




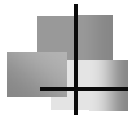
Review

- This is a summary/review of the *model of software* as presented in 221/222 and discussed in "Software Component Engineering With Resolve/C++"
- <http://www.cse.ohio-state.edu/sce/book/index.html>



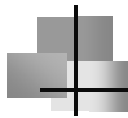
Model of Software

- How should software be designed and built?
- Modern software is component-based
- What are the essential pieces of the Resolve/C++ component-based model of software?
 - Components
 - Relationships
 - Discipline

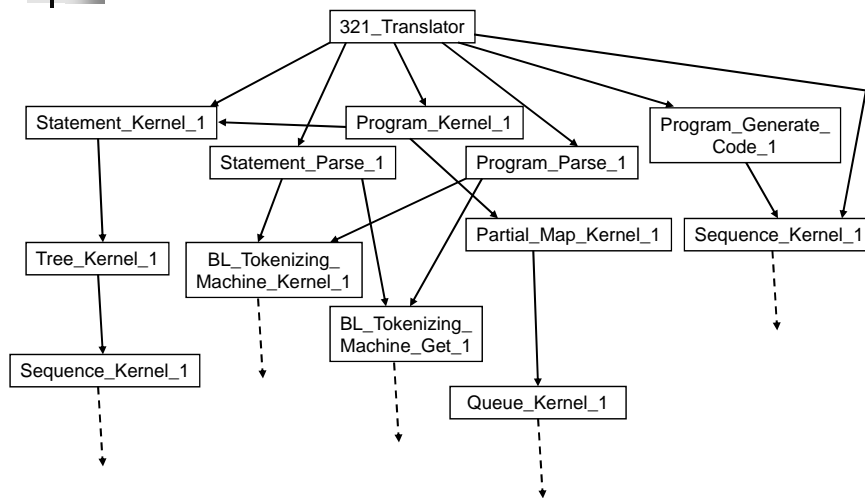


Components

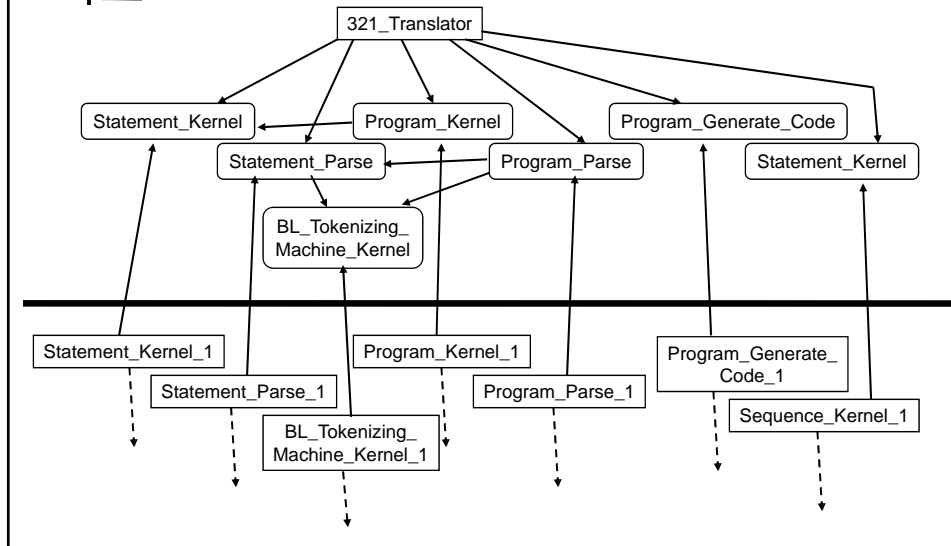
- Abstract
 - Client view, specifications, behavior, contract
 - Describe the "what"
- Concrete
 - Implementer view, code
 - Describe the "how"
- Main program



Concrete Components Only

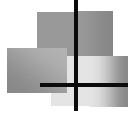


With Abstract Components



Other Dimensions

- There are other (secondary) dimensions along which components can be organized and distinguished:
 - Instance vs. template
 - Kernel vs. extension
 - Kernel/extension vs. utility
- They are important, but not quite as fundamental as abstract vs. concrete



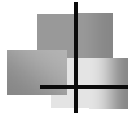
Component Dependencies

- Whenever a component $C1$ mentions in its code another component $C2$, we say that $C1$ depends on $C2$
- There are different ways a component can depend on another component
- We have names for the most common and meaningful relationships



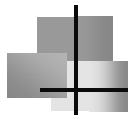
Relationships

- Implements $C \rightarrow A$
- Extends $A \rightarrow A, C \rightarrow C$
- Checks $C \rightarrow C$
- Instantiates $AI \rightarrow AT, CI \rightarrow CT$
- Encapsulates $C \text{ kernel} \rightarrow \text{Rep}$
- Specializes $AT \rightarrow AT, CT \rightarrow CT$
- Uses



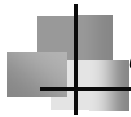
Our Approach vs. Others

- In standard OO approaches/languages
 - Components = classes
 - Relationships = inheritance
- Our approach (model of software) is more *specific* in defining what a component is and what kinds exist, and in defining how components can depend on each other
- This is not meant to restrict our creativity—it is meant as guide toward good designs

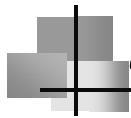
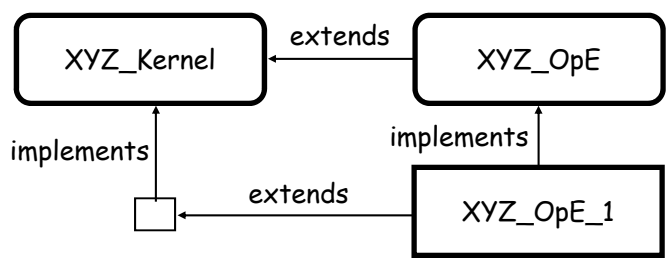


Component Coupling Diagrams

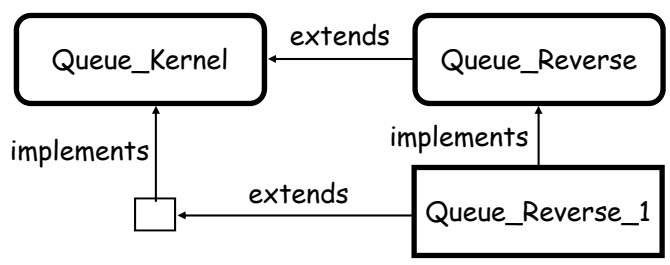
- Component coupling diagrams (CCDs) are used to depict graphically *all* component dependencies
- They mostly follow a few standard patterns

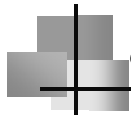


Extension Component CCD

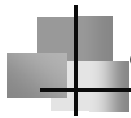
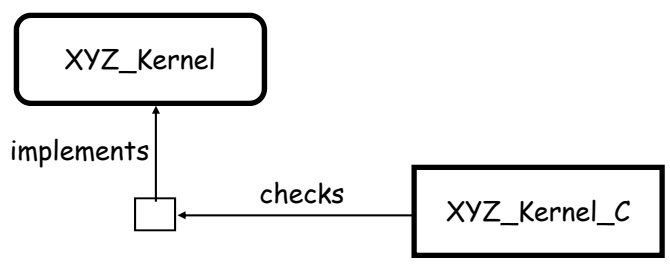


Extension CCD: Example

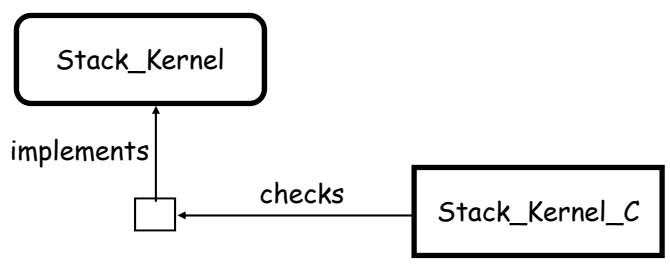




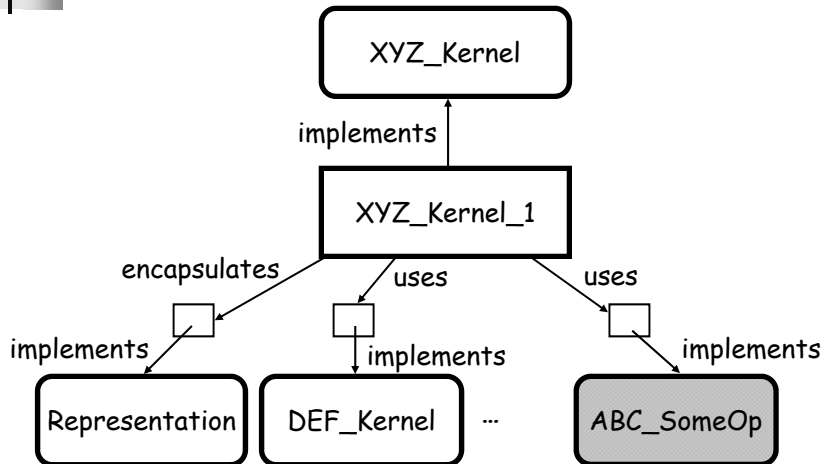
Checking Component CCD



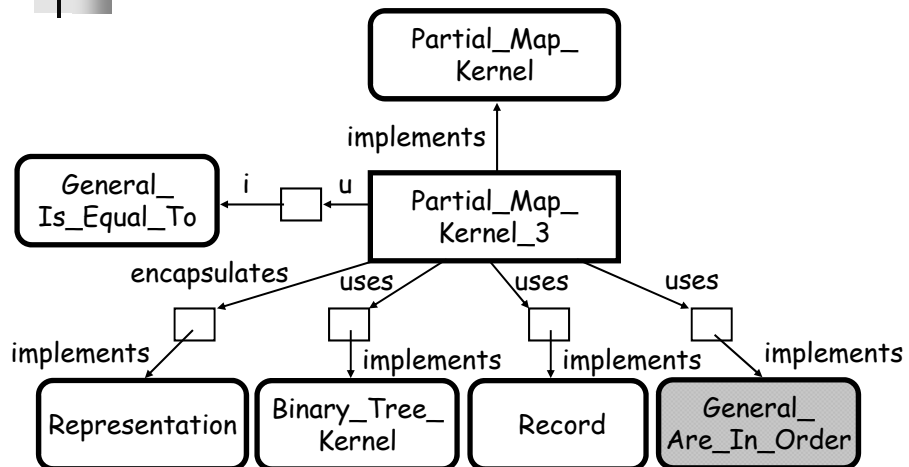
Checking CCD: Example

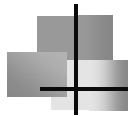


Kernel Component CCD



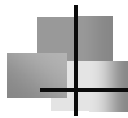
Kernel CCD: Example





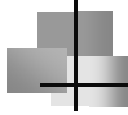
Resolve/C++ Discipline

- A variety of rules and guidelines adhering to best practices in software engineering
- They range from very basic rules on formatting code to fairly technical and deep rules to avoid bad surprises



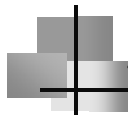
Coding Conventions

- Use only `if`, `if-else`, `if-else-if`, `case_select`, and `while` (no `for`, `do-while`, etc.). Always include blocks in `{ }`, even single-statement blocks.
- Use **and**, **or** and **not** instead of `&&`, `||`, `!` (but it is OK to use `!=`)—note that **and** and **or** do not use short-circuit evaluation.
- All operations must include `requires` and `ensures` and appropriate parameter modes for all formal parameters.
- Class names should be nouns, in mixed case with the first letter of each word capitalized and words separated by `'_'`.
- Operation names should be capitalized like class names; procedure names should be verbs and function names should be nouns.
- Object names are entirely lower case with words separated by `'_'`.



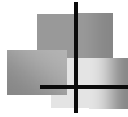
Basic Rules

- Formal parameter declaration rule—Declare every formal parameter of every operation as a reference parameter (i.e., put "&" after the type name), *unless* it is a preserves-mode parameter of type *Boolean*, *Character*, *Integer*, *Real*, or *Text*. All the parameters of functions must be preserves mode (and it is desirable to design functions so that there is no requires clause).
- Repeated argument rule—Never repeatedly use the same object as an actual parameter to a single operation call.
- Object declaration rule—Declare each object at the very beginning of the innermost block possible; do not declare an object with a global name.
- Object redeclaration rule—Do not redeclare an object with a name that is already visible at the point of declaration.



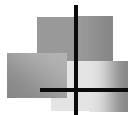
Programming By Contract

- Design-by-Contract Rule—Do not call an operation when its precondition is not satisfied.
- Precondition Rule—When implementing any operation body, *assume* that the operation's precondition is satisfied upon entry to the operation body.



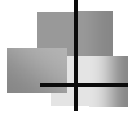
Component Dependecies

- Limited Dependence Rule—Design each component so it depends on as few other components as possible.
- Least Information Rule—Design each component so it depends only on components that contain information that is required to understand the new component.
- Concrete Component Dependence Rule—Avoid introducing a dependence on a concrete component. If dependence of a concrete component on another component is required, create that dependence to an abstract component.



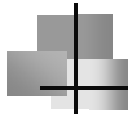
Component Organization

- Components are organized in catalogs
- Catalogs are organized by component families
- Component families usually contain
 - Abstract kernel—defining the type (math model), initial value, and kernel operations (with specs)
 - Abstract extensions—defining one or more operations (with specs)
 - Concrete kernels—various implementations of abstract kernel with different representations and algorithms (and thus performance)
 - Checking kernel—checking implementation layered on top of (and parameterized by) other kernel implementations
 - Concrete extensions—implementation of extensions layered on top of (and parameterized by) kernel implementations
- http://www.cse.ohio-state.edu/sce/rcpp/RESOLVE_Catalog-HTML/



Catalog Filesystem Structure

- A catalog in Resolve/C++ is a directory tree with the following structure:
 - Some Catalog
 - AI
 - FamilyA
 - FamilyB
 - ...
 - AT
 - FamilyB
 - Kernel.h
 - ExtensionX.h
 - FamilyC
 - ...
 - CI
 - FamilyA
 - FamilyB
 - ...
 - CT
 - FamilyB
 - Kernel_1.h
 - Kernel_1a.h
 - Kernel_1a_C.h
 - Kernel_2.h
 - Kernel_C.h
 - ExtensionX_1.h
 - FamilyC
 - ...



Abstract Kernels

- An abstract kernel includes:
 - A new programming type
 - Name
 - Math model
 - Initialization value
 - Kernel (primary) operations
 - Name
 - Formal parameters and parameter modes
 - Requires and ensures
 - For templates, template parameters

http://www.cse.ohio-state.edu/sce/rcpp/RESOLVE_Catalog-HTML/AT/XYZ/Kernel.html



Abstract Kernel Design

- Pick a meaningful component *family* name and name the component **Family_Kernel**
- Think about what values Family_Kernel objects can have and describe them using a mathematical model
 - boolean, integer, real, character, string, set, multiset, tuple (, ,), binary tree, tree
 - Name the various pieces of the model
 - Include any constraints on the possible values of each piece
- Specify the initial value as a value of the model type
- Think about how the component should be parameterized
 - Choose appropriate *abstract template* parameters
- Think about how you want to manipulate the new objects, and choose the kernel operations
 - Select meaningful operation and parameter names
 - Specify parameter types and modes, and requires and ensures
 - Kernel (*primary*) operations should provide *minimal*, but *complete* functionality (in particular, client needs operations to check requires of all operations)
- Remember you are designing the *client* view!!



Requirements

- The new component needs to allow the client to count the number of occurrences of arbitrary items.
- It should also support easy computation of item frequencies.
- It should be possible to iterate over *distinct* items without having to iterate on each instance of each item (e.g., to print a table of occurrences).
- The new component should allow memory efficient implementations (where memory usage is proportional to the number of *distinct* items)



Concrete Kernels

- A concrete kernel includes:
 - The type representation
 - Instance of Representation record
 - Fields (type, name)
 - Convention and correspondence
 - Public kernel operations implementation
 - Private (local) operations
 - Name
 - Formal parameters and parameter modes
 - Requires and ensures
 - Implementation
 - Template parameters (some from abstract component and others specific to the implementation)
 - For implementation-specific template parameters provide default choices whenever possible)

http://www.cse.ohio-state.edu/sce/rcpp/RESOLVE_Catalog-HTML/CT/XYZ/Kernel_1.html



Concrete Kernel Design

- Choose a representation that satisfies the following:
 - It is expressive enough to represent all the possible abstract values
 - It is good enough to allow efficient implementation of all kernel operations
- Think about the mapping between abstract and concrete (representation) values and describe this mapping in the *correspondence* clause
- Think about any constraints on the acceptable representation values and describe them in the *convention* clause
- Include all components needed (including those used in template parameter restrictions, even those they appear in comments)
- Specify the template parameters by including
 - The parameters to the abstract kernel, and
 - Parameters for each non-built-in component used in the representation (specify defaults for each of these whenever possible)
- Implement the kernel operations and design and implement any local operations that help improve your implementation
- See the Resolve/C++ catalog for many examples of kernel implementations



Kernel Implementation Rules

- **Single-Object Data Representation Rule**—Always combine the data representation objects, even if there is only one of them anyway, into a single representation record whose type is an instance of the built-in *Representation* template.
- **Representation Record Naming Rule**—Use the name *Rep* for the formal template parameter that is an instance of *Representation*.
- **Default Template Parameter Rule**—For every type used in the data representation of a concrete kernel component, if it is not a built-in type then declare that type as a formal template parameter. Create an appropriate default binding for each formal template parameter for which you (as implementer) can create an appropriate instance on behalf of the client. List the formal template parameters without default bindings first, followed by those with default bindings, making *Rep* the last one.



Kernel Implementation cont.

- **Representation Invariant Rule**—When implementing any kernel operation body (except the constructor), *assume* that the convention clause holds upon entry to the operation body. For any kernel operation body (except the destructor), *make sure* that the convention clause again holds upon return from the operation body. The operation body may, however, change the representation in such a way that the convention clause temporarily does not hold during execution of the operation body.
- **Abstraction Relation Rule**—When implementing any kernel operation body (except the constructor), *assume* that the correspondence clause holds upon entry to the operation body. For any kernel operation body (except the destructor), *assume* that the correspondence clause holds upon return from the operation body. Do *not* assume that the correspondence clause holds during execution of the operation body.



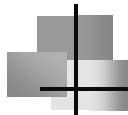
Kernel Implementation cont.

- Layered Standard Operation Rule—If the representation record's default initial value (i.e., a record with each field having an initial value for the type of that field) satisfies the convention, and the correspondence says that it represents an abstract initial value, then do not write any code for the constructor. Otherwise, write as the body of the private procedure *Initialize* the code that transforms the initialized representation record to some representation of an abstract initial value. Do not write any code for the destructor, the swap operator, or the *Clear* operation.



Kernel Implementation cont.

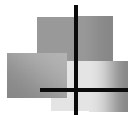
- Kernel Purity Rule—In the body of a kernel operation, do not call any kernel operation of the same component (including a recursive call).
- Local Operation Rule—In the body of any local operation other than *Initialize* or *Finalize*—as in the body of a global operation—do not mention **self**. In a call to a local operation—as in a call to a global operation—do not use a distinguished argument.



Checking Kernels

- A (concrete) checking kernel includes:
 - Public kernel operations with requires
 - Implemented by layering on checked kernel
 - Template parameters
 - Abstract kernel template parameters
 - Family_Base implementing abstract Family_Kernel
 - Private (local) operations, if needed
 - Name
 - Formal parameters and parameter modes
 - Requires and ensures
 - Implementation

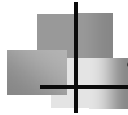
http://www.cse.ohio-state.edu/sce/rcpp/RESOLVE_Catalog-HTML/CT/XYZ/Kernel_C.html



Checking Kernel Rules

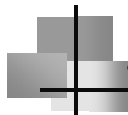
- Checking Kernel Implementation Rule—Implement checking kernels by parameterizing them by an implementation of the kernel and by layering them on top of it.
- Checked Operation Rule—Provide a checked implementation for any public kernel operation with a non-true requires clause. In the body of checked operation *OpA*, check the requires clause, and then call the checked implementation's operation *OpA* with the same parameters and with the following syntax:

self.Family_Base::OpA (...)



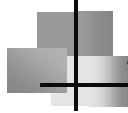
Lab 6: General Structure

1. Read words and add them to counting component (loop 1)
2. Extract the N (100?) words with highest count
 - take words out of counter and put them in SM1 (loop 2)
 - take top N words out of SM1 and put them in SM2 (loop 3)
3. Generate tag cloud of N words in alphabetical order
 - take words out of SM2 and output tag cloud (loop 4)



Lab 6: Other Details

- Include and instantiate components needed
- Declare records and utility classes
- Choose appropriate implementations (for performance)



Lab 6: Template Instantiation

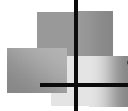
```
concrete_instance
class Name :
    instantiates
        Template <parameters>
    {};
```



Lab 6: Record Instantiation

```
concrete_instance
class Name :
    instantiates
        Record <Type1, Type2, ... >
    {};
```

```
field_name (Name, 0, Type1, fName1);
field_name (Name, 0, Type2, fName2);
...
```



Lab 6: Utility Classes

```
concrete_instance
utility_class Name :
  implements
    abstract_instance General_Are_In_Order <Type>
{
  /*! math definition !*/
  utility_function_body Boolean Are_In_Order (
    preserves Type& x,
    preserves Type& y
  )
  {
    // implement function
  }
};
```

see http://www.cse.ohio-state.edu/sce/rcpp/RESOLVE_Catalog-HTML/CI/Text/Are_in_Order_1.html



Abstract Extensions

- An abstract extension includes:
 - One or more extension (secondary) operations
 - Name
 - Formal parameters and parameter modes
 - Requires and ensures
 - Template parameters
 - Abstract kernel template parameters

http://www.cse.ohio-state.edu/sce/rcpp/RESOLVE_Catalog-HTML/AT/XYZ/OpE.html



Layered Extensions

- A concrete (layered) extension includes:
 - Public extension operation(s) implementation
 - Layered on top of extended kernel
 - Template parameters
 - Abstract kernel template parameters
 - Family_Base implementing abstract Family_Kernel
 - Private (local) operations
 - Name
 - Formal parameters and parameter modes
 - Requires and ensures
 - Implementation

http://www.cse.ohio-state.edu/sce/rcpp/RESOLVE_Catalog-HTML/CT/XYZ/OpE_1.html



Layered Extensions Rules

- Extension Implementation Rule—Implement extensions by parameterizing them by an implementation of the kernel and by layering them on top of it.
- Local Operation Rule—In the body of any local operation—as in the body of a global operation—do not mention **self**. In a call to a local operation—as in a call to a global operation—do not use a distinguished argument.