Techniques for RE

Chapters 6, 7 and 9

see also our Chap 2.x about UML
and Chap 2.y about Architecture and Patterns

be aware of two possible, feasible starting points for techniques:

**data-centred**
starting from
**what exist**,
**what is present**
(there)

**behaviour-centred**
starting from
**what happen**s,
**how does it change**
(there)



data-centred:

data model / EER diagram / class diagram

built around logical, structural units (entities):
in OO: the classes

users & items & (sub)systems as classes

roles & services as methods

apart from structural relationships as
is-a & part-of & general "has-relation-with"

"call relationships" express the communication:
: I want you to give this service
or to play this role



behaviour-centred:

scenario's:
structured English
DFDs (data flow diagrams)
STDs (state transition diagrams)
automata, finite state machines, Petri nets
statechart diagrams
activity diagrams
interaction overview diagrams
sequence diagrams

built around logical dynamical units
(functionalities)
in OO: the use cases

role descriptions
interaction descriptions

either by example or exaustive



and with users & items & subsystems as actors

interaction relationships;

I send you this
message / trigger / event / signal /
/ information
enabling you to start behaving accordingly
as well as committing myself to this sending

From you I receive this
message / trigger / event / signal /
/ information
enabling me to start behaving accordingly
and aware of the sender's committing



3 common OO approaches
and 1 (still) unusual OO approach:

1. **use cases**
scenario's: examples of behaviour
in a sequential form:

this one does this
then, that one does that
then, ...

always related to Rs
(perhaps in groups)

always **declarative**



2. **class diagram**
consisting of
users / items / interfaces / (sub)systems

relations:
logical / structural / <<uses>>

methods:
roles / services / functionalities

attributes:
properties / characteristics / status(ses)

always related to Rs
(perhaps in groups)

always **declarative** (often: one system)



3. related to classes

**dynamics** according to scenario's as in 1

where such scenario's can be

- **visible**: external behaviour
meant for "method sequencing"

- **hidden**: internal behaviour
per method: specs of functionality

and

**influencing** of these dynamics
by means of **interaction** (send / receive)



in RE phase (our phase of SE) this means:

ad 2:

- interfaces of system-to-be (StB)

- no internal classes of StB
(unless these directly correspond to
problem domain)

ad 3:

- no hidden behaviours within StB

- visble behaviours correspond to interfaces

( again unless there is a direct
correspondence to problem domain
via classes present, reflecting that)



4: **Integration-Orientation**

as part of modelling within analysis subphase

step 1:

OO model (UML2.0-like) of
- business as-is
- environment (of business) as-is

this is a fully-fledged model
- structure (data-like)
- behaviour (eg. visible and hidden)
- interaction
(communication & coordination)

step 2:

1 package of software system to-be
- visible behaviour only
- interaction with business & env
both as-is and to-be

step 3:

align business & environment as-is
with software system to-be

- according to Rs
- in consequence of Rs
- despite Rs
then: extra negotiate / adapt



step 4:

show / explain / discuss
the **integrated**, resulting

**sofsys & bus & env to-be**

with the stakeholders

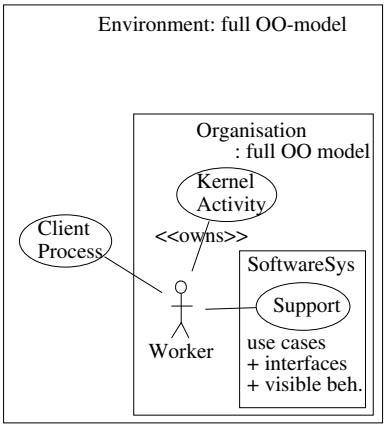
so we have now 3 RE-like processes:

step1: existing Rs for the as-is situation
moreover: design too

step 2: the classical RE process

step 3&4: 1&2 but well-integrated towards to-be








Ch.7. Viewpoint-Oriented RE Approaches

VOSE Chap 7.3
VORD Chap 7.4 & 9

VOSE:
Viewpoint-Oriented System Engineering

each stakeholder role gives a viewpoint (VP)

VOSE is of historical relevance only!!

template of VP in VOSE:
- **style** concrete formalism used
- problem domain (source)
- plan: schedule / approach / who does it
- record: history + reasons
- **specs**: Rs & model fragments for it



integration of huge number of VPs

the old drawback: **different styles**

are very hard to integrate

furthermore: standard problems

homonyms:
called the same, being different

heteronyms:
called different, being the same
(or overlapping)

overlaps --> consistency required
- generalized vs specialized
sometimes called **vertical** consistency
- aggregated vs decomposed
sometimes called **horizontal** consistency

consistency management needed



VORD:
VP-Oriented Rs Definition

the system to-be as client-server system

each client gives a VP

2 types of clients:

- direct VPs:
those asking for / receiving a service

- indirect VPs:
those observing any service providing
(engineering / organisational /
/ environmental / ...)



direct VPs

- users

- tuners: system administrators, operators, ...

- extra!: items, such as product
in view of proactive process support

note:
fully-fledged OO / UML

so the well-known difficulties with
integrating different styles
now belong to the realm of OO

where it has been rather partially solved
up to now

see Chap 9 for a large example