Summary
Software Architecture

Tanja Werthmüller, Patrick Bänziger, Simon Stelling, Fabian Hahn

October 4, 2009

1 Principles

Decomposability
A modular design method must help decompose complex problems into subproblems.

Composability
A modular design method must support production of software elements that may be freely combined with each other for new software.

Direct Mapping
A modular design method must yield software with a structure in direct correspondence with the structure of the specification.

Few Interfaces Principle
Every module must communicate with as few others as possible.

Small Interfaces Principle
If two modules communicate, they must exchange as little information as possible.

Explicit Interfaces Principle
Whenever two modules communicate, this must be clear from the text of one or both of them.

Continuity
A modular design method must ensure that small changes in specification yield small changes in architecture.

Uniform Access Principle
It doesn't matter to the client whether you look up or compute.

Information Hiding Principle
The designer of every module must select a subset of its properties as the official information about the module, made available to authors of client modules.

The Open-Closed Principle
Modules should be open and closed. (Closed: Usable by clients / Open: may be extended)

The Single Choice Principle
Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.

Object technology definition
Object-oriented software construction is the construction of sofware systems as structured collections of (possibly partial) abstract data type implementations.

2 ADTs

In computing, an abstract data type (ADT) is a data type that is independent of any actual implementation. It is a specification of data and the operations that can be performed on the data.

e.g.

Types:
- STACK [G]   – G: Formal generic parameter

Functions:
- put: STACK [G] $\times$ G $\rightarrow$ STACK [G]
- remove: STACK [G] $\not\rightarrow$ STACK [G]
- item: STACK [G] $\not\rightarrow$ G
- empty: STACK [G] $\rightarrow$ BOOLEAN
- new: STACK [G]

Preconditions:
- remove (s: STACK [G]) require not empty (s)
- item (s: STACK [G]) require not empty (s)

Axioms:
$\forall$ x: G, $\forall$ s: STACK [G]
- item (put (s,x)) = x
- remove (put (s,x)) = s
- empty (new)
- not empty (put (s,x))

A partial function, identified by $\not\rightarrow$, is a function that may not be defined for all possible arguments in the specified input domain.

e.g. In mathematics the inverse function inverse(x) := 1/x is a partial function, because it is not defined for x = 0.

Three forms of functions in the specification of an ADT T:

Creators:	
OTHER $\rightarrow$ T	e.g. new
Queries:	
T $\times$ ... $\rightarrow$ OTHER	e.g. item, empty
Commands:	
T $\times$ ... $\rightarrow$ T	e.g. put, remove

Sufficiently Complete ADT specification: A specification for T is sufficiently complete if its axioms make it possible, for any expression of the form f(...) where f is a query:

1. To determine whether the expression is correct.
2. To reduce it to a form not involving T.

Correct ADT specification: An ADT expression f(a,b,c...) is correct if it is well-formed and:

1. Every one of a, b, c... is (recursively) correct.
2. Their values satisfy the precondition of f, if any.

Unfortunately sufficient completeness is undecidable.

3 Contracts

The design by contract principle is based on the following three central notions:

- **precondition**: predicate that has to hold before a feature is executed, responsibility of the client (feature caller)
- **postcondition**: predicate that has to hold after a feature is executed, responsibility of the supplier (feature implementor)
- **invariant**: predicate that has to hold after creation of an object and before and after any exported feature of the object is executed, responsibility of the supplier (class implementor)

Contracts get more complicated with inheritence:

- A inherited precondition can only be weakened. It is written as **require else**.
- A inherited postcondition can only be strengthened. It is written as **ensure then**.
- The inherited invariants of all parents (and the own invariant) are and'ed.

4 Design Patterns

A design pattern is a document that describes a general solution to a design problem that recurs in many applications so that developers can adapt the pattern to their specific applications. A pattern is not reusable. It has to be adapted to the specific problem every time it is used.

Creational:

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural:

- Adapter
- Bridge
- Composite/Decomposite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral:

- Chain of responsobility
- Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Template Method
- Visit

4.1 Observer

When an object changes state, all its dependents (observers) are notified. The observer then decides whether to ignore or handle the notification. Abstract coupling between Subject and Observer: The subject doesn't know the concrete class of any observer. Observers can be added and removed at any time.

4.2 Command

The command pattern is a design pattern in which an object is used to save all the information needed to call a method at a later time. It is particularly used to implement the undo/redo function. In this case all methodobjects that are executed are pushed on a stack. If a client wants to undo a command, the program pops the top element of the stack and executes its undofunction.

4.3 Visitor

The visitor pattern encapsulates functions for a given set of classes in objects. This is desirable if the set of classes to be visited is fixed but the set of functions that are applied to these objects should be extendible. The idea is that every function is implemented in a specific class, a VISITOR, for every object. For example the PRICE_VISITER has functions: *visit_house*, *visit_car*, *visit_boat*... The big advantage of this pattern is that the objects don't have to know what kind of VISITOR visits them, they just have to implement a function *accept(vis: VISITOR)*, which calls the appropriate *visit_* function of the passed visitor object. In Eiffel, this double dispatching mechanism just makes sure that the *visit_* functions can be exported only to the appropriate visited class.

In other languages that support function overloading, however, the mechanism is used to circumvent static dispatching of overloaded functions.
Note that adding a new VISITOR is straightforward, whereas adding a new object type would require adding a function *visit_new_object* to all VISITORs, which might be quite cumbersome.

E.g. We have a set of objects (HOUSE, CAR,...) that should be visitable.

```
Class Main
  make is
    do
      ...
      a_house.accept(a_price_visitor)
      if (a_price_visitor.price <= maxprice) then
        a_house.accept(a_buy_visitor)
      end
    end
end

deferred Class VISITOR
  visit_house (house: HOUSE) is
    deferred  end

  visit_car (car: CAR) is
    deferred  end
end

Class PRICE_VISITOR
inherit VISITOR
  price: INTEGER

  visit_house (house: HOUSE) is
    do
      --calculate houseprice
      price:= houseprice
    end

  visit_car (car: CAR) is
    do
      --calculate carprice
      price:= carprice
    end
  ...
end

Class BUY_VISITOR
inherit VISITOR
  owner: STRING

  visit_house (house: HOUSE) is
    do
      --find new_owner
      owner:= new_owner
    end

  visit_car (car: CAR) is
    do
      --find new_owner
      owner:= new_owner
    end
  ...
end

...

Class HOUSE
  accept(vis: VISITOR) is
    do
      vis.visit_house(Current)
    end
end

Class CAR
  accept(vis: VISITOR) is
    do
      vis.visit_car(Current)
    end
end
```

4.4 Strategy

The Strategy pattern allows to **select an algorithm at runtime** of a family of interchangable algorithms. The **context** saves the current strategy object and allows to change it.
Negative: Clients must know and instantiate the strategies.

4.5 Chain of responsibility

Give more than one object a chance to handle a request, by avoid coupling the sender to a receiver. Like exception handling in Java. A receiver of a request can decide to handle the request or to pass it up the chain. The pattern does not guarantee that the request is handled. Use when: You want to send a request without knowing who handles it (reduced coupling). The set objects that can handle the request can be specified at runtime. Related patterns: Often applied in conjunction with Composite.

4.6 State

The state pattern is for example used to represent an automaton. Each is represented as a class all inheriting from a common class STATE and the automaton only knows its current state. Each state has a function *next*, which takes arguments susch as the transition variables to decide what state will be next. The automaton has a function *set_state*. To perform a state transition, the automaton calls the *next* function of its current state with some transition variables, which calls the *set_state* function of the automaton with its successor as argument. Instead of an automaton we can also just use some kind of object, which performs differently depending on its current state.

4.7 Abstract Factory

The abstract factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. Usually, this is achieved by having an abstract (deferred) class which defines features that return instances of another abstract product class to be created. A concrete implementation of the abstract factory would now implement the effective features to create and return concrete implementations of the abstract product. As a result, a client could use the abstract product's interface without knowing which concrete product he has got back from the factory.

4.8 Factory Method

The factory method pattern allows to abstract the creation of an object without knowing the type of the object at compile-time. The **AbstractProduct** is created by **AbstractCreator**. The effective class **Product** is the created by an effective **ConcreteCreator**. (Which can be able to create multiple object types, depending on the type given by the customer).

4.9 Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations. E.g. If you pass a description of a house to a drawer, he will draw a picture of the house. If you pass it to a constracter, he will construct the real house. Construction or drawing or whatever builder you use, is done step-by-step.

4.10 Singleton

The singleton pattern is used to ensure a class only has a single instance and to provide a global access point to it. This is usually achieved by declaring the constructor of the class as private and using a static method *getInstance()* that will always return the same instanciated object. Since the Eiffel language does not support static methods, a seperate singleton creater class can be used to replace the static method by a feature with a constructor call that is only executed a single time (using the *once* keyword).

E.g. If we want to make sure that we can have at most two different PLAYER in our systeme.

```
Class PLAYER
feature {NONE}
  instances: ARRAY[PLAYER_INSTANCE] is
    once
      create Result.make
    end
feature
  make (a_name: STRING) is
    do
      if (instances.count < 2) then
        create instance.make(a_name)
        instances.put(instance)
      else
        instance := instance[2]
      end
    end
  instance: PLAYER_INSTANCE
end

CLASS PLAYER_INSTANCE
feature {PLAYER}
  name: STRING
  make (a_name: STRING) is
    do
      name := a_name
    end
```

4.11 Bridge

The bridge pattern is used to decouple an abstraction from its implementation so that they can vary independently. It is an alternative to inheritence which isn't always flexible enough. For example if we want to have different implementations for different abstractions, there has to be a specific subclass for every combination. That's why we want to split implementation and abstraction. We have a superclass for all IMPLEMENTORs and one for all ABSTRACTIONs. Each instance of ABSTRACTION contains a reference to an object of type IMPLEMENTOR.

E.g. We want to be able to draw different shapes with different implementations. Our implementations are a DRAWER and a PRINTER, the shapes we want to draw are a RECTANGLE and a CIRCLE. Instead of having a RECTANGLE_DRAWER inheriting from DRAWER and RECTANGLE, a RECTANGLE_PRINTER inheriting from PRINTER and RECTANGLE, a CIRCLE_DRAWER... we use the bridge pattern to combine them freely in the MAIN class.

```
Class MAIN
  make is
    ...
    rectangle.imp := drawer
    rectangle.draw
    rectangle.imp := printer
    rectangle.draw
    circle.imp := drawer
    circle.draw
    circle.imp := printer
    circle.draw
    ...
  end
end

deferred Class SHAPE
  imp: SHAPE_IMP
    --implementation
  draw is
    deferred  end
end

Class RECTANGLE
inherit SHAPE
  x1,x2,y1,y2: INTEGER
  draw is
    do
      imp.draw_line (x1,y1,x1,y2)
      imp.draw_line (x1,y1,x2,y1)
      imp.draw_line (x1,y2,x2,y2)
      imp.draw_line (x2,y1,x2,y2)
    end
end

Class CIRCLE
inherit SHAPE
  x,r: INTEGER
  draw is
    do
      imp.draw_circle(x,r)
    end
```

```
deferred Class SHAPE_IMP
  draw_line (x1,y1,x2,y2: INTEGER) is
    deferred end

  draw_circle (x,r: INTEGER) is
    deferred  end
end

Class SHAPE_DRAWER
inherit SHAPE_IMP
  draw_line (x1,y1,x2,y2: INTEGER) is
    do
      ... --draw this line
    end

  draw_circle (x,r: INTEGER) is
    do
      ... --draw this circle
    end
end

Class SHAPE_PRINTER
inherit SHAPE_IMP
  draw_line (x1,y1,x2,y2: INTEGER) is
    do
      ... --print this line
    end

  draw_circle (x,r: INTEGER) is
    do
      ... --print this circle
    end
end
```

4.12 Composite

The composite pattern allows to represent hierarchical structures such as trees. It consists of **leaves** and **composites** (internal nodes). Both inherit a common interface from a **component**. The idea is that composites allow to treat a single leaf and a group of leaves uniformly.
Example: The 'get_price' method returns a price for a leaf and for a composite it returns the sum of all leaf prices.

4.13 Decorator

The decorator pattern is used to add some functionalities to a given object without having to change the whole class of it. It adds responsibilites to individual objects dynamically and transparently, that is, without affecting other objects. The decorator pattern resembles inheritance but on the level of objects instead of classes. It is also used, when subclassing is impractical, because structuring the whole problem into subclasses would produce a explosion of subclasses to support every combination.

E.g. We have a class TEXTBOX inheriting from some class VISUAL_COMPONENT. One of our textbox-objects now grows to big to fit on a page, so we want to add a scrollbar.

```
Class MAIN
  make is
    do
      textbox := create TEXTBOX.make
      textbox.write
      if (textbox.size > max_size) then
       textbox := create SCROLLER.make(textbox)
      end
      textbox.draw
    end
end

Class SCROLLER
inherit VISUAL_COMPONENT
  component: VISUAL_COMPONENT
  make (obj: VISUAL_COMPONENT) is
    do
      component := obj
    end
  draw is
    do
      draw_srollbar
      component.draw
    end
  ...
end
```

Here the TEXTBOX class would have to implement a draw function which would be given by the VISUAL_COMPONENT class.

4.14 Facade

The facade pattern is a very intuitive one: Its intent is to provide a unified interface for a complicated system. Take a compiler suite, containing a lexer, a parser, a machine code generator etc. as an example of a complicated system. Most clients probably would just like to access the whole compiler through a single compile (code) method. The facade just provides this simple interface, abstracting away the complexity of the other interfaces in the system.

4.15 Flyweight

This pattern uses sharing to support large numbers of fine-grained objects efficiently. The goal is to reduce the amount of created instances of a class and their memory consumption by using the same instance over and over again. This is achieved by a distinction of state in the objects: The intrinsic state denotes information that can be shared across multiple usages of an operation whereas the extrinsic state denotes information that is unique to the operation's context. As a result, there is only need to create a seperate object for every intrinsic state, which is usually handled by a factory class implementing a pool of shared flyweight objects. If a client now requires an instance of the flyweight class, it requests the shared flyweight object from the factory matching the needed intrinsic state and then uses the resulting object by passing the extrinsic state as parameters its methods.

5 Exception

An interrupt is an **abnormal event** in logical control flow-major causes are signals from the OS (interrupts, div by zero, out of mem...), contract violation, programmer triggered or void calls (not in EIFFEL).

There are two acceptable ways for a recipient of an exception:

Failure is an 'organized panic' - it forwards the exception upwards to its caller (ultimatively the runtime system).

Retry Exactly that: Try again, maybe different strategy.

The basic idea of error handling in Eiffel is:

- A routine may have a **rescue** clause.
- A rescue clause may contain a retry instruction.
- A rescue clause which does not retry results in a **failure**.
- A rescue clause **must restore the invariant**.

6 UML

The *Unified Modelling Language* allows to represent models (mostly of software) as diagrams. There are a variety of different diagrams. The most important entities and are pictured below:

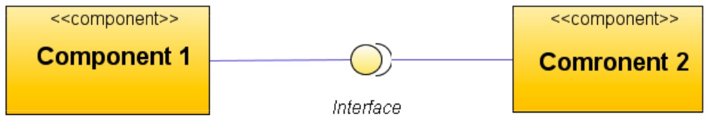

Figure 1: Structural Entities (uniform since UML 2) with interface

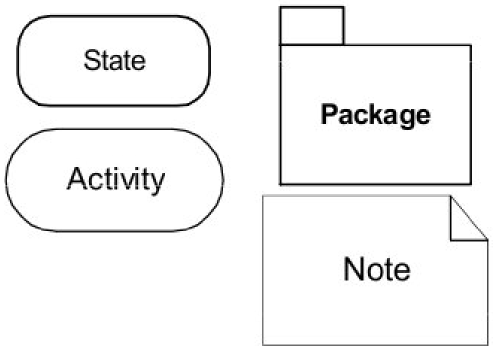

Figure 2: Entities - except Structural

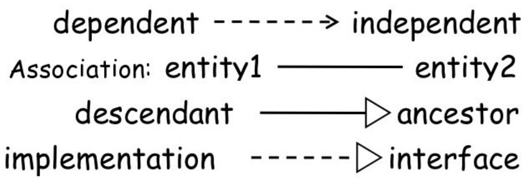

Figure 3: Relations

7 Software Architecture styles

The aim is to classify styles of software architecture. An architectual style is defined by:

- Type of basic architectural components
  (e.g. classes, filters, databases, layers)
- Type of connectors
  (e.g. calls, pipes, inheritance, event broadcast)