(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
10 September 2021 (10.09.2021)

WIPO | PCT

(10) International Publication Number
WO 2021/178056 A1

(51) International Patent Classification:
*G06F 11/36* (2006.01) *G06F 8/41* (2018.01)

(21) International Application Number:
PCT/US2021/014127

(22) International Filing Date:
20 January 2021 (20.01.2021)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
16/809,025 04 March 2020 (04.03.2020) US

(71) Applicant: **ORACLE INTERNATIONAL CORPORATION** [US/US]; 500 Oracle Parkway M/S 5OP7, Redwood Shores, California 94065 (US).

(72) Inventor: **BZOCH, Pavel**; 500 Oracle Parkway M/S 5OP7, Redwood Shores, California 94065 (US).

(74) Agent: **BERGSTROM, James T.** et al.; 1100 Peachtree Street NE, Suite 2800, Atlanta, Georgia 30309 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, IT, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, WS, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV,

(54) Title: OBJECT ATTRIBUTES DYNAMIC VALIDATION

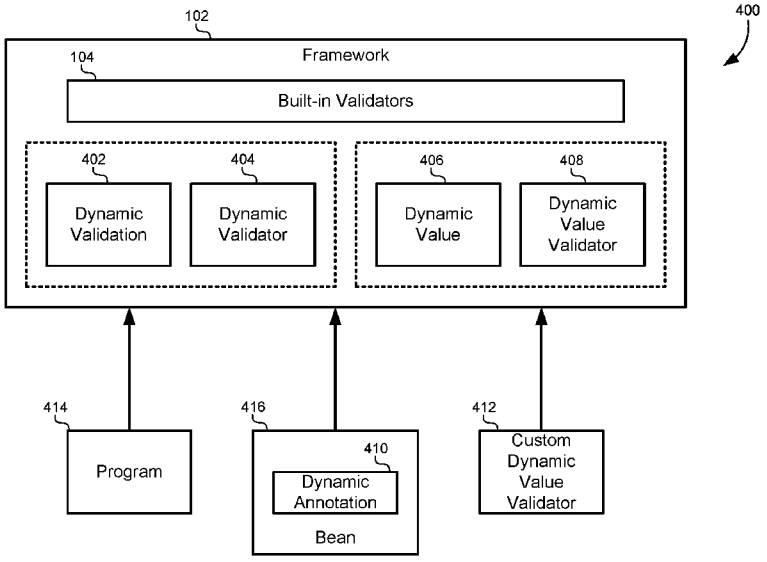



FIG. 4

(57) Abstract: A programming language framework may be enhanced to provide for dynamic validation. Dynamic validation allows the validator function for any variable to be selected at runtime rather than statically declared at programming-time. Instead of annotating a variable with an annotation that refers to a specific validator function or constraint type, programmers can annotate a variable with an annotation that indicates that the validator function will be selected dynamically at runtime. When a runtime instance of the variable is created, the programming language framework may identify the dynamic validation annotation on the variable, and then use the runtime values in the variable to determine which validator function(s) should be used.

WO 2021/178056 A1

*[Continued on next page]*

MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

**Published:**

— *with international search report (Art. 21(3))*

OBJECT ATTRIBUTES DYNAMIC VALIDATION

## CROSS-REFERENCE TO RELATED APPLICATIONS

**[0001]** This application claims the benefit and priority of U.S. Patent Application No. 16/809,025, filed March 4, 2020, entitled "DYNAMIC VALIDATION FRAMEWORK EXTENSION", which is hereby incorporated by reference in its entirety.

## BACKGROUND

**[0002]** Programming langauges have recently provided advanced mechanisms for enforcing constraints on member variables, classes, and functions. Constraints may be used to validate different data fields to ensure that they meet some one or more predefined requirements. Traditionally, constraints have been enforced using customized error-checking code that was included throughout the software. However, modern programming languages have begun providing mechanisms for validating data against sets of constraints as part of the programming language framework. Instead of rewriting the constraints and validation code for each project and each data set, programmers can instead use the constraint-enforcement mechanisms of the programming language framework to improve the efficiency of software development and the consistency with which common datatypes may be validated.

**[0003]** For example, the Java programming language provides validation for constraints imposed on class modules or "beans." Bean validation is defined in the JSR 380 specification, which allows for static validation on Java beans. In static validation, an attribute (e.g., a member variable) inside the bean meets some predefined criteria, such as being non-null, non-empty, within a specified numeric range, and so forth. If these predefined criteria are not met, the framework reports a constraint violation. The validation framework also allows users to create user-defined constraints and user-defined validators. These allow users to customize various validation routines to match a desired data format.

## BRIEF SUMMARY

**[0004]** Modern programming language frameworks provide constructs that perform static validation on runtime values using predefined constraints. When runtime instances are created,

the runtime values stored in variables and/or member attributes may be provided to a validator function that ensures that the value conforms to the predefined constraint. Predefined constraints may include numerical ranges, acceptable string patterns, date ranges, specific formatting, and/or other requirements that may be enforced on a data value. To activate a constraint, the variable or member attribute may be annotated with a text string that refers to a specific validator function. Thus, the validator function to be used for a specific variable is set at programming-time, not at runtime.

**[0005]** The embodiments described herein enhance programming language frameworks to provide for dynamic validation. Dynamic validation allows the validator function for any variable to be selected at runtime rather than statically declared at programming-time. Instead of annotating a variable with an annotation that refers to a specific validator function or constraint type, programmers can annotate a variable with an annotation that indicates that the validator function will be selected dynamically at runtime. When a runtime instance of the variable is created, the programming language framework may identify the dynamic validation annotation on the variable, and then use the runtime values in the variable to determine which validator function(s) should be used.

**[0006]** The programming language framework may be modified to include additional annotation definitions. One annotation definition may be used to annotate variables that should be subject to dynamic validation as decribed above. Another annotation definition may be used to annotate the user-defined dynamic value validator functions themselves. The second annotation may receive names and/or values for attributes to be dynamically validated. For example, a class definition may include two member attributes that operate as a key-value pair (e.g., string key, string value). The second annotation may reference the "key" attribute and determine whether the value in the "key" attribute indicates that this key-value pair stores a phone number (e.g., key = "phone"; value = "571-555-1534"). The annotation used on the validator function can specify that it should be used when an attribute named "key" has a value of "phone," and the code inside the validator function can then validate the "value" attribute to determine whether it stores a properly formatted phone number. If the "key" value stores something different, such as "date" or "time," then this this validator would not be executed on this class instance. This decision is made at runtime by the framework.

**[0007]** When the programming language framework receives a variable at runtime annotated for dynamic validation, the framework may first execute any traditional static validations. If a static validation fails, then no dynamic validation may be necessary. After static validations are complete, the programming language framework may generate a list of all dynamic validator functions available in a class path or program directory. The list of available dynamic validators may include any custom dynamic value validator functions that have been defined by programmers. The framework may then cycle through the list of available dynamic value validator functions and identify any for which the runtime values of the annotated variable satisfy the annotated contraints on the dynamic value validator function. Continuing with the example above, the framework may identify the phone number value validator function in the class path and compare the attributeName = "key" and attributeValue = "phone" constraint in the function annotation to the runtime values of the variable. Any identified validator functions may then be executed on the runtime values. In some embodiments, the framework may provide an abstract base class that includes protected helper functions for executing dynamic validations, and this abstract base class may be overridden in the custom dynamic value validators written by the programmer.

**[0008]** These programming language framework enhancements allow a programmer to encapsulate different dependencies between member attributes in different validator functions that are selected dynamically at runtime. When only static validation is used, a validator function was required to include branches of if/then statements that compare different possible values. These statements are difficult to maintain over time as code evolves. However, when using dynamic validation, each variable dependency for validation can be handled separately and independently, and new dependencies can be easily added without limitation by annotating a new validator function with the added dependency.

## BRIEF DESCRIPTION OF THE DRAWINGS

**[0009]** A further understanding of the nature and advantages of various embodiments may be realized by reference to the remaining portions of the specification and the drawings, wherein like reference numerals are used throughout the several drawings to refer to similar components.

In some instances, a sub-label is associated with a reference numeral to denote one of multiple similar components. When reference is made to a reference numeral without specification to an existing sub-label, it is intended to refer to all such multiple similar components.

**[0010]** **FIG. 1** illustrates a diagram of a constraint validation framework for a programming language, according to some embodiments.

**[0011]** **FIG. 2A** illustrates an example of a bean that uses a built-in constraint annotation, according to some embodiments.

**[0012]** **FIG. 2B** illustrates definitions for a custom validator/constraint.

**[0013]** **FIG. 3** illustrates a program that validates the constraints in the bean, according to some embodiments.

**[0014]** **FIG. 4** illustrates a diagram of a framework extension for dynamic validation, according to some embodiments.

**[0015]** **FIG. 5** illustrates a dynamic validation constraint annotation, according to some embodiments.

**[0016]** **FIG. 6** illustrates an example of a bean that uses dynamic validation, according to some embodiments.

**[0017]** **FIG. 7** illustrates a new annotation added to the framework that defines relationships between dynamic values, according to some embodiments.

**[0018]** **FIG. 8** illustrates an example of an abstract class for a DynamicValueValidator, according to some embodiments.

**[0019]** **FIG. 9A** illustrates a definition for a DynamicValidator class, according to some embodiments.

**[0020]** **FIG. 9B** illustrates an isValid( ) function for the DynamicValidator, according to some embodiments.

**[0021]** **FIG. 9C** illustrates methods for the DynamicValidator to perform static validation and matching between metadata defined in the DynamicValue annotation and the corresponding bean, according to some embodiments.

**[0022]** **FIG. 10A** illustrates an example of a user-defined custom dynamic value validator for dynamic validation, according to some embodiments.

**[0023]** **FIG. 10B** illustrates an example of a custom dynamic value validator with multiple constraint annotations, according to some embodiments.

**[0024]** **FIG. 11** illustrates an example of a program where dynamic validation may be invoked, according to some embodiments.

**[0025]** **FIG. 12** illustrates a flowchart of a method for dynamically validating variable dependencies at runtime, according to some embodiments.

**[0026]** **FIG. 13** illustrates a simplified block diagram of a distributed system for implementing some of the embodiments.

**[0027]** **FIG. 14** illustrates a simplified block diagram of components of a system environment by which services provided by the components of an embodiment system may be offered as cloud services.

**[0028]** **FIG. 15** illustrates an exemplary computer system, in which various embodiments may be implemented.

## DETAILED DESCRIPTION

**[0029]** In the Java programming language, various standards (JSR 380, JSR 349, JSR 303, etc.) describe a framework that may be used for the validation of attributes within Java beans. Bean validation is commonly used when data flows from a presentation layer to a persistence layer. Before this framework, users typically had to duplicate their validation code in each layer of their application. The validation framework allows users to perform a validation using a single mechanism via metadata that is added to domains or classes. The metadata may include constraints that are added via XML descriptors. For example, the default metadata source for a

Java class may include a Java annotation that is added to a target. The target data may then be validated against the constraint referenced by the annotation at runtime.

**[0030]** A constraint generally is comprised of two separate parts. A first part of the constraint includes a constraint annotation, which includes an annotation that may be added to the code to identify a runtime constraint. Generic constraint annotations may target a field, a method, a constructor, a parameter, a type, and so forth. A second part of the constraint may include a collection of validators that implement the constraint. For example, a constraint may include an allowable numeric range for a member variable, and a validator may include an isValid( ) function comprising code that determines whether the runtime value of the member variable falls within the allowable numeric range. Thus, a constraint annotation and a validator for that constraint annotation work together to perform runtime validation of constraints on program data.

**[0031]** **FIG. 1** illustrates a diagram 100 of a constraint validation framework 102 for a programming language, according to some embodiments. In computer programming, a software framework is an abstraction in which software providing generic functionality can be augmented by user code. The framework 102 generally provides a standardized library and methods to build and deploy applications, and it may provide functionality that is part of a larger software platform to facilitate the development of software applications. For example, the framework 102 may include support programs, compilers, code libraries, toolsets, application programming interfaces (APIs), and other components or development environments. For example, the Java programming language may be combined with various frameworks that provide utilities, programming language features, and code libraries that can be used to implement constraint validation. As used herein, the framework 102 itself may be distinguished from user code or custom, user-defined validators and constraint annotations.

**[0032]** A user program 106 may include user code that is compiled, assembled, interpreted, and/or executed as a software application. For example, the program 106 may include a main( ) function that instantiates class objects, executes functions, stores variables, and/or otherwise implements various aspects of a software application. The program 106 may be considered user code, and may be distinguished from the framework 102. Additionally, the user code may

include one or more custom class/object definitions. In the Java programming language, these classes may be organized in packages referred to as "beans." The program 106 may instantiate runtime objects using the class definitions in a bean 108.

**[0033]** As is common in many class definitions, classes within the bean 108 may include member variables, member functions, and other data. A constraint annotation 110 may be added in various locations to the class definitions in the bean 108. The scope of the annotation 110 may vary based on the location of the annotation 110 and/or the definition of the corresponding constraint annotation definition. As will be described in the examples below, the annotation 110 typically includes a line of code that identifies the constraint and may provide values for the constraint, such as messages and/or other information that may be used when validating the constraint.

**[0034]** In order to validate the constraint, a validator may be executed by the program 106. In some cases, the framework 102 may include one or more built-in validators 104. The built-in validators may include code that validates a set of built-in constraint definitions. For example, some Java frameworks may include built-in support for simple constraints. The javax.validation.constraints package includes built-in constraints, such as constraints that enforce minimum/maximum values, enforce null/non-null values, enforce predefined regular expression patterns, enforce predetermined sizes, and so forth. The bean 108 may utilize these built-in validators/constraints 104 by simply adding the corresponding annotation 110 to the code of the bean 108.

**[0035]** In addition to using the built-in validators/constraints 104, the program 106 may also use custom a validator/constraint 112. To use a custom validator and/or a custom constraint, a user-defined validator definition and/or constraint definition may be provided to the program 106. The custom validator/constraint 112 may each implement predefined interfaces and/or abstract classes provided by the framework 102 such that the custom validator/constraint 112 can be executed by the framework 102 the same way that the built-in validators/constraints 104 are used at runtime.

**[0036]** **FIG. 2A** illustrates an example of a bean 108 that uses a built-in constraint annotation 110, according to some embodiments. This bean includes a definition of a public class that

encapsulates a key-value data structure. Many modern data structures use this paradigm for storing many different datatypes in a unified format. A generic format may use a key-value structure where data is stored as a key-value pair. The key portion of the pair may be used to describe a data type (or any other type of metadata) for the value portion of the pair. For example, the key portion of the pair may define a data type, such as a phone number, an address, a username, etc. While the corresponding value portion of the pair may include a specific phone number, a specific address, a specific user's name, etc. This flexible system allows any type of data to be stored, so long as the key defines the data type and the value defines the data value. This KeyValue class may be used herein as an example of a bean that may use dynamic validation as described below. However, this class is used only by way of example it is not meant to be limiting. Dynamic validation may be used with any data structure, including classes and other encapsulations of data.

**[0037]** The bean 108 may include private member variables for both the key and the value. The example of FIG. 2A includes a constraint annotation 110 that may be used for validating the value of the string. Some of the built-in constraints in some Java frameworks may include the @NotBlank constraint, which indicates that the string value in the key variable should not be blank. The annotation 110 includes a parameter that sets the message accompanying a constraint violation to a specified value, such as "key is blank." Instead of writing custom validation code, the user can simply include the constraint annotation 110 to enforce predefined constraints that can be automatically validated by functions in the framework without writing additional code.

**[0038]** In addition to using built-in constraints, programming language frameworks may also allow users to define their own custom constraint annotations and custom validators for validating those constraints. **FIG. 2B** illustrates definitions for a custom validator/constraint 112. First, an example of a constraint definition 220 is provided that allows users to define their own constraint annotations. The constraint definition 220 may specify a target upon which the constraint may be applied (e.g., a field, a method, a type, etc.). The constraint definition 220 may also include message, group, and/or payload fields. Other fields may be added to provide information or other settings that may enhance validation. For example, the message field may be used to create an error message, which may be overwritten by a parameter as illustrated in FIG. 2A. Group fields may define groups to which the constraint belongs. The payload field

may specify other data with which the constraint may be associated. For example, the payload may be used to associate a severity with the constraint.

**[0039]** Along with the constraint definition 220, a validator definition 222 is also provided to specifically validate the constraint of the constraint annotation definition 220. The validator definition 222 in the context of the Java programming language may implement the ConstraintValidator interface, and may override two public functions. The initialize( ) function may be used to receive data from the constraint annotation and initialize the validator. The isValid( ) function holds the code that is used to validate the corresponding constraint against the corresponding object. If an annotation for the constraint definition 220 is used on a target object that corresponds to the object type in the validator definition 222, then the constraint may be validated. In some cases, the validator definition 222 may be defined generically such that the target class is of the Object type. In this case, the framework invokes the user-defined validation code for all targets annotated with the constraint. For example, the validator definition 222 may validate all targets bean annotated with the @MyConstraint constraint having a type of MyObject. Note that the validator definition 222 cannot be selected based on runtime values of attributes with in the object.

**[0040]** **FIG. 3** illustrates a program 106 that validates the constraints in the bean 108, according to some embodiments. First, the program 106 instantiates a validator 302 of a default type. The program 106 then defines a validate( ) function 308 that performs a validation on a specified list of objects in a parameter list. The main( ) function of the program 106 then instantiates a new KeyValue object with specified strings for the key and the value attributes. This instantiated object is then passed as a parameter 306 to the validate function. The validate function 308 then executes the specified validator on the object and reports any constraint violations.

**[0041]** Although users may define custom validators and constraints in existing programming language frameworks, the embodiments described herein improve on existing programming language frameworks to provide mechanisms for validating dependencies between attributes through the framework based on runtime values. These improvements allow custom validators and constraints to be readily implemented to provide deep validation where a value for one

attribute can influence the type of constraint that is validated on another attribute. For example, if the value for the key in a KeyValue object includes the string value of "phone," then the user may wish to validate the string in the value attribute to ensure that it is a properly formatted phone number. In another example, if the key is "date," then the user may wish to validate the value to ensure that the string is a properly formatted date. This cannot be performed with a static validation because only a single value of the target is validated. And while the user can validate relationships by defining their own constraints and providing validation code, the user would need to define a new combination of constraint and validator for each relationship in the target. This code is very hard to maintain and very hard to parse. The embodiments described herein instead describe an extension to a validation framework that solves this problem in a generic way rather than requiring specific code for each relationship.

**[0042]** **FIG. 4** illustrates a diagram 400 of a framework extension for dynamic validation, according to some embodiments. This diagram 400 is similar to the diagram 100 of FIG. 1, except that the framework 102 has been extended to include a number of additional modules: a dynamic validation 402 annotation, a dynamic validator 404, a dynamic value annotation 406, and a dynamic value validator 408. Each of these modules will be described in greater detail below. In short, these modules allow the framework to handle a @DynamicValidation annotation to then dynamically validate different relationships between different data fields based on runtime values, such as relationships between keys and values in the KeyValue class described above. It should be noted that these modules are part of the programming language framework and are not user-defined classes that the user is required to write. These are modules that may be deployed with the framework 102, and may thus be distinguished from code that a user may write to interact with the framework 102.

**[0043]** In this example, a bean 416 may include a dynamic annotation 410, such as the @DynamicValidation annotation described in detail below. When a program 414 creates an instance of an object defined in the bean 416, the dynamic annotation 410 will instruct the framework 102 to perform a dynamic validation using a custom dynamic value validator 412 defined by the user. As described below, the custom dynamic value validator 412 may extend abstract base classes made available in the extension to the framework 102, thus minimizing the code that a user may be required to write.

**[0044]** **FIG. 5** illustrates a dynamic validation constraint annotation, according to some embodiments. The starting point for dynamic validation is the definition of the constraint annotation and its corresponding validator. The dynamic validation constraint definition 402 defines the annotation that may be added to a class to indicate to the framework that dynamic validation should be executed rather than a traditional validation. Since the dynamic validation may be used to evaluate relationships between different member variables of a class, the constraint definition 402 sets the target value to be TYPE, which allows access to all of the attributes within the object definition. The name given for this constraint definition 402 is DynamicValidation (which corresponds to the @DynamicValidation annotation). However, this name is used only by way of example and is not meant to be limiting. Any other name may be used for the dynamic validation constraint definition 402. In some embodiments, no default message 502 needs to be defined with the annotation. As described below, any message defined within this annotation may be omitted and replaced during validation by the validator. Also note that the definition 502 specifies that this constraint may be validated by the DynamicValidator class, the name of which is also used only by way of example, and the function of which is described in detail below.

**[0045]** **FIG. 6** illustrates an example of a bean 416 that uses dynamic validation, according to some embodiments. Using the definition described above in FIG. 5, the KeyValue class may be annotated with the @DynamicValidation constraint annotation. This allows the framework to invoke a validator that dynamically evaluates the relationship between the runtime values of the key and value member attributes. Note that because the target value for the dynamic validation constraint definition 402 is TYPE, the dynamic annotation 410 may be applied at the class level rather than at the individual member variable levels. This dynamic annotation 410 based on the dynamic validation constraint definition 402 may be added at a similar location of any bean where dynamic validation should be invoked.

**[0046]** **FIG. 7** illustrates a new annotation added to the framework that defines relationships between dynamic values, according to some embodiments. As described in detail below, this annotation may be used for custom dynamic value validators to indicate runtime attribute values that may cause the validator to be invoked. Specifically, FIG. 7 illustrates a dynamic value annotation definition 406 that defines relationships between values. This particular annotation

includes two methods: attributeName and valuePattern. The attributeName method returns a name of an attribute for which the value will be checked in the target bean. The valuePattern method returns a regular expression for checking the value. This annotation may be used on instances of the DynamicValueValidator described in detail below. This extension ensures that only beans satisfying the relationship definition defined in the @DynamicValue need to be further validated. Additionally, this annotation may be used on the target DynamicValueValidator instances multiple times to allow the user to define multiple relationships. For example, if there are more @DynamicValue annotations for the DynamicValueValidator, each of these relationships defined by the annotations can be further validated.

**[0047]** **FIG. 8** illustrates an example of an abstract class for a DynamicValueValidator, according to some embodiments. First, note that the DynamicValueValidator class is an abstract class, which means that any custom dynamic value validator classes defined by the user should extend the DynamicValueValidator 408, and the extension should be annotated with at least one @DynamicValue as described above. This abstract class may be patterned after the javax.validation.ConstraintValidator abstract class in some Java frameworks that is extended by regular user-defined validators.

**[0048]** Each validator that extends the DynamicValueValidator class should provide an implementation of the isValid( ) method 802. This method may be called by the extended class if the relationship defined in the @DynamicValue is met for the validated bean. Specifically, the code for providing the actual validation may be provided in the custom dynamic value validator in an isValid( ) function that overrides this abstract function in the abstract base class.

**[0049]** Additionally, the DynamicValueValidator 408 includes two protected methods: validateCustomConstraints( ) 804 and useConstraint( ) 806, which may be used to facilitate user-defined validators. The validateCustomConstraints( ) method 804 may be used in a user-created validator if it defines its own object to be statically validated. One of the core concepts behind validation is that the validator defines its own inner class having attributes annotated with constraints. Then, these attributes are populated with data from the bean being validated. The instance of the class is then passed to the validateCustomConstraints( ) method 804. If this

method 804 identifies any constraint violation in the validated object, it takes the first message from the violation, passes it to the ConstraintValidationContext, and returns false. The helper method, useConstraint( ) 806, may be used to provide a user-defined message if the user specifies a deep validation of the bean that fails (e.g., the user validates whether a connection to a provided URL can be opened). Both of these functions may be called by the custom dynamic value validator written by the user that extends this class. As part of the framework extension, these functions reduce the amount of code required to write custom dynamic value validators.

**[0050]** **FIG. 9A** illustrates a definition for a DynamicValidator 404 class, according to some embodiments. The DynamicValidator implements the ConstraintValidator template that is already defined in the framework. Importantly, the DynamicValidator 404 serves as an entry point to the framework extension added by the embodiments described herein for performing dynamic validatio. This class first creates a validator for checking statically defined constraints, creates a list of dynamic validators that are loaded from a class path scan, and creates a static instance of a VALIDATOR. It should be noted that the DynamicValidator may be used for beans of any type. When the instance of the ConstraintValidator is created, it loads all classes extending the DynamicValueValidator described above. A constructor 902 creates an instance of a DynamicValidator according to the code in FIG. 9A. Specifically, if any of the dynamic validators in the class path are not yet instantiated, they are synchronously loaded and instantiated. As shown in FIG. 9A, the code filters out any abstract classes, any classes without no-argument constructors, and classes that do not have at least one correct @DynamicValidation annotation. Next, each of these classes are instantiated and cached. Some embodiments may use a full class path scan, while other embodiments may use Contexts and Dependency Injection (CDI) to obtain all instances of the DynamicValueValidator having the @DynamicValidation annotation. If CDI is used, then user-defined dynamic bean validators should define their scope to be visible for the CDI lookup. This approach may be preferable in Java EE environments.

**[0051]** **FIG. 9B** illustrates an isValid( ) 910 function for the DynamicValidator, according to some embodiments. As soon as the isValid( ) method 910 is invoked in the DynamicValidator 404, it may perform a validation of any statically defined constraints within the bean (912). If any static constraint is violated, no dynamic validation may be necessary, and thus the isValid( ) method may return and exit. Statically defined constraints may have a higher priority than

dynamically defined constraints in some embodiments. Alternatively, if there are no constraint violations on statically defined constraints, then dynamic validation may begin. The list of dynamic validators populated above in the constructor 902 may be filtered to find validators that are defined for this particular type of bean, and for which the relationship defined in the @DynamicValue is satisfied (914). Note that if the attribute does not have the type String, the toString( ) method may be called to retrieve a string representation of the attribute. Finally, any validator satisfying these conditions may then be passed to their respective isValid( ) methods (916).

**[0052]** **FIG. 9C** illustrates methods for the DynamicValidator to perform static validation and matching between metadata defined in the DynamicValue annotation and the corresponding bean, according to some embodiments. The isStaticValid( ) function 920 may perform validation on all properties found in the current class, as well as recursively in any parent classes. This private function 920 is called above in FIG. 9B to first determine whether any static validations fail before performing any dynamic validations. The matchDynamicAnnotation( ) function 922 may perform matching between metadata defined in the DynamicValue annotation and the bean. This private member function may return true if data defined in the annotation matches data found in the bean. This function 922 is called above in FIG. 9B when filtering dynamic validators that do not have the specified relationship condition.

**[0053]** FIGS. 7-9C illustrate implementations of various modules that may be included in a framework extension for performing dynamic validation, according to some embodiments. These example modules may use the Java programming language and specific Java constructs as an example to provide an enabling disclosure. However, these examples are not meant to be limiting. The principles for extending a framework to provide dynamic validation may be applied to any programming language, and may have different implementations other than those specifically used by way of example in these figures.

**[0054]** **FIG. 10A** illustrates an example of a user-defined custom dynamic value validator 412 for dynamic validation, according to some embodiments. The custom dynamic value validator 412 may be the only code that needs to be written by a user to employ dynamic validation. First, the validator 412 should include instances of one or more @DynamicValue annotations 1002.

The validator 412 should also extend 1004 the DynamicValueValidator abstract class described above in FIG. 8. This extended class provides an implementation for the abstract isValid( ) function 1006 from the parent class. Some embodiments may also provide private helper functions to perform the validation.

**[0055]** The dynamic @DynamicValue annotation 1002 specifies the variable and value for that variable that will trigger this dynamic validation. In this example, dynamic validation will be triggered when the target object includes a variable with the attribute name "key," and where the value of that attribute includes a string with the "phone number" text depicted in the annotation 1002. The validator 412 extends 1004 the DynamicValueValidator abstract parent class specifically for objects of the type KeyValue described above in FIG. 2A and FIG. 6. The isValid( ) receives a target KeyValue object, then uses the private PhoneNumber class to validate the phone number. At this point, the value variable in the KeyValue target object is expected to have a valid phone number to be validated because the key variable dynamically indicated such. Note that if the value of the attribute in the KeyValue object did not match the value in the annotation of the custom dynamic value validator 412, then the validator would not be executed on the instance of the object.

**[0056]** **FIG. 10B** illustrates an example of a custom dynamic value validator with multiple constraint annotations, according to some embodiments. Although the example in FIG. 10A only uses a single @DynamicValue annotation, other implementations may use a plurality of @DynamicValue annotations. For example, the KeyValue class described above includes a single key in a single value. Other embodiments include a TwoKeyValue class that includes two keys and a single value. The first key (key1) may still be used to specify that the value is a phone number, while the second key (key2) may be used to specify a country code. Because each country may have different formats for their phone numbers, both keys may be included individually in the @DynamicValue annotations 1010 in order for this particular custom dynamic value validator 412 to be invoked. This validator 412 may operate the same as the validator in FIG. 10A, except the class may be extended 1012 using the TwoKeyValue class rather than the KeyValue class. Similarly, the isValid( ) function 1014 may receive a TwoKeyValue object rather than a KeyValue object.

**[0057]** **FIG. 11** illustrates an example of a program 414 where dynamic validation may be invoked, according to some embodiments. As described above in relation to FIG. 6, the KeyValue class has been annotated with the @DynamicValidation constraint. When a new instance 1104 of the KeyValue class is created, string values may be provided for the key and value attributes. In this example, the key is set to be "phone", which will qualify for the custom dynamic value validator 412 in FIG. 10. The validate( ) function may then be called to dynamically check the value of the key ("phone") and invoke the custom dynamic value validator 412 to ensure that the phone number provided in the value attribute is of the proper format.

**[0058]** **FIG. 12** illustrates a flowchart 1200 of a method for dynamically validating variable dependencies at runtime, according to some embodiments. The method may include receiving, at a programming language framework, an instance of an object (1202). The instance of the object may be instantiated from an object definition. For example, a class object may be instantiated based on a class definition file. The class definition may be part of a bean or other software module. The definition of the object, such as the class definition, may be annotated with a constraint. The constraint may indicate that instances of the object should be subject to dynamic validation rather than static validation or other forms of custom, user-defined validation routines. Dynamic validation may imply that a type of validation routine and constraints that are applied to the instance of the object depend on variable runtime values in the object instance. For example, the @DynamicValidation constraint annotation may be used on a class definition as illustrated in FIG. 6. An instance of the object (e.g., the KeyValue object) may be instantiated and passed to the framework as part of a validate function as illustrated in FIG. 3.

**[0059]** The method may also include receiving one or more validators that are annotated with an annotation that identifies an attribute in the definition of the object, and a value for the attribute (1204). For example, the custom dynamic value validators in FIG. 10A and FIG. 10B may be annotated using the @DynamicValue annotation from FIG. 7. The annotation may include an attribute name that identifies the attribute in the definition of the object, along with a corresponding value for the attribute. For example, the attribute may be identified by an attribute name, such as "key," and the value may include a value string, such as "phone number" as used in the examples above. The validator may be received by the framework as it extends an abstract

base class. The validator may also implement a Boolean function that returns an indication of whether the value passed the constraint (e.g., an isValid( ) function). The validator may include user-defined code that implements a constraint, such as a format, minimum, maximum, range, and/or any other variable constraint.

**[0060]** The method may additionally include identifying a validator in the one or more validators for which a value of the attribute in the instance of the object matches the value of the attribute in the annotation of the validator (1206). For example, the DynamicValidator class in FIG. 9A may identify all of the validators in a runtime path or otherwise associated with a program. The constructor for the DynamicValidator may then cycle through each of the available validators and identify validators for which the annotation (e.g., the DynamicValue parameters) match the value for the corresponding attribute in the object instance. In some embodiments, each validator may have more than one (i.e., a plurality of) DynamicValue annotations, each of which may be satisfied to execute the validator on the object instance.

**[0061]** The method may further include executing the validator using the instance of the object (1208). The identified validator, along with any other validators that match the annotation criteria, may be executed on the instance of the object. For example, an attribute value may be validated against a constraint, and the constraint may be selected based on the value of another attribute, thus allowing attribute dependencies to influence which validators are executed and the results of those validators.

**[0062]** It should be appreciated that the specific steps illustrated in FIG. 12 provide particular methods of performing a dynamic validation according to various embodiments. Other sequences of steps may also be performed according to alternative embodiments. For example, alternative embodiments may perform the steps outlined above in a different order. Moreover, the individual steps illustrated in FIG. 12 may include multiple sub-steps that may be performed in various sequences as appropriate to the individual step. Furthermore, additional steps may be added or removed depending on the particular applications.

**[0063]** Each of the methods described herein may be implemented by a computer system. Each step of these methods may be executed automatically by the computer system, and/or may be provided with inputs/outputs involving a user. For example, a user may provide inputs for

each step in a method, and each of these inputs may be in response to a specific output requesting such an input, wherein the output is generated by the computer system. Each input may be received in response to a corresponding requesting output. Furthermore, inputs may be received from a user, from another computer system as a data stream, retrieved from a memory location, retrieved over a network, requested from a web service, and/or the like. Likewise, outputs may be provided to a user, to another computer system as a data stream, saved in a memory location, sent over a network, provided to a web service, and/or the like. In short, each step of the methods described herein may be performed by a computer system, and may involve any number of inputs, outputs, and/or requests to and from the computer system which may or may not involve a user. Those steps not involving a user may be said to be performed automatically by the computer system without human intervention. Therefore, it will be understood in light of this disclosure, that each step of each method described herein may be altered to include an input and output to and from a user, or may be done automatically by a computer system without human intervention where any determinations are made by a processor. Furthermore, some embodiments of each of the methods described herein may be implemented as a set of instructions stored on a tangible, non-transitory storage medium to form a tangible software product.

**[0064]** **FIG. 13** depicts a simplified diagram of a distributed system 1300 for implementing one of the embodiments. In the illustrated embodiment, distributed system 1300 includes one or more client computing devices 1302, 1304, 1306, and 1308, which are configured to execute and operate a client application such as a web browser, proprietary client (e.g., Oracle Forms), or the like over one or more network(s) 1310. Server 1312 may be communicatively coupled with remote client computing devices 1302, 1304, 1306, and 1308 via network 1310.

**[0065]** In various embodiments, server 1312 may be adapted to run one or more services or software applications provided by one or more of the components of the system. In some embodiments, these services may be offered as web-based or cloud services or under a Software as a Service (SaaS) model to the users of client computing devices 1302, 1304, 1306, and/or 1308. Users operating client computing devices 1302, 1304, 1306, and/or 1308 may in turn utilize one or more client applications to interact with server 1312 to utilize the services provided by these components.

**[0066]** In the configuration depicted in the figure, the software components 1318, 1320 and 1322 of system 1300 are shown as being implemented on server 1312. In other embodiments, one or more of the components of system 1300 and/or the services provided by these components may also be implemented by one or more of the client computing devices 1302, 1304, 1306, and/or 1308. Users operating the client computing devices may then utilize one or more client applications to use the services provided by these components. These components may be implemented in hardware, firmware, software, or combinations thereof. It should be appreciated that various different system configurations are possible, which may be different from distributed system 1300. The embodiment shown in the figure is thus one example of a distributed system for implementing an embodiment system and is not intended to be limiting.

**[0067]** Client computing devices 1302, 1304, 1306, and/or 1308 may be portable handheld devices (e.g., an iPhone®, cellular telephone, an iPad®, computing tablet, a personal digital assistant (PDA)) or wearable devices (e.g., a Google Glass® head mounted display), running software such as Microsoft Windows Mobile®, and/or a variety of mobile operating systems such as iOS, Windows Phone, Android, BlackBerry 10, Palm OS, and the like, and being Internet, e-mail, short message service (SMS), Blackberry®, or other communication protocol enabled. The client computing devices can be general purpose personal computers including, by way of example, personal computers and/or laptop computers running various versions of Microsoft Windows®, Apple Macintosh®, and/or Linux operating systems. The client computing devices can be workstation computers running any of a variety of commercially-available UNIX® or UNIX-like operating systems, including without limitation the variety of GNU/Linux operating systems, such as for example, Google Chrome OS. Alternatively, or in addition, client computing devices 1302, 1304, 1306, and 1308 may be any other electronic device, such as a thin-client computer, an Internet-enabled gaming system (e.g., a Microsoft Xbox gaming console with or without a Kinect® gesture input device), and/or a personal messaging device, capable of communicating over network(s) 1310.

**[0068]** Although exemplary distributed system 1300 is shown with four client computing devices, any number of client computing devices may be supported. Other devices, such as devices with sensors, etc., may interact with server 1312.

**[0069]** Network(s) 1310 in distributed system 1300 may be any type of network that can support data communications using any of a variety of commercially-available protocols, including without limitation TCP/IP (transmission control protocol/Internet protocol), SNA (systems network architecture), IPX (Internet packet exchange), AppleTalk, and the like. Merely by way of example, network(s) 1310 can be a local area network (LAN), such as one based on Ethernet, Token-Ring and/or the like. Network(s) 1310 can be a wide-area network and the Internet. It can include a virtual network, including without limitation a virtual private network (VPN), an intranet, an extranet, a public switched telephone network (PSTN), an infra-red network, a wireless network (e.g., a network operating under any of the Institute of Electrical and Electronics (IEEE) 802.11 suite of protocols, Bluetooth®, and/or any other wireless protocol); and/or any combination of these and/or other networks.

**[0070]** Server 1312 may be composed of one or more general purpose computers, specialized server computers (including, by way of example, PC (personal computer) servers, UNIX® servers, mid-range servers, mainframe computers, rack-mounted servers, etc.), server farms, server clusters, or any other appropriate arrangement and/or combination. In various embodiments, server 1312 may be adapted to run one or more services or software applications described in the foregoing disclosure. For example, server 1312 may correspond to a server for performing processing described above according to an embodiment of the present disclosure.

**[0071]** Server 1312 may run an operating system including any of those discussed above, as well as any commercially available server operating system. Server 1312 may also run any of a variety of additional server applications and/or mid-tier applications, including HTTP (hypertext transport protocol) servers, FTP (file transfer protocol) servers, CGI (common gateway interface) servers, JAVA® servers, database servers, and the like. Exemplary database servers include without limitation those commercially available from Oracle, Microsoft, Sybase, IBM (International Business Machines), and the like.

**[0072]** In some implementations, server 1312 may include one or more applications to analyze and consolidate data feeds and/or event updates received from users of client computing devices 1302, 1304, 1306, and 1308. As an example, data feeds and/or event updates may include, but are not limited to, Twitter® feeds, Facebook® updates or real-time updates received from one or

more third party information sources and continuous data streams, which may include real-time events related to sensor data applications, financial tickers, network performance measuring tools (e.g., network monitoring and traffic management applications), clickstream analysis tools, automobile traffic monitoring, and the like. Server 1312 may also include one or more applications to display the data feeds and/or real-time events via one or more display devices of client computing devices 1302, 1304, 1306, and 1308.

**[0073]** Distributed system 1300 may also include one or more databases 1314 and 1316. Databases 1314 and 1316 may reside in a variety of locations. By way of example, one or more of databases 1314 and 1316 may reside on a non-transitory storage medium local to (and/or resident in) server 1312. Alternatively, databases 1314 and 1316 may be remote from server 1312 and in communication with server 1312 via a network-based or dedicated connection. In one set of embodiments, databases 1314 and 1316 may reside in a storage-area network (SAN). Similarly, any necessary files for performing the functions attributed to server 1312 may be stored locally on server 1312 and/or remotely, as appropriate. In one set of embodiments, databases 1314 and 1316 may include relational databases, such as databases provided by Oracle, that are adapted to store, update, and retrieve data in response to SQL-formatted commands.

**[0074]** **FIG. 14** is a simplified block diagram of one or more components of a system environment 1400 by which services provided by one or more components of an embodiment system may be offered as cloud services, in accordance with an embodiment of the present disclosure. In the illustrated embodiment, system environment 1400 includes one or more client computing devices 1404, 1406, and 1408 that may be used by users to interact with a cloud infrastructure system 1402 that provides cloud services. The client computing devices may be configured to operate a client application such as a web browser, a proprietary client application (e.g., Oracle Forms), or some other application, which may be used by a user of the client computing device to interact with cloud infrastructure system 1402 to use services provided by cloud infrastructure system 1402.

**[0075]** It should be appreciated that cloud infrastructure system 1402 depicted in the figure may have other components than those depicted. Further, the embodiment shown in the figure is

only one example of a cloud infrastructure system that may incorporate some embodiments. In some other embodiments, cloud infrastructure system 1402 may have more or fewer components than shown in the figure, may combine two or more components, or may have a different configuration or arrangement of components.

**[0076]** Client computing devices 1404, 1406, and 1408 may be devices similar to those described above for 1302, 1304, 1306, and 1308.

**[0077]** Although exemplary system environment 1400 is shown with three client computing devices, any number of client computing devices may be supported. Other devices such as devices with sensors, etc. may interact with cloud infrastructure system 1402.

**[0078]** Network(s) 1410 may facilitate communications and exchange of data between clients 1404, 1406, and 1408 and cloud infrastructure system 1402. Each network may be any type of network that can support data communications using any of a variety of commercially-available protocols, including those described above for network(s) 1310.

**[0079]** Cloud infrastructure system 1402 may comprise one or more computers and/or servers that may include those described above for server 1312.

**[0080]** In certain embodiments, services provided by the cloud infrastructure system may include a host of services that are made available to users of the cloud infrastructure system on demand, such as online data storage and backup solutions, Web-based e-mail services, hosted office suites and document collaboration services, database processing, managed technical support services, and the like. Services provided by the cloud infrastructure system can dynamically scale to meet the needs of its users. A specific instantiation of a service provided by cloud infrastructure system is referred to herein as a "service instance." In general, any service made available to a user via a communication network, such as the Internet, from a cloud service provider's system is referred to as a "cloud service." Typically, in a public cloud environment, servers and systems that make up the cloud service provider's system are different from the customer's own on-premises servers and systems. For example, a cloud service provider's system may host an application, and a user may, via a communication network such as the Internet, on demand, order and use the application.

**[0081]** In some examples, a service in a computer network cloud infrastructure may include protected computer network access to storage, a hosted database, a hosted web server, a software application, or other service provided by a cloud vendor to a user, or as otherwise. For example, a service can include password-protected access to remote storage on the cloud through the Internet. As another example, a service can include a web service-based hosted relational database and a script-language middleware engine for private use by a networked developer. As another example, a service can include access to an email software application hosted on a cloud vendor's web site.

**[0082]** In certain embodiments, cloud infrastructure system 1402 may include a suite of applications, middleware, and database service offerings that are delivered to a customer in a self-service, subscription-based, elastically scalable, reliable, highly available, and secure manner. An example of such a cloud infrastructure system is the Oracle Public Cloud provided by the present assignee.

**[0083]** In various embodiments, cloud infrastructure system 1402 may be adapted to automatically provision, manage and track a customer's subscription to services offered by cloud infrastructure system 1402. Cloud infrastructure system 1402 may provide the cloud services via different deployment models. For example, services may be provided under a public cloud model in which cloud infrastructure system 1402 is owned by an organization selling cloud services (e.g., owned by Oracle) and the services are made available to the general public or different industry enterprises. As another example, services may be provided under a private cloud model in which cloud infrastructure system 1402 is operated solely for a single organization and may provide services for one or more entities within the organization. The cloud services may also be provided under a community cloud model in which cloud infrastructure system 1402 and the services provided by cloud infrastructure system 1402 are shared by several organizations in a related community. The cloud services may also be provided under a hybrid cloud model, which is a combination of two or more different models.

**[0084]** In some embodiments, the services provided by cloud infrastructure system 1402 may include one or more services provided under Software as a Service (SaaS) category, Platform as a Service (PaaS) category, Infrastructure as a Service (IaaS) category, or other categories of

services including hybrid services. A customer, via a subscription order, may order one or more services provided by cloud infrastructure system 1402. Cloud infrastructure system 1402 then performs processing to provide the services in the customer's subscription order.

**[0085]** In some embodiments, the services provided by cloud infrastructure system 1402 may include, without limitation, application services, platform services and infrastructure services. In some examples, application services may be provided by the cloud infrastructure system via a SaaS platform. The SaaS platform may be configured to provide cloud services that fall under the SaaS category. For example, the SaaS platform may provide capabilities to build and deliver a suite of on-demand applications on an integrated development and deployment platform. The SaaS platform may manage and control the underlying software and infrastructure for providing the SaaS services. By utilizing the services provided by the SaaS platform, customers can utilize applications executing on the cloud infrastructure system. Customers can acquire the application services without the need for customers to purchase separate licenses and support. Various different SaaS services may be provided. Examples include, without limitation, services that provide solutions for sales performance management, enterprise integration, and business flexibility for large organizations.

**[0086]** In some embodiments, platform services may be provided by the cloud infrastructure system via a PaaS platform. The PaaS platform may be configured to provide cloud services that fall under the PaaS category. Examples of platform services may include without limitation services that enable organizations (such as Oracle) to consolidate existing applications on a shared, common architecture, as well as the ability to build new applications that leverage the shared services provided by the platform. The PaaS platform may manage and control the underlying software and infrastructure for providing the PaaS services. Customers can acquire the PaaS services provided by the cloud infrastructure system without the need for customers to purchase separate licenses and support. Examples of platform services include, without limitation, Oracle Java Cloud Service (JCS), Oracle Database Cloud Service (DBCS), and others.

**[0087]** By utilizing the services provided by the PaaS platform, customers can employ programming languages and tools supported by the cloud infrastructure system and also control the deployed services. In some embodiments, platform services provided by the cloud

infrastructure system may include database cloud services, middleware cloud services (e.g., Oracle Fusion Middleware services), and Java cloud services. In one embodiment, database cloud services may support shared service deployment models that enable organizations to pool database resources and offer customers a Database as a Service in the form of a database cloud. Middleware cloud services may provide a platform for customers to develop and deploy various business applications, and Java cloud services may provide a platform for customers to deploy Java applications, in the cloud infrastructure system.

**[0088]** Various different infrastructure services may be provided by an IaaS platform in the cloud infrastructure system. The infrastructure services facilitate the management and control of the underlying computing resources, such as storage, networks, and other fundamental computing resources for customers utilizing services provided by the SaaS platform and the PaaS platform.

**[0089]** In certain embodiments, cloud infrastructure system 1402 may also include infrastructure resources 1430 for providing the resources used to provide various services to customers of the cloud infrastructure system. In one embodiment, infrastructure resources 1430 may include pre-integrated and optimized combinations of hardware, such as servers, storage, and networking resources to execute the services provided by the PaaS platform and the SaaS platform.

**[0090]** In some embodiments, resources in cloud infrastructure system 1402 may be shared by multiple users and dynamically re-allocated per demand. Additionally, resources may be allocated to users in different time zones. For example, cloud infrastructure system 1430 may enable a first set of users in a first time zone to utilize resources of the cloud infrastructure system for a specified number of hours and then enable the re-allocation of the same resources to another set of users located in a different time zone, thereby maximizing the utilization of resources.

**[0091]** In certain embodiments, a number of internal shared services 1432 may be provided that are shared by different components or modules of cloud infrastructure system 1402 and by the services provided by cloud infrastructure system 1402. These internal shared services may include, without limitation, a security and identity service, an integration service, an enterprise

repository service, an enterprise manager service, a virus scanning and white list service, a high availability, backup and recovery service, service for enabling cloud support, an email service, a notification service, a file transfer service, and the like.

**[0092]** In certain embodiments, cloud infrastructure system 1402 may provide comprehensive management of cloud services (e.g., SaaS, PaaS, and IaaS services) in the cloud infrastructure system. In one embodiment, cloud management functionality may include capabilities for provisioning, managing and tracking a customer's subscription received by cloud infrastructure system 1402, and the like.

**[0093]** In one embodiment, as depicted in the figure, cloud management functionality may be provided by one or more modules, such as an order management module 1420, an order orchestration module 1422, an order provisioning module 1424, an order management and monitoring module 1426, and an identity management module 1428. These modules may include or be provided using one or more computers and/or servers, which may be general purpose computers, specialized server computers, server farms, server clusters, or any other appropriate arrangement and/or combination.

**[0094]** In exemplary operation 1434, a customer using a client device, such as client device 1404, 1406 or 1408, may interact with cloud infrastructure system 1402 by requesting one or more services provided by cloud infrastructure system 1402 and placing an order for a subscription for one or more services offered by cloud infrastructure system 1402. In certain embodiments, the customer may access a cloud User Interface (UI), cloud UI 1412, cloud UI 1414 and/or cloud UI 1416 and place a subscription order via these UIs. The order information received by cloud infrastructure system 1402 in response to the customer placing an order may include information identifying the customer and one or more services offered by the cloud infrastructure system 1402 that the customer intends to subscribe to.

**[0095]** After an order has been placed by the customer, the order information is received via the cloud UIs, 1412, 1414 and/or 1416.

**[0096]** At operation 1436, the order is stored in order database 1418. Order database 1418 can be one of several databases operated by cloud infrastructure system 1418 and operated in conjunction with other system elements.

**[0097]** At operation 1438, the order information is forwarded to an order management module 1420. In some instances, order management module 1420 may be configured to perform billing and accounting functions related to the order, such as verifying the order, and upon verification, booking the order.

**[0098]** At operation 1440, information regarding the order is communicated to an order orchestration module 1422. Order orchestration module 1422 may utilize the order information to orchestrate the provisioning of services and resources for the order placed by the customer. In some instances, order orchestration module 1422 may orchestrate the provisioning of resources to support the subscribed services using the services of order provisioning module 1424.

**[0099]** In certain embodiments, order orchestration module 1422 enables the management of business processes associated with each order and applies business logic to determine whether an order should proceed to provisioning. At operation 1442, upon receiving an order for a new subscription, order orchestration module 1422 sends a request to order provisioning module 1424 to allocate resources and configure those resources needed to fulfill the subscription order. Order provisioning module 1424 enables the allocation of resources for the services ordered by the customer. Order provisioning module 1424 provides a level of abstraction between the cloud services provided by cloud infrastructure system 1400 and the physical implementation layer that is used to provision the resources for providing the requested services. Order orchestration module 1422 may thus be isolated from implementation details, such as whether or not services and resources are actually provisioned on the fly or pre-provisioned and only allocated/assigned upon request.

**[0100]** At operation 1444, once the services and resources are provisioned, a notification of the provided service may be sent to customers on client devices 1404, 1406 and/or 1408 by order provisioning module 1424 of cloud infrastructure system 1402.

**[0101]** At operation 1446, the customer's subscription order may be managed and tracked by an order management and monitoring module 1426. In some instances, order management and monitoring module 1426 may be configured to collect usage statistics for the services in the subscription order, such as the amount of storage used, the amount data transferred, the number of users, and the amount of system up time and system down time.

**[0102]** In certain embodiments, cloud infrastructure system 1400 may include an identity management module 1428. Identity management module 1428 may be configured to provide identity services, such as access management and authorization services in cloud infrastructure system 1400. In some embodiments, identity management module 1428 may control information about customers who wish to utilize the services provided by cloud infrastructure system 1402. Such information can include information that authenticates the identities of such customers and information that describes which actions those customers are authorized to perform relative to various system resources (e.g., files, directories, applications, communication ports, memory segments, etc.) Identity management module 1428 may also include the management of descriptive information about each customer and about how and by whom that descriptive information can be accessed and modified.

**[0103]** **FIG. 15** illustrates an exemplary computer system 1500, in which various embodiments may be implemented. The system 1500 may be used to implement any of the computer systems described above. As shown in the figure, computer system 1500 includes a processing unit 1504 that communicates with a number of peripheral subsystems via a bus subsystem 1502. These peripheral subsystems may include a processing acceleration unit 1506, an I/O subsystem 1508, a storage subsystem 1518 and a communications subsystem 1524. Storage subsystem 1518 includes tangible computer-readable storage media 1522 and a system memory 1510.

**[0104]** Bus subsystem 1502 provides a mechanism for letting the various components and subsystems of computer system 1500 communicate with each other as intended. Although bus subsystem 1502 is shown schematically as a single bus, alternative embodiments of the bus subsystem may utilize multiple buses. Bus subsystem 1502 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. For example, such architectures may include an Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnect (PCI) bus, which can be implemented as a Mezzanine bus manufactured to the IEEE P1386.1 standard.

**[0105]** Processing unit 1504, which can be implemented as one or more integrated circuits (e.g., a conventional microprocessor or microcontroller), controls the operation of computer system 1500. One or more processors may be included in processing unit 1504. These processors may include single core or multicore processors. In certain embodiments, processing unit 1504 may be implemented as one or more independent processing units 1532 and/or 1534 with single or multicore processors included in each processing unit. In other embodiments, processing unit 1504 may also be implemented as a quad-core processing unit formed by integrating two dual-core processors into a single chip.

**[0106]** In various embodiments, processing unit 1504 can execute a variety of programs in response to program code and can maintain multiple concurrently executing programs or processes. At any given time, some or all of the program code to be executed can be resident in processor(s) 1504 and/or in storage subsystem 1518. Through suitable programming, processor(s) 1504 can provide various functionalities described above. Computer system 1500 may additionally include a processing acceleration unit 1506, which can include a digital signal processor (DSP), a special-purpose processor, and/or the like.

**[0107]** I/O subsystem 1508 may include user interface input devices and user interface output devices. User interface input devices may include a keyboard, pointing devices such as a mouse or trackball, a touchpad or touch screen incorporated into a display, a scroll wheel, a click wheel, a dial, a button, a switch, a keypad, audio input devices with voice command recognition systems, microphones, and other types of input devices. User interface input devices may include, for example, motion sensing and/or gesture recognition devices such as the Microsoft Kinect® motion sensor that enables users to control and interact with an input device, such as the Microsoft Xbox® 360 game controller, through a natural user interface using gestures and spoken commands. User interface input devices may also include eye gesture recognition devices such as the Google Glass® blink detector that detects eye activity (e.g., 'blinking' while taking pictures and/or making a menu selection) from users and transforms the eye gestures as input into an input device (e.g., Google Glass®). Additionally, user interface input devices may include voice recognition sensing devices that enable users to interact with voice recognition systems (e.g., Siri® navigator), through voice commands.

**[0108]** User interface input devices may also include, without limitation, three dimensional (3D) mice, joysticks or pointing sticks, gamepads and graphic tablets, and audio/visual devices such as speakers, digital cameras, digital camcorders, portable media players, webcams, image scanners, fingerprint scanners, barcode reader 3D scanners, 3D printers, laser rangefinders, and eye gaze tracking devices. Additionally, user interface input devices may include, for example, medical imaging input devices such as computed tomography, magnetic resonance imaging, position emission tomography, medical ultrasonography devices. User interface input devices may also include, for example, audio input devices such as MIDI keyboards, digital musical instruments and the like.

**[0109]** User interface output devices may include a display subsystem, indicator lights, or non-visual displays such as audio output devices, etc. The display subsystem may be a cathode ray tube (CRT), a flat-panel device, such as that using a liquid crystal display (LCD) or plasma display, a projection device, a touch screen, and the like. In general, use of the term "output device" is intended to include all possible types of devices and mechanisms for outputting information from computer system 1500 to a user or other computer. For example, user interface output devices may include, without limitation, a variety of display devices that visually convey text, graphics and audio/video information such as monitors, printers, speakers, headphones, automotive navigation systems, plotters, voice output devices, and modems.

**[0110]** Computer system 1500 may comprise a storage subsystem 1518 that comprises software elements, shown as being currently located within a system memory 1510. System memory 1510 may store program instructions that are loadable and executable on processing unit 1504, as well as data generated during the execution of these programs.

**[0111]** Depending on the configuration and type of computer system 1500, system memory 1510 may be volatile (such as random access memory (RAM)) and/or non-volatile (such as read-only memory (ROM), flash memory, etc.) The RAM typically contains data and/or program modules that are immediately accessible to and/or presently being operated and executed by processing unit 1504. In some implementations, system memory 1510 may include multiple different types of memory, such as static random access memory (SRAM) or dynamic random access memory (DRAM). In some implementations, a basic input/output system (BIOS),

containing the basic routines that help to transfer information between elements within computer system 1500, such as during start-up, may typically be stored in the ROM. By way of example, and not limitation, system memory 1510 also illustrates application programs 1512, which may include client applications, Web browsers, mid-tier applications, relational database management systems (RDBMS), etc., program data 1514, and an operating system 1516. By way of example, operating system 1516 may include various versions of Microsoft Windows®, Apple Macintosh®, and/or Linux operating systems, a variety of commercially-available UNIX® or UNIX-like operating systems (including without limitation the variety of GNU/Linux operating systems, the Google Chrome® OS, and the like) and/or mobile operating systems such as iOS, Windows® Phone, Android® OS, BlackBerry® 10 OS, and Palm® OS operating systems.

**[0112]** Storage subsystem 1518 may also provide a tangible computer-readable storage medium for storing the basic programming and data constructs that provide the functionality of some embodiments. Software (programs, code modules, instructions) that when executed by a processor provide the functionality described above may be stored in storage subsystem 1518. These software modules or instructions may be executed by processing unit 1504. Storage subsystem 1518 may also provide a repository for storing data used in accordance with some embodiments.

**[0113]** Storage subsystem 1500 may also include a computer-readable storage media reader 1520 that can further be connected to computer-readable storage media 1522. Together and, optionally, in combination with system memory 1510, computer-readable storage media 1522 may comprehensively represent remote, local, fixed, and/or removable storage devices plus storage media for temporarily and/or more permanently containing, storing, transmitting, and retrieving computer-readable information.

**[0114]** Computer-readable storage media 1522 containing code, or portions of code, can also include any appropriate media, including storage media and communication media, such as but not limited to, volatile and non-volatile, removable and non-removable media implemented in any method or technology for storage and/or transmission of information. This can include tangible computer-readable storage media such as RAM, ROM, electronically erasable programmable ROM (EEPROM), flash memory or other memory technology, CD-ROM, digital

versatile disk (DVD), or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or other tangible computer readable media. This can also include nontangible computer-readable media, such as data signals, data transmissions, or any other medium which can be used to transmit the desired information and which can be accessed by computing system 1500.

**[0115]** By way of example, computer-readable storage media 1522 may include a hard disk drive that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive that reads from or writes to a removable, nonvolatile magnetic disk, and an optical disk drive that reads from or writes to a removable, nonvolatile optical disk such as a CD ROM, DVD, and Blu-Ray® disk, or other optical media. Computer-readable storage media 1522 may include, but is not limited to, Zip® drives, flash memory cards, universal serial bus (USB) flash drives, secure digital (SD) cards, DVD disks, digital video tape, and the like. Computer-readable storage media 1522 may also include, solid-state drives (SSD) based on non-volatile memory such as flash-memory based SSDs, enterprise flash drives, solid state ROM, and the like, SSDs based on volatile memory such as solid state RAM, dynamic RAM, static RAM, DRAM-based SSDs, magnetoresistive RAM (MRAM) SSDs, and hybrid SSDs that use a combination of DRAM and flash memory based SSDs. The disk drives and their associated computer-readable media may provide non-volatile storage of computer-readable instructions, data structures, program modules, and other data for computer system 1500.

**[0116]** Communications subsystem 1524 provides an interface to other computer systems and networks. Communications subsystem 1524 serves as an interface for receiving data from and transmitting data to other systems from computer system 1500. For example, communications subsystem 1524 may enable computer system 1500 to connect to one or more devices via the Internet. In some embodiments communications subsystem 1524 can include radio frequency (RF) transceiver components for accessing wireless voice and/or data networks (e.g., using cellular telephone technology, advanced data network technology, such as 3G, 4G or EDGE (enhanced data rates for global evolution), WiFi (IEEE 802.11 family standards, or other mobile communication technologies, or any combination thereof), global positioning system (GPS) receiver components, and/or other components. In some embodiments communications

subsystem 1524 can provide wired network connectivity (e.g., Ethernet) in addition to or instead of a wireless interface.

**[0117]** In some embodiments, communications subsystem 1524 may also receive input communication in the form of structured and/or unstructured data feeds 1526, event streams 1528, event updates 1530, and the like on behalf of one or more users who may use computer system 1500.

**[0118]** By way of example, communications subsystem 1524 may be configured to receive data feeds 1526 in real-time from users of social networks and/or other communication services such as Twitter® feeds, Facebook® updates, web feeds such as Rich Site Summary (RSS) feeds, and/or real-time updates from one or more third party information sources.

**[0119]** Additionally, communications subsystem 1524 may also be configured to receive data in the form of continuous data streams, which may include event streams 1528 of real-time events and/or event updates 1530, that may be continuous or unbounded in nature with no explicit end. Examples of applications that generate continuous data may include, for example, sensor data applications, financial tickers, network performance measuring tools (e.g. network monitoring and traffic management applications), clickstream analysis tools, automobile traffic monitoring, and the like.

**[0120]** Communications subsystem 1524 may also be configured to output the structured and/or unstructured data feeds 1526, event streams 1528, event updates 1530, and the like to one or more databases that may be in communication with one or more streaming data source computers coupled to computer system 1500.

**[0121]** Computer system 1500 can be one of various types, including a handheld portable device (e.g., an iPhone® cellular phone, an iPad® computing tablet, a PDA), a wearable device (e.g., a Google Glass® head mounted display), a PC, a workstation, a mainframe, a kiosk, a server rack, or any other data processing system.

**[0122]** Due to the ever-changing nature of computers and networks, the description of computer system 1500 depicted in the figure is intended only as a specific example. Many other configurations having more or fewer components than the system depicted in the figure are

possible. For example, customized hardware might also be used and/or particular elements might be implemented in hardware, firmware, software (including applets), or a combination. Further, connection to other computing devices, such as network input/output devices, may be employed. Based on the disclosure and teachings provided herein, other ways and/or methods may be used to implement the various embodiments.

**[0123]** In the foregoing description, for the purposes of explanation, numerous specific details were set forth in order to provide a thorough understanding of various embodiments. It will be apparent, however, that these embodiments may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form.

**[0124]** The foregoing description provides exemplary embodiments only, and is not intended to limit the scope, applicability, or configuration of the disclosure. Rather, the foregoing description of the exemplary embodiments provide an enabling description for implementing at least one embodiment. It should be understood that various changes may be made in the function and arrangement of elements without departing from the spirit and scope of some embodiments as set forth in the appended claims.

**[0125]** Specific details are given in the foregoing description to provide a thorough understanding of the embodiments. However, the embodiments may be practiced without these specific details. For example, circuits, systems, networks, processes, and other components may have been shown as components in block diagram form in order not to obscure the embodiments in unnecessary detail. In other instances, well-known circuits, processes, algorithms, structures, and techniques may have been shown without unnecessary detail in order to avoid obscuring the embodiments.

**[0126]** Also, it is noted that individual embodiments may have beeen described as a process which is depicted as a flowchart, a flow diagram, a data flow diagram, a structure diagram, or a block diagram. Although a flowchart may have described the operations as a sequential process, many of the operations can be performed in parallel or concurrently. In addition, the order of the operations may be re-arranged. A process is terminated when its operations are completed, but could have additional steps not included in a figure. A process may correspond to a method, a function, a procedure, a subroutine, a subprogram, etc. When a process corresponds to a

function, its termination can correspond to a return of the function to the calling function or the main function.

**[0127]** The term "computer-readable medium" includes, but is not limited to portable or fixed storage devices, optical storage devices, wireless channels and various other mediums capable of storing, containing, or carrying instruction(s) and/or data. A code segment or machine-executable instructions may represent a procedure, a function, a subprogram, a program, a routine, a subroutine, a module, a software package, a class, or any combination of instructions, data structures, or program statements. A code segment may be coupled to another code segment or a hardware circuit by passing and/or receiving information, data, arguments, parameters, or memory contents. Information, arguments, parameters, data, etc., may be passed, forwarded, or transmitted via any suitable means including memory sharing, message passing, token passing, network transmission, etc.

**[0128]** Furthermore, embodiments may be implemented by hardware, software, firmware, middleware, microcode, hardware description languages, or any combination thereof. When implemented in software, firmware, middleware or microcode, the program code or code segments to perform the necessary tasks may be stored in a machine readable medium. A processor(s) may perform the necessary tasks.

**[0129]** In the foregoing specification, aspects of various embodiments are described with reference to specific embodiments thereof, but not all embodiments are limited thereto. Various features and aspects of the above-described embodiments may be used individually or jointly. Further, embodiments can be utilized in any number of environments and applications beyond those described herein without departing from the broader spirit and scope of the specification. The specification and drawings are, accordingly, to be regarded as illustrative rather than restrictive.

**[0130]** Additionally, for the purposes of illustration, methods were described in a particular order. It should be appreciated that in alternate embodiments, the methods may be performed in a different order than that described. It should also be appreciated that the methods described above may be performed by hardware components or may be embodied in sequences of machine-executable instructions, which may be used to cause a machine, such as a general-

purpose or special-purpose processor or logic circuits programmed with the instructions to perform the methods. These machine-executable instructions may be stored on one or more machine readable mediums, such as CD-ROMs or other type of optical disks, floppy diskettes, ROMs, RAMs, EPROMs, EEPROMs, magnetic or optical cards, flash memory, or other types of machine-readable mediums suitable for storing electronic instructions. Alternatively, the methods may be performed by a combination of hardware and software.

WHAT IS CLAIMED IS:

1. A non-transitory computer-readable medium comprising instructions that, when executed by one or more processors, cause the one or more processors to perform operations comprising:

receiving, at a programming language framework, an instance of an object, wherein a definition of the object is annotated with a constraint;

receiving one or more validators that are annotated with an annotation that identifies:

an attribute in the definition of the object; and

a value for the attribute;

identifying a validator in the one or more validators for which a value of the attribute in the instance of the object matches the value for the attribute in the annotation of the validator; and

executing the validator using the instance of the object.

2. The non-transitory computer-readable medium of claim 1, wherein the validator validates a value of a second attribute in the definition of the object.

3. The non-transitory computer-readable medium of claim 1, wherein the one or more validators comprises a second validator for which the value of the attribute in the instance of the object does not match the value for the attribute in the annotation of the second validator.

4. The non-transitory computer-readable medium of claim 1, wherein the constraint with which the definition of the object is annotated causes the programming language framework to use dynamic validation rather than static validation.

5. The non-transitory computer-readable medium of claim 4, wherein the programming language framework comprises a definition for the constraint with which the definition of the object is annotated, wherein the definition for the constraint specifies that the constraint may be executed on object type targets.

6. The non-transitory computer-readable medium of claim 1, wherein the programming language framework comprises a definition for the annotation with which the one or more validators are annotated, wherein the definition comprises an attribute name and an attribute value pattern.

7. The non-transitory computer-readable medium of claim 1, wherein the programming language framework comprises a definition of an abstract class from which each of the one or more validators is inherited.

8. The non-transitory computer-readable medium of claim 7, wherein the abstract class comprises an abstract function that determines if a constraint is valid for an object.

9. The non-transitory computer-readable medium of claim 7, wherein the abstract class comprises a protected function that validates an object against custom constraints.

10. The non-transitory computer-readable medium of claim 1, wherein the programming language framework comprises a definition of a class that scans a class path and identifies each of the one or more validators in the class path.

11. The non-transitory computer-readable medium of claim 10, wherein the definition of the class includes a function that filters out ones of the one or more validators for which the value of the attribute in the instance of the object does not match the values for the attributes in the annotations of the ones of the one or more validators.

12. The non-transitory computer-readable medium of claim 10, wherein the programming language framework creates an instance based on the definition of the class as an entry point for dynamic validation when receiving the instance of the object.

13. The non-transitory computer-readable medium of claim 10, wherein the definition of the class first executes any static validations before executing any dynamic validations.

14. The non-transitory computer-readable medium of claim 1, wherein the instance of the object comprises the attribute and a second attribute.

15. The non-transitory computer-readable medium of claim 14, wherein the attribute determines which of the one or more validators should be used to validate a value assigned to the second attribute.

16. The non-transitory computer-readable medium of claim 15, wherein the attribute comprises a key and the second attribute comprises a value in a key-value pair.

17. The non-transitory computer-readable medium of claim 16, wherein the key indicates a datatype of the value.

18. The non-transitory computer-readable medium of claim 1, wherein:

the validator is annotated with a second annotation that identifies a second attribute in the definition of the object and a second value for the second attribute; and

a value of a second attribute in the instance of the object matches the second value for the second attribute in the second annotation of the validator.

19. A system comprising:

one or more processors; and

one or more memory devices comprising instructions that, when executed by the one or more processors, cause the one or more processors to perform operations comprising:

receiving, at a programming language framework, an instance of an object, wherein a definition of the object is annotated with a constraint;

receiving one or more validators that are annotated with an annotation that identifies:

an attribute in the definition of the object; and

a value for the attribute;

identifying a validator in the one or more validators for which a value of the attribute in the instance of the object matches the value for the attribute in the annotation of the validator; and

executing the validator using the instance of the object.

20. A method of executing dynamic validation in programming language frameworks, the method comprising:

receiving, at a programming language framework, an instance of an object, wherein a definition of the object is annotated with a constraint;

receiving one or more validators that are annotated with an annotation that identifies:

an attribute in the definition of the object; and

a value for the attribute;

identifying a validator in the one or more validators for which a value of the attribute in the instance of the object matches the value for the attribute in the annotation of the validator; and

executing the validator using the instance of the object.

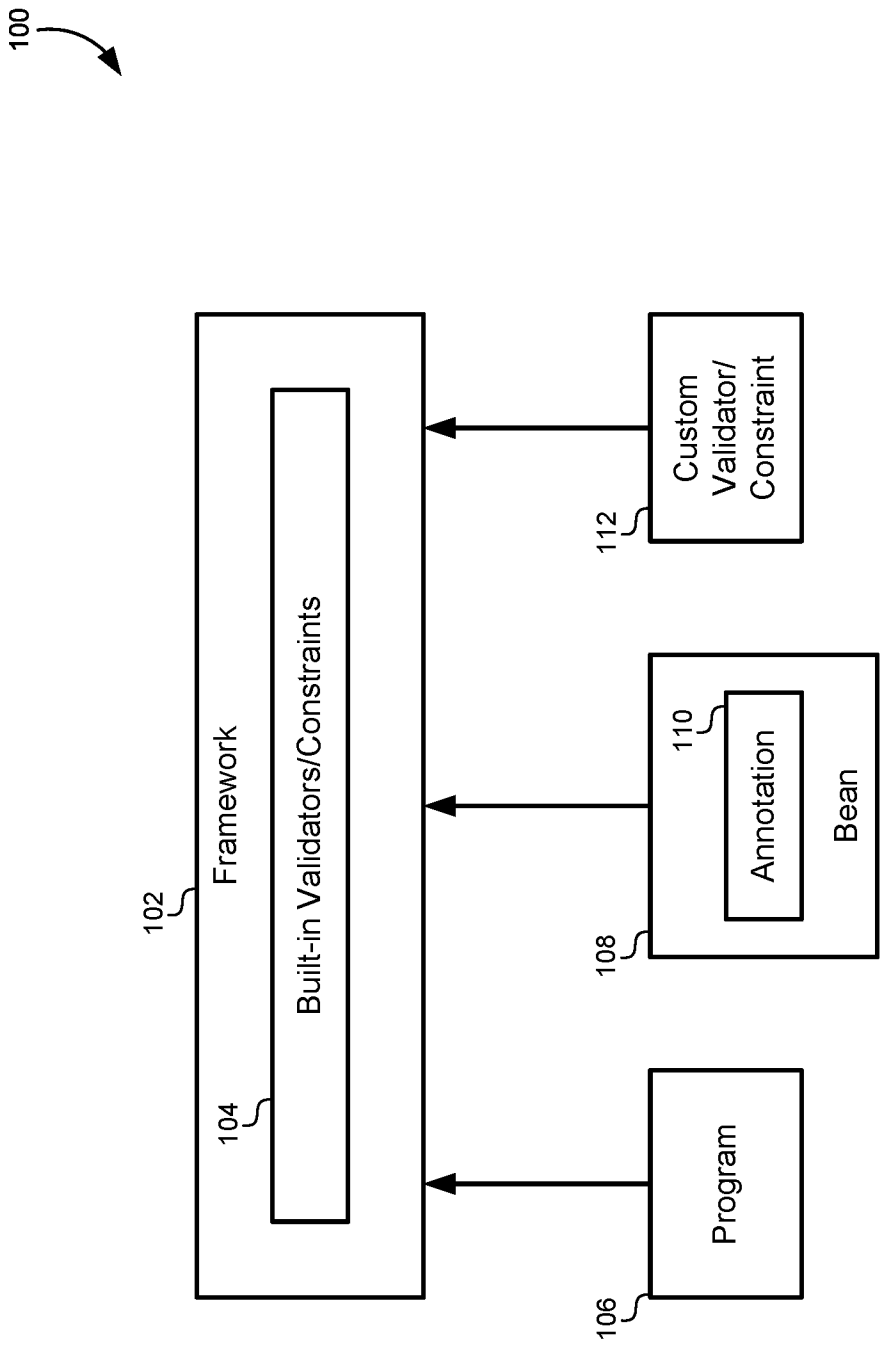

100
102
Framework
104
Built-in Validators/Constraints
Custom Validator/ Constraint
112
110
Annotation
Bean
108
Program
106

FIG. 1

```
public class KeyValue {

   @NotBlank(message = "key is blank")
   private String key;

   private String value;

   public KeyValue(String key, String value){
      this.key = key;
      this.value = value;
   }

   public String getKey() {
      return key;
   }

   public void setKey(String key){
      this.key = key;
   }

   public String getValue() {
      return value;
   }

   public void setValue() {
      this.value = value;
   }

   @Override
      public String toString(){
      return "KeyValue{" + "key='" + key + '\'' + ", value='" + value + '\''+ '}';
   }
}
```

110

108

FIG. 2A

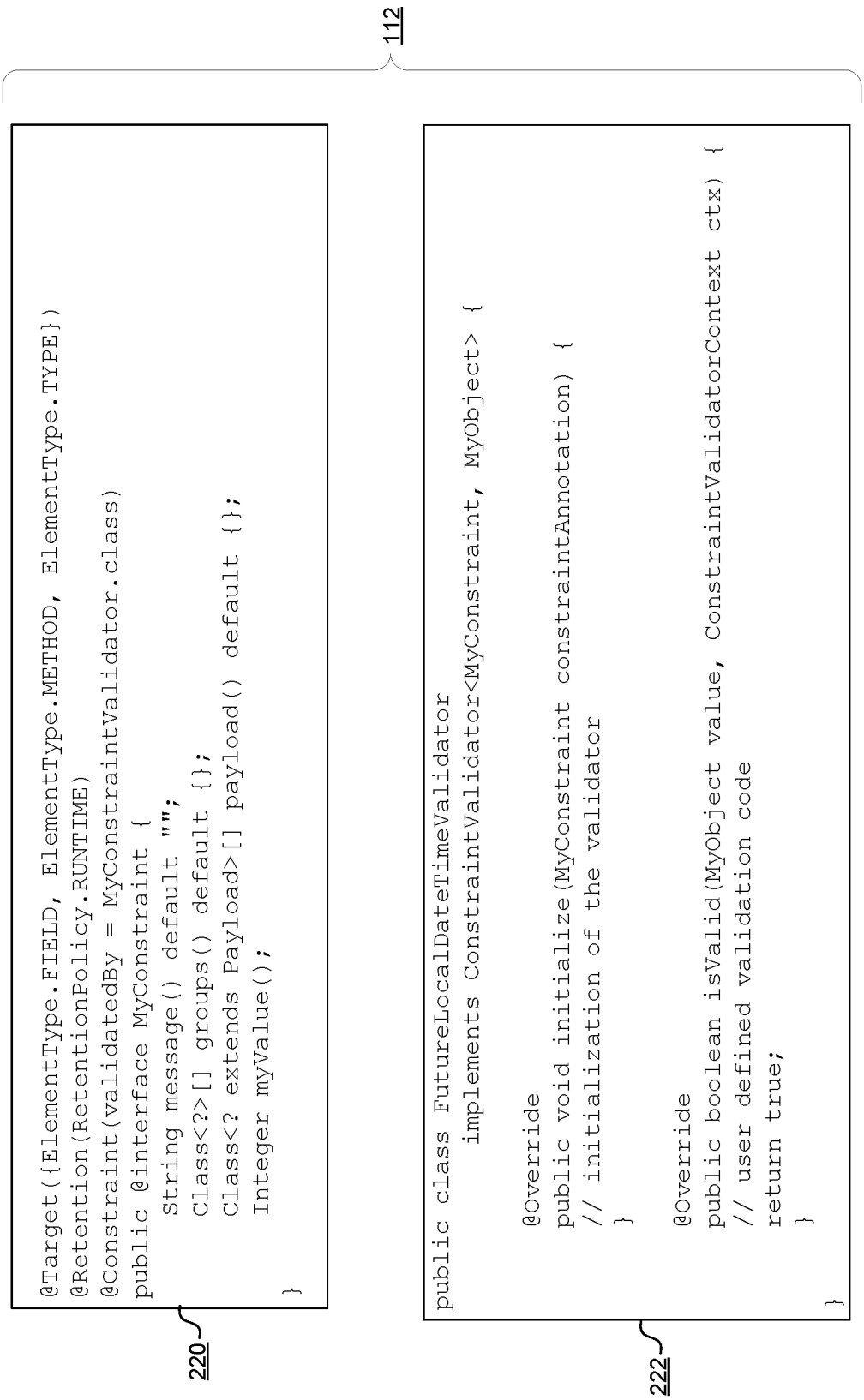

112
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = MyConstraintValidator.class)
public @interface MyConstraint {
String message() default "";
Class<?>[] groups() default {};
Class<? extends Payload>[] payload() default {};
Integer myValue();
}
220
public class FutureLocalDateTimeValidator
implements ConstraintValidator<MyConstraint, MyObject> {
@Override
public void initialize(MyConstraint constraintAnnotation) {
// initialization of the validator
}
@Override
public boolean isValid(MyObject value, ConstraintValidatorContext ctx) {
// user defined validation code
return true;
}
}
222

FIG. 2B

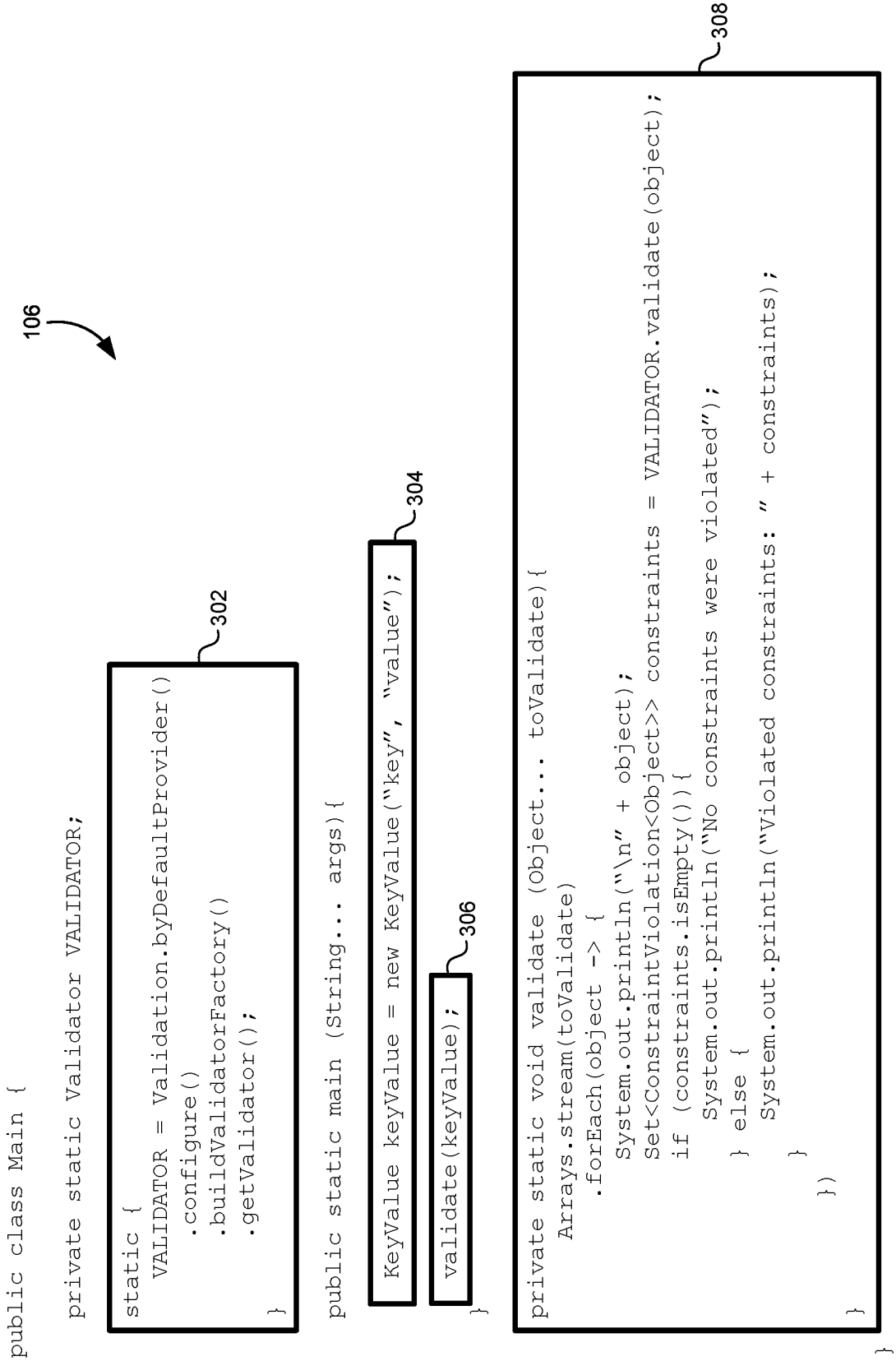

106
public class Main {
private static Validator VALIDATOR;
static {
VALIDATOR = Validation.byDefaultProvider()
.configure()
.buildValidatorFactory()
.getValidator();
}
302
public static main (String... args){
KeyValue keyValue = new KeyValue("key", "value");
304
validate(keyValue);
306
}
private static void validate (Object... toValidate){
Arrays.stream(toValidate)
.forEach(object -> {
System.out.println("\n" + object);
Set<ConstraintViolation<Object>> constraints = VALIDATOR.validate(object);
if (constraints.isEmpty()){
System.out.println("No constraints were violated");
} else {
System.out.println("Violated constraints: " + constraints);
}
})
}
308
}

FIG. 3

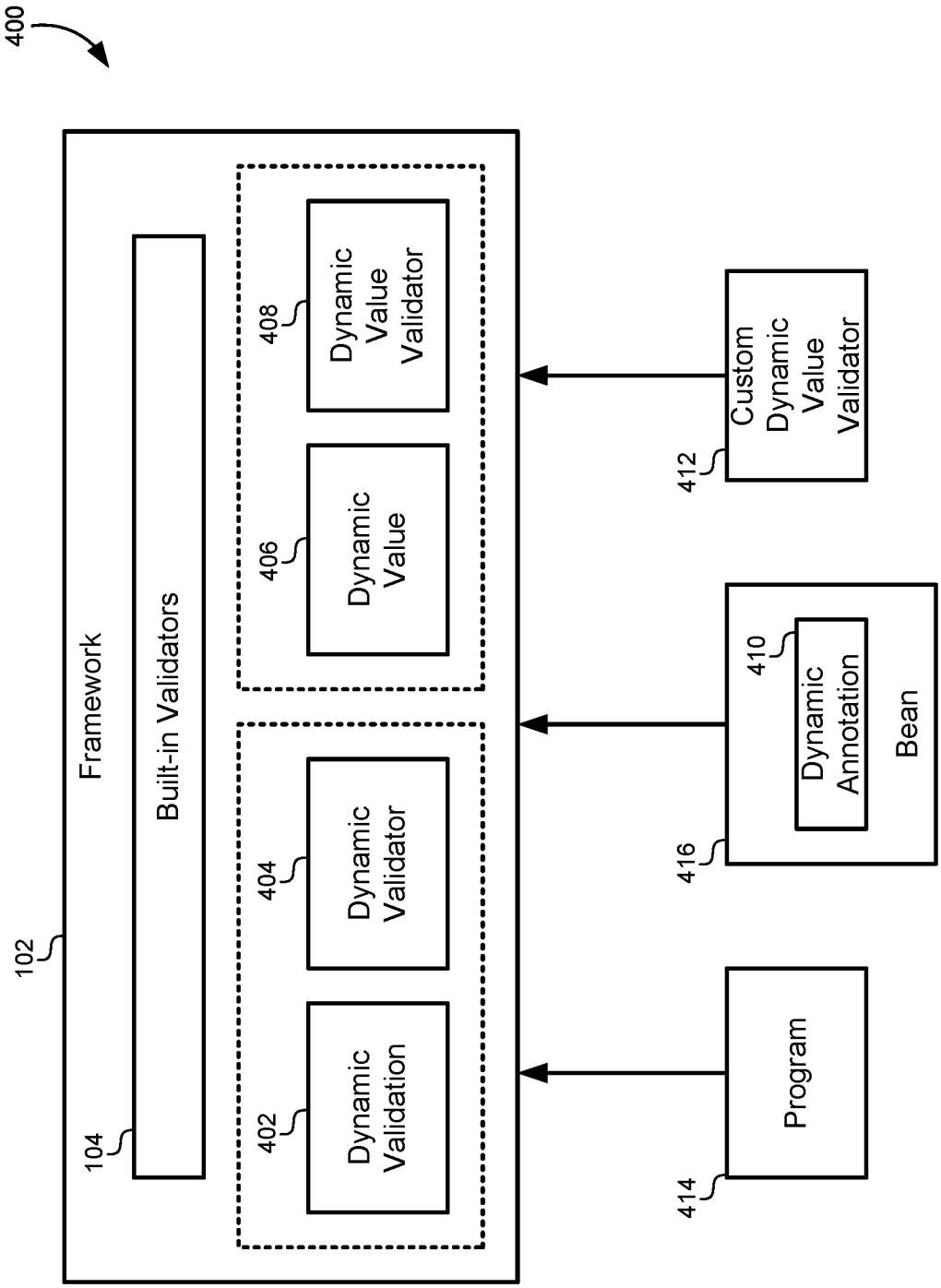

400
Framework
102
Built-in Validators
104
Dynamic Validation
402
Dynamic Validator
404
Dynamic Value
406
Dynamic Value Validator
408
Custom Dynamic Value Validator
412
Dynamic Annotation
410
Bean
416
Program
414

FIG. 4

402
DynamicValidation
@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = DynamicValidator.class)
502
public @interface DynamicValidation {
String message() default "";
504
Class<?>[] groups() default {};
Class<? extends Payload>[] payload() default {};
}

FIG. 5

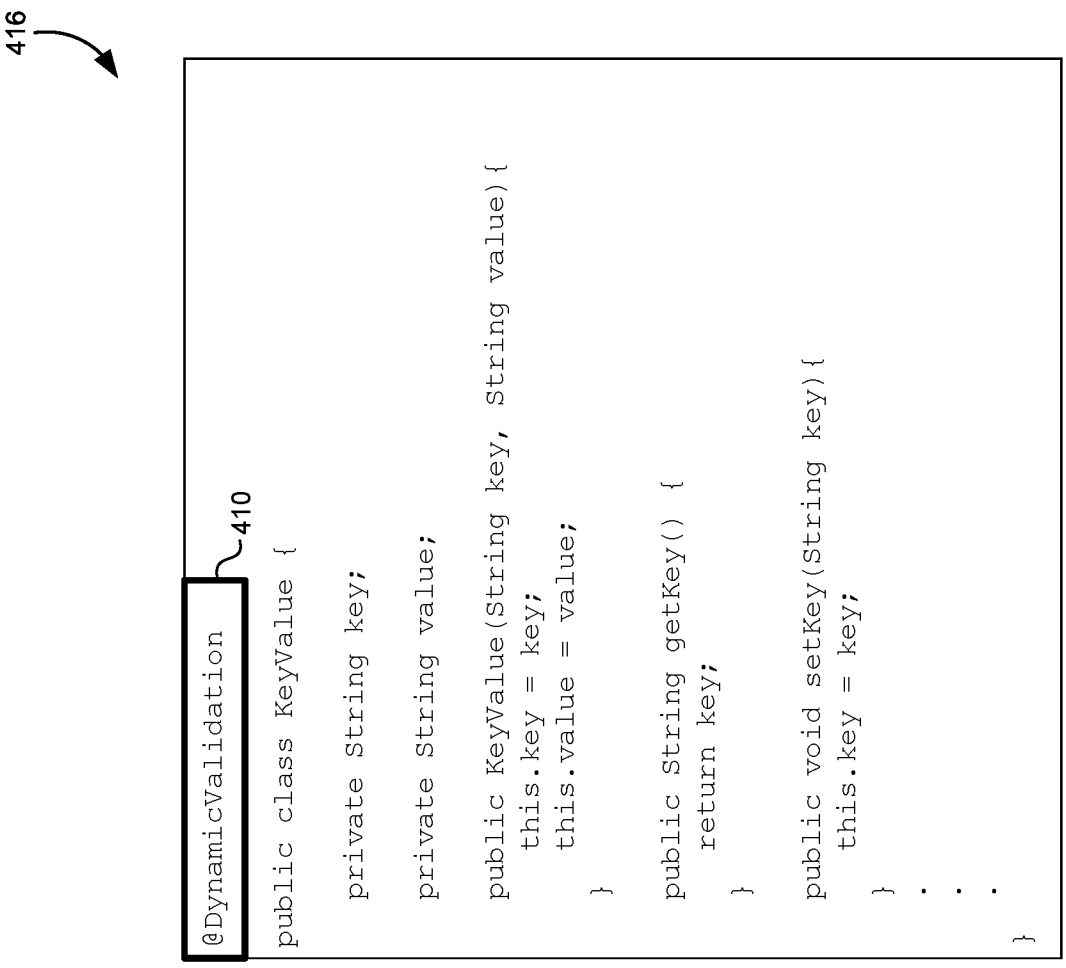

416
@DynamicValidation
410
public class KeyValue {
private String key;
private String value;
public KeyValue(String key, String value){
this.key = key;
this.value = value;
}
public String getKey() {
return key;
}
public void setKey(String key){
this.key = key;
}
. . .
}

FIG. 6

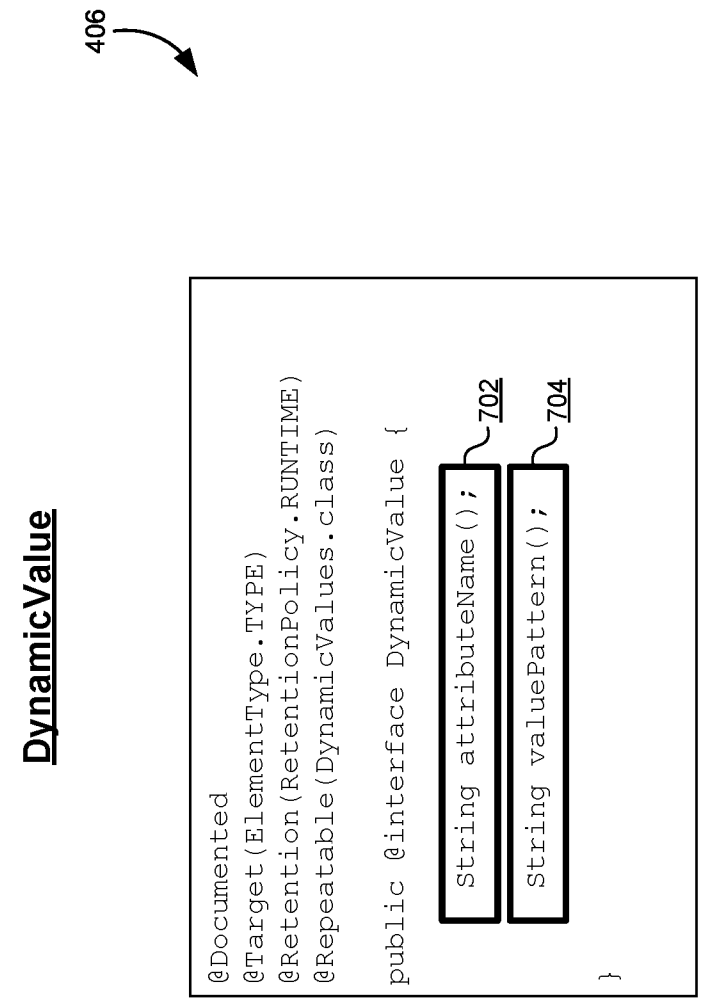

406
DynamicValue
@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(DynamicValues.class)
public @interface DynamicValue {
String attributeName();
String valuePattern();
}
702
704

FIG. 7

**DynamicValueValidator**

408

```
public abstract class DynamicValueValidator<E> {

    private static Validator VALIDATOR;

    static {
        VALIDATOR = Validation.byDefaultProvider()
                .configure()
                .buildValidatorFactory()
                .getValidator();
    }

    public abstract boolean isValid(E value, ConstraintValidatorContext ctx);

    protected final boolean validateCustomConstraints(Object toValidate,
                                                      ConstraintValidatorContext ctx) {
        Set<ConstraintViolation<Object>> violations = VALIDATOR.validate(toValidate);
        if (!violations.isEmpty()) {
            useConstraint(violations.iterator().next().getMessageTemplate(), ctx);
            return false;
        }
        return true;
    }

    protected void useConstraint(String template,
                                 ConstraintValidatorContext ctx) {
        ctx.disableDefaultConstraintViolation();
        ctx.buildConstraintViolationWithTemplate(template).addConstraintViolation();
    }
}
```

802

804

806

FIG. 8

```
public class DynamicValidator implements ConstraintValidator<DynamicValidation, Object> {

   private static Validator VALIDATOR;
   private static List<DynamicValueValidator> dynamicValidators;

   static {
      VALIDATOR = Validation.byDefaultProvider()
               .configure()
               .buildValidatorFactory()
               .getValidator();
    }

   public DynamicValidator() {
      synchronized (DynamicValueValidator.class) {
         if (dynamicValidators == null) {
            Reflections reflections = Reflections(DynamicValidator.class.getClassLoader());
            Set<Class<? extends DynamicValueValidator>>allValidatorCLasses =
                     reflections.getSubTypesOf(DynamicValueValidator.class);
            dynamicValidators = allValidatorCLasses.stream()
                     .filter(c -> !c.isInterface() && !Modifier.isAbstract(c.getModifiers()))
                     .filter(c -> c.isAnnotationPresent(DynamicValue.class)
                     .filter(c -> Stream.of(c.getConstructors()).anyMatch(cnst ->
                                                         cnst.getParameterCount() == 0))
                     .map(c -> {
                         try {
                              return c.newInstance();
                         } catch (InstantiationException | IllegalAccessException e) {
                              return null;
                         }
                     })
                     .filter(Objects::nonNull)
                     .collect(Collectors.toList());
    }   }   }
```

404a

902

FIG. 9A

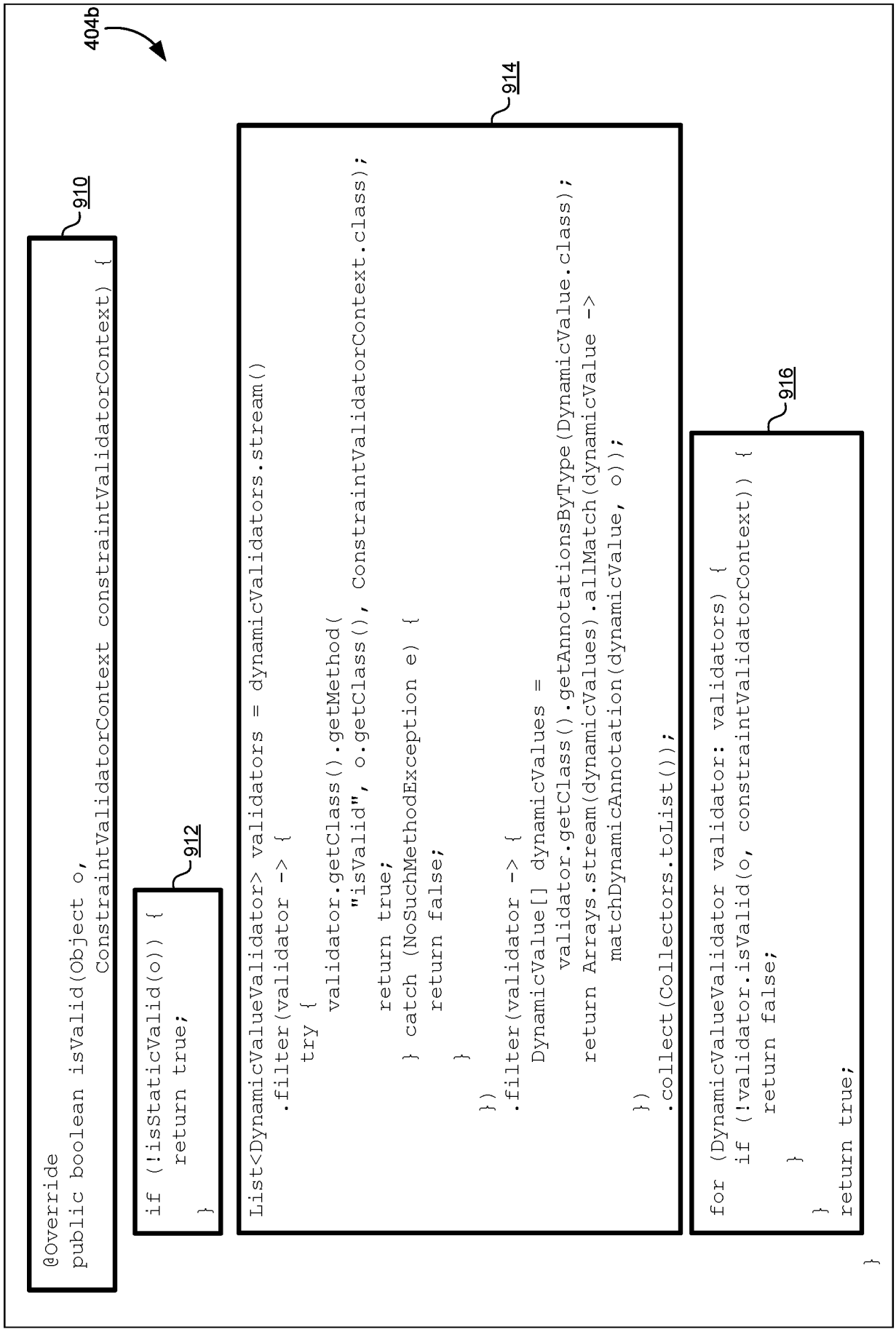

404b
910
912
914
916
@Override
public boolean isValid(Object o,
ConstraintValidatorContext constraintValidatorContext) {
if (!isStaticValid(o)) {
return true;
}
List<DynamicValueValidator> validators = dynamicValidators.stream()
.filter(validator -> {
try {
validator.getClass().getMethod(
"isValid", o.getClass(), ConstraintValidatorContext.class);
return true;
} catch (NoSuchMethodException e) {
return false;
}
})
.filter(validator -> {
DynamicValue[] dynamicValues =
validator.getClass().getAnnotationsByType(DynamicValue.class);
return Arrays.stream(dynamicValues).allMatch(dynamicValue ->
matchDynamicAnnotation(dynamicValue, o));
})
.collect(Collectors.toList());
for (DynamicValueValidator validator: validators) {
if (!validator.isValid(o, constraintValidatorContext)) {
return false;
}
}
return true;
}

FIG. 9B

```
private boolean isStaticValid(Object o) {                                    920

        Set<String> propertyNames = new HashSet<>();
        Class<?> clazz = o.getClass();
        while (clazz != null) {
            propertyNames.addAll(Arrays.stream(clazz.getDeclaredFields())
                    .map(Field::getName)
                    .collect(Collectors.toSet())
            );
            clazz = clazz.getSuperclass();
        }
        return propertyNames.stream().allMatch(p -> VALIDATOR.validateProperty(o, p).isEmpty());
    }

private boolean matchDynamicAnnotation(DynamicValue dynamicValue, Object o) {   922

    Class<?> clazz = o.getClass();
    while (clazz != null) {
        try {
            Field f = clazz.getDeclaredField(dynamicValue.attributeName());
            f.setAccessible(true);
            Object propertyValue = f.get(o);
            return propertyValue != null
                    && propertyValue.toString().matches(dynamicValue.valuePattern());
        } catch (NoSuchFieldException ignored) {

        } catch (IllegalAccessException e) {
            return false;
        }
        clazz = clazz.getSuperclass();
    }
    return false;
}
}
```

404c

FIG. 9C

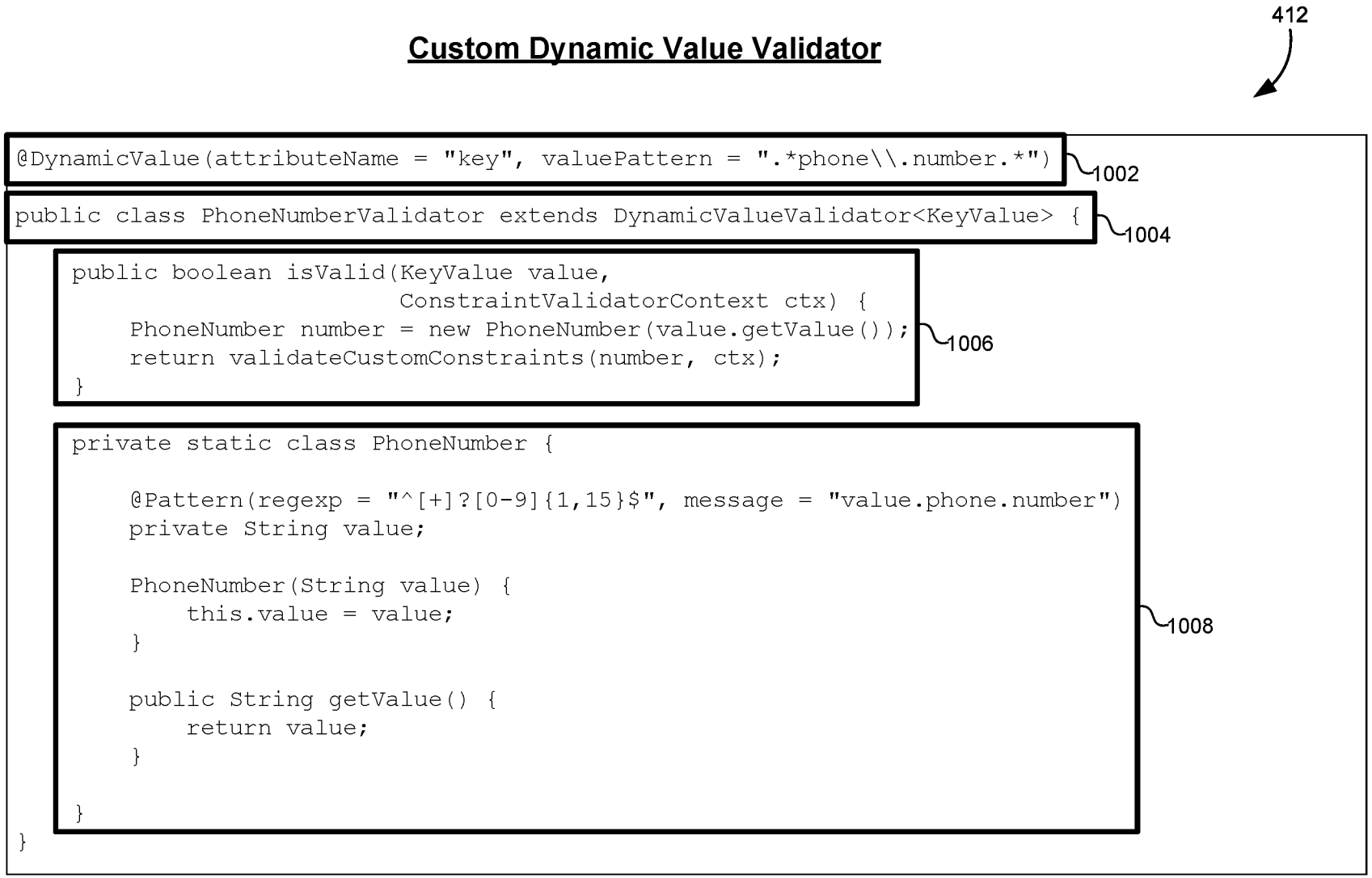

Custom Dynamic Value Validator
412
@DynamicValue(attributeName = "key", valuePattern = ".*phone\\.number.*")
1002
public class PhoneNumberValidator extends DynamicValueValidator<KeyValue> {
1004
public boolean isValid(KeyValue value,
ConstraintValidatorContext ctx) {
PhoneNumber number = new PhoneNumber(value.getValue());
return validateCustomConstraints(number, ctx);
}
1006
private static class PhoneNumber {
@Pattern(regexp = "^[+]?[0-9]{1,15}$", message = "value.phone.number")
private String value;
PhoneNumber(String value) {
this.value = value;
}
public String getValue() {
return value;
}
}
1008
}

FIG. 10A

## Custom Dynamic Value Validator

412

```
@DynamicValue(attributeName = "key1", valuePattern = ".*phone\\.number.*")
@DynamicValue(attributeName = "key2", valuePattern = "CZ")
```
1010

```
public class PhoneNumberValidator extends DynamicValueValidator<TwoKeyValue> {
```
1012

```
    public boolean isValid(TwoKeyValue value,
                           ConstraintValidatorContext ctx) {
        PhoneNumber number = new PhoneNumber(value.getValue());
        return validateCustomConstraints(number, ctx);
    }
```
1014

```
    private static class PhoneNumber {

        @Pattern(regexp = "^[+]?[0-9]{1,15}$", message = "value.phone.number")
        private String value;

        PhoneNumber(String value) {
            this.value = value;
        }

        public String getValue() {
            return value;
        }

    }
}
```

FIG. 10B

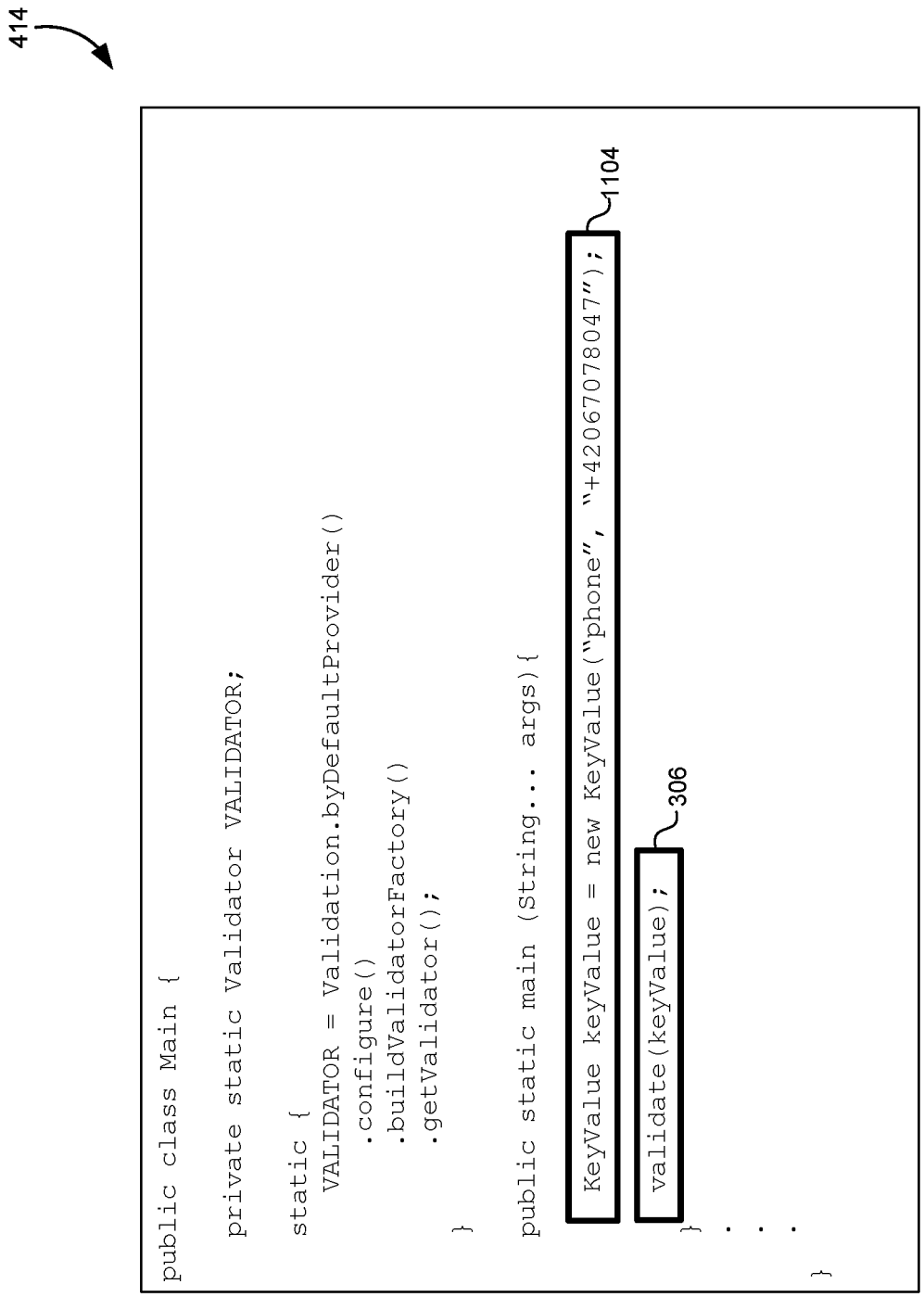

414
public class Main {
private static Validator VALIDATOR;
static {
VALIDATOR = Validation.byDefaultProvider()
.configure()
.buildValidatorFactory()
.getValidator();
}
public static main (String... args){
KeyValue keyValue = new KeyValue("phone", "+420670780 47");
1104
validate(keyValue);
306
}
. . .
}

FIG. 11

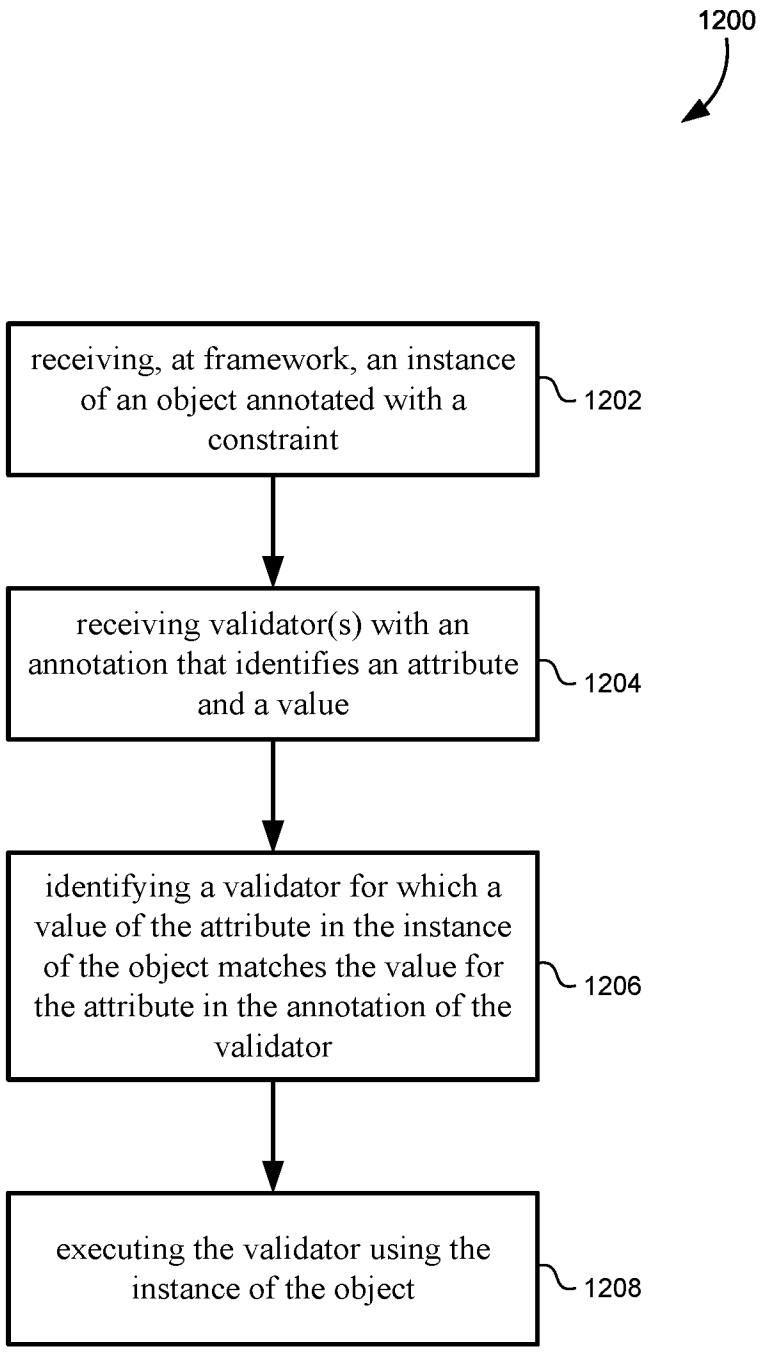

1200
receiving, at framework, an instance of an object annotated with a constraint
1202
receiving validator(s) with an annotation that identifies an attribute and a value
1204
identifying a validator for which a value of the attribute in the instance of the object matches the value for the attribute in the annotation of the validator
1206
executing the validator using the instance of the object
1208

FIG. 12

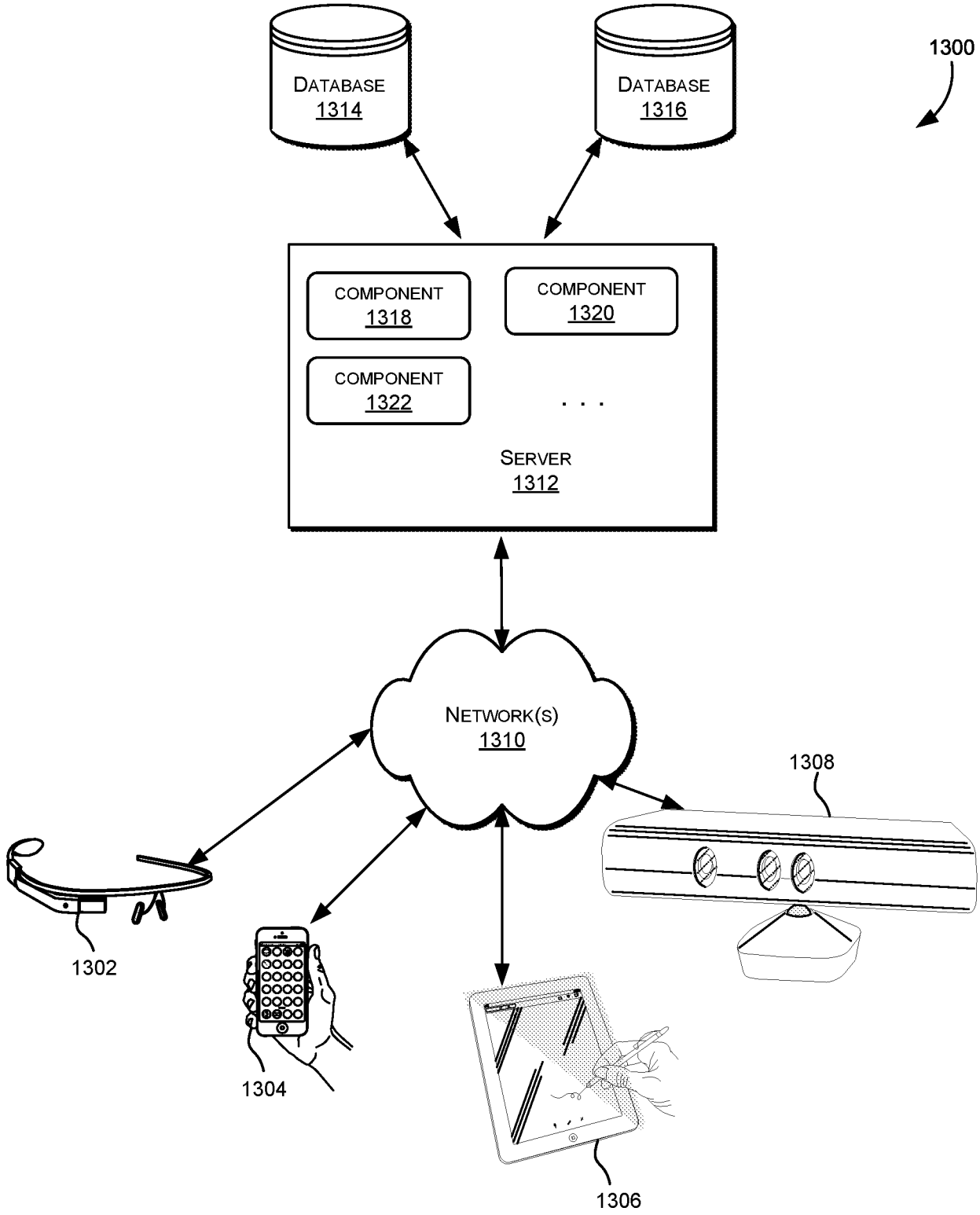

1300
DATABASE
1314
DATABASE
1316
COMPONENT
1318
COMPONENT
1320
COMPONENT
1322
. . .
SERVER
1312
NETWORK(S)
1310
1308
1302
1304
1306

FIG. 13

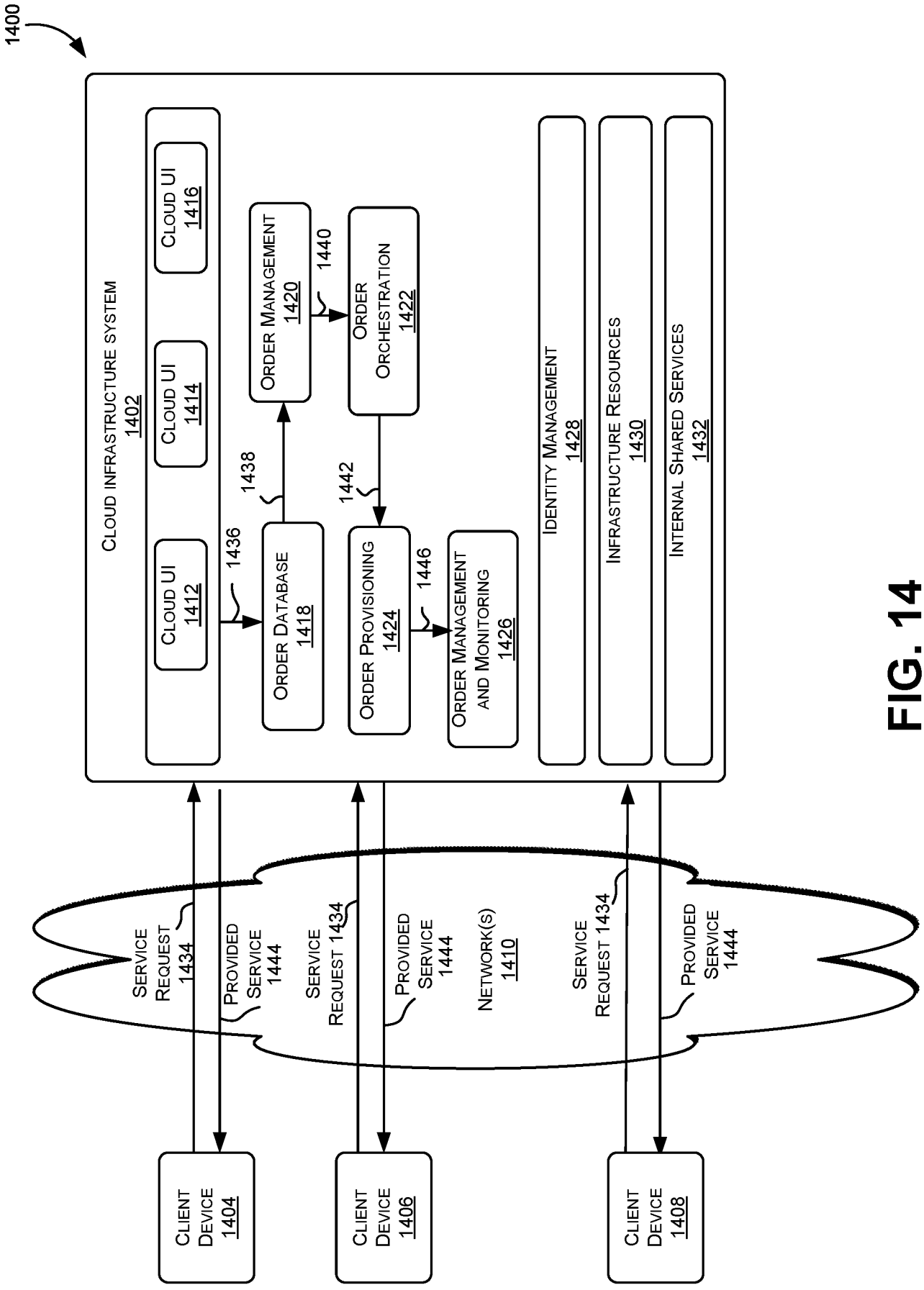

1400
CLOUD INFRASTRUCTURE SYSTEM
1402
CLOUD UI
1412
CLOUD UI
1414
CLOUD UI
1416
ORDER DATABASE
1418
ORDER MANAGEMENT
1420
ORDER
ORCHESTRATION
1422
ORDER PROVISIONING
1424
ORDER MANAGEMENT
AND MONITORING
1426
IDENTITY MANAGEMENT
1428
INFRASTRUCTURE RESOURCES
1430
INTERNAL SHARED SERVICES
1432
1436
1438
1440
1442
1446
SERVICE
REQUEST
1434
PROVIDED
SERVICE
1444
NETWORK(S)
1410
CLIENT
DEVICE
1404
CLIENT
DEVICE
1406
CLIENT
DEVICE
1408

FIG. 14

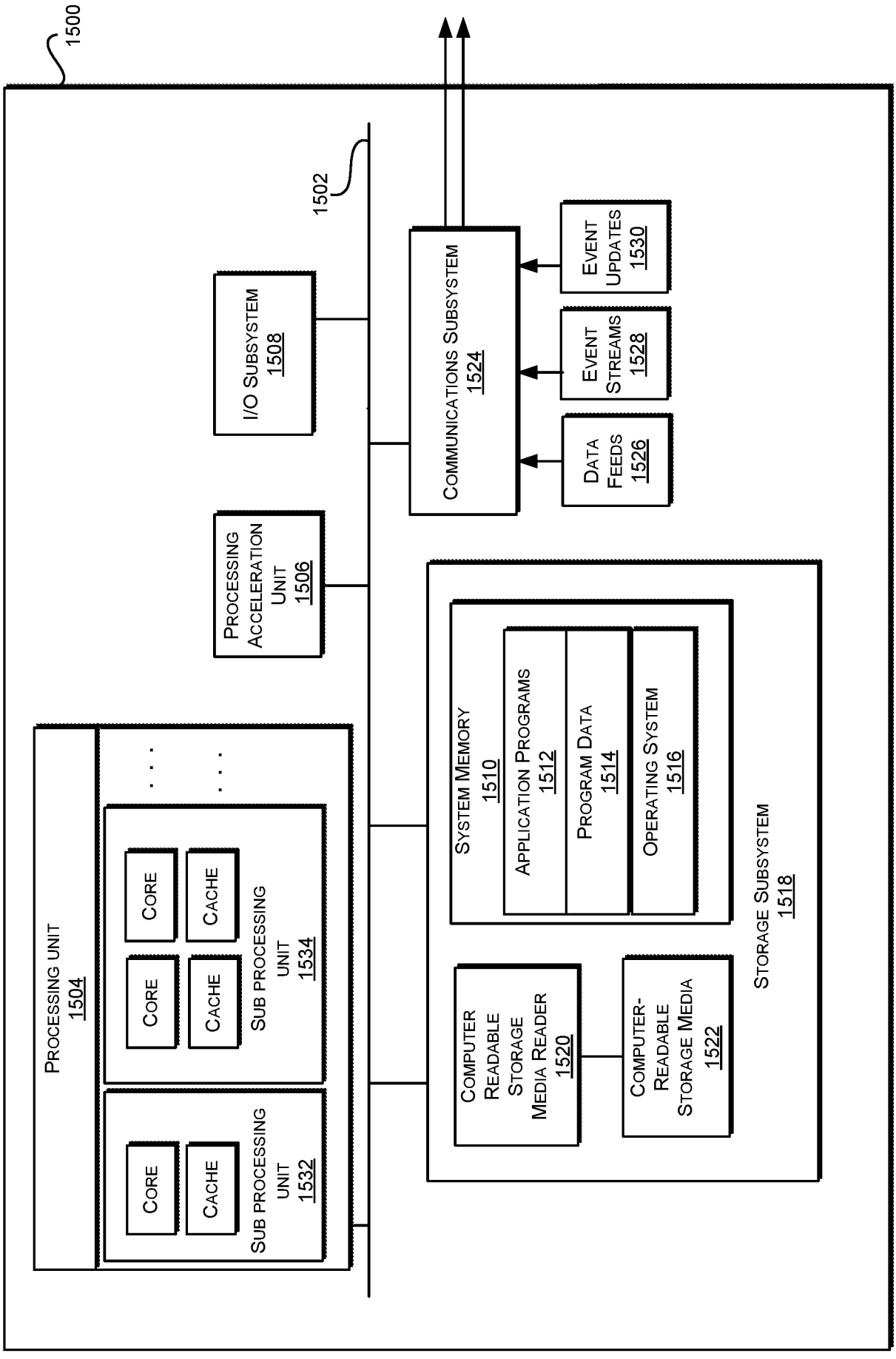

1500
1502
I/O SUBSYSTEM 1508
COMMUNICATIONS SUBSYSTEM 1524
EVENT UPDATES 1530
EVENT STREAMS 1528
DATA FEEDS 1526
PROCESSING ACCELERATION UNIT 1506
PROCESSING UNIT 1504
CORE
CACHE
SUB PROCESSING UNIT 1532
SUB PROCESSING UNIT 1534
SYSTEM MEMORY 1510
APPLICATION PROGRAMS 1512
PROGRAM DATA 1514
OPERATING SYSTEM 1516
COMPUTER READABLE STORAGE MEDIA READER 1520
COMPUTER-READABLE STORAGE MEDIA 1522
STORAGE SUBSYSTEM 1518

FIG. 15

# INTERNATIONAL SEARCH REPORT

International application No
PCT/US2021/014127

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F11/36 G06F8/41
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EPO-Internal

C. DOCUMENTS CONSIDERED TO BE RELEVANT

| Category* | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| X | That Java Guy: "The Wayback Machine -https Bean Validation: Validation based on multiple properties", , 3 September 2018 (2018-09-03), XP055796279, Retrieved from the Internet: URL:https://web.archive.org/web/20190628044600/http://that-java-guy.blogspot.com:80/2018/09/bean-validation-validation-based-on.html [retrieved on 2021-04-16] the whole document ----- -/-- | 1-20 |

[X] Further documents are listed in the continuation of Box C. [ ] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

| Date of the actual completion of the international search | Date of mailing of the international search report |
|---|---|
| 16 April 2021 | 26/04/2021 |

| Name and mailing address of the ISA/ | Authorized officer |
|---|---|
| European Patent Office, P.B. 5818 Patentlaan 2<br>NL - 2280 HV Rijswijk<br>Tel. (+31-70) 340-2040,<br>Fax: (+31-70) 340-3016 | Hoareau, Samuel |

1

Form PCT/ISA/210 (second sheet) (April 2005)

# INTERNATIONAL SEARCH REPORT

International application No
PCT/US2021/014127

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

| Category* | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| A | Anonymous: "Explore Annotations in Java 8 - DZone Java", , 9 November 2019 (2019-11-09), XP055796193, Retrieved from the Internet: URL:https://web.archive.org/web/20191109143033/https://dzone.com/articles/explore-annotations-in-java-8 [retrieved on 2021-04-16] the whole document ----- | 1-20 |

1