Performance Specifications Based upon Complete Profiles

Joan Krone

William F. Ogden

Murali Sitaraman

Our Starting Point

D. Parnas:

- A good specification should tell a client everything he needs to know about a component and nothing more.

Us:

- A client needs to know not only about the functionality provided by a component, but also about its performance.

Goals for a Performance Specification Mechanism

It should support:

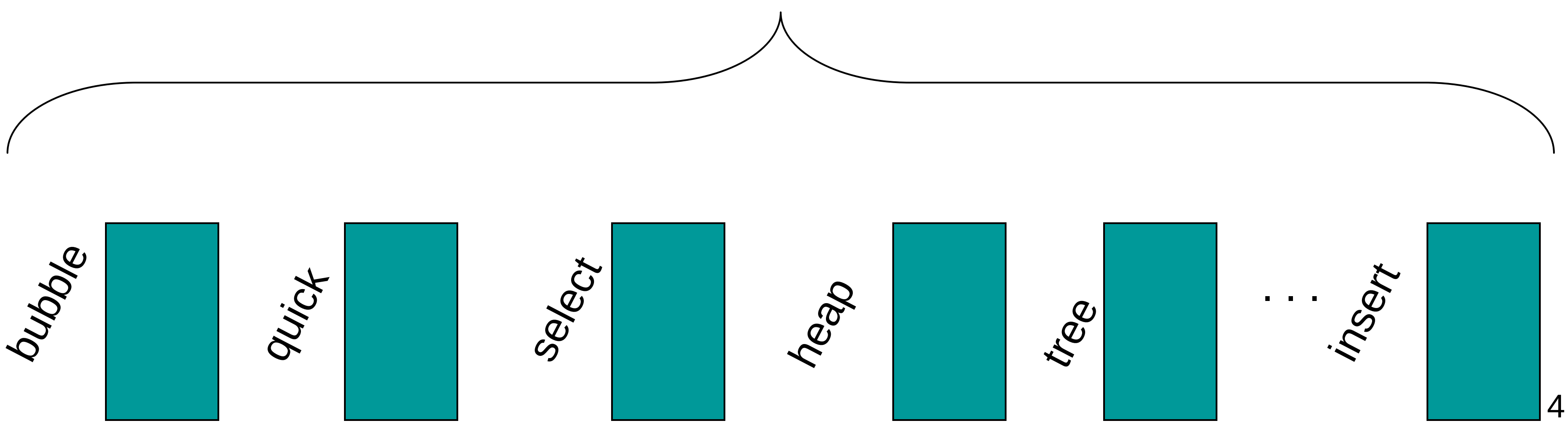
- Abstracting away confusing details
- Retaining adequate precision (completeness)
- Scaling for arbitrarily large components
- Verifying correctness of compositions
- Extending functional specifications

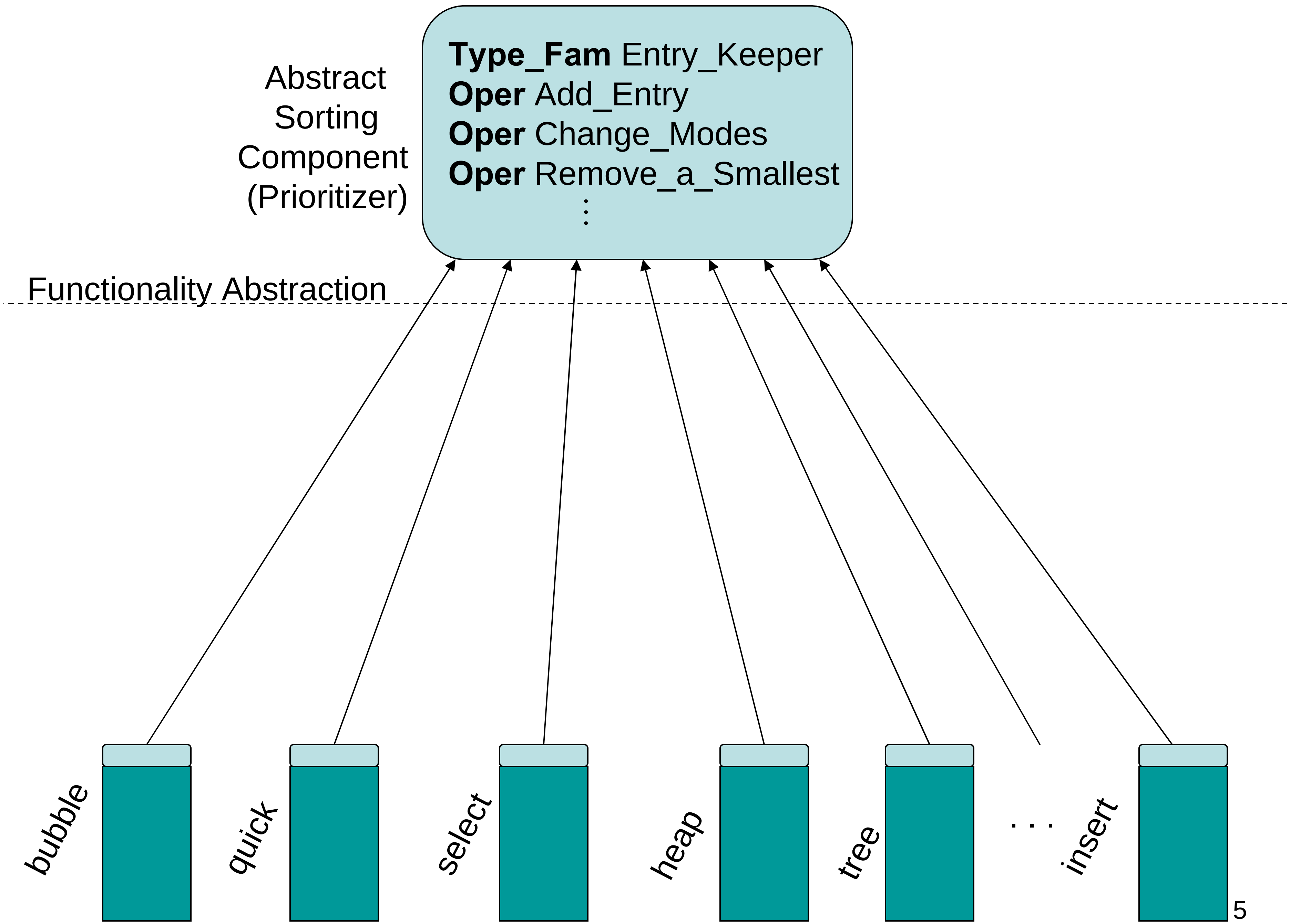
and critically:

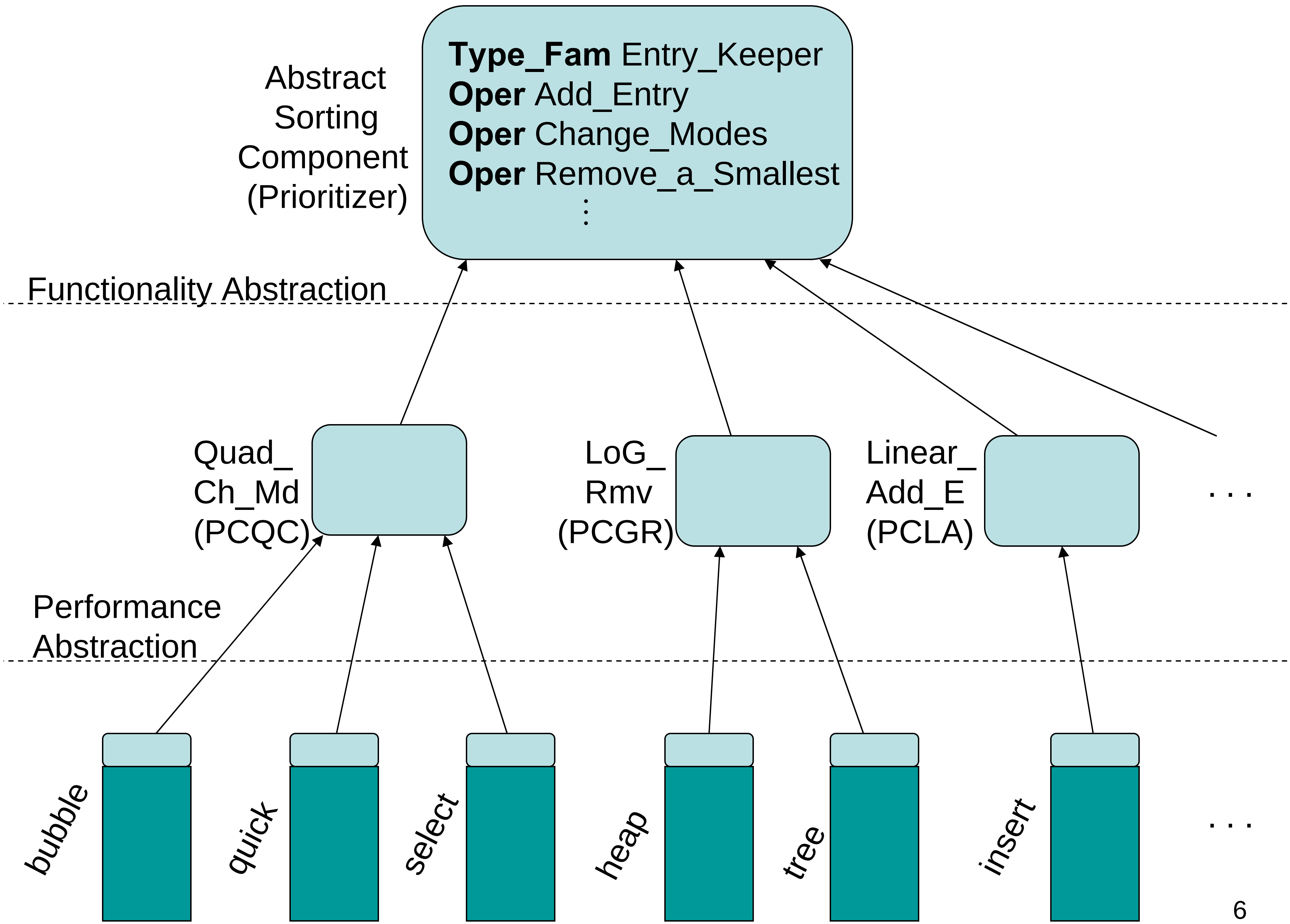
- Describing commonalities

Commonality Identification Example

Various “Sorting” Implementations




Type_Fam Entry_Keeper
Oper Add_Entry
Oper Change_Modes
Oper Remove_a_Smallest
⋮
Abstract Sorting Component (Prioritizer)
Functionality Abstraction
bubble
quick
select
heap
tree
. . .
insert


Abstract
Sorting
Component
(Prioritizer)
Type_Fam Entry_Keeper
Oper Add_Entry
Oper Change_Modes
Oper Remove_a_Smallest
⋮
Functionality Abstraction
Quad_
Ch_Md
(PCQC)
LoG_
Rmv
(PCGR)
Linear_
Add_E
(PCLA)
. . .
Performance
Abstraction
bubble
quick
select
heap
tree
insert
. . .

Simple Profile Example with Stacks

```
Concept Stack_Template( type Entry; eval Max_Depth:
        ⋮
  Type_Family Stack ⊆ Str(Entry);
        ⋮
  Operation Push( alters E: Entry; updates S: Stack );
      requires |S| < Max_Depth;
      ensures S = ⟨#E⟩∘#S;
  Operation Pop( replaces R: Entry; updates S: Stack );
      requires |S| > 0 ;
      ensures #S = ⟨R⟩∘S;
  Operation Depth_of( preserves S: Stack ): Integer;
      ensures Depth_of = ( |S| );
        ⋮
```

```
Enhancement Flipping_Capability for Stack_Template;
   Operation Flip( updates S: Stack );
      ensures S = #S^Rev;
end Flipping_Capability;
```

Possible Implementation:

```
   Realization Obvious_F_C_Realiz for Flipping_Capability
      Procedure Flip( updates S: Stack );
         Var Next_Entry: Entry;
         Var S_Flipped: Stack;
         While Depth_of( S ) ≠ 0
            affecting S, S_Flipped, Next_Entry;
            maintaining #S = S_Flipped^Rev ∘ S and
                                          Entry.Is_Init(Next_Entry);
            decreasing |S|;
         do
            Pop( Next_Entry, S );
            Push( Next_Entry, S_Flipped );
         end;
         S :=: S_Flipped;
      end Flip;
   end Obvious_F_C_Realiz;
```

An Example Profile

Profile SSCF **short_for** Stack_Space_Conscious_Flip **for**
Flipping_Capability **for** Stack_Template **with_profile** SSC;
Defines $SSCF_{F1}$, $SSCF_{F2}$: $\mathbb{R}^{\geq 0}$;
Defines $SSCF_{FMC1}$, $SSCF_{FMC2}$: $\mathbb{N}$;
Operation Flip(**updates** S: Stack);
duration $SSCF_{F1}$ + Entry.**I_Dur** + Stack.**I_Dur** +
Entry.**F_IV_Dur** + Stack.**F_IV_Dur** +
($SSCF_{F2}$ + Entry.**I_Dur** + Entry.**F_IV_Dur**)·|#S|;
manip_disp ($SSCF_{FMC1}$ + Entry.**I_Disp** + Stack.**I_Disp**) +
Max($SSCF_{FMC2}$, Entry.**IM_Disp**, Entry.**F_IVM_Disp**);
end SSCF;

```
        duration SSCF_F1 + Entry.I_Dur + Stack.I_Dur + Entry.F_IV_Dur +
            Stack.F_IV_Dur + (SSCF_F2 + Entry.I_Dur + Entry.F_IV_Dur)·|#S|;
```

```
Realization Obvious_F_C_Realiz for Flipping_Capability
    Definition SSCF_F1: ℝ≥0 = ( Dur_Call(1) + SSC_Dp + Int.Dur_≠ + Dur_:=: );
    Definition SSCF_F2: ℝ≥0 = ( SSC_Dp + Int.Dur_≠ + SSC_Po1 + SSC_Pu );
    Definition SSCF_FMC1: ℕ = ⋯
        ⋮
    Procedure Flip( updates S: Stack );
        Var Next_Entry: Entry;
        Var S_Flipped: Stack;
        While Depth_of( S ) ≠ 0
            affecting S, S_Flipped, Next_Entry;
            maintaining #S = S_Flipped^Rev ∘ S and Entry.Is_Init(Next_Entry);
            decreasing |S|;
            elapsed_time ( SSCF_F2 + Entry.I_Dur +
                                                  Entry.F_IV_Dur )·|S_Flipped|;
        do
            Pop( Next_Entry, S );
            Push( Next_Entry, S_Flipped );
        end;
        S :=: S_Flipped;
    end Flip;
```

Profile SSC **short_for** Space_Conscious **for** Stack_Template;

Defines SSC_I, SSC_{I1}, SSC_F, SSC_{Po1}, SSC_{Pu}, SSC_C, SSC_{C1}, SSC_{Dp}, SSC_{RC}: $\mathbb{R}^{\geq 0}$;

Type_Family Stack;

Initialization

duration $SSC_I + (SSC_{I1} + \text{Entry.}\mathbf{I\_Dur})\cdot\text{Max\_Depth}$;

Operation Pop(**replaces** R: Entry; **updates** S: Stack);

duration SSC_{Po1} + Entry.**I_Dur** + Entry.**F_Dur**(#R);†

Operation Push(**alters** E: Entry; **updates** S: Stack);

ensures Entry.**Is_Init**(E);‡

duration SSC_{Pu};

Operation Depth_of(**preserves** S: Stack): Integer;

duration SSC_{Dp};

⋮

end SSC;

†Note that this duration expression is split between the externally defined terms for the duration of an Entry initialization, Entry.**I_Dur**, and the finalization of the incoming value of R, Entry.**F_Dur**(#R) and the internally defined term SSC_{Po1}.

‡Note that this extension of the functional specification for Push is essential for achieving tight performance specifications.

So, the **Profile** construct is a performance specification mechanism that supports:

- Abstracting away confusing details
- Retaining adequate precision (completeness)
- Scaling for arbitrarily large components
- Verifying correctness of compositions
- Extending functional specifications
- Describing commonalities

It (or something quite similar) should be included in any serious language for component specification and verification.

What Else is There?

- What happens with displacement (space).
- How component composition works.
  - Multiple profiles for a constituent.
- How large components can have simple profiles.
- How to formally verify profiles.
- When to supplement an object model.