

A Domain Modeling Methodology

How does one create a GenVoca domain model?

This lecture will review the domain model behind ADAGE (avionics software) and the approach taken to develop it. This was the first attempt to apply GenVoca ideas on a “fresh” domain (i.e., one in which the domain modeler was not also a domain expert).

Note the following:

- similarities in the way data structures and avionics software are described
- although avionics is an unfamiliar domain for most, modeling processes and component interpretations are universal

exactly the same ideas can be applied to other domains

- note: modeling emphasis is on finding primitive *refinements*, less on fundamental “objects” or “classes”

Motivation and Basic Beliefs or When to Use GenVoca

New (avionics) software systems tend to be similar to those previously built

- on average, 50% to 80% of “new” system could be taken from component libraries

(Avionics) software generator produces (only) untuned systems:

- many aircraft-specific constants whose values are known only after extensive simulations and field tests
- “tuning” phase is not fully automatable
- work is aimed at less ambitious but still important goal of avionics software synthesis

Avionics software is obviously “hierarchical” and “mature” ... i.e., suitable for standardization

Domain Modeling Objectives

- (1) to identify the primitive components of avionics software

components can be designed, built, debugged independently of each other; they export and import standardized interfaces
- (2) to explain how components fit together to form systems and subsystems
- (3) to explain variations in avionics software as different combinations of components
- (4) to outline features of an avionics system generator that is plausible

domain model is a theory; until validated through implementation and experimentation, theory is suspect

Rest of the Lecture

Present overview of model

- review realms, components, compositions

Present lessons learned

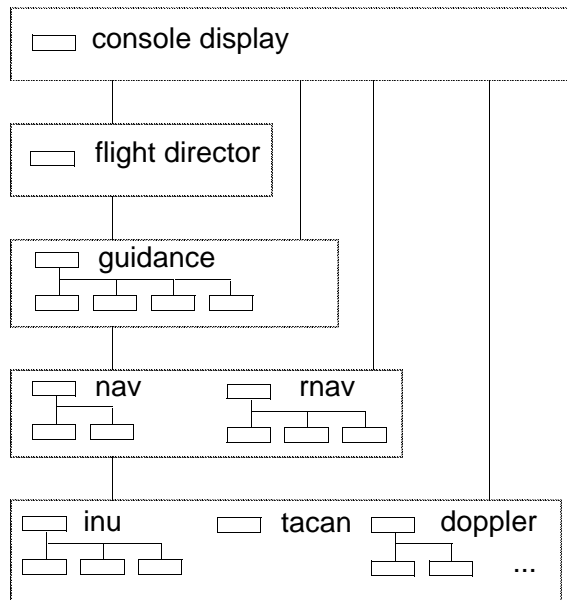
- big picture: how to decompose a domain?
- what worked and what didn't?
- see how domain/software is being described... same for data structures, databases, etc...

Results are collaborative effort with Lou Coglianesse, Steve Shafer, and Mark Goodwin...

see 1995 paper in *Symposium on Software Reuse*

Hierarchical Decomposition of Software

Avionics software systems are layered:



Took a bottom-up approach to subsystem decomposition and realm development...

Data Source Objects (DSOs)

Data Source Objects are:

- physical sensors and their device drivers
- located along exterior of aircraft
- report changes in aircraft height, position, speed, direction, etc. during flight
- data is reported at different rates per device

DSO subsystems:

- can be as simple as a single DSO
- can be complicated, involving selectors and filters of redundant DSOs
- in all cases, output is a single state vector (leaving the navigation subsystem impression that there is only one physical DSO in a DSO subsystem — where in fact there could be several)
- would like (during ground testing) to remove actual devices and plug in simulators (to drive navigation and higher-level software subsystems)

Data Source Objects (Continued)

- each type of DSO has unique interface, realm
- realm interface has state vector, operations

```

realm T;                                // "generic" realm
T = { dso1, dso2,...                    // DSO equipment list
      dso_simulator1,...                // DSO simulators
      dso_select[ {T} ],               // DSO selector
      dso_filter[ T ],...              // DSO filter
      dso_integrator[ {T} ],...        // integration
    }

```

- 20 distinct DSO types/realms identified containing 100+ components

```

adc      Air Data Computers
ahrs     Attitude Heading Reference System
dmg      Digital Map Generator
...
pls      Personnel Locating System
tacan    Tactical Air Navigation sensor

```

Example: Inertial Navigation System

```

realm ins;
ins = { ins_asn_141, ins_asn_143, ins_asn143rlg,
        ins_asn_90, ins_ln_15j, ins_ltn_51,
        ins_skn_2443, ..., ins_simulator,
        ins_select[{ins}], ins_filter[ins],
        ins_integrator[{ins}], ...
    }

```

DSO subsystem is a type equation:

- example: SOH has single `ins` component:

```
soh_ins = ins_asn_141;
```

- example: FWT has multiple `ins` components:

```

fwt_ins = ins_select[ { ins_skn_2443,
                        ins_skn_2443 } ]

```

NAV Subsystems

NAV subsystem returns position of aircraft w.r.t. earth coordinates

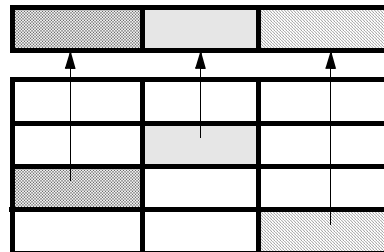
- most elaborate subsystem in avionics software that was studied

Navigation software works by:

- creates different estimates of aircraft position from DSO outputs or from combinations of DSO outputs — called *navigation modes*
- multiple estimates are fused into a single estimate by best-of-column approach (*note symmetry*)

“best” estimate

estimate from mode 1
estimate from mode 2
estimate from mode 3
estimate from mode 4



NAV Subsystems (Continued)

Lots of filters, selectors possible in NAV computations

Derived data added to vectors prior to their output

Realms of NAV subsystems are:

```
earth_model
earth_geometry
atmosphere_model
inav      (internal navigation)
nav_data  (derived data)
```

NAV subsystems modeled as *recursive* type equations involving components in 5 realms

- recursion needed to capture feedback loops
(see example shortly)

Internal Navigation Realm (INAV)

INAV components used internally with NAV subsystems

Navigation modes are components of INAV

Components that rank navigation mode outputs, perform selections, and washout filtering...

```
inav = { // List of known navigation modes.
    addr_nav_mode [atmosphere_model],
    ahrs_nav_mode [ahrs],
    dns_nav_mode [dns, earth_geometry],
    gps_nav_mode [gps],
    ins_nav_mode [ins],
    mhs_nav_mode [mhs],
    ...
    // Combined mode components.

    ahrs_dns_nav_mode [ahrs, dns, earth_geometry],
    gps_dns_nav_mode [gps, dns, earth_geometry],
    gps_ins_nav_mode [gps, ins],
    ins_dns_nav_mode [ins, dns, earth_geometry],
    ...
}
```

INAV (Cont)

```
// Filter components (note symmetry)

high_pass_filter [inav],
low_pass_filter [inav], ...

// Filters used to smooth transition due
// to changed input.

washout_filter_inav_dns [inav, dns,
                        earth_geometry],
washout_filter_inav [inav],

// ranking or mode-control components

static_nav_mode_ranking [{inav}],
dynamic_nav_mode_ranking [{inav}],

// selection and averaging components

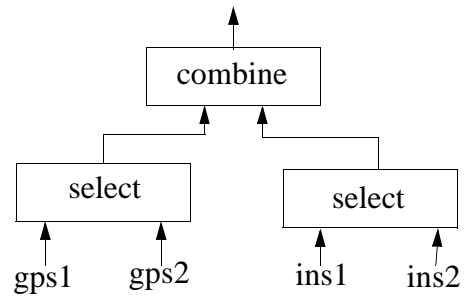
inav_select [{inav}],
inav_average [{inav}],

... }
```

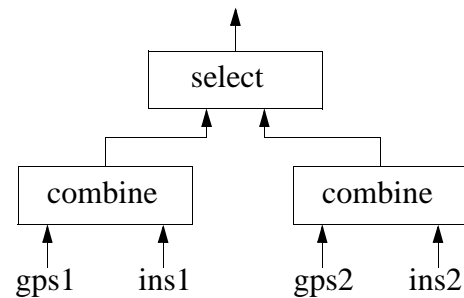
Compositions of these components model common software designs in avionics ...

Common Configurations of Components

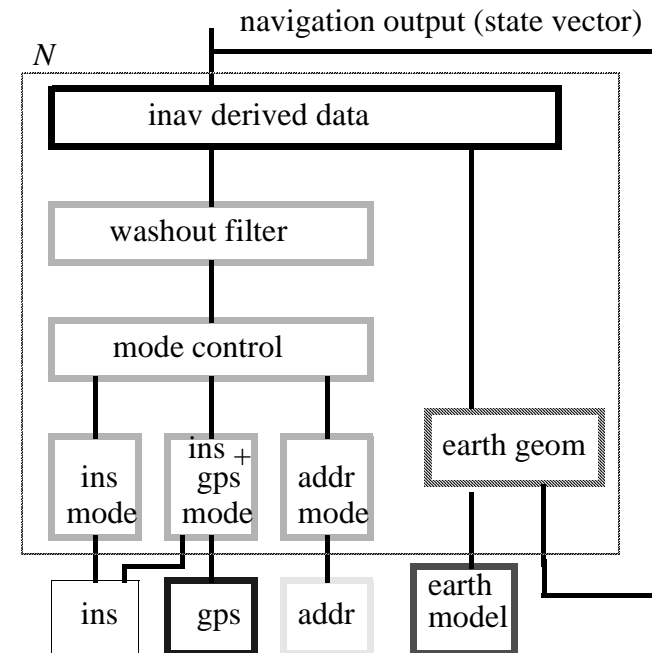
Selection before combining (aiding):



Combining before selection:



Recursive Equations Define NAV Subsystems



$$N = \text{navigation}(ins, gps, addr, earth_model, N)$$

Rest of the Model and Features Not Covered

Overall, ADAGE reference architecture has:

- > 40 realms
- > 350 components
- simple system > 50 components, > 15 layers deep

Model does not address:

- design rules... (see **Design Rule Checking** lecture)
- different implementation styles...
 - + simple executive where components interchange state vectors through global variables
 - + potential parallelism; components can be Ada tasks... (see **Architectural Styles** lecture)
- precise realm interface specifications — *not needed at this level of modeling/analysis*

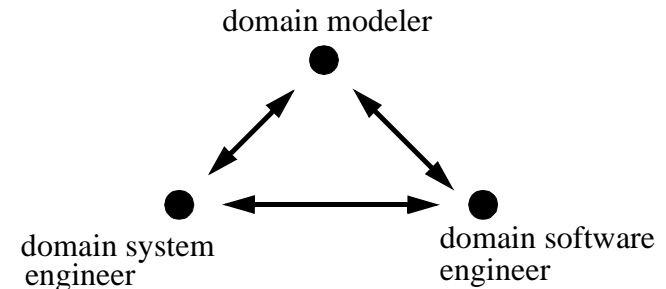
Lesson: Triad of Expertise

Creating a domain model requires expertise in:

domain modeling
domain-specific systems engineering
domain-specific software engineering

Ideally and previously, a single person assumed all three roles. Unfortunately, this situation is rare.

Triad of expertise:



Lesson: Exemplar Systems

Ideally, domain modeling requires study of many different systems

- Genesis model distilled from O(10) systems

We examined two avionics software systems in detail:

special operations helicopter (SOH)
fixed-wing transport (FWT)

Made up for lack of multiple exemplar systems by availability of considerable expertise at Loral FSD

Lesson: Avoid Looking at Code

Our work centered on components, not implementation

- examined the implementation of two similar navigation components in FWT and SOH
- general agreement on the set of equations used by both

disaster

Uncountable number of ways to code a set of equations

- terms can be given different names, appear in completely different places
- “individual coding preferences” common explanation

Not surprising why software reuse doesn't occur

- almost impossible to reuse *existing* code
- almost impossible to understand what an existing software module does (or how it does it)
- domain modeling must be at *conceptual*, not code, level
- *look at code only if absolutely necessary!*

Lesson: Look for Fundamental Programming Abstractions

Start by recognizing fundamental “layering” abstractions that are known to exist:

- ex: DSOs, Navigation, Radio Navigation, Guidance, etc.
- hint: for mature domains, these abstractions were probably discovered long ago...
- create realms for each abstraction

Sketch an OOVM interface for each realm abstraction:

- an abstraction may be a single object/class

ex: basic unit of exchange is a state vector
 avionics interfaces are operations on state vectors,
 such as read, write, and initialize

- an abstraction may be multiple objects/classes

ex: data structures have cursors, containers, elements

Note: identify abstractions first, classes later

Lesson: Verify Abstractions Through Implementation

Verify the OOVM abstractions by writing code:

- to ensure that the correct model/interface has been defined
- make sure that there are efficient implementations of the interface

just because interface is defined doesn't mean that it lends itself to efficient implementations

- using multiple, different OOVM implementations to ensure that interface isn't biased toward a particular implementation

to retrofit the interface later on will be very costly!

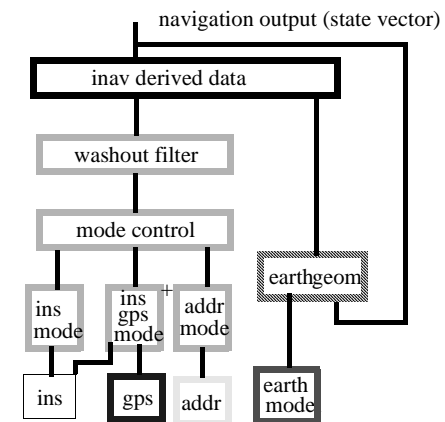
Lesson: Look for Large-Scale Refinements

Decompose subsystems into primitive components by looking for large-scale refinements:

- components are mappings/xforms between realm interfaces
- example: look for primitive refinements on state vectors — in particular look for *symmetric* components
(e.g., filtering, selecting, etc.)
- each time output state vector different from input — create new realm

Look for Refinements (Continued)

Goal: *wanted each component to match basic term or processing step used by avionics engineers*



Decomposition stopped when:

engineers were comfortable
& continuation not obvious
& evident that component could be encapsulated

Lesson: Information Gathering

Amazed at difficulty of obtaining seemingly elementary information

key is finding the domain experts(s)

Domain modeling requires clear understanding of how all parts of a class of systems work

few have necessary experience and insight

Expertise tends to be focussed on particular project coupled with diffuse information (or rumors) on what was done elsewhere

Curtis, et al. "A Field Study of the Software Design Process for Large Systems", CACM November 1988.

Current software engineering practices *hinder or preclude* successful domain modeling

Fundamental about understanding and designing software systems (or system families) in their entirety

Lesson: Need Right Team Attitude

Note: lesson from another project...(i.e., *not* ADAGE)

Each team member (esp. leader) must believe that a building-block approach is possible

- goal is to make it work (one way or another) knowing the real problems of domain-specific software development

i.e., don't shy away from hard problems

to convince others, must show that you're "not drilling holes in the thinnest part of the wood" and that you're solving the "hard" problems of that domain

"only get one shot; so do it right"

- if there are "negative" team members/leaders, the project has failed before it begins

Lesson: Don't Collect Useless Data

There is the tendency to collect all sorts of useless data on devices, entities, etc., that have little or nothing to do with a domain model

- it is necessary to define a common set of terminology and concepts for communication, but...
- don't collect a largely useless catalog of statistics and features that gives little or no clue on what the building blocks are and how they compose
- a GenVoca domain model isn't merely a classification of objects; it is a model of how complex suites of algorithms can compose
- *emphasis of modeling is on components and their compositions, not cataloging artifacts and features*

Lesson: Tools are Essential

We wrote a simple syntax checker `le`:

- to define realms
- to define realm components
- to declare systems as typed variables
- to define systems as equations

`le` reports type errors, unused variables, undefined variables, ambiguous equations, etc.

Big win

modeling is an art: any precision injected is always beneficial

clarified discussions

forced users to think in terms of building blocks

Example le Program

```
// realm declaration
realm s, s1, s2, r;

// enumerate realm membership or use inheritance
s1 = { m[s] }
s2 = { x[s,s], y[s], z[r] }
s :: s1, s2;      // s is a supertype of s1, s2
r = { t, p }

// declare subsystem variables
system <s> sys_1, sys_2;
system <r> sys_3, sys_4;

// specify equations that define systems
sys_3 = t;
sys_4 = p;
sys_1 = x[ y[ z[sys_3]], z[sys_3]];
sys_2 = y[ z[sys_4]];
```

Lesson: Design and Composition is the Key

Reuse libraries should *not* be populated with randomly-harvested components *if you want to generate software automatically*

Difficulties are legion: components are:

- not interoperable (different assumptions)
- not composable (ad hoc interfaces)
- customized, composed with great difficulty (*hacking*)

see Garlan 1995 ICSE paper

Components that are designed to be:

- plug-compatible, interoperable, interchangeable is key to *automated component-based software development*

Periodic Table example

Lesson: It Takes Time

Easy to underestimate the effort to design and build a generator:

*At least twice as difficult,
twice as costly to develop
generators than single system*

Why?

- hard enough to design a large software system, but...
 - in addition to all of the usual constraints, have constraints that designed components must be “reusable”
 - i.e., they must be interoperable, interchangeable
- must build more components than needed for a single system
 - can’t demonstrate generation of system families without a good-sized library of components

Must be prepared to expend resources to “do it right”

Conclusions — Lessons Learned

Repeatability of a domain modeling approach is important

- ADAGE was the first time a domain model was developed with GenVoca in mind
- has been repeated several times since...

Keys to “success”:

- *domain was mature*
- *software hierarchical*
- *not difficult to identify fundamental programming abstractions, standardized interfaces*
 - ... known for years ...*
- *components were identifiable as refinements between realm interfaces*
- *components corresponded to basic processing step used by (avionics) engineers to explain software*
- *prepared to invest resources, time*

Further Reading

- D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, "Creating Reference Architectures: An Example From Avionics", *ACM SIGSOFT Symposium on Software Reusability*, Seattle, 1995, 27-37.
- T. Biggerstaff, "An Assessment and Analysis of Software Reuse", in *Advances in Computers*, Volume 34, Academic Press, 1992.
- B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems", *Communications of the ACM*, November 1988.
- R. Prieto-Diaz and G. Arango (ed.), *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press 1991.
- D. Garlan, et al, "Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts", *ICSE 1995*.
- H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tavakoli, "A Prototype Domain Modeling Environment for Reusable Software Architectures", *Third International Conference on Software Reuse, Rio de Janeiro*, November 1-4, 1994, 74-83.
- R.E. Johnson and B. Foote, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, June/July 1988.
- R.E. Johnson, "Documenting Frameworks using Patterns", *OOPSLA 1992*, 63-76.
- R.E. Johnson, "How to Design Frameworks", Tutorial Notes, 1993.
- J.S. Heidemann and G.J. Popek, "File-System Development with Stackable Layers", *ACM Transactions on Computer Systems*, 12(1), 58-89.
- N. Hutchinson and L. Peterson, "The x-kernel: an Architecture for Implementing Network Protocols", *IEEE Trans. Software Engineering*, January 1991.
- S. O'Malley and L. Peterson, "A Dynamic Network Architecture", *ACM Transactions on Computer Systems*, 10, 2, May 1992.