** elements is done using: `put_front`, `put_left`, `put_right`, `extend`, `remove` (item at cursor position)

5.2.2 Arrays

- An array is characterized by:
    - Constant time for random reads/writes
    - Costly to resize (including inserting elements in the middle of the array)
    - Must be indexed by an integer
    - Generally very space efficient

- Even though arrays are of fixed size with a capacity of `upper-lower+1`, they can be resized. Instead of `put` you may use `force`, which has no precondition and resizes the array when needed.

5.2.3 Hash Tables

- Both arrays and hash tables are indexed based structures: Item manipulation requires an index, or, in case of hash tables, a key.

- Unlike arrays, table allow keys other than integers: `HASH_TABLE[ITEM_TYPE, KEY_TYPE->HASHABLE]`

- Hash tables depend on hash functions, which map K, the set of possible keys, into an integer interval $a \dots b$. A perfect hash function gives a different integer value for every element of K. Whenever two different keys give the same hash value, a collision occurs.

- In reality hash functions are never perfect, so we need to deal with collisions. The following approaches deal with the problem:

  - **Open hashing**: The data structure for this strategy is `ARRAY[LINKED_LIST[G]]`. In each entry of the array, for a certain index l, you find the list of objects whose keys hash to i. The costs of search are $O(1)$ to find the correct array index by hashing the key and $O(c)$ to find the item in the linked list. c is therefore a collision factor. With constant capacity of our array, the costs will become $O(\text{count}/\text{capacity})$, so this approach is somehow limited

  - **Closed hashing** (used by `HASH_TABLE`): Here we have a single `ARRAY[G]` where at any time some of its positions are occupied and some free. If for an insertion the hash function yields an already occupied position, the mechanism will try a succession of other positions until it finds a free one.
    A common technique, of the hash function yields a first candidate position `i=f(key)mod capacity`, is to try successive positions `i+k*increment`, where increment is `f(key) mod (capacity+1)`.
    These are remarkably good results, since search with a good hash function becomes essentially $O(1)$

5.2.4 Tuples

- In mathematics, computer science, linguistics, and philosophy a tuple is an ordered list of elements. In set theory, an (ordered) $n-$tuple is a sequence (or ordered list) of elements, where n is a non-negative integer

- Tuples can be declared with

  - Labeled arguments: `TUPLE[INTEGER, STRING, PERSON]`
  - Unlabeled arguments: `tup: TUPLE[number: INTEGER, street: STRING, resident: PERSON]`

- Is used like so: `tup := [99, "Weg", god_himself]`

- One may think of tuples as of classes with only attributes. They therefore are also more interesting as a language mechanism to describe simple structures in a clear and simple way, rather than they are in the sense of a data structure.

- Tags (like `number` in the above example) are optional, they do not affect the type of a Tuple.

- A Tuple with no arguments can hold any tuple, one with only one argument may hold any tuples with at least one element with the first of same type, etc.

- **Conformance**: You may assign an expression of type `TUPLE[A,B,C]` to a variable of same type, or one of the following: `TUPLE[A,B]` or `TUPLE[A]` or `TUPLE`

5.2.5 Dispensers

- These structures use no key or other identifying information for items, you insert an item just by itself. You also cannot choose what element you get by the query `item`, Note: `item` follows the command/query separation and therefore does not remove the item returned (you would need to make your own function, say `get`). The policy used to determine which item to return, may be one of the following:
  - **LIFO** (**L**ast **I**n, **F**irst **O**ut): Choose the element inserted most recently. Such dispenser is called a **stack**, and the stack operations are known as:
    - **push**: Pushes an item to the top of the stack (command `put`)
    - **pop**: Pops the top item (command `remove`). The body of a stack is what remains after popping
  - **FIFO** (**F**irst **In**, **F**irst **O**ut): Choose the oldest element not yet removed. Such dispenser is called **queue**.
    - Queues do also have many applications, such as simulations (represent pending events) and GUI (handling user inputs)
    - As with stacks you may use arrayed or linked implementations. The linked one is straightforward, where the arrayed needs some special treatment. Because we always add new items at the end of the array and remove old ones from the beginning, at some point we will reach the end. Therefore, `ARRAYED_QUEUE` is conceptually a ring.
  - With a **priority queue**, items have an associated priority.

6 Agents

- An object whose sole role is to describe an operation, for example if you want to treat an operation as an object
- Every agent has an associated routine, which the agent wraps and is able to invoke.
- To get an agent, use the `agent` keyword, e.g. `a_agent := agent my_routine`
- Applications include:
  - Iteration
  - Undoing (redo-undo)
  - Event-driven programming
- The type of an agent is one of the following:
  - `PROCEDURE [BASE_TYPE, OPEN_ARGS -> TUPLE]`

- FUNCTION [BASE_TYPE, OPEN_ARGS -> TUPLE, RESULT_TYPE]
- PREDICATE [BASE_TYPE, OPEN_ARGS -> TUPLE]

Note: All these types inherit from the deferred class `ROUTINE`, and `PREDICATE` is furthermore a descendant of `FUNCTION`

- The important features of agents include `call([open_args])`, and `last_result`
- **Open/closed arguments**:
  - An agent can have both "closed" and "open" arguments:
    * **Closed arguments** are set at agent definition time
    * **Open arguments** are set at agent call time
  - To keep an argument open, replace it by a question mark, i.e. `v := agent a0.f(a1,?,a3)`
  - All closed argument agent: `u := agent a0.f(a1,a2,a3)`
  - If you omit the argument list, all arguments of the underlying function are considered open.
- **Open target**: It is also possible to keep the target of an agent open, but then you need to specify its type. The target is then passed as first item in the argument tuple to the agent.
- Here are some examples of agents:

– `u := agent TARGET_TYPE.f(?, ?, z)`	`u.call([target,a,b])`
– `u := agent a0.f(x, ?)`	`u.call([a])`
– `u := agent a0.f`	`u.call([])`
– `u := agent TARGET_TYPE.f`	`u.call([target])`

- Some example types:
  - `PROCEDURE[ANY, TUPLE]` (no open arguments)
  - `PROCEDURE[ANY, TUPLE[X,Y,Z]]` (3 open args, with types `X, Y, Z`)
  - `FUNCTION[ANY, TUPLE[X,Y],RES]` (2 open args, result of type `RES`)

7 Event Driven Programming

7.1 Terminology

- An **event** is a run-time operation, executed by a software element to make some information (including that it occurred) available for potential use by the software elements not specified by the operation. The information associated with an event (other than it actually occurred) constitutes the event's **arguments**
- To **trigger** (or **publish**) an event is to execute it
  - A software element that may trigger events is a **publisher**
  - A software element that may use the event's information is a **subscriber**

- Any event belongs to an **event type**, and therefore shares the same argument list **signature**
- Note: Event type/event may suggest corresponding to a type of OO programming, but in our model, an event is not an object! There is the `EVENT_TYPE` in Eiffel, which denotes the **general idea** event types, and a particular event type is then an instance of this class.

7.2 The event-driven scheme

- Some elements, publishers, make known to the rest of the system what event types they may trigger
- Some elements, subscribers, are interested in handling events of certain event types. They register the corresponding actions.
- At any time, a publisher can trigger an event. This will cause execution of actions registered by subscribers for the event's type. These actions can use the event's arguments.
- In event-driven design, a **context** is a Boolean expression specified by a subscriber at registration time, but evaluated at triggering time, such that the registered action will only be executed if the evaluation yields true.

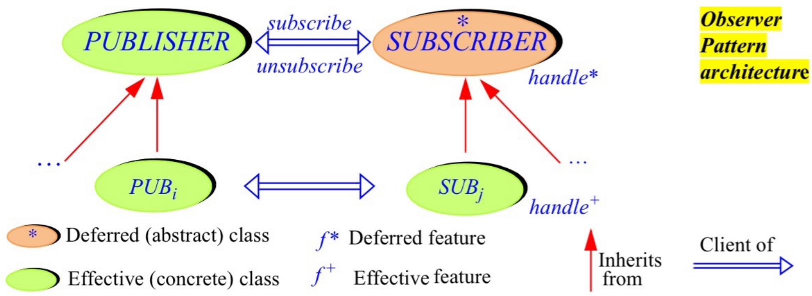
7.3 Publish-Subscribe program

In devising a software architecture supporting the publish-subscribe paradigm, we should consider the following requirements:

- Publishers must not need to know who the subscribers are
- Any event triggered by one publisher may be consumed by several subscribers
- The subscribers should not need to know about the publishers. **This goal cannot be provided by the observer pattern**
- The subscribers can register and deregister while the application is running
- It should be possible to make events dependent or not on a context
- It should be possible to connect publishers and subscribers with minimal work

7.3.1 Observer Pattern

- Note that this pattern is limited, and therefore should not be used. Regardless the existence of superior methods, the observer pattern is still used very frequently.



- **Publisher**: Describes the properties of a typical publisher in charge of an event type. It can trigger events through `publish`, and subscribers can subscribe and unsubscribe
- **Subscriber**: on the other hand do also have subscribe and unsubscribe to (un-) subscribe to a particular publisher. The feature calls (un-)subscribe of the desired publisher, passing itself as argument. Furthermore there is the deferred feature handle which expects a `LIST` as argument. This is not ideal, since there is no type safety with the `LIST` as argument, but since the very general classes `PUBLISHER` and `SUBSCRIBER` need to know about the arguments, there is not really a better way. We could only specify the two classes further (e.g. one for each argument list), but therefore loosing generality. (Using genericity with tuples would work, but who does have them other than Eiffel? And Eiffel can do better..)
- **Drawbacks** of the observer pattern:
  - The argument business, see above
  - Subscribers subscribe directly to publishers rather than event types
  - A subscriber may register with only one publisher, with that publisher, it can register only one action
  - It is not possible to directly reuse an existing routine. The classes need to inherit from `PUBLISHER`/`SUBSCRIBER` resulting in too much glue code
  - The last problem gets even worse without multiple inheritance.

7.3.2 Event Type Library (A Much Better Approach)

- There is only a single class `EVENT_TYPE[ARGS -> TUPLE]` with a generic parameter, ensuring type safety.
- One can create event types, for instance:

```
left_click: EVENT_TYPE[TUPLE[x:REAL,y:REAL]]
-- Event type representing left-button click events
        once
                create Result
        end
```

- This declaration can take place in "facilities" class to which all others who need have access to (no context required). But it is also possible to put it in an ordinary class like `BUTTON`, then having something like `your_button.left_click.subscribe(agent p)` (context required)
- To trigger an event, one may write: `left_click.publish([your_x, your_y])`
- To subscribe to an event, one may write: `left_click.subscribe(agent p)`, requiring that `p` represents a function with proper signature.