

How to Draw a CCD from Code

Suppose you want to understand a new component. What other components do you need to understand first? The main purpose of a *component coupling diagram* (CCD) is to show you exactly this information. How do you start from the code to be understood, such as that in Figure 2, and produce a CCD, such as that of Figure 1, which concisely summarizes the intercomponent dependencies of interest? The purpose of this document is to demonstrate that process, which could be automated by a tool. Many tools of this sort are available to draw UML class diagrams, for example—another notation for depicting dependency information.

There are six steps involved in drawing a CCD from Resolve/C++ code, of which only the **highlighted** portion is relevant from the standpoint of a CCD. These steps are illustrated below in piece-by-piece construction of the example CCD of Figure 1, which shows all the dependencies to all the components that you need to understand in order to understand *Queue_Sort_1*.

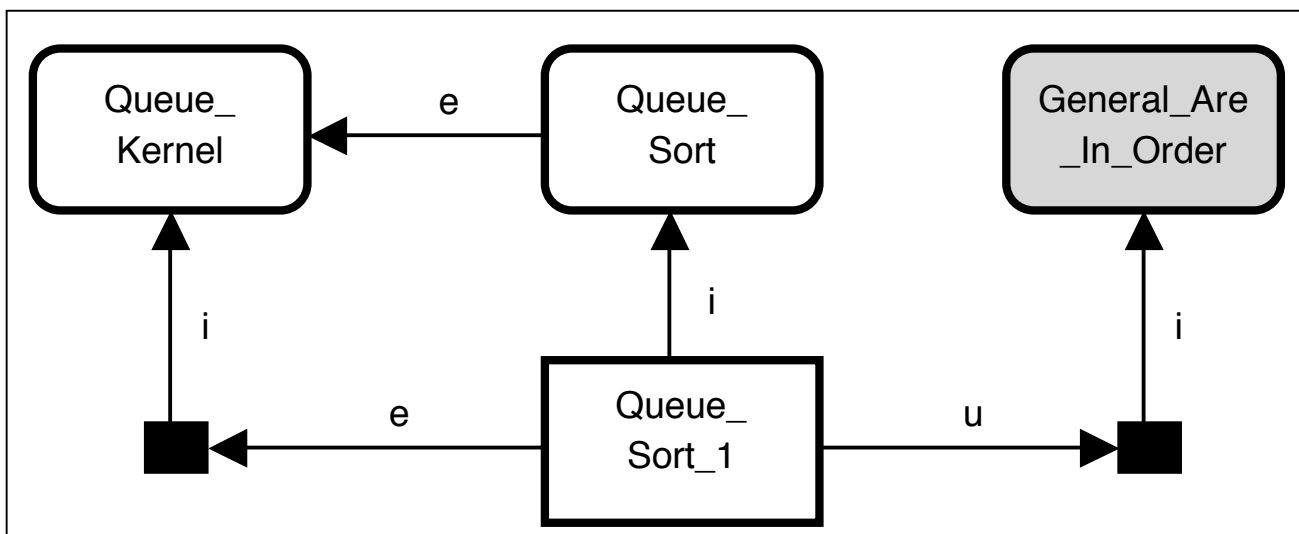


Figure 1: CCD showing dependencies from *Queue_Sort_1*

```
// /*-----*\
// | Concrete Template : Queue_Sort_1
// \*-----*/

#ifndef CT_QUEUE_SORT_1
#define CT_QUEUE_SORT_1 1

///-----
/// Global Context -----
///-----

#include "AT/Queue/Sort.h"
/*!
#include "AT/General/Are_In_Order.h"
#include "AT/Queue/Kernel.h"
```

```

!*/
///-----
/// Interface -----
///-----

concrete_template <
  concrete_instance class Item,
  concrete_instance utility_class Item_Are_In_Order,
  !*/
    implements
      abstract_instance General_Are_In_Order <Item>
  !*/
  concrete_instance class Queue_Base
  !*/
    implements
      abstract_instance Queue_Kernel <Item>
  !*/
  >
class Queue_Sort_1 :
  implements
    abstract_instance Queue_Sort <Item>,
  extends
    concrete_instance Queue_Base
{
private:

  local_procedure_body Remove_Min (
    alters Queue_Sort_1& q,
    produces Item& min
  )
  !*/
  requires
    q /= empty_string
  ensures
    (q * <min>) is permutation of #q and
    for all x: Item
      where (x is in elements (q))
        (ARE_IN_ORDER (min, x))
  !*/
  {
    object catalyst Queue_Sort_1 temp;

    q.Dequeue (min);
    while (q.Length () > 0)
    !*/
      alters q, min, temp
      maintains
        (temp * q * <min>) is permutation of
          (#temp * #q * <#min>) and
        for all x: Item
          where (x is in elements (temp))
            (ARE_IN_ORDER (min, x))
      decreases
        |q|
    !*/
  }
}

```

```

    {
        object catalyst Item x;

        q.Dequeue (x);
        if (Item_Are_In_Order::Are_In_Order (x, min))
        {
            min &= x;
        }
        temp.Enqueue (x);
    }
    q &= temp;
}

public:

procedure_body Sort ()
{
    object catalyst Queue_Sort_1 sorted;

    while (self.Length () > 0)
    /*!
        alters self, sorted
        maintains
            (self * sorted) is permutation of (#self * #sorted) and
            IS_ORDERED (sorted)
        decreases
            |self|
    !*/
    {
        object catalyst Item min;

        Remove_Min (self, min);
        sorted.Enqueue (min);
    }
    self &= sorted;
}

};

#endif // CT_QUEUE_SORT_1

```

Figure 2: Code for *Queue_Sort_1* in CT/Queue/Sort_1.h

Step 1

In the code, find the name of the component whose CCD you're drawing. This name is the identifier that appears in the code immediately after the keyword **class** or **utility_class**. Then determine which kind of component it is by looking for the keyword **abstract_instance** or **concrete_instance** immediately before **class** or **utility_class**, or for the keyword **abstract_template** or **concrete_template** immediately before the template's formal parameter list. Draw a box in the CCD for this component, using the kind of component to tell you what characteristics the box should have:

- rounded corners for an abstract component vs. square corners for a concrete component;
- thin sides for an instance, thick sides for a template; and
- no shading for a regular **class**, shaded gray for a **utility_class**.

Write the name of the component inside this box.

For example, in the code of Figure 2 it is immediately evident from the opening comment (not to mention from the code) that the component name is "Queue_Sort_1" and that this component is a **concrete_template class**. You should therefore draw the following figure to begin the CCD.

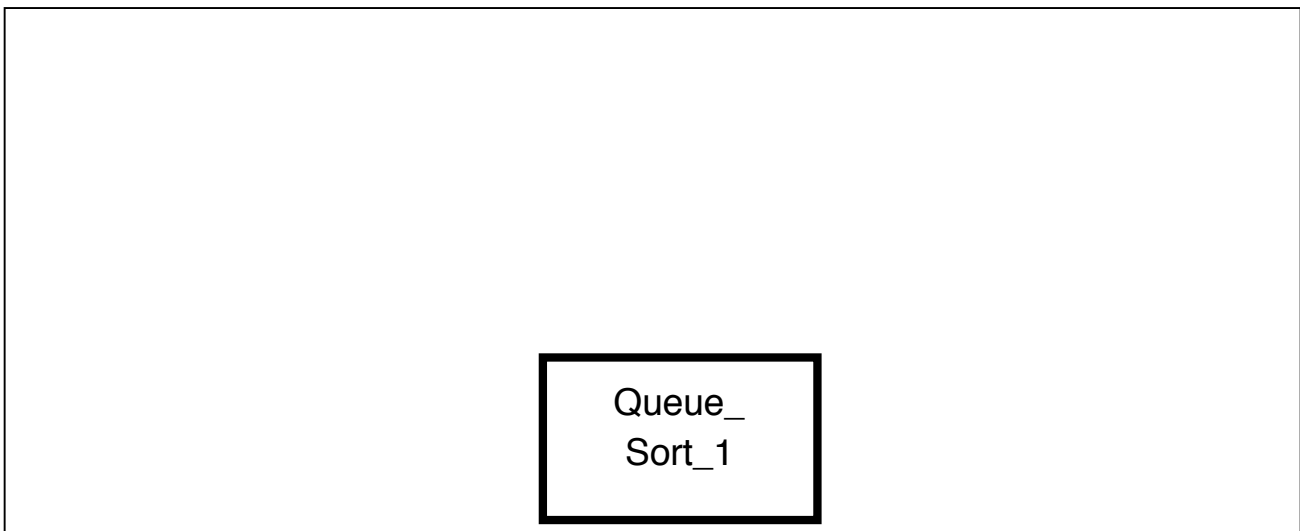


Figure 3: CCD after Step 1

Step 2

Find all the other components that this component directly depends on by examining the “Global Context” section of the code. *Recursively* draw the complete CCD for every component that appears in a **#include** statement — even for every component that appears in a **#include** statement inside a formal comment.

For example, the “Global Context” section of the code in Figure 2 is:

```
#include "AT/Queue/Sort.h"  
/*!  
    #include "AT/General/Are_In_Order.h"  
    #include "AT/Queue/Kernel.h"  
!*/
```

The components defined in these files are all abstract components: *Queue_Sort*, *General_Are_In_Order*, and *Queue_Kernel*. You may examine these files in any order; the order illustrated in the sequence of figures below is the order of inclusion. If you follow the six-step process for drawing a CCD from code for each of these three components, you will first have Figure 4a, then Figure 4b.

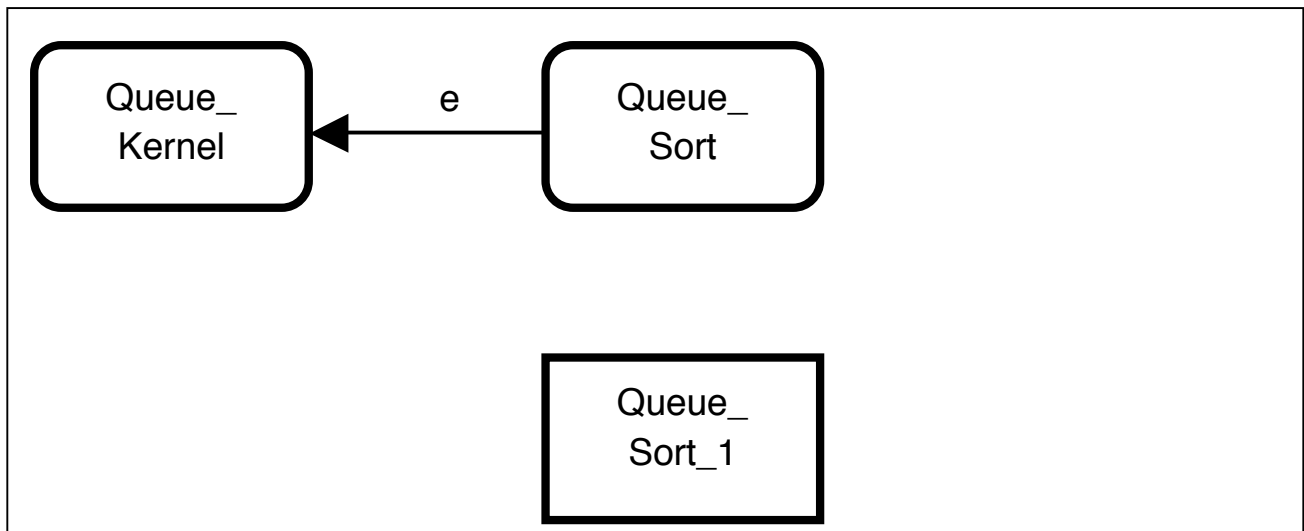


Figure 4a: CCD after adding *Queue_Sort* CCD

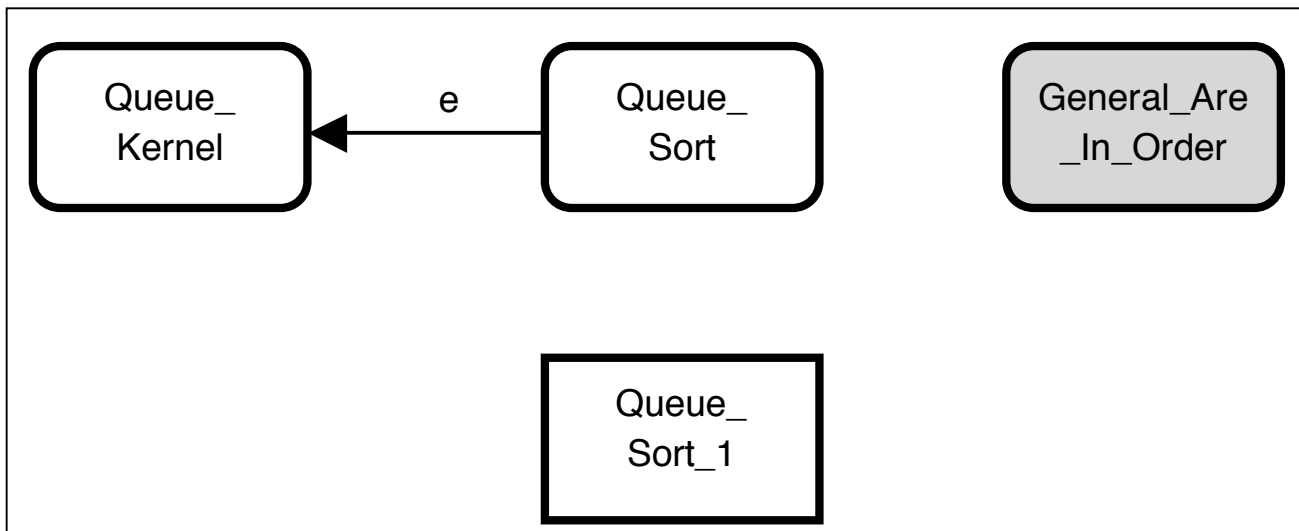


Figure 4b: CCD after adding *General_Are_In_Order* CCD

One question you might have at this point is simply, “How do I know *where* in the figure to draw these things?” There are three answers to this:

- You can get a general idea of where to draw things by looking at a CCD for a similar kind of component. Here, for instance, you’re drawing the CCD for a concrete component that implements an extension, and the general pattern for this kind of CCD is a square arrangement in which the component for which the CCD is being created is in the lower right corner. (You can already see three of the four corners of this square, and one of its sides, in Figure 4b.)
- As a general rule, if you’re drawing a CCD for a concrete component, the abstract components it depends on should generally be drawn above and/or below it based on the relationships: *implements* arrows generally go up, and other dependencies go across or down.
- Don’t worry about it yet! Draw the complete CCD in draft form and then redraw it so it looks nice.

The last four steps of the process are based on the following taxonomy of component dependencies, and on the terminology shown in the cells.

<p>Taxonomy of component dependencies</p>	<p><i>direct</i> — the second component to the relationship is a <i>component identified in Step 2</i></p>	<p><i>indirect</i> — the second component to the relationship is a <i>template parameter</i> that is restricted to implement a component identified in Step 2</p>
<p><i>explicit</i> — a component relationship keyword <i>appears in the code</i> after the semicolon that comes immediately after the component name identified in Step 1</p>	<p><i>explicit</i> <i>direct</i> <i>dependency</i> (Step 3)</p>	<p><i>explicit</i> <i>indirect</i> <i>dependency</i> (Step 4)</p>
<p><i>implicit</i> — a component relationship keyword <i>does not appear in the code</i>, so the relationship is uses</p>	<p><i>implicit</i> <i>direct</i> <i>dependency</i> (Step 5)</p>	<p><i>implicit</i> <i>indirect</i> <i>dependency</i> (Step 6)</p>

Step 3

Find each *explicit direct dependency* between the original component (identified in Step 1) and a component it depends on (identified in Step 2). For each explicit direct dependency, draw an arrow between the two components and label it with their relationship.

For example, in the code of Figure 2 there is one explicit direct dependency: *Queue_Sort_1* **implements** (some instance of) *Queue_Sort*. The other explicit dependency here is that *Queue_Sort_1* **extends** *Queue_Base*. But *Queue_Base* is a template parameter, not a component identified in Step 2; this situation is handled in Step 4.

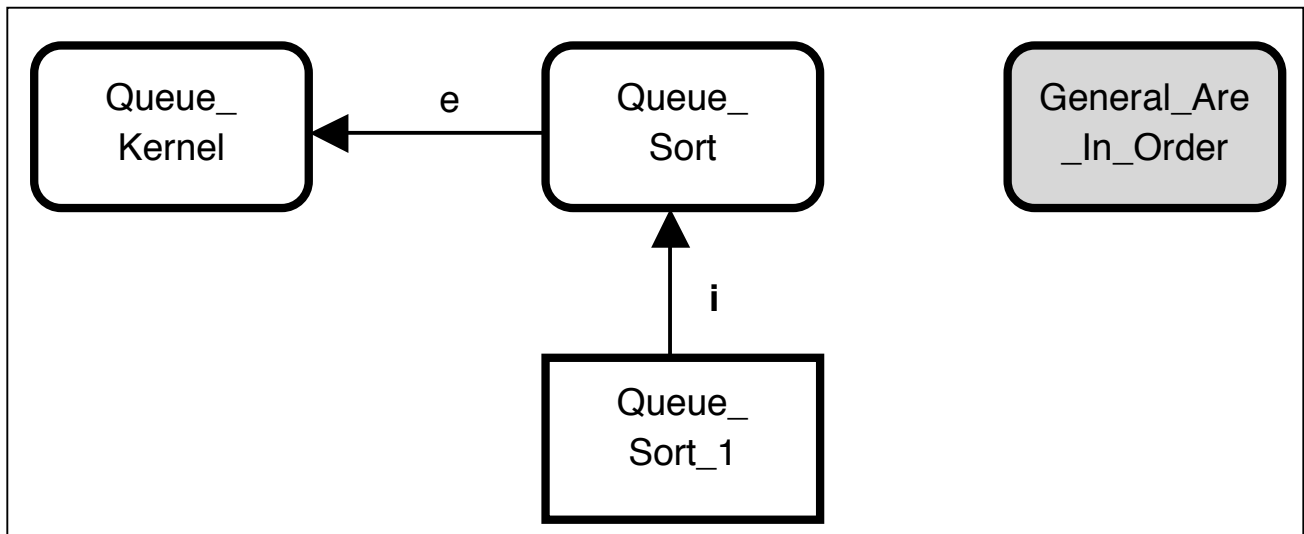


Figure 5: CCD after adding explicit direct dependencies

Step 4

Find each *explicit indirect dependency* between the original component (identified in Step 1) and a component it depends on (identified in Step 2). For each explicit indirect dependency, draw a small black rectangle to denote the template parameter (which, it turns out, is always a concrete component). Draw an arrow between the original component and the small black rectangle and label it with their relationship. Then draw arrows between the small black rectangle and any components it is restricted to implement in the template's formal parameter list. Finally, label these arrows **implements** (or **i** for short).

For example, in the code of Figure 2 there is one explicit indirect dependency: *Queue_Sort_1* **extends** *Queue_Base*. In the CCD, this relationship may be shorted to **e**. Looking back to the template parameter list for *Queue_Base*, you can see that the client must supply as an actual parameter a component that **implements** (some instance of) *Queue_Kernel*. This is the restriction on *Queue_Base* that appears in the CCD.

There is no need to write in names for the small black rectangles that stand for template parameters, since these names would be those of the formal parameters, which are entirely arbitrary. However, if it helps you connect the CCD to the code, you *may* write these names in the CCD.

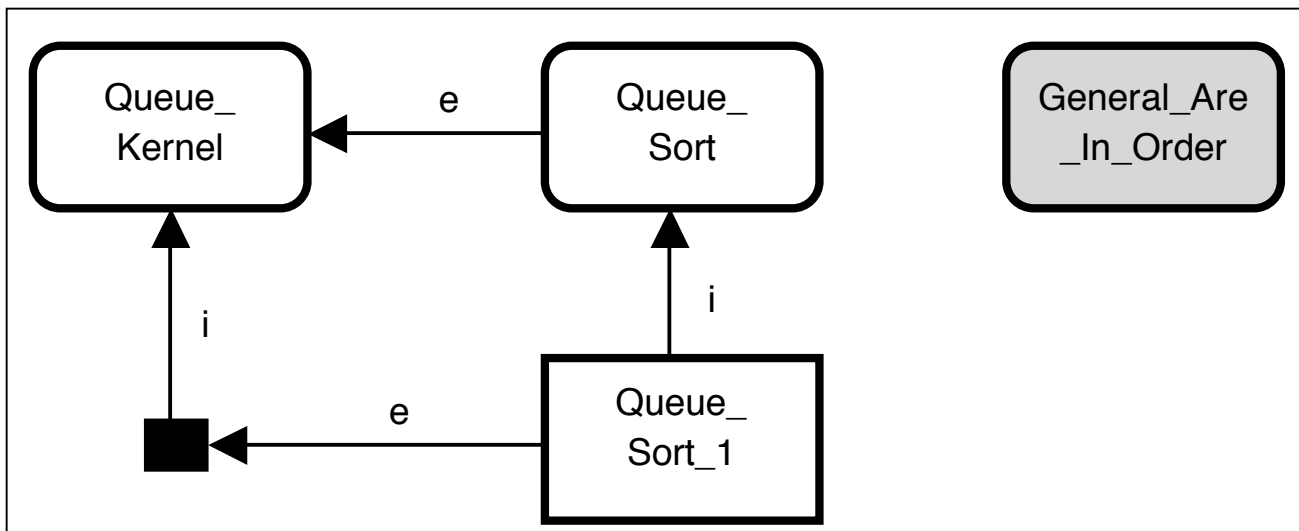


Figure 6: CCD after adding explicit indirect dependencies

Step 5

Find each *implicit direct dependency* between the original component (identified in Step 1) and a component it depends on (identified in Step 2). For each implicit direct dependency, draw an arrow between the two components and label it with the relationship **uses**.

In the code of Figure 2 there are no implicit direct dependencies. There is, however, one implicit indirect dependency; see Step 6.

Step 6

Find each *implicit indirect dependency* between the original component (identified in Step 1) and a component it depends on (identified in Step 2). For each implicit indirect dependency, draw a small black rectangle to denote the template parameter (which, it turns out, is always a concrete component). Draw an arrow between the original component and the small black rectangle and label it with the relationship **uses** (or **u** for short). Then draw arrows between the small black rectangle and any components it is restricted to implement in the template's formal parameter list. Finally, label these arrows **implements** (or **i** for short).

For example, in the code of Figure 2 there is one implicit indirect dependency: *Queue_Sort_1* **uses** *Item_Are_In_Order*. There is a dependency here because in the template parameter list for *Item_Are_In_Order* you can see that the client must supply as an actual parameter a component which **implements** (some instance of) *General_Are_In_Order*. This is the restriction on *Item_Are_In_Order* that appears in the CCD.

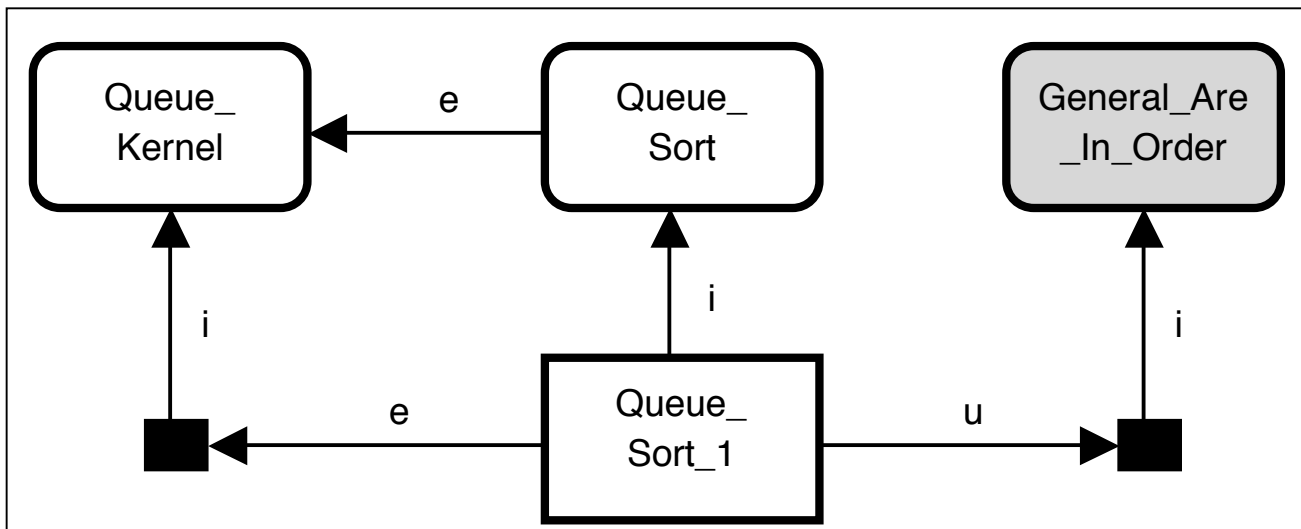


Figure 7: CCD after adding implicit indirect dependencies (final)

Notice that not all template parameters appear in the CCD. In particular, the formal template parameter *Item* does not introduce any dependencies between *Queue_Sort_1* and any other component. There is no small black rectangle in the CCD that corresponds to *Item* because there is no restriction on which **concrete_instance** classes might be supplied as the associated actual parameter.