Builder's Guide for WaterBeans Components

Daniel Plakosh
Dennis Smith
Kurt C. Wallnau

December 1999

TECHNICAL REPORT
CMU/SEI-99-TR-024
ESC-TR-99-024

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Builder's Guide for WaterBeans Components

CMU/SEI-99-TR-024
ESC-TR-99-024

Daniel Plakosh
Dennis Smith
Kurt C. Wallnau

December 1999

Dynamic Systems

Unlimited distribution subject to the copyright.

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Norton L Compton

Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

This work is sponsored by the Environmental Protection Agency and the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 1999 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Preface

This paper describes WaterBeans, a proof-of-feasibility system for building software applications through a process of assembling prefabricated software components. WaterBeans was originally developed for the U.S. Environmental Protection Agency (EPA) as a proof of feasibility that software component technology could be used to develop software applications in the domain of water-quality modeling. This paper builds on the original WaterBeans contribution. It documents the programming interface for component developers and provides a brief description of the composition environment.

Abstract

WaterBeans is a proof-of-feasibility system for building software applications through a process of assembling (composing) prefabricated software components. WaterBeans was originally developed as a proof of feasibility that software component technology could be used to develop software applications in the domain of water-quality modeling. (In particular, WaterBeans supports modeling and simulating the hydrology of urban storm water sewage and runoff.) WaterBeans includes a component model for component developers, a visual composition environment for importing and assembling components into applications, and several families of components. One family of components supports modeling and simulating urban sewage systems. Another family of components was developed to prove the generality of WaterBeans; this family of components allows visualization and manipulation of digital waveforms. This report documents the programming interface for component developers. It also provides a brief description of the composition environment.

1 Background

Today most software applications for water-quality modeling are monolithic applications, consisting of computational modeling engines that are tightly coupled to user interfaces, data formats, vendor-specific compiler extensions, operating systems, and hardware. Typically the user interfaces provided with the modeling application are simple, somewhat unfriendly, and decades behind the current practice in graphical user interface technology and data visualization. Due to the lack of sophistication in these user interfaces, users are becoming dissatisfied with this software and are looking elsewhere (mainly overseas) for software to solve their water-quality modeling needs. This recent trend has resulted in a high demand for water-quality modeling applications with advanced user interfaces and sophisticated graphics presentation packages. Because of the difficulty associated with extracting the computational engines from the water-quality modeling software applications, software developers that are now building products to meet this demand are reluctant to use existing modeling engines in their new applications. Instead, developers have chosen to develop new modeling engines. These new mathematical modeling engines are often proprietary and have not undergone the extensive peer review that is usually required to validate a model. The water-quality simulation community and the municipalities that they work for are willing to rely on these proprietary engines because the applications

- are user friendly and much easier to use due to the advanced user interfaces
- provide sophisticated graphics with real-time simulations that are often desirable for use in community presentations (High-tech data visualization displays are often used to help convince the public that the correct choices are being made.)
- are better suited for the newer operating systems and include user support

The EPA recognized that it needed to change its software development approach to meet the needs of the users and software developers who desire to market products that provide enhanced graphical user interfaces (GUIs) and visualization tools but still use standard modeling engines. It was felt that a solution to this software dilemma was the adoption of a commercial component specification for building modeling engines. Perhaps ActiveX, Java Beans, or some other component specification would be adequate. After further investigation, it was determined that none of the commercial component technologies met the water-quality community's unique software requirements. This realization resulted in the development of a customized component specification. It is expected that this new approach will make future water-quality modeling software reusable and attractive to both end users and software developers.

2 Introduction

When developing any type of software specification, it is desirable to also build a prototype implementation that conforms to the specification. The prototype is intended to serve as a proof of concept that validates the specification. Additionally, the implementation aids in the refinement of the specification by applying the lessons learned during the development phase. Typically, the development of a good specification is an iterative process, where the specification evolves as a result of modifications to the specification as well as the reference prototype. The importance of a prototype implementation can not be overstated. In the past, many have tried to develop specifications without expending the additional resources required for a prototype implementation, and these efforts have often resulted in specifications that can not be implemented or are unacceptable to the community that they were intended to serve.

Naturally, we recognized that the successful development of a component specification would require a prototype implementation. This prototype included the development of sample components that conform to the initial component specification and an integration framework application to test and exercise the components. We decided that the integration framework should be generic and not component specific. This decision resulted in a generic component integration framework that can be viewed as a visual programming environment (VPE) or meta-application for building applications from components. We felt that if the components worked well in a very generic application, then the components would probably work well in a custom-built modeling application.

This document contains the programmer's guide for the first iteration of the custom component model that was developed. It describes the details of the component model along with the visual programming environment. Using this document, a programmer should be able to create components for use within the VPE or a custom modeling application.

The rest of this report is organized of as follows: Section 3 describes the motivation behind the custom component model, while Section 4 describes details of the component model. The visual programming environment is described in Section 5. Finally, we present our summary in Section 6.

3 Motivation for the Custom Component Model

Software development within the water-quality community is much different from development in other organizations because most of the software is developed by scientists and engineers with no formal background in computer science. The typical water-quality software package is written in the FORTRAN programming language using primitive software design techniques and without the use of any type of coding standards or implementation guidelines. Given the state of software development within the community, the SEI determined that any proposed component model had to be simple and easily implemented by a typical software developer.

Based upon the current water-quality software development practices and the desire to develop reusable software that is attractive to both end users and developers, it was determined that an acceptable component model must have the following qualities:

- **compatible with PC x86 Microsoft Windows** – The majority of water-quality modeling is performed on personal computers using Windows 95, Windows 98, and Windows NT.
- **language independent** – The component model should allow components to be developed using different programming languages such as FORTRAN, C, and C++.
- **ability to support introspection** – Components should be capable of providing information that describes itself at runtime.
- **usable to both naïve end users and application developers** – Components should be usable by programmers and by end users in some type of visual programming environment or as an application plug-in.
- **no distributed computing support** – Components need to support only a single user, without any remote communication.
- **ability to support simulation** – The component coordination model needs to support water-quality simulations.
- **ability to contain components from other component frameworks** – EPA modeling components should be able contain components from other component frameworks such as ActiveX or Borland's Visual Component Library (VCL). Additionally it should be possible to incorporate EPA components in other component frameworks.
- **"simple" application programmer interface (API) and component model** – The component model and API had to be simple due to the nature of most component developers and end users.

- **stimulate component market** – The model should encourage the development of components and stimulate the component market.
- **low overhead** – The component model must be lightweight to support high-performance simulations.

Using the above desired component qualities as a guide, an initial component specification to support components for modeling water quality was developed.

As this specification evolves, it could offer a standard architecture framework for modeling urban water quality, which could act as a counterbalance to the emergence of a closed, proprietary software solution as a potential de facto industry standard.

It is expected that this specification will prove the feasibility of using a component-based approach to urban runoff modeling that will

- enable the development of alternative conformant implementations of a standard component model (i.e., components and infrastructure)
- define an application-specific component model for water-quality modeling
- enable third-party integration of water-quality software components
- facilitate a software component marketplace for water-quality modeling

4 Component Specification

This section describes the custom component specification called “WaterBeans” that was developed for water-quality modeling components. Although this specification was developed to support a specific environment, it is still robust enough to be used in many types of applications.

4.1 Fundamentals

The WaterBeans component model is quite different when compared to other component models such as the Visual Component Library (VCL), Java Beans, or ActiveX. The WaterBeans component model is based upon the following concepts:

- The component model is intended to support components such as computational engines, graphical data visualization interfaces, and data input and management interfaces. These types of components can be viewed as being on the level of a plug-in. This component model is not suited for low-level components such as buttons, graphs, etc.
- Components export only a standard set of functions. These functions make a standard component interface. Most component models allow developers to define their own methods, which results in component-specific APIs as well as the semantics that go along with them.
- Communication with the component is achieved through streams-oriented input and output interfaces. A component may export up to 32 separate input interfaces and output interfaces. Each interface is strongly typed and must adhere to a developer-defined data-interface specification. This approach is quite uncommon with the commercial component models, where data exchange and control are achieved through component-specific methods.
- The component model supports developer-defined properties and developer-defined property editors. Properties are similar to variables. An application at runtime can read the value of a property and change the value of the property if it is writeable. Properties allow a component to be customized at runtime and provide important information to an application.

  The following property types are supported:

  – ASCII string
  – 32-bit integer
  – 32-bit float
  – Boolean
- A component can describe itself at runtime to an application. Introspection functions allow an application to determine the following information about a component at runtime:

    - description of a component
    - detailed information about each input/output interface of the component
    - detailed information about each property that the component exports
- The component coordination/execution model is based upon data-driven semantics.
- The component supports user-defined graphical displays.
- The component is intended for use with Microsoft Windows (WIN32) systems such as Windows 95, Windows 98, and Windows NT.

A component is physically represented as a dynamic load library (commonly referred to as a DLL). Each component is required to export the standard set of functions commonly referred to as a component interface. The developer's conformance to the component specification is what separates a component from a DLL. The appendix contains the header file to define or use a WaterBeans component within the C programming language.

An application must load components using *runtime-dynamic linking*. In *runtime-dynamic linking,* the application loads a component using Microsoft® Windows® *LoadLibrary()* function. Next, the application calls the Microsoft® Windows® *GetProcAddress()*, which returns a function pointer for the specified function.

4.2 Common Data Types

Each component must declare the data types and defined values that are shown in Table 1. This information is for use in the standard set of exported component functions.

Defined Data Types and Constants	**Description**
`enum TCPropertyTypes {ptString=0,ptFloat=1,ptInteger=2,ptBool=3 };`	Enumerated list of property types that can be exported by a component
`#define COMPFALSE 0`	Value of false for the CompBool data type
`#define COMPTRUE  1`	Value of true for the CompBool data type
`typedef void * TcomponentInstance;`	Pointer to an instance of a component
`typedef int    CompBool;`	Boolean value
`typedef void * TParentWindow;`	Pointer to the parent window of a component

Table 1: Common Data Types and Constants

4.3 Exported Functions

Each component must export the standard set of functions. These functions can be grouped into the following categories:

- introspection
- data transfer
- persistance
- property manipulation
- control

4.3.1 Introspection

4.3.1.1 GetComponentInfo

The `GetComponentInfo` function retrieves general information about the component. Typically, this information is needed by an application immediately after a component is loaded.

Export Definition

```
extern "C" __declspec(dllexport)
  TComponentInfo * __stdcall GetComponentInfo(void);
```

Parameters

None

Return Values

If the function succeeds, the return value is a pointer to the structure `TComponentInfo` shown in Figure 1.

If the function fails, the return value is NULL.

```
typedef struct
{
 char      * Name;
 char      * Category;
 char      * Hint;
 char      * VendorName;
 char      * VendorContactNumber;
 char      * VendorWebAddr;
 char      * IconResourceName;
 int         MajorVersion;
 int         MinorVersion;
 int         NumInputs;
 int         NumOutputs;
 CompBool    PropertyEditor;
 CompBool    OutputDisplay;
 CompBool    Runable;
} TComponentInfo;
```

Figure 1: TComponentInfo Structure

The definition of each field in the TComponentInfo structure is shown in Table 2.

Field	**Definition**
`Category`	An ASCII string that describes the category of the component. Components are grouped into categories that are somewhat ad hoc in nature and are defined by the component designer. An example component category could be “Hydraulic Modeling Engines” to indicate that the component is of a particular type.
`Name`	An ASCII string that indicates the name of the component. This name should be descriptive enough for the user to recognize it from other components. Different components can not have the same name and category.
`Hint`	An ASCII string to provide amplifying information about the component. This information is intended for use in visual environments.
`VendorName`	An ASCII string that indicates the component developer.
`VendorContact-Number`	An ASCII string that indicates the telephone number of the component developer.
`IconResource-Name`	An ASII string that identifies the resource name of the icon embedded in the component. An icon is a graphical representation of the component for use in a visual environment.
`MajorVersion`	A 32-bit integer that represents the major version number of the component.
`MinorVersion`	A 32-bit integer that represents the minor version number of the component.
`NumInputs`	A 32-bit integer that represents the number of input interfaces. Each input interface is defined by its name and type.
`NumOutputs`	A 32-bit integer that represents the number of output interfaces. Each output interface is defined by its name and type.
`PropertyEditor`	A Boolean that indicates the presence of a graphical property editor.
`OutputDisplay`	A Boolean that indicates the presence of a graphical output display.
`Runable`	A Boolean that indicates if the component has an execute function. Some components may compute data on demand only through a data request on an interface. This type of interface is called a passive interface.

Table 2: Description of TComponentInfo Fields

4.3.1.2 GetInputInfo and GetOutputInfo

`GetInputInfo()` and `GetOutputInfo()` provide information that describes a particular interface. Each interface has a data type and name associated with it.

Export Definition

```
extern "C" __declspec(dllexport)
  TIOInfo * __stdcall GetInputInfo(
      int InterfaceIndex);
```

And

```
extern "C" __declspec(dllexport)
  TIOInfo * __stdcall GetOutputInfo(
      int InterfaceIndex);
```

Parameters

`InterfaceIndex`: The index number of the input or output interface. This index can be determined through the function call `GetComponentInfo()` where the value of fields `NumInputs` and `NumOutput` are returned in the reference to the `TGetComponentInfo` structure. The indexes are zero-based; that is, the first interface has an index of zero, the next has an index of one, and so on.

Return Values

If the function succeeds, the return value is a pointer to the structure `TIOInfo` shown in Figure 2.

If the function fails, the return value is `NULL`.

```
typedef struct
{
 char      * InterfaceName;
 char      * InterfaceType;
 CompBool    Required;
 CompBool    Active;
} TIOInfo;
```

Figure 2: TIOInfo Structure

The definition of each field in the TIOInfo structure is shown in Table 3.

Field	**Description**
`InterfaceName`	An ASCII string that represents the name of the interface.
`InterfaceType`	An ASCII string that represents the particular type of data that passed over the interface.
`Required`	A Boolean that indicates if the data are required or optional for proper execution of the component. This field typically pertains only to input interfaces.
`Active`	A Boolean that indicates if the interface is active or passive. This field pertains only to output interfaces.

Table 3: Description of TIOInfo Fields

4.3.1.3 GetNumPropertiesC

The function call `GetNumPropertiesC()` returns the number of properties exported by the component.

Export Definition

```
extern "C" __declspec(dllexport)
  int  __stdcall GetNumPropertiesC(
       TComponentInstance Instance);
```

Parameters

`Instance`: Particular instance of the component. This function can be called only after the `GetNewInstance()` function.

Return Values

If the function succeeds, the return value is the number of properties exported by the component.

If the function fails, the return value is less than zero.

4.3.1.4 GetPropertyDescriptionC

The function call `GetPropertyDescriptionC()` returns information about a particular component property.

Export Definition

```
extern "C" __declspec(dllexport)
  int __stdcall GetPropertyDescriptionC(
       TcomponentInstance         Instance,
       Int                        PropertyIndex,
       TCPropertyDescription *    Description);
```

Parameters

`Instance`: Instance of the component. This function can be called only after the `GetNewInstance()` function.

`PropertyIndex:` Index of the particular property. The number of properties is determined through the `GetNumPropertiesC()` function. The indexes are zero-based; that is, the first property has an index of zero, the next has an index of one, and so on.

`Description`: A pointer to a TCPropertyDescription structure shown in Figure 3. This structure is filled upon successful completion of the function call.

```
typedef struct
{
 char              Name[256];
 TCPropertyTypes  Type;
 CompBool          ReadOnly;
}TCPropertyDescription;
```

Figure 3: TCPropertyDescription Structure

The definition of each field in the `TCPropertyDescription` structure is shown in Table 4.

Field	**Definition**
`Name`	An ASCII string that indicates the name of the property. Property names have a maximum length of 255 characters.
`Type`	An enumerated field that indicates that type of the property (integer, string, etc.). See Table 1 for more information.
`ReadOnly`	A Boolean field that indicates if the property is changeable.

Table 4: Definition of TCPropertyDescription Fields

4.3.2 Data Transfer

Data communication with a component is achieved through a component-defined streams interface where each interface is strongly typed. There are two types of output interfaces and one type of input interface. The first type of output interface is called a passive interface, where the component provides data as it is requested on the interface. For the requestor of the data, it is similar to reading an input file in which the component supplies the data only as it is requested. The second type of output interface is known as an active interface where data are generated by the component as it executes via the `Execute()` function. In this case, the data are accumulating for the reader. Input interfaces have read request just like any other stream interface.

4.3.2.1 SetInputCallBack and SetOutputCallBack

The `SetInputCallBack` and the `SetOutputCallBack` functions are used to specify the function to be called to get input data or put output data on a particular interface. Additionally, the `SetOutputCallBack` function is used only for the active type of output interfaces.

Export Definition

```
extern "C" __declspec(dllexport)
  int __stdcall SetInputCallBack(
       TcomponentInstance    Instance,
       int                   InterfaceIndex,
       TGetorPutDataCBInfo*  GetorPutDataCB);
```

And

```
extern "C" __declspec(dllexport)
  int __stdcall SetOutputCallBack(
       TcomponentInstance    Instance,
       int                   InterfaceIndex,

       TGetorPutDataCBInfo*  GetorPutDataCB);
```

Parameters

`Instance:` Particular instance of the component. This function can be called only after the `TGetNewInstance()` function.

`InterfaceIndex:` Index number of the interface. The number of input and output interfaces is determined through the function call `GetComponentInfo().` The values of `NumInputs` and `NumOutputs` indicate the number of input and output interfaces. The indexes are zero-based; that is, the first interface has an index of zero, the next has an index of one, and so on.

`GetorPutDataCB:` A pointer to a callback information structure (`TGetorPutDataCBInfo`) that specifies the callback used to input data or output data. The structure of the `TGetorPutDataCBInfo` is shown in Figure 4.

```
typedef struct
{
 TGetorPutData        CallBack;
 TComponentInstance   CBInstance;
} TGetorPutDataCBInfo;
```

Figure 4: TGetorPutDataCBInfo Structure

TGetorPutDataCBInfo Structure Fields

`Callback:` A pointer to the callback function that is defined as the following type:

```
typedef int __stdcall (*TGetorPutData)(
       TcomponentInstance  Instance,
       unsigned char *          Buffer,
       int                      Length);
```

Callback Parameters

`Instance`: This is set to the value of CBInstance defined in the `TGetorPutDataCBInfo` structure.

`Buffer:` A pointer to the buffer that is to be filled with data or output.

`Length:` Length of the buffer to be filled or written.

Callback Return Value

If the callback function succeeds, the value returned is the actual number of bytes read or written.

If the callback function fails, the return value is less than or equal to zero.

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is nonzero.

4.3.2.2 GetPassiveInputCallBack

The `GetPassiveInputCallBack()` function returns a pointer to the input function of a passive output interface. Output data from this interface are then obtained using this function. This function is used only with passive outputs.

Export Definition

```
extern "C" __declspec(dllexport)
  int __stdcall GetPassiveInputCallBack(
       TcomponentInstance    Instance,
       int                   InterfaceIndex,
       TGetorPutDataCBInfo * GetDataDB);
```

Parameters

`Instance:` Particular instance of a component. This function can be called only after the `TGetNewInstance()` function.

`InterfaceIndex:` Index number of the interface. The number of input and output interfaces is determined through the function call `GetComponentInfo()`. The values of `NumInputs` and `NumOutputs` indicate the number of input and output interfaces. The indexes are zero-based; that is, the first interface has an index of zero, the next has an index of one, and so on.

`GetorPutDataCB:` A pointer to the callback information structure (`TGetorPutDataCBInfo`) . This structure is filled upon successful completion of the operation. It then specifies the callback to be used to obtain the input data from a passive output interface. The structure of the `TGetorPutDataCBInfo` is shown in Figure 4.

TGetorPutDataCBInfo Structure Fields

`Callback:` A pointer to the callback function that is defined as the following type:

```
typedef int __stdcall (*TGetorPutData)(
           TcomponentInstance    Instance,
           unsigned char *       Buffer,
           int                   Length);
```

Callback Parameters

`Instance`: This is set to the value of CBInstance defined in the `TGetorPutDataCBInfo` structure.

`Buffer:` A pointer to the buffer to be filled.

`Length:` The length of the buffer.

Callback Return Value

If the callback function succeeds, the value returned is the actual number of bytes read or written.

If the callback function fails, the return value is less than or equal to zero.

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is nonzero.

4.3.2.3 GetInputRun Requirements

The `GetInputRunRequirements()` function returns the input interfaces that will require data for the next execution step of a component instance. This function should be called before calling the components `Execute`() function, described in Section 4.3.5.2. This function returns the input interface that must have data available.

Export Definition

```
extern "C" __declspec(dllexport)
  unsigned int __stdcall GetInputRunRequirements(
       TcomponentInstance Instance);
```

Parameters

`Instance:` Particular instance of a component. This function can be called only after the `GetNewInstance()` function.

Return Value

This function returns a bit string indicating the interfaces that will require input data. Bits 0 through 31 map directly to input interfaces 0 through 31 respectively.

4.3.3 Persistence

Persistence provides the ability to save and restore the state of a component. The persistence model requires only the two operations `GetPersistence()` and `PutPersistence()`.

4.3.3.1 GetPersistence

The GetPersistence function returns a buffer filled with the persistence data for an instance of a component. Persistence data include the value of each property that the component exports and the component’s state data. A `NULL` buffer pointer will cause this function to return the actual number of bytes required to store the persistence data.

Export Definition

```
extern "C" __declspec(dllexport)
  int __stdcall GetPersistance(
      TcomponentInstance    Instance,
      unsigned char *       Buffer,
      int                   Length);
```

Parameters

`Instance`: Particular instance of the component.

`Buffer:` A pointer to the buffer to be filled with the persistence data. A NULL pointer will return the actual number of bytes required to store the persistence data.

`Length:` The length of the buffer.

Return Value

If the function succeeds and the buffer pointer was non-zero, the return value is the actual number of bytes placed into the buffer

If the function succeeds and the buffer pointer was NULL, the return value is the size of the persistence data in bytes.

If the function fails, the return value is less than zero.

4.3.3.2 PutPersistence

The `PutPersistence()` function restores a component's state from the supplied persistence data.

Export Definition

```
extern "C" __declspec(dllexport)
  int __stdcall PutPersistance(
      TcomponentInstance    Instance,
      unsigned char *       Buffer,
      int                   Length);
```

Parameters

`Instance`: Particular instance of the component.

`Buffer:` A pointer to the persistence data.

`Length:` The size of the persistence data in bytes.

Return Value

If the function succeeds, the return value is the actual number of bytes used to restore the state of the component.

If the function fails, the return value is less than zero.

4.3.4 Property Manipulation

The functions in this section provide the capability to read and write properties both programmatically and through a component GUI interface when available.

4.3.4.1 GetPropertyC

The `GetPropertyC()` function returns the current value of a property.

Exported Definition

```
extern "C" __declspec(dllexport)
  int __stdcall GetPropertyC(
      TcomponentInstance    Instance,
      int                   PropertyIndex,
      TCPropertyValue *     Value);
```

Parameters

`Instance`: Particular instance of the component.

`PropertyIndex:` Index number of the particular property. The number of properties is determined through the `GetNumPropertiesC()` function. The indexes are zero-based; that is, the first property has an index of zero, the next has an index of one, and so on.

`Value`: Pointer to a TCPropertyValue structure that is set to the value of the property upon successful completion of the function call. The format of the `TCPropertyValue` structure is shown in Figure 5.

```
typedef struct
{
 TCPropertyTypes Type;
 union
 {
  char  StringValue[256];
  int   IntegerValue;
  int   BoolValue;
  float FloatValue;
 } Value;
} TCPropertyValue;
```

Figure 5: TCPropertyValue Structure

The definition of each field in the `TCPropertyValue` structure is shown in Table 5.

Field	Definition
`Type`	An enumerated field that indicates the type of the property such as integer, string, etc. See Table 1 for more information. This value is used to determine the correct reference from the union field.
`Value`	A union used to access all valid property types.
`StringValue`	Union value used to represent a string. A string property can not be greater then 255 bytes.
`IntegerValue`	Union value used to represent an integer.
`BoolValue`	Union value used to represent a Boolean.
`FloatValue`	Union value used to represent a floating point number.

Table 5: Description of TCPropertyValue Fields

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is nonzero.

4.3.4.2 SetPropertyC

The `SetPropertyC()` function sets the value of a property.

Exported Definition

```
extern "C" __declspec(dllexport)
  int __stdcall SetPropertyC(
       TcomponentInstance    Instance,
       int                   PropertyIndex,
       TCPropertyValue *     Value);
```

Parameters

`Instance`: Particular instance of the component.

`PropertyIndex:` Index number of the particular property. The number of properties is determined through the `GetNumPropertiesC()` function. The indexes are zero-based; that is, the first property has an index of zero, the next has an index of one, and so on.

`Value`: Pointers to a TCPropertyValue structure that contains the new value of the property. For more information about the TCPropertyValue structure, see Figure 5 and Table 5.

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is nonzero

4.3.4.3 PropertyEditor

A component may export the graphical user interface to manipulate component properties. If this interface exists, the value of the `PropertyEditor` field of the `TComponentInfo` structure returned from the `GetComponentInfo()` function shall be set to true. The `PropertyEditor()` function shall make the graphical property editor visible on the display.

Exported Definition

```
extern "C" __declspec(dllexport)
  int __stdcall PropertyEditor(
       TcomponentInstance Instance);
```

Parameters

`Instance`: Particular instance of the component.

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is nonzero.

4.3.5 Control

The functions in this section provide the capability to initialize a component, create and delete an instance of a component, track the state a component's properties, and control a component's execution and displays.

4.3.5.1 Initialize

The `Initialize()` function sets the state of a component instance to its initial state as if it was just created.

Export Definition

```
extern "C" __declspec(dllexport)
  int __stdcall Initialize(
       TcomponentInstance Instance);
```

Parameters

`Instance`: Particular instance of the component.

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is nonzero.

4.3.5.2 Execute

The `Execute()` function executes one computational step of the component.

Export Definition

```
extern "C" __declspec(dllexport)
  int __stdcall Execute(
       TcomponentInstance Instance);
```

Parameters

`Instance`: Particular instance of the component.

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is nonzero.

4.3.5.3 SetPropertyChangedCBC

This function is used to specify the callback that is used to notify the application that the value of a property has changed via the Property Editor.

Export Definition

```
extern "C" __declspec(dllexport)
  int __stdcall SetPropertyChangedCBC(
       TcomponentInstance         Instance,
       TPropertyChangedCBInfo *   Callback);
```

Parameters

`Instance`: Particular instance of the component.

`Callback:` Pointer to a `TPropertyChangedCBInfo` structure that contains the callback information. This structure shown in Figure 6.

```
typedef struct
{
 TPropertyChangedC  CallBack;
 TComponentInstance Instance;
} TPropertyChangedCBInfo;
```

Figure 6: TPropertyChangedCBInfo Structure

TPropertyChangedCBInfo Structure Fields

`Callback:` A pointer to the callback function that is defined as the following type:

```
typedef int __stdcall (*TPropertyChangedC)(
TcomponentInstance    Instance,
Int                   PropertyIndex,
TCPropertyValue *     Value);
```

Callback Parameters

`Instance`: This is set to the value of CBInstance defined in the `TPropertyChanged-CBInfo` structure.

`PropertyIndex:` Index of the property that has been changed.

`Value:` Pointer to the new value of the property.

Callback Return Value

If the callback function succeeds, the value returned is the actual number of bytes read or written.

If the callback function fails, the return value is less than or equal to zero.

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is nonzero.

4.3.5.4 GetNewInstance

The `GetNewInstance()` function creates a new instance of a component.

Export Definition

```
extern "C" __declspec(dllexport)
  TcomponentInstance __stdcall GetNewInstance(
       TParentWindow ParentWindow);
```

Parameters

`ParentWindow:` Handle of the parent window.

Return Value

If the function succeeds, the return value is a nonzero value representing the new instance.

If the function fails, the return value is NULL.

4.3.5.5 DeleteInstance

The `DeleteInstance()` function deletes an instance of a component.

Export Definition

```
extern "C" __declspec(dllexport)
  int __stdcall DeleteInstance(
       TcomponentInstance Instance);
```

Parameters

`Instance`: Particular instance of the component.

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is nonzero.

4.3.5.6 Display

The function `Display`() shall cause a component's graphical display to be visible.

Export Definition

```
extern "C" __declspec(dllexport)
  int __stdcall Display(
       TcomponentInstance Instance);
```

Parameters

`Instance`: Particular instance of the component.

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is nonzero.

5 Visual Programming Environment

Based on the component specification, a meta-application, which is a VPE for building applications from components (in this case, other urban runoff applications), has been developed for using WaterBeans components. This VPE enables engines and other components to be plugged in and to work together. The WaterBeans specification defines how components, such as engines, are integrated through their interfaces. As long as components conform to these specifications, they can be plugged into the VPE and used with other components that comply with the specification.

As a result, the VPE does not know what the components actually do. The components will use their own algorithms and produce a set of results. These results are passed by the VPE either to another engine that will do additional work on them, to a graphics analyzer that will produce a chart of their results, or to a database that will store the results.

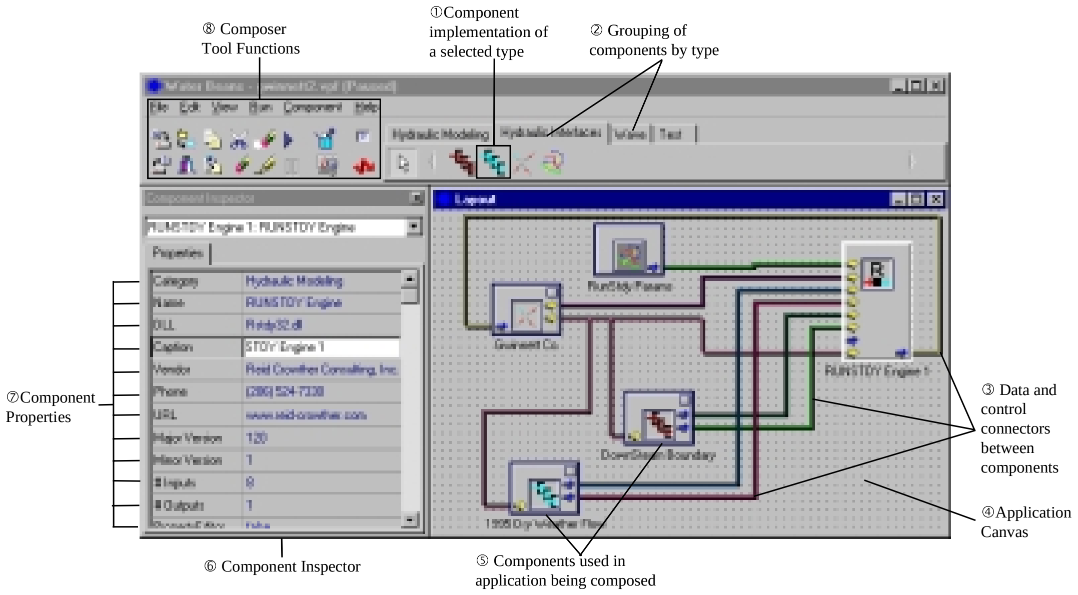
Components can consist of engines re-implemented from current applications, future engines to be built, output analyzers, or graphic displays. Components can come from different domains or have different levels of granularity. Thus, the VPE can represent a foundation for combining modeling software from different applications.

The current VPE implementation enables components to work together and produce results. Currently, WaterBeans is implemented within urban sewer modeling, and several components derived from SewerCAT[1] work within WaterBeans.

The WaterBeans VPE is an architecture framework that conforms to the constraints of the software architecture and demonstrates the application of component-based integration to water-quality modeling. The scope of this proof-of-concept demonstration is urban loading models.

Figure 7 depicts an annotated screen shot of the WaterBeans interface. The regions of this interface correspond to capabilities discussed in the remainder of this section.

[1] SewerCat is an application developed by Reid Crowther for modeling dynamics of sewer networks.


⑧ Composer Tool Functions
①Component implementation of a selected type
② Grouping of components by type
Water Beans - jennett2.wpf (Paused)
File Edit View Run Component Help
Hydraulic Modeling
Hydraulic Interfaces
Wave
Test
Component Inspector
Layout
RunStdy Params
Gravnent Co.
RUNSTDY Engine 1
DownStream Boundary
1995 Dry Weather Flow
⑦Component Properties
③ Data and control connectors between components
④Application Canvas
⑥ Component Inspector
⑤ Components used in application being composed

RUNSTDY Engine 1: RUNSTDY Engine

Properties

Category	Hydraulic Modeling
Name	RUNSTDY Engine
DLL	Rstdy32.dll
Caption	STDY Engine 1
Vendor	Reid Crowther Consulting, Inc.
Phone	(206) 524-7330
URL	www.reid-crowther.com
Major Version	120
Minor Version	1
# Inputs	8
# Outputs	1
PropertyEditor	false

Figure 7: WaterBeans Visual Composer

Several categories of components are supported by WaterBeans (①), including engines,[2] output visualization, data management, and data translators. Within each category there will be two or more interchangeable implementations (②). For example, RUNSTEADY and Superlink will be provided as engine components (① and ② are referred to as a “component palette”).[3] Urban runoff applications can be developed using a visual programming metaphor within the application canvas (③): selected components (⑤) are composed (integrated) through the use of different kinds of component connectors (④). The component inspector (⑥) provides an interface to a component’s application programming interfaces (APIs) and component-specific properties (⑦).[4] The overall functionality of WaterBeans is available through a menu-based and “quick-access” interface (⑧).

To clearly differentiated the architecture framework (i.e., the WaterBeans implementation) from the architectural style, the components used in the WaterBeans demonstration have been reused in Reid Crowther’s SewerCat. This demonstrates that WaterBeans is based upon an open software architecture, and that WaterBeans and SewerCat are both simply *conformant implementations* of this software architectural style. This is a crucially important point of the proof of concept, and it distinguishes the WaterBeans proof of concept from all of the available “point-to-point” integrated applications currently in use.

[2] Engines are components that implement numerical solvers for mathematical equations. These solvers are referred to as “engines” because they frequently “drive” the simulation process.
[3] RUNSTEADY and Superlink are computational engines from the SewerCat application.
[4] Component-specific properties can be modified from the component inspector interface.

6 Summary

WaterBeans proves the feasibility of using a component-based approach to urban runoff modeling that will

- define an application-specific component model for water-quality modeling
- enable the development of alternative *conformant* implementations of the standard component model (i.e., components and component framework)
- enable third-party integration of water-quality software components
- facilitate a software component marketplace for water-quality modeling

WaterBeans demonstrates in a highly visible way the feasibility of achieving these benefits. Rather than being an application for constructing and analyzing urban runoff models, Water-Beans is an application for building urban runoff applications from urban runoff components. The WaterBeans meta-application demonstrates the following two complementary capabilities that prove the feasibility and benefits of a component-based approach for urban water-quality modeling:

1. Third-party integrators can build new urban water-quality modeling applications through a compositional style of development, in which applications are implemented as a composition of software components. WaterBeans provides the build-time services to support compositional development and the runtime services to support inter-component coordination.
2. Third-party integrators can extend WaterBeans by importing alternative implementations of defined component types. Third-party integrators can also extend WaterBeans by defining new component types, although these new types must conform to the constraints defined by the architecture and be implemented by the WaterBeans framework.

The first capability described above demonstrates the use of component technology to reduce the complexity of application development drastically, but, ultimately, it is the second capability that has the largest impact on the water-quality community. This second capability (which is the consequence of defining a standard component model for urban runoff applications) will establish an open environment for water-quality scientists, component developers, application developers, water-quality modelers, and design engineers.

The WaterBeans component specification and prototype demonstrate the utility of a standard component model for urban-loading models. However, the scope of the standard in the prototype is limited to engine components. The other categories of components—data management, output visualization, and data translators—have not received the care required to make an assertion that they have standard interfaces. This may be a desirable generalization of WaterBeans.

Appendix: Component Header File

```
//---------------------------------------------------------------------------
#ifndef DLLSpecH
#define DLLSpecH
//---------------------------------------------------------------------------
#define COMPTRUE  1
#define COMPFALSE 0
typedef void * TComponentInstance;
typedef int    CompBool;
typedef void * TParentWindow;
enum TCPropertyTypes {ptString,ptFloat,ptInteger,ptBool};

typedef struct
{
 TCPropertyTypes Type;
 union
 {
  char  StringValue[256];
  int   IntegerValue;
  int   BoolValue;
  float FloatValue;
 } Value;
} TCPropertyValue;

typedef struct
{
 char             Name[256];
 TCPropertyTypes  Type;
 CompBool         ReadOnly;
}TCPropertyDescription;

typedef struct
{
 char      * Name;
 char      * Category;
 char      * Hint;
 char      * VendorName;
 char      * VendorContactNumber;
 char      * VendorWebAddr;
 char      * IconResourceName;
 int         MajorVersion;
 int         MinorVersion;
 int         NumInputs;
 int         NumOutputs;
 CompBool    PropertyEditor;
 CompBool    OutputDisplay;
 CompBool    Runable;
} TComponentInfo;

typedef struct
{
 char      * InterfaceName;
 char      * InterfaceType;
 CompBool    Required;
 CompBool    Active;
} TIOInfo;

//Callback Defs

// NULL Buffer returns bytes avail
typedef int                  __stdcall (*TGetData)(TComponentInstance,
                                                    unsigned char *,/*Buffer */
                                                    int             /*Length */);
typedef int                  __stdcall (*TPutData)(TComponentInstance,
                                                    unsigned char *,/*Buffer */
                                                    int             /*Length */);
typedef int                  __stdcall (*TPropertyChangedC) (TComponentInstance,
                                                             int Idx,
                                                             TCPropertyValue *);
typedef struct
{
 TPutData           CallBack;
 TComponentInstance Instance;
} TPutDataCBInfo;

typedef struct
{
 TGetData           CallBack;
 TComponentInstance Instance;
```

```
} TGetDataCBInfo;

typedef struct
{
 TPropertyChangedC  CallBack;
 TComponentInstance Instance;
} TPropertyChangedCBInfo;

#ifndef DLL_EXPORT
// Exported Functions
typedef TComponentInfo     * __stdcall (*TGetComponentInfo)(void);
typedef TComponentInstance   __stdcall (*TGetNewInstance)(TParentWindow);
typedef int                  __stdcall (*TDeleteInstance)(TComponentInstance);
typedef int                  __stdcall (*TInitialize)(TComponentInstance);
typedef int                  __stdcall (*TExecute)(TComponentInstance);
typedef TIOInfo            * __stdcall (*TGetInputInfo)(int);
typedef TIOInfo            * __stdcall (*TGetOutputInfo)(int);
typedef int                  __stdcall (*TSetInputCallBack) (TComponentInstance,
                                                   int,TGetDataCBInfo *);
typedef int                  __stdcall (*TSetOutputCallBack)(TComponentInstance,
                                                   int,TPutDataCBInfo *);
typedef unsigned int         __stdcall (*TGetInputRunRequirements)
                                       (TComponentInstance);
typedef int                  __stdcall (*TGetPassiveInputCallBack)
                                       (TComponentInstance,int,
                                                       TGetDataCBInfo *);
typedef int                  __stdcall (*TGetPersistance)
                                       (TComponentInstance,unsigned char *,int);
typedef int                  __stdcall (*TPutPersistance)
                                       (TComponentInstance,unsigned char *,int);
typedef int                  __stdcall (*TCPropertyEditor)(TComponentInstance);
typedef int                  __stdcall (*TDisplay)(TComponentInstance);
typedef int                  __stdcall (*TGetNumPropertiesC)(TComponentInstance);
typedef int                  __stdcall (*TGetPropertyDescriptionC)(
                                        TComponentInstance,
                                        int,
                                        TCPropertyDescription *);
typedef int                  __stdcall (*TSetPropertyC)(TComponentInstance,
                                                int,
                                                TCPropertyValue *);
typedef int                  __stdcall (*TGetPropertyC)(TComponentInstance,
                                                int,
                                                TCPropertyValue *);
```

```
typedef int                 __stdcall  (*TSetPropertyChangedCBC)(
                                                    TComponentInstance,
                                                    TPropertyChangedCBInfo *);

#define PROCADDR_GETCOMPONENTINFO(Inst)  \
     (TGetComponentInfo) GetProcAddress(Inst,"GetComponentInfo")

#define PROCADDR_GETNEWINSTANCE(Inst)  \
     (TGetNewInstance) GetProcAddress(Inst,"GetNewInstance")

#define PROCADDR_DELETEINSTANCE(Inst)  \
     (TDeleteInstance) GetProcAddress(Inst,"DeleteInstance")

#define PROCADDR_INITALIZE(Inst)  \
     (TInitialize) GetProcAddress(Inst,"Initialize")

#define PROCADDR_EXECUTE(Inst)  \
     (TExecute) GetProcAddress(Inst,"Execute")

#define PROCADDR_GETINPUTINFO(Inst)  \
     (TGetInputInfo)  GetProcAddress(Inst,"GetInputInfo")

#define PROCADDR_GETOUTPUTINFO(Inst)  \
     (TGetOutputInfo)  GetProcAddress(Inst,"GetOutputInfo")

#define PROCADDR_SETINPUTCALLBACK(Inst) \
     (TSetInputCallBack)  GetProcAddress(Inst,"SetInputCallBack")

#define PROCADDR_SETOUTPUTCALLBACK(Inst) \
     (TSetOutputCallBack)  GetProcAddress(Inst,"SetOutputCallBack")

#define PROCADDR_GETINPUTRUNREQUIREMENTS(Inst) \
     (TGetInputRunRequirements)  GetProcAddress(Inst,"GetInputRunRequirements")

#define PROCADDR_GETPASSIVECALLBACK(Inst ) \
     (TGetPassiveInputCallBack)  GetProcAddress(Inst,"GetPassiveInputCallBack")

#define PROCADDR_GETPERSISTANCE(Inst ) \
     (TGetPersistance)  GetProcAddress(Inst,"GetPersistance")

#define PROCADDR_PUTPERSISTANCE(Inst ) \
     (TPutPersistance)  GetProcAddress(Inst,"PutPersistance")
```

```
#define PROCADDR_PROPERTYEDITOR(Inst) \
     (TCPropertyEditor)  GetProcAddress(Inst,"PropertyEditor")

#define PROCADDR_DISPLAY(Inst) \
     (TDisplay)  GetProcAddress(Inst,"Display")

#define PROCADDR_GETNUMPROPERTIESC(Inst) \
    (TGetNumPropertiesC) GetProcAddress(Inst,"GetNumPropertiesC");

#define PROCADDR_GETPROPERTYDESCRIPTIONC(Inst) \
    (TGetPropertyDescriptionC) GetProcAddress(Inst,"GetPropertyDescriptionC");

#define PROCADDR_SETPROPERTYC(Inst) \
    (TSetPropertyC) GetProcAddress(Inst,"SetPropertyC");

#define PROCADDR_GETPROPERTYC(Inst) \
    (TGetPropertyC) GetProcAddress(Inst,"GetPropertyC");

#define PROCADDR_SETPROPERTYCHANGEDCBC(Inst) \
    (TSetPropertyChangedCBC) GetProcAddress(Inst,"SetPropertyChangedCBC");
#else
extern "C" __declspec(dllexport) TComponentInfo  *
                  __stdcall GetComponentInfo(void);
extern "C" __declspec(dllexport) TComponentInstance
                  __stdcall GetNewInstance(TParentWindow);
extern "C" __declspec(dllexport) int
                  __stdcall DeleteInstance(TComponentInstance);
extern "C" __declspec(dllexport) int
                  __stdcall Initialize(TComponentInstance);
extern "C" __declspec(dllexport) int
                  __stdcall Execute(TComponentInstance);
extern "C" __declspec(dllexport) TIOInfo *
                  __stdcall GetInputInfo(int);
extern "C" __declspec(dllexport) TIOInfo *
                  __stdcall GetOutputInfo(int);
extern "C" __declspec(dllexport) int
                  __stdcall SetInputCallBack(TComponentInstance,int,
                                             TGetDataCBInfo *);
extern "C" __declspec(dllexport) int
                  __stdcall SetOutputCallBack(TComponentInstance,int,
                                              TPutDataCBInfo *);
extern "C" __declspec(dllexport) unsigned int
                  __stdcall GetInputRunRequirements(TComponentInstance);
```

```
extern "C" __declspec(dllexport) int
                  __stdcall GetPassiveInputCallBack(TComponentInstance,int,
                                                    TGetDataCBInfo *);
extern "C" __declspec(dllexport) int
                  __stdcall GetPersistance(TComponentInstance,
                                               unsigned char *,int);
extern "C" __declspec(dllexport) int
                  __stdcall PutPersistance(TComponentInstance,
                                               unsigned char *,int);
extern "C" __declspec(dllexport) int
                  __stdcall PropertyEditor(TComponentInstance);
extern "C" __declspec(dllexport) int
                  __stdcall Display(TComponentInstance);

extern "C" __declspec(dllexport) int
                  __stdcall GetNumPropertiesC(TComponentInstance);

extern "C" __declspec(dllexport) int
                  __stdcall GetPropertyDescriptionC(TComponentInstance,
                                                   int,
                                                   TCPropertyDescription *);
extern "C" __declspec(dllexport) int
                  __stdcall SetPropertyC(TComponentInstance,
                                        int,
                                        TCPropertyValue *);
extern "C" __declspec(dllexport) int
                  __stdcall GetPropertyC(TComponentInstance,
                                        int,
                                        TCPropertyValue *);

extern "C" __declspec(dllexport) int
                  __stdcall SetPropertyChangedCBC(TComponentInstance,
                                                 TPropertyChangedCBInfo *);
#endif
//---------------------------------------------------------------------------
#endif
//---------------------------------------------------------------------------
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (LEAVE BLANK)	2. REPORT DATE **December 1999**	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE **Builder's Guide for WaterBeans Components**		5. FUNDING NUMBERS C — F19628-95-C-0003
6. AUTHOR(S) **Daniel Plakosh, Dennis Smith, Kurt C. Wallnau**		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER **CMU/SEI-99-TR-024**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER **ESC-TR-99-024**
11. SUPPLEMENTARY NOTES		
12.A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12.B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

WaterBeans is a proof-of-feasibility system for building software applications through a process of assembling (composing) prefabricated software components. WaterBeans was originally developed as a proof of feasibility that software component technology could be used to develop software applications in the domain of water-quality modeling. (In particular, WaterBeans supports modeling and simulating the hydrology of urban storm water sewage and runoff.) WaterBeans includes a component model for component developers, a visual composition environment for importing and assembling components into applications, and several families of components. One family of components supports modeling and simulating urban sewage systems. Another family of components was developed to prove the generality of WaterBeans; this family of components allows visualization and manipulation of digital waveforms. This report documents the programming interface for component developers. It also provides a brief description of the composition environment.

14. SUBJECT TERMS **Environmental Protection Agency (EPA), graphical user interface, software development, urban runoff modeling, visual programming environment, water-quality modeling**			15. NUMBER OF PAGES **36**
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL