

Architectural Styles

A software system is represented by a type equation, which defines the composition of algorithms, classes that comprise a system.

In certain problem domains (e.g., avionics), a software system is defined by a type equation *and* a specification of how components are to communicate.

In this lecture:

- review basic ideas of “architectural styles” and component “glue”
- relate to layering of components in GenVoca
- show examples
- explain relevance to software reuse, domain modeling, subjectivity

Introduction

Architectural styles fundamental concept in software design

- computations inside components should be *invariant* to protocols that enable components to interchange results

Example: wide range of styles in avionics software

- time-line executive where components exchange data through global state vectors
- Ada tasks that exchange state vectors thru inter-task communication
- many others...

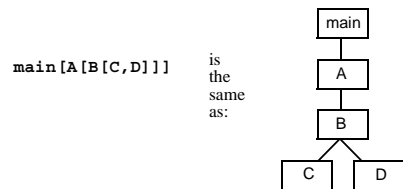
Using different component communication styles will generate substantially different software implementations for the same type equation: yet core of the generated systems would execute exactly the same algorithms in exactly the same order (sans communication protocols)

How to introduce styles into type equations?

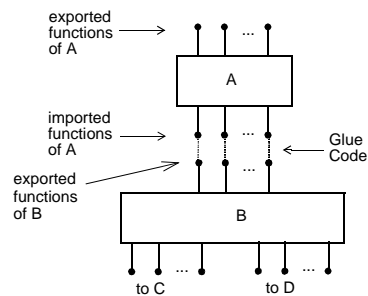
first answer: glue

Architectural Styles and Glue

Graphical depiction of simple type expression



Expanded view shows set of functions, interconnections:



Glue code is means by which connections between components is achieved - there's lots of different "glue"

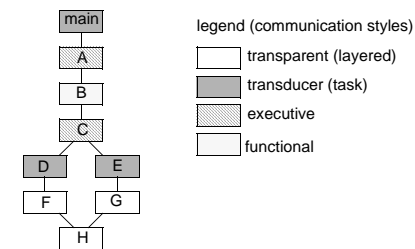
- Garlan & Shaw's "architectural styles"

First Idea

"Glue" is really an interface *envelope* that encloses or packages the algorithms of a component

GenVoca layout editor will have "glue" menu

- designers will paint the "glue style" onto components of a type expression to specify the communication style to use:



Problem: need way to express computations of component that is distinct from means of component communication

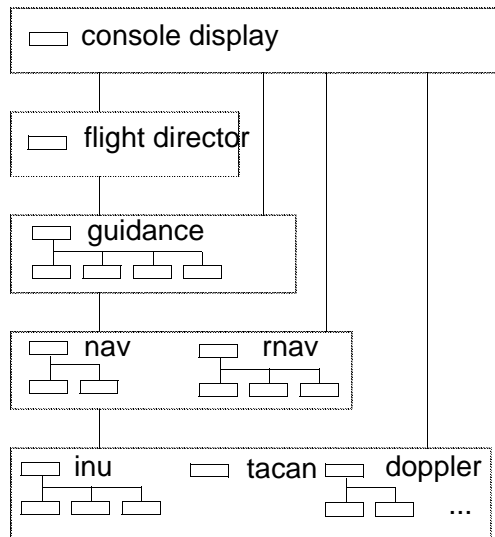
Solution: define canonical representation of components

- motivate development by looking at ADAGE

ADAGE Realm Background

Avionics components exchange aircraft state vector

- recall **Domain Modeling Methodology Lecture**:



- note: execution is bottom-up; different realms export different state vectors
- realms export single class (i.e., state vector) and operations on state vector

Abstract Operations

Canonical operation specification in pseudo-Ada:

```

procedure operation_name (
    STATE_VEC_IN1 : in STATE_VECT_TYPE_1 ;
    ...
    STATE_VEC_INn : in STATE_VECT_TYPE_n ;
    STATE_VEC_OUT : out STATE_VECT_TYPE_n+1
) is
begin
    COMPUTATIONS    -- algorithms that take STATE_VEC_IN*
                   -- and assign values to STATE_VEC_OUT
end
  
```

Component (realm) interface in ADAGE has operations:

- initialize state vector
- read (current state vector)
- update (current state vector)
- terminate, etc.

Focus on read operation; others are similar

Abstract Operations (Continued)

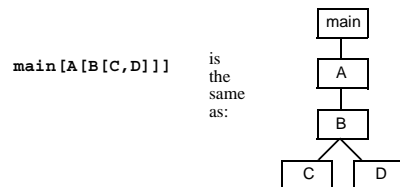
Example: read vector operation of component A:

```
procedure READ_A ( x : in TYPE_B; y : out TYPE_A ) is
begin
  y = a(x);  -- compute y using algorithm a() and input x
end
```

Note that $a(x)$ denotes an algorithm that is parameterized by x .

Now look at some examples of glue... observe that examples are:

- semantically equivalent
- syntactic transformations of each other
- use following type equation



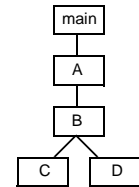
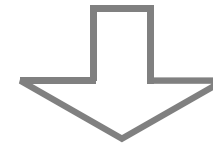
Executive Implementation

Outputs of operations stored in global state vectors

Communication via read & write of global state vectors

Before:

```
procedure READ_A ( x : in TYPE_B; y : out TYPE_A ) is
begin
  y = a(x);
end
```



After:

```
vec_a : TYPE_A;  -- global state vectors
vec_b : TYPE_B;

procedure READ_A is  -- read operation for component A
begin
  vec_a = a(vec_b);  -- read/write global state vectors
end;
```

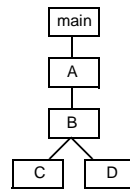
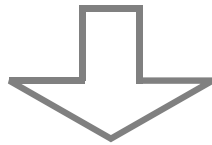
“Functional” Implementation

State vectors passed as parameters to function calls

No global state vectors/variables:

Before:

```
procedure READ_A ( x : in TYPE_B; y : out TYPE_A ) is
begin
  y = a(x);
end
```



After:

```
function READ_A ( vec_b : TYPE_B ) return TYPE_A is
begin
  vec_a : TYPE_A;
  vec_a = a(vec_b);
  return vec_a ;
end;
```

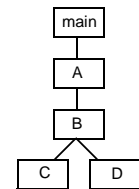
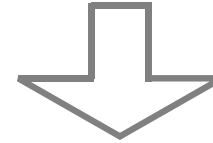
Layered Implementation

Each component only calls functions of immediate lower layer (A calls B, B calls C and D)

Abstract operations encoded as parameterless functions that return their state vector:

Before:

```
procedure READ_A ( x : in TYPE_B; y : out TYPE_A ) is
begin
  y = a(x);
end
```



After:

```
function READ_A return TYPE_A is
begin
  vec_a : TYPE_A;
  vec_b : TYPE_B;

  vec_b = READ B; -- call lower layer B for state vector
  vec_a = a(vec_b);
  return vec_a ;
end;
```

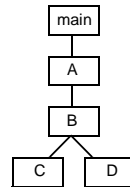
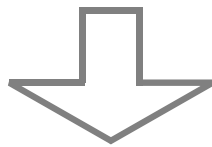
Transducer/Task Implementation

Each component is an Ada task (*transducer*)

State vectors exchanged between tasks:

Before:

```
procedure READ_A ( x : in TYPE_B; y : out TYPE_A ) is
begin
  y = a(x);
end
```



After:

```
task body TASK_A
use TASK_B is
begin
  loop
    accept READ_A( vec_a : out TYPE_A ) do
      vec_b : TYPE_B;
      -- read state vector from TASK_B
      TASK_B.READ B(vec_b);
      vec_a = a(vec_b);
    end;
    ...
  end loop;
end;
```

Synthesis

Each of previous examples of “style” are:

- semantically equivalent
- syntactic transformations of each other
- “simple” styles accomplished via *compositional* means

```
// customized interface
procedure READ_A_FUNC( x : in TYPE_B ) return TYPE_A is
begin
  READ_A ( x , y );    // actual interface
  return y;
end;
```

- “complex” styles accomplished via *generative* or *transformational* means:

parse tree
of original
system



parse tree
of stylized
system

semantics-preserving program transformations
see Griswold, Weigert papers

Synthesis (Continued)

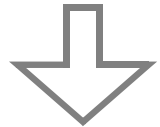
An architectural style is a refinement

GenVoca Component:

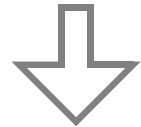
Architectural Style:

export interface

canonical interface



component
mapping



architectural
stylization

import interface

stylized interface

- look familiar???

*Architectural styles are
GenVoca components*

A Model of Architectural Styles

Every realm interface has its own architectural “style”

Define a GenVoca domain model (realms + components) as before:

- note: components are written in one particular “style”

$S = \{ s, \dots \}$

$R = \{ r[x:S], \dots \}$

$T = \{ t[x:R], \dots \}$

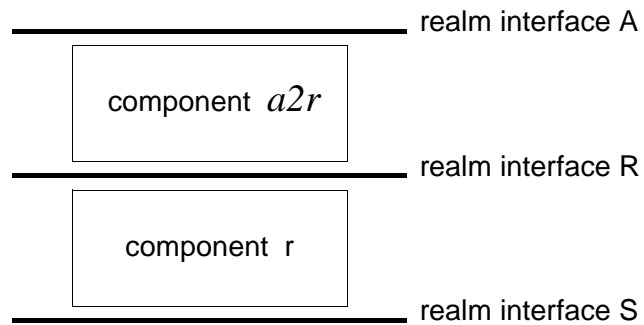
```
// realm R interface in style  $R$ 
// realm S interface in style  $S$ 
// realm T interface in style  $T$ 
```

Typically, a default style is “inlining” - i.e., algorithms of a component are inlined when they are invoked

Model (Continued)

Suppose we want realm **R** components to *export* the customized OOVM interface **A** (in style *A*)

- but how?



- Ans: defn. realm **A** and layer *a2r* to translate from **A** to **R**:

$$\mathbf{A} = \{ a2r[y:\mathbf{R}] \}$$

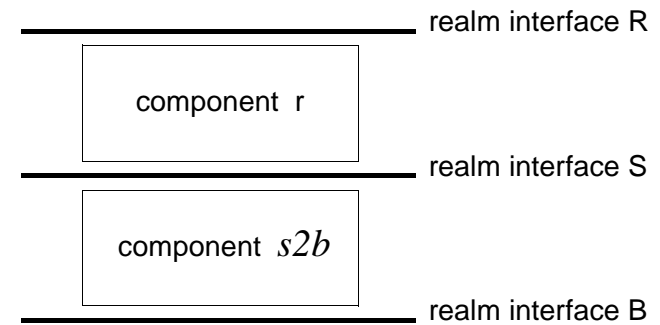
- new version of **r** is **r'**, which is defined by the equation:

$$\mathbf{r}'[x:\mathbf{S}] = a2r[\mathbf{r}[x:\mathbf{S}]];$$

Model (Continued)

Suppose we want realm **R** components to *import* the customized OOVM interface **B** (in style *B*)

- but how?



- Ans: defn. layer *s2b* in realm **S** to translate from **S** to **B**:

$$\mathbf{S} = \{ \mathbf{s}, \dots, s2b[y:\mathbf{B}] \}$$

- new version of **r** is **r''**, which is defined by the equation:

$$\mathbf{r}''[y:\mathbf{B}] = \mathbf{r}[s2b[y:\mathbf{B}]];$$

Model (Continued)

$a2r$ and $s2b$ are *style components*:

- i.e., components that encode realm interfaces in different architectural styles

*don't have to modify basic
GenVoca model to account
for architectural styles*

- in general, if there are m realms in a domain model and each is to support n styles, there will be:

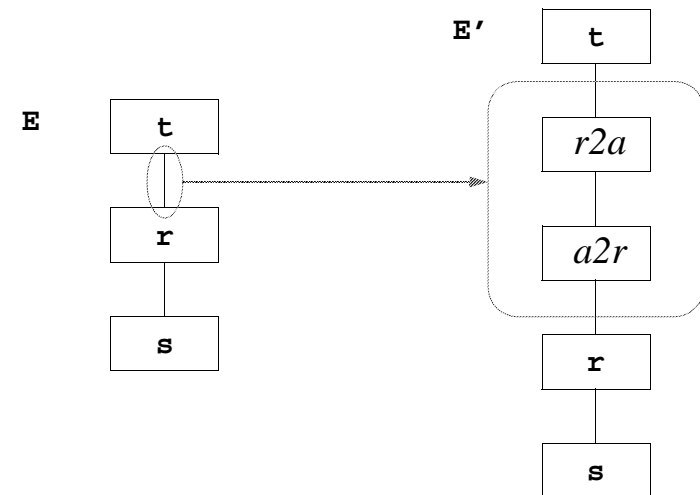
$m*n$ realms added and $2*m*n$ components added

manageable since m is ~ 10 -20 and n is ~ 1 -10

Style Usage

Suppose we want to A -stylize component r :

- we A -stylize the import interface of t and
- A -stylize the export interface of r

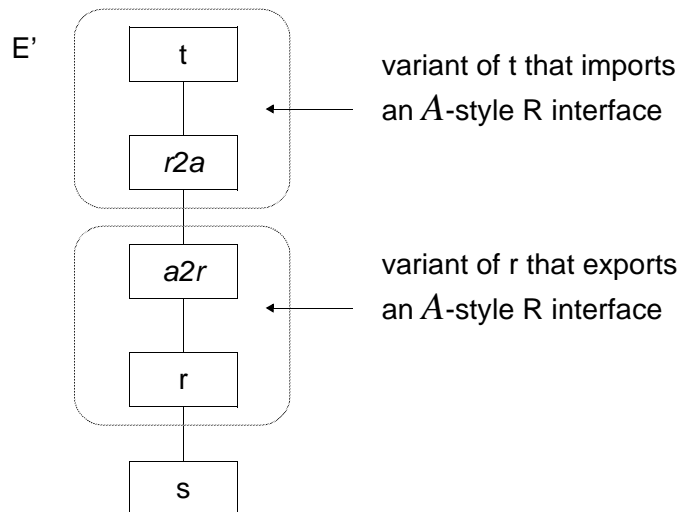


Note: composition $r2a [a2r]$ is *algebraic identity*

- domain-specific computations of E are identical to E'
- stylizing TEs does not alter their computations

Usage (Continued)

Net effect of algebraic identity (reparenthesize):



$$\begin{aligned}
 E' &= t''[r'[s]] \\
 &= t[r2a[a2r[r[s]]]] \\
 &= E
 \end{aligned}$$

Big Picture

A single type equation represents a large family of related systems

- members of family differ on their choice of “styles”
- members are similar because they implement the same fundamental algorithms

produce new variations of system via style algebraic identities

There are many other “styles” from which to choose:

- inlining
- Unix pipes/file filters
- dispatch tables (for dynamic reconfiguration)
- Gorlick’s weaves
- OO classes
- domain-specific styles (e.g., executive)
- ...

Big Picture (Continued)

What we're doing is not new:

- CORBA based on instance of same idea
see Udell's paper
- Module Interconnection Languages
see White and Purtilo's paper
- Design Patterns: Adaptor and Facade
see Gamma text

Architectural styles have an impact on:

- domain modeling
- subjectivity
- software reuse (and library construction)

Impact

Domain Modeling:

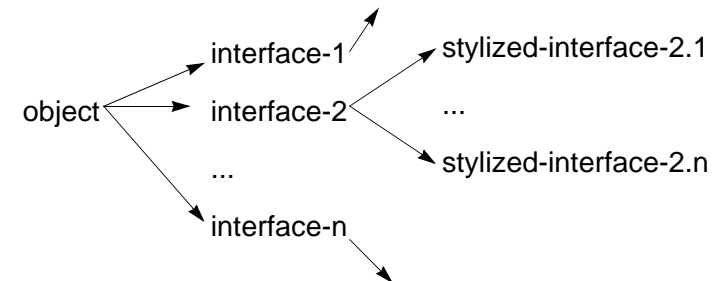
Look for fundamental algorithms

- *ignore communication / architectural "styles"*
- note "styles" that are used - maybe domain-specific

Subjectivity

New twist: not only do objects have families of interfaces

- *each interface has a family of different encodings*



- *architectural styles = architectural subjectivity*
the subjective choice of architectural styles for interface encoding for a particular application/system

Impact (Continued)

Software Reuse:

Don't want libraries of components with fixed styles

- precludes or hinders reuse
- don't want n filter components in a reuse library, 1 per style

Want library of components that are “style” free

Don't want the “packaging” of a component to preclude its reuse

- ex: Unix sort function and sort filter

Approach is *scalable* — each new “style” increases the number of systems that can be generated exponentially

Conclusions — Lessons Learned

- *architectural styles can be encoded as refinements - i.e., GenVoca components*
- *type equations + styles provides a very powerful set of concepts to describe and reason about software systems in terms of large-scale components*

can describe vast families of systems

- *separate component communication mechanisms from computations*

not new idea, but very important

simpler specification of components

scalability arguments for library construction

- *component communication protocols/styles can be domain specific (e.g., executive) or generic (e.g., inlining)*

Further Reading

- D. Batory and Y. Smaragdakis, “Another Look at Architectural Styles and ADAGE”, Loral FSD Owego T.R. ADAGE-UT-95-02.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- D. Garlan and M. Shaw, “An Introduction to Software Architecture”, in *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific Publishing Company, 1993.
- M.M. Gorlick and R.R. Razouk, “Using Weaves for Software Construction and Analysis”, *Proc. ICSE* 1991, 23-34.
- W.G. Griswold, “Direct Update of Data Flow Representations for a Meaning-Preserving Program Restructuring Tool”, *ACM SIGSOFT* 1993.
- D.E. Perry and A.L. Wolf, “Foundations for the Study of Software Architecture”, *ACM SIGSOFT Software Engineering Notes*, October 1992, 40-52.
- J. Udell, “Componentware”, *BYTE*, May 1994.
- E. White and J. Purtilo, “Integrating the Heterogeneous Control Properties of Software Modules”, *ACM SIGSOFT* 1992.
- T.J. Weigert, J.M. Boyle, T.J. Harmer, “Knowledge-Based Derivation of Programs from Specifications”, *Artificial Intelligence in Automation*, World Scientific Press, 1996.