7. Coherent Descriptions

the idea is to discuss in this section

descriptions of (software) systems

that actually improve the insight into
the coherency
between all (software) system parts

in so doing, we mostly concentrate on
software systems

but once in a while we take into account
the (business) environment too

in view of integration-orientation,
discrimination between software and business
is quite irrelevant, however

we actually cover 2 topics
arcitecture: components and connectors
patterns: classes/objects in collaboration

note: both topics are relevant across systems

the notion of architecture
illustrates the relevance of the SE principle
abstraction
as an architectural description really aims
at being global, giving essence,
omitting everything else

the notion of pattern
illustrates the relevance of the SE principle
generalisation
as a pattern description really aims
at catching reoccurring essence,
by extracting / combining essential ingredients,
and putting them in place where needed

architecture, software architecture mainly:
components and connectors

architecture has to do with mastering a system’s

(large) size and (high) complexity

architecture does this by

- globally structuring the system into
a manageable number of parts
- globally gluing, connecting these parts

in the context of architecture,
parts are called components or elements
: composition
glue is called a connector or a relation
: connectivity

choosing suitable components
as well as suitable connectors has to do with
SEprinciples of
abstraction mainly (already mentioned),
furthermore grouping and viewing

in addition, components have “interfaces”
regulating their mutual visibility
this is viewing again: how a component
is to be viewed by another component
(interface as view provided / requested)

very important:

usually there is
not just one architecture of a system

but there are “many”,

each one geared to some aspect(s) or to some point(s) of view or the essence chosen

material is partly based on

the stimulating book

P.Clements, F.Bachmann, L.Bass, D.Garlan,
J.Ivers, R.Little, R.Nord, J.Stafford:

Documenting Software Architectures
Views and Beyond

Addison Wesley, 2003

ISBN 0-201-70372-6

with respect to software systems

one often discriminates between
3 types of architecture

- wrt what it does: main functionality

this type of architecture gives
structure of logical design corresponding to
the (classical) use case diagram

- wrt how / when it does it: main execution

this type of architecture gives
structure of runnable parts:
the classical components (plug-ins, COTS)

the third type of architecture
may seem somewhat ill-focused
as it covers two rather unrelated ways for
globally describing a software system

- wrt where it does it: allocation

this type of architecture gives
: the machine(s) where each part is running
or is stored
as well as

: the SE people responsible for each part

one might argue, responsibility is so different from physical presence
that they could be considered as two different types of architecture

architecture of the functionality

components are modules;
in UML: class/object, package, component

such a module is seen as
a bundling of functionality

a module in principle offers its functionality
not only to itself
but also to the other modules

via the connectivity it is specified
whether / how such functionality
can be used by the other modules

relevant relations between modules are:
- is-part-of
- is-dependent-on
- is-a

the aim of the functionality architecture is:

understanding the logical design

this is important for

- construction: modules serve as blueprint for the design as well as for the code

- analysis: in particular with respect to the functional requirements
  : traceability: from high level requirements
    to the detailed invocation sequences
  : impact analysis: on the basis of high level
    problem report or change request,
    insight in the detailed consequences
    (in terms of functionality)

- communication: conveying insight into the system's functionality to someone else
  module (de)composition supports both
    top-down and bottom-up presentation

functionality architecture can be presented
on the basis of various styles

recurring elements of a style: not unlike pattern

4 styles:

- decomposition style: ◇

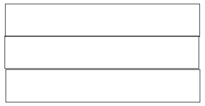
- uses style <<uses>> ⇢

- generalization style: △

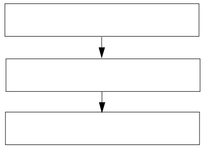
- layered style / tier style:

for the layered style there is
no specific UML notation

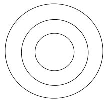
but one often sees diagrams like



or



or



only the 4th (layered) is referred to as a pattern

interesting example of layered style is
ArchiMate
being an architectural framework language for
business, software and hardware architecture

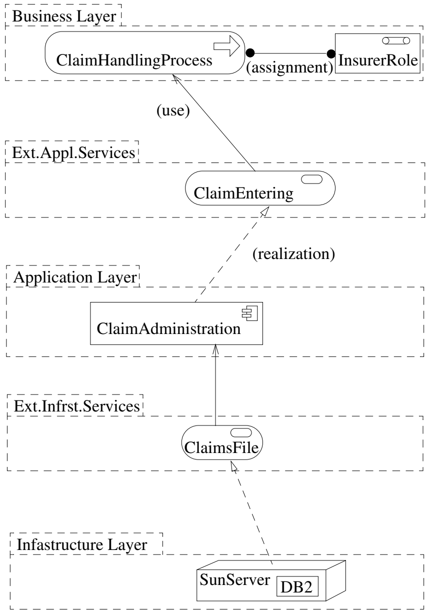
a layer in ArchiMate is a grouping
(package-like)

above 3 different architectures
- business, software and hardware -
are put into 3 different, hierarchical layers:

- business layer (top)
- application layer (middle)
- infrastructure layer (bottom)
  comprising real and virtual machines,
  and (lower level) system software

in between top-middle and middle-bottom
there are two additional layers
containing the services provided
by middle to top and by bottom to middle

characteristic structure of an ArchiMate model:



the ArchiMate layering can be extended with

Environment Layer (and ExtBusServices)

containing eg. Clients, other organisations
and their processes

so ArchiMate's layering

indeed has the 3 tiers:
Business, Application, Infrastructure

but (commonly) the layer structure
is bipartite

regulating the strictly hierarchical
use of / via the externally offered services
via separate layers in between

(back to general functional architectures)
what the styles are for

decomposition style:
- understanding, learning
- distributing development among a team

uses style:
- incremental development
- testing, debugging (of functionality mainly)

generalization style:
- extension, evolution
- local change, variation
- reuse

layered style:
based on information hiding, support for virtual machine, so
- modifyability
- portability

architecture of the system in execution

often referred to as components and connectors
C&C,
but substantially more restricted than
the components and connectors from composition and connectivity as in general architecture
(called elements and relations for this reason!)

a component is a type description of
a runtime entity

a connector is a type description of
a physical link between components at run time

interfaces are referred to as ports
via a port a component sends to / receives from
unknown elsewhere signals
(triggers, messages, data; no calls)

the signals are transmitted via a connector, linking ports of components the aim of the C&C architecture is:

understanding the execution of the system

this is important for runtime requirements like

performance, reliability, availability

leads to insight into

- (main) running components, their interaction
- shared data stores
- shared applications
- replication
- protocols
- sequentialization, true concurrency
- flow of data
- flow of control
- tuning of runtime configuration

insight is sometimes based on formal analysis,
more often based on experience, heuristics

for this type of architecture the styles are
very often referred to as architectural patterns

6 styles:

- pipe and filter

- shared data

- publish subscribe

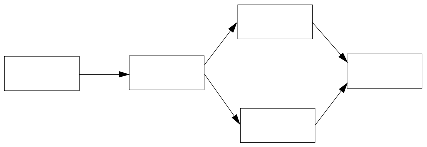
- client server

- peer to peer

- communicating processes

what the styles are for,
together with some common representation

pipe and filter:
- (subsequent) data transformation and their scheduling
- latency between input and eventual output
- buffer capacity and speed at pipes



filter: □

pipe: →

note: this is not a very UML-like notation

it is interesting to remark how ArchiMate

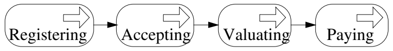
integrates features

from different architectural types:

1 (business/application /infrastructure) process

ClaimHandlingProcess

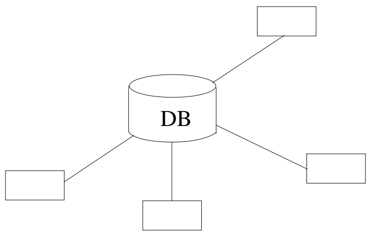
can be refined into smaller process steps:



trigger: →

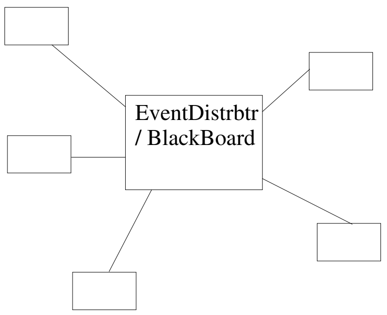
via different roles (or collaborations) assigned the execution (filtering) result is being pumped further
(choices are common, loops not common)

shared data:
- decoupling of data production and consumption (not necessarily destroying)
- bottle-neck analysis
- security, privacy, authorization
- coupling storage and access: mapping data and computation
- data persistence

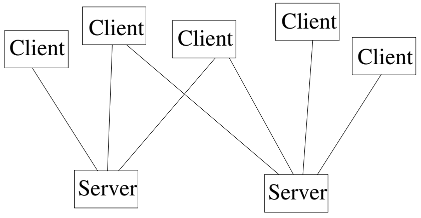


publish subscribe:
- decoupling sending and receiving: set of receivers is unknown
- modifyability of number of receivers,
even on the fly

blackboard architecture is even more specific:
also (number of) sender(s) is unknown



client server:
- decoupling applications from services used
- deploying of often used services on specific hardware
- interoperability
- integration with legacy systems
- scalability
- reliability
- quality of service: both functional and non functional requirements
- quality of service usage



again

ArchiMate integrates not only features from
different architectural types
but also features from
different architectural styles

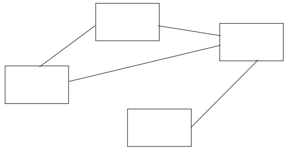
apart from the pipe and filter style
for its processes

it has the service layers to connect
business, application, infrastructure layers

so ArchiMate has the flavour of a SOA
(Service-Oriented Architecture)

peer to peer:
- collaboration, allowing for all kinds of roles
- flexibility in distributing the separate collaborations
- local sharing of data, resources within set of collaborating peers

eg. CORBA (Common Object Request Broker Architecture) is peer-to-peer



the connectors are of the type:

invoke procedure

in accordance to the interface specifications

communicating processes:
- true concurrency versus bundling of threads of control (interleaving)
- detailed performance and reliability issues
- protocol conformity

a visualization would be equal to
the peer to peer picture

but now the connectors can be of

any type of communication,

be it that most often for any connector the type of communication is fixed

note:
pre-emption / explicit interruption,
actually hidden in the other C&C styles,
can be addressed straightforwardly
as can every gradation of asynchronity

ArchiMate has also collaborations:

a grouping of roles

together being responsible for process (step)

it leaves unspecified which role does which part of the process (step)

neither is there communication between roles indicated:

only the collaborative result counts

(a typically managerial view)

such an ArchiMate collaboration then is
an underspecified peer-to-peer or even
communicating processes style - yet another!

architecture of the allocation

elements are
software units allocated to “physical unit”

the software units are
either (sets of) modules or (sets of) components

the physical units depend on the style:

deployment style:
piece of hardware: processor, storage, router

implementation style:
configuration item: file, directory

work assignment style:
human: person, team, subcontractor

note

3rd style in particular is rather divergent

but for this ArchiMate’s eclecticism
is quite clarifying:
(eclecticism: combining everything useful)

- roles are assigned to teams, people

- application(step)s are coupled to platforms
  (be it via services, e.g. provided
  by the right application server)

what the styles are for

deployment:

- performance: tuning by adapting
- reliability, security: keeping copies elsewhere, migration at runtime
- cost estimation: of deploying the system

implementation:

- configuration management, both during development and production
- version management and specification of differences
- highlighting, isolating an item for special purposes, eg. testing, refactoring

work assignment:

- team resource management: responsibility, skills, experience
- understanding project structure, internally and externally
- project planning: work break down, cost estimation, scheduling

note

different architectures can “coincide”:

modules or components can serve as unit for work assignment

eg hiding of internal details, as in modules or in components, can be similar to how team members are to integrate their software elements

recapitulating the above 3 architectures

main functionality covers
structuring **what** the system does

main execution covers
structuring **how** the system acts

allocation covers
where the system resides

beginning awareness of fourth architecture:

"impact"

again it makes sense to discriminate between

what in the organisation / environment
how in the organisation / environment
where in the organisation / environment

(also for this see ArchiMate)

so it seems:

not only modelling can be extended to
the domains of organisation / environment

but also the architectural views and styles
analogous to the what-how-where division

this reinforces ideas about
integration-orientation

on the other hand,

one can also start from architectural concepts in the org/env domain

and extend these to the software domain,

possibly via systematic translation to eg UML

(approach as in ArchiMate)

this could be a topic of study in

(process) integration, alignment

discussion about role of objects in architecture

some authors state:

OO is absolutely unfit for architectural specifications

as OO paradigm has been built on
calling a method of a certain object, and to that aim the object must be known at runtime at the moment of calling

this then is considered to be
fundamentally different from the C&C idea where signals are sent and received via ports, without knowing which component or object for that matter is out there

but:
the protocols and their local interpretations guarantee equivalent behavioural reaction
moreover, UML 2.0 has ports etc

the above type-style combinations
of an architecture
are examples of

often used or often recognized
global structures
of software systems as a whole

however, on a smaller scale too
ie. within models / software

one uses / recognizes again and again

particular structures with particular behaviour and communication
as eg. wrt. UML's collaborations

such often occurring structures are called

patterns

well-known book about patterns:

E.Gamma, R.Helm, R.Johnson, J.Vlissides:

Design Patterns
Elements of Reusable Object-Oriented Software

Addison Wesley, 1995

ISBN 0-201-63361-2

discusses 23 patterns in 3 categories:

5 creational patterns
7 structural patterns
11 behavioural patterns

remember:
often occurring --> generalization principle

some design pattern examples follow here
(very superficially only)

creational patterns:

Singleton:

restricts number of instances of a class to 1,
offers a global access point for it

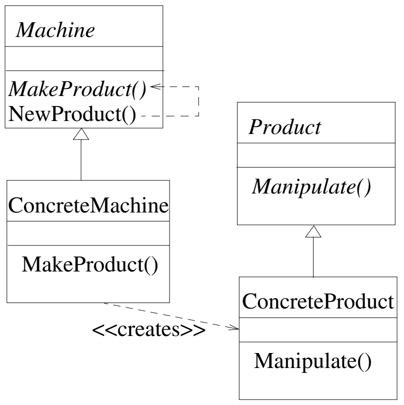
upon instantiating (construction) a specific counter is checked

same idea (pattern!) works for a different fixed maximum of instances

also allows for subclasses of the singleton class

Factory Method:

provides an interface on a general level for instantiating an object, by letting subclasses determine which class the object is an instance of; so it delegates a class instantiation to subclasses



structural patterns:

Adapter

changes an interface into another, such that the new form corresponds to what is expected elsewhere

AKA (also known as) Wrapper or Envelope

Often by renaming,
but also by rearranging functionality

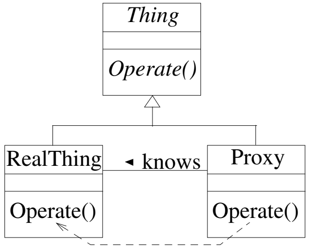
Decorator (AKA Wrapper !!)

Extending functionality of an object dynamically (eg instead of subclassing)

Combining Adapter and Decorator can result in a completely different functionality look-and-feel

Proxy

offers a substitute, placeholder for an object to hide the actual access



behavioural patterns:

Observer

AKA Publish-Subscribe !!, Dependent

assures a one-to-many dependency between objects such that
the (many) dependents of the (one) object
are kept informed and adjusted as soon as the one is undergoing / performing a state change

Mediator

AKA Broker

arranges the interaction between objects such that the objects can remain unknown to each other

since the GangOfFour (GOF: Gamma et al)

patterns “always” have a fixed structure:

- name: to facilitate discussion and usage
- synonyms (AKA)
- intent: very short characterization
- motivation: reasons and rough idea
- applicability: conditions, criteria, situations

- structure
- participants
- collaborations
    the 3 together constitute the pattern’s model
    a bit obsolete: should be more complete

- consequences: analysis, discussion, variants
- implementation: discussion about it
- sample code: usually in well-known language
- known uses: real examples,
    from different domains
- related patterns: comparison,
    complementarity, successful combinations

(remarks on) examples of a business pattern:

often there is
some architectural pattern for organizations

workflow: pipe and filter pattern
    for any business activity

examples:

all waterfall-like process descriptions, e.g.

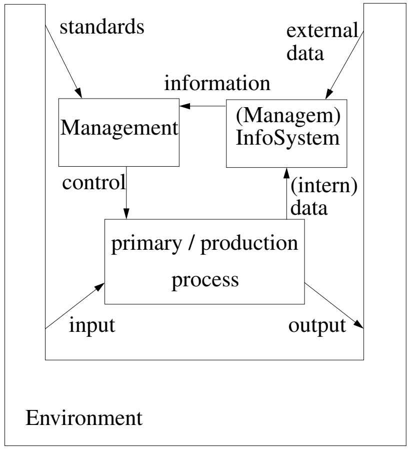
- complete lifecycle process
    of software engineering

- complete RE process as in chapter 2

- complete elicitation&analysis process
    as in beginning of this chapter

for general, managed organizations:

embedded feedback loop pattern
(Dutch: besturingsparadigma)



some remarks:

- this is not a UML diagram
  rather it is a data flow process diagram

- information and control are both data too

- standards and external data are optional

- pattern is recursive: it can re-occur
    inside primary process
    inside information system (IS)
    inside management
    or inside a combination of these

- ICT can have overlap with Management and with primary process (not only with IS)

of the above 13 structural requirements
only “sample code” is not easily fulfilled
unless
any business implementation counts too

roughly summarizing:

architectures:

reflect macrostructure and macrodynamics

patterns:

reflect microstructure and microdynamics

both are intended for improving
recognition, identification, discussion, analysis, application of
    essence

- global essence (macro)
- recurring essential features (micro)