Descriptions for Figures (to illustrate our GUI-API)

Terminology used for describing the figures:

CCG and **CGM**: Component Code Generator (or CCG) is a class or an object implemented for generating code for a component. Code Generation Method (or CGM) is a virtual function implemented in each of the CCG. This method or function is called to generate presentation code for a component, after properly initializing object instance of a CCG-object.

ACi is Object instance of **AC_Info** Class: An object comprising application context and other information such as user profile, browser type and version. This object is properly initialized at the beginning and passed to each of the CCGs. Each CCG can use all this information ACi to generate code.

GUI-CCG: Reusable Component Code Generator for a GUI-component such as pie-chart or line-chart, and so on. That is, a reusable GUI-CCG class is implemented to generate presentation code for pie-chart; another reusable GUI-CCG class is implemented to generate presentation code for line-chart; and so on. The reusable GUI-CCGs only implement presentation logic. To present a GUI-component, the application instantiates an object instance of an appropriate reusable GUI-CCG class and initializes the object instance, by inputting all the application data and configuration data into the object. See FIG-1

ACCG or **RCC** (Replaceable Component Class): Application Component Code Generator (ACCG) generates presentation code for a component in an application (or AC-Application Component). Each ACCG implements all the necessary application code and uses one or more GUI-CCGs for generating presentation code for one or more GUI-components. An object instance of RCC is referred to as RSCC (Replaceable Self-Contained Component).

Brief Description for each of the figures (FIG-1 to FIG-1):

FIG-1, illustrates application code and a pie-chart for presenting portfolio of a customer. The pseudo shown section 106 generates SVG/JavaScript code for presenting the pie chart 104 shown. The application logic instantiates an object instance at line 4 by using a reusable CCG-class for pie-chart for presenting the GUI-component 104. Assume that the pseudo code in section-106 implements application logic to access data at run-time, process the data (e.g. by iterating over raw data) for using the data to initialize the object instance pie-chart GUI-components at line 16. Once the object instance of pie-chart is fully initialized and configured, it can be requested to generate browser compatible SVG/JavaScript (e.g. by inputting browser type and version at run-time to the CCG-object) or attached to parent for parent to request generation of the code.

FIG-2 provides another perspective for the pie-chart in FIG-1 for presenting portfolio of a customer. The code described in the FIG-1 is encapsulated in a custom CCG-class 200, referred to a Replaceable Component (or Replaceable Self-Contained Component) class. This RCC (Replaceable Component Class) implements application logic and uses a reusable GUI-CCG class for pie-chart to present portfolio pie-chart of a customer (whose account number is passed at run-time). This RCC can be used to present this GUI-component 209 in any application by implementing the few lines in section 207 (by imputing the account number of the user such as 40021729).

FIG-3 shows a flow chart for CGM of a container component (e.g. RCC or ACCG) that uses multiple GUI-CCGs. The ACCG of the container component uses multiple reusable GUI-CCGs and implements application logic (e.g. for accessing data, processing data and using data) to initialize each of the GUI-CCGs by using application data. For example, code sections 301 or 302 instantiates an object instance of a reusable GUI-CCG and initializes the object instance of the GUI-CCG by using application data. Please refer to FIG-1 and

FIG-2, which illustrates code for application logic code for using each GUI-CCG for presenting a GUI-component. If any two GUI-components in the ACCG need to collaborate with each other, the ACCG also implements necessary communication code. For example, code section 303 generates necessary communication code to allow communication between CCG1 and CCG2. Likewise, the code section 306 for generating necessary communication code to allow communication between CCG3 and CCG4.

FIG-4 shows an ACCG (or a RSCC – Replaceable Self-Contained Component) that uses multiple reusable GUI-CCGs for presenting GUI-components such as a line-chart 421, a table 425 and a bar-chart 422. The ACCG implements application logic and uses multiple instances of reusable GUI-CCG objects (e.g. 401, 402 and 408) for presenting multiple GUI-components. Please refer to FIG-1 and FIG-2, which illustrates code for application logic and code for using each GUI-CCG for presenting a GUI-component. For an example, please refer to flaw chart in FIG-4 for steps to implement CGM of an ACCG.

FIG-5 shows a CGM of a container-ACCG (or a RSCC – Replaceable Self-Contained Component) that uses multiple other ACCGs (501, 502 and 503) for presenting subcomponents (511, 512 and 513). Since each of the RCCs or ACCGs (or RCC) for subcomponents already implemented all the application logic, it is not necessary for the CGM to implement any application logic. If the subcomponents need to collaborate with each other, it is necessary for the CGM to generate necessary communication code to allow collaboration between the subcomponents (see FIG-6 below).

FIG-6 shows a flow chart of steps implemented in a CGM of a container-ACCG (e.g. see FIG-5) comprising subcomponent, where each subcomponent is included by using an ACCG. It requires 2-steps to include each subcomponent (1) instating and initializing an object instance of a RCC as shown in 601 or 603 and (2) attaching the subcomponent to the container component at proper place

as shown in 602 and 604. If the subcomponents need to collaborate with each other, it is necessary for the CGM to generate necessary communication code to allow collaboration between the subcomponents as in steps 607 or 609.

FIG-7 shows another perspective for the FIG-5, where each of the ACCGs (e.g. 721, 722 or 723) encapsulates application logic and presentation logic (e.g. by using GUI-CCGs) for presenting a subcomponent (731, 732 or 733) respectively. The container-ACCG 720 just implements few lines of code to assemble each of the subcomponents for the container-component 730.

FIG-8a illustrates a container-CCG 820 that uses two CCGs 821 and 822 for generating two subcomponents 824 and 825 respectively. If subcomponents 824 and 825 need to collaborate with each other, how is it possible to include necessary communication code in the code of container-component 823?

FIG-8b illustrates a container-CCG 830 that uses two CCGs, which are CCG1 831 and CCG2 832 for generating two subcomponents AC1 835 and AC2 836 respectively. If subcomponents AC1 and AC2 need to collaborate with each other, the container-CCG 830 implements a special code ‘integration-logic’ that interacts with CCG-objects CCG1 831 and CCG2 832 for generating necessary communication code. In general, one component (e.g. AC2) needs to call a function (e.g. AC1_service_func of AC1) of another component upon a given event. That is, a first component AC2 initiates communication upon an occurrence of a given event. To allow this collaboration, the integration logic implements a call-back function for calling the service function of AC1 and registers the call-back with AC2. Up on the event, AC2 calls the call-back that in-turn calls the function of the AC1.

FIG-9 illustrates the integration-logic 952 that generates communication code 957 to allow communication between two subcomponents AC1 956 and AC2 958. The communication code 957 is a callback function that calls service

function of AC1. The callback function is registered with AC2 by using CCG2. Upon a given event AC2 calls the callback, which in turn calls the function of AC1. To accomplish this CCG1 is designed to return a function-names implemented in AC1, where code for AC1 is generated by CCG1; and CCG2 is designed to register a callback that can be called by AC2 upon a given event, where code for AC2 is generated by CCG2.

Although at first glance, the process to generate communication code is confusing or complex, it is always generating 3 to 5 lines of similar code by following 3 simple steps. The three simple steps to generate communication code between any two components are (1) implementing a simple callback function in the code of application (2) to call a function implemented in a first component by getting the function name by using CCG of the first component, and (3) registering the callback with a second component by using CCG of the second component. This method for generating communication code become simpler with little practice, since it is always following the 3 simple steps to generate 3 to 5 lines of code.

FIG-10 shows a component-hierarchy 1040 generated comprising multiple ACs (i.e. AC1 to AC20) by a server application 1000, by using multiple RCCs (i.e. RCC-01 to RCC-20) respectively. In this component-hierarchy, each container RCC (e.g. RCC-12) uses other RCCs (e.g. RCCs 8, 9 & 10) for generating subcomponents, where the container-CCGs (e.g. RCC-12, RCC-15 or RCC-07) are in turn used by other container-CCGs for generating subcomponents. This kind of hierarchy can go to a tree of any length and width. For example, RCC-20 can be used to generate a subcomponent of another RCC. The application grows and evolves by adding more components (or branches) by using new RCCs (or container-RCCs) and by redesigning each of the components (i.e. RCCs as illustrated in sample City-GIS application).