2 Categories of Constraints

In an attempt to find out what kind of constraints are typical for object-oriented libraries and application frameworks, we have searched the Java 1.2 API documentation [Sun Microsystems, Inc. 1999a] of the standard java classes [Chan, Lee 1997, Chan et al. 1998, Chan et al. 1999] for occurrences of the words "should", "must", "require", "ensure", "expect", and "need".

The standard java classes consist of 35 packages whose names start with "`java.`", containing a total of 823 classes. According to the conventions for documenting Java source code, the standard Java classes contain *formal comments* starting with "`/**`" which describe the classes, interfaces, methods, and fields. Using JAVADOC, a tool that is part of Sun's Java development kit (JDK), hyperlinked HTML files can be generated which contain the documentation. We have used the UNIX tool GREP on the HTML files for searching for occurrences of the above-mentioned words.

Each occurrence of the search words "should", "must", "require", "ensure", "expect", and "need" has been checked for whether it can be understood as a constraint or whether it is a spurious occurrence. Using this method, a total of 592 constraints have been found in 210 classes. Most probably, more constraints could be found in the standard Java classes which either are not documented at all or are documented using words that do not include one of the words we have searched for. However, we believe that the number of constraints that have been found is large enough to be a representative set of constraints found in today's object-oriented software.

Two constraints seem to be impossible to enforce programmatically, because they refer to semantic properties of the program which probably can only be understood by a human. For example, the documentation of the empty interface `java.util.EventListener` states:

> "A tagging interface that all event listener interfaces must[1] extend"

This interface, which is not used at all in the Java standard classes, should be used to mark classes that play an event listener role as opposed to classes that produce events, in order to make it easier for a human reader to distinguish between the two. The other constraint can be found in `java.rmi.server.UnicastRemoteObject` ("Objects that require remote behavior should extend `RemoteObject`").

In a sense, the two constraints that cannot be enforced automatically are similar to the following important rule: "Use meaningful names!" Clearly, this rule can only be checked by human readers, not by a tool that performs checks algorithmically.

[1] Occurrences of the searched words are underlined in cited documentation text.

Each of the remaining 590 occurrences has been classified using three independent classification dimensions, which emerged during several passes over the list of occurrences of the search words. The classification dimensions are described in Section 2.1. The resulting constraint categories are explained by means of example constraints from the Java standard classes in Sections 2.2 and 2.3. In Section 2.4, other possible categorization schemes are discussed.

2.1 Classification dimensions

By iterating repeatedly over the list of search word occurrences in the documentation of the Java standard classes, we have found three independent classification dimensions. In this section, we will explain each classification dimension in turn.

2.1.1 Static constraints vs. dynamic constraints

One can distinguish between constraints that can be checked at compile-time and constraints that can only be checked at run-time. We call the former *static constraints* and the latter *dynamic constraints*.

A good example of a static constraint can be found in the documentation of the class `java.rmi.RemoteException`:

> "Each method of a remote interface, an interface that extends `java.rmi.Remote`, must list `RemoteException` in its throws clause."

Clearly, this constraint can be checked at compile-time.

The documentation of the static method `copy` in the class `java.util.Collections` contains a dynamic constraint regarding its arguments `src` and `dest` of type `java.util.List`:

> "The destination list must be at least as long as the source list. If it is longer, the remaining elements in the destination list are unaffected."

This constraint refers to run-time entities, objects of type `java.util.List`, the lenghts of which cannot be known – except for some special cases – at compile-time.

Of the 590 checkable constraints, we have found 401 dynamic constraints and 189 static constraints. For some of the constraints, the distinction was not as clear as in the above examples. When in doubt, we have classified a constraint as a dynamic constraint — see Section 2.3 for a discussion of the types of dynamic constraints which could be classified as static constraints under certain circumstances.

2.1.2 Type constraints vs. method constraints vs. field constraints

All constraints that have been found are *associated* with a particular program element (i.e., a class, an interface, a method, a constructor, or a field) without which the constraints would not exist. For example, the documentation of the class `java.awt.PrinterJob` states:

> [A] "`PrinterJob` object should be created using the `static getPrinterJob` method".

In other words, objects of class `PrinterJob` should not be created using the `new` operator. This constraint is associated with a type (i.e., a class or an interface; in this case: the class `PrinterJob`) because it would not exist if the class `PrinterJob` did not exist.

If a constraint is associated with a class (as in our example) or an interface, we call it a *type constraint*. If it is associated with a method or a constructor, we call it a *method constraint*, regarding constructors as a special kind of methods. Finally, if the constraint is associated with a field, we call it a *field constraint*.

As an example for a method constraint, i.e., a constraint that is associated with a method, consider the documentation for method `readObjectOverride` in class `java.io.ObjectInputStream`:

> "This method is called by trusted subclasses of `ObjectInputStream` that construct an `ObjectInputStream` using the `protected` no-arg constructor. The subclass is expected to provide an override method with the modifier `final`."

This constraint is a method constraint because if this method was not needed in the interface of `ObjectInputStream`, the constraint would not exist.

There are a small number of field constraints as well. For example, the documentation for `java.awt.AWTEvent` states:

> "The event masks defined in this class are needed ONLY by component subclasses which are using `Component.enableEvents()` to select for event types not selected by registered listeners."

This constraint, which is associated with certain static fields in class `AWTEvent`, is a field constraint because it would not exist if these fields did not exist.

Of the 590 checkable constraints, we have found 109 type constraints, 479 method constraints, and 2 field constraints, respectively.

2.1.3 Definition constraints vs. usage constraints

Given a particular program element (type, method, or field), and a constraint that is associated with it, one can distinguish between constraints that govern the *definition* of that program element and constraints that govern the *usage* of that program element. The places

in a program where a program element might be used differ depending on the used program element: Types can be used in declarations, as the declared type, or in expressions, as the expression's static type, or in the definition of other types, as their supertypes. Methods can be used in method call expressions, or in the special case of constructors, they can be used in implicit or explicit constructor calls. Fields can be used in field access expressions.

To understand the distinction between the two kinds of constraints, it is helpful to proceed in two steps: In the first step, we explain the distinction as if subtyping did not exist, and in the second step, we explain how subtyping is handled.

First, without taking subtyping into account, it is easy to explain the distinction by considering a constraint which is violated: If the violation is due to the program element with which the constraint is associated, it is a *definition constraint*. If, however, the violation is due to another part of the program in which the program element is used, it is a *usage constraint*.

The constraint associated with the method `getAlignment` in `java.awt.Component` is a good example for a definition constraint[2]:

> "The [returned] value should be a number between 0 and 1 where 0 represents alignment along the origin, 1 is aligned the furthest away from the origin, 0.5 is centered, etc."

A violation of this constraint would be caused by an incorrect implementation of `getAlignment`, and hence, the constraint is a definition constraint.

In the same class, there is a usage constraint associated with the public method `addNotify`:

> "This method is called internally by the toolkit and should not be called directly by programs[3] ."

A violation of this constraint would be caused by a non-toolkit class calling `addNotify`, and hence, the constraint is a usage constraint.

After motivating the difference between definition and usage constraints, we now proceed to the second step and take subtyping into account:

In general, a definition constraint applies not only to the program element associated with the constraint, but also – by the principle of substitutability [Liskov, Wing 1994] – to derived types or overriding methods that might, at run-time, be substituted for the associated program element. Thus, a definition constraint that is associated with a class `C` applies not only to `C` itself, but also to all subclasses of `C`. Similarly, a definition constraint that is associated with a method `m` applies not only to `m` itself, but also to all methods that override `m`. In the above example, the constraint applies not only to the implementation of `getAlignment` in `java.awt.Component`, but to all implementations of `getAlignment` in subclasses of `java.awt.Component` as well. In fact, a subclass of `java.awt.Component`, the class

[2] As we will see, the given constraint, a method postcondition, is a dynamic definition constraint. Note that method preconditions are dynamic usage constraints, not dynamic definition constraints.

[3] The reason for this constraint (instead of making the method non-public) is that some toolkit classes belong to other packages.

`java.awt.Container`, overrides the method `getAlignment` and notes the same definition constraint in its documentation. The reason for definition constraints applying to derived program elements as well is that users of the particular program element rely on the constraint being satisfied. However, at run-time, they might actually be using a derived program element, which therefore must satisfy the constraint as well.

A usage constraint applies to all places in the complete program where the program element associated with the constraint is used. An important question is whether usage constraints can be weakened for derived program elements, or whether they should also apply to all places where derived program elements are used. The derived program elements of types are their subtypes, and the derived program elements of methods are their overriding methods. Based on the – limited – experience with CoffeeStrainer, we have decided that usage constraints cannot be weakened for derived program elements; they can only be made stronger. We have made this decision for pragmatic reasons listed below, despite the fact that the vaguely related concept of preconditions (as known, for example, from Eiffel [Meyer 1992]), can indeed be weakened in subclasses. The reasons are as follows:

- From the constraints found in the Java standard classes, no usage constraints were weakened for derived program elements. Instead, usage constraints always apply to derived program elements as well. In the above example, non-toolkit classes were not allowed to call the method `addNotify` in class `java.awt.Component`. Clearly, if a subclass of `java.awt.Component` had a method `addNotify` (overriding the original method), it should not be possible for non-toolkit classes to call this method either. In fact, a subclass of `java.awt.Component`, the class `java.awt.Container`, overrides the method `addNotify` and notes the same usage constraint in its documentation.

- The context of the particular program element (in our example, the "toolkit") may assume that a constraint established for the usage of this program element cannot be weakened by derived program elements. This is particularly important in cases where subtypes may be defined that are outside of the context, for example when a subclass of `java.awt.Component` is defined which is not part of the "toolkit".

- There are usage constraints which are based on syntactic notions – usages of program elements –, unlike preconditions, which are based on a precise semantic notion, namely the invocation of a method at run-time. This makes it difficult to argue formally whether or not usage constraints can be weakened for derived program elements. The safe choice, then, is to say that usage constraints cannot be weakened.

To sum up, a constraint that, for example, applies to all usages of an interface `I` (e.g., in variable declarations) also applies to all usages of subtypes of `I` as well (e.g., it applies to variable declarations with a static type `U` which is a subtype of `I`), and a constraint that applies to all usages (i.e., method calls) of a method `m` also applies to all usages of methods that override `m`. Because there are no derived program elements from a field, field usage constraints only apply to usages (i.e., field accesses) of the field itself.

Of the 590 checkable constraints, we have found 175 definition constraints and 415 usage constraints.

It is noteworthy that both field constraints that have been found are field usage constraints. This is no surprise when considering the two main motivations for definition constraints: first, to constrain overriding definitions in subtypes, and second, to ensure properties of the defined program element itself. The first motivation does not exist for field definitions because in Java fields cannot be overridden by subclasses. The second motivation only applies if the definition to which it applies is more complex than the specification of the constraint, which can be true for class, interface, or method definitions, but hardly for field definitions.

2.1.4 Resulting categories

Figure 2.1 shows how many example constraints have been found in the twelve resulting categories of constraints. As already noted, field definition constraints do not make much sense given that Java does not allow overriding of fields in subclasses, leading to only ten categories of constraints. Note that dynamic field usage constraints might be useful, although no example for this category has been found in the Java standard classes. For example, dynamic field usage constraints might be used to disallow access to static fields in classes that are not yet properly initialized during class loading. Such field accesses, if allowed, could lead to different and difficult to understand behavior depending on the order in which classes are loaded.

	static constraints		dynamic constraints	
	definition	usage	definition	usage
type constraints	68	19	6	16
method constraints	43	57	58	321
field constraints	(0)	2	(0)	0
total	111	78	64	337
	189		401	

Figure 2.1: Constraint categories

2.2 Static constraints

The following five sections on the different categories of static constraints give an overview of the constraints that have been found in the Java standard classes for each category. In each section, we identify the three packages that contain the most constraints of the corresponding category and discuss the constraints in these packages. Each section also describes a typical constraint from one category in detail. In Chapter 3, we will use these constraints as examples to explain how CoffeeStrainer works.

2.2.1 Type definition constraints

Static type definition constraints can be found in many packages, as shown in Figure 2.2. We will first describe the constraints that can be found in the three packages that contain most

of the static type definition constraints.

- In package `java.util`, containing mostly collection classes, virtually all of the static type definition constraints are constraints associated with interfaces that require that implementing classes have certain kinds of constructors in order to be consistent with the existing collection classes. In Java, interfaces cannot contain abstract constructors—within the language, it is not possible to express the requirement that implementing classes should provide certain constructors.
- The package `java.rmi.activation` contains classes that support remote object activation in the context of the remote method invocation (RMI) framework. Like in package `java.util`, there are constraints that require activatable objects to have constructors of a certain form. Additionally, there are static type definition constraints that require certain `export` methods to be called during a remote object's activation.
- In package `java.awt`, two new kinds of static type definition constraints can be found: First, some of the classes in the package need to be immutable, i.e., instance fields should only be written to in the classes' constructors. Second, there is an interesting constraint for class `FontMetrics`. Many methods of this class form closed, mutually recursive loops — with all methods being concrete methods — so that subclasses need to override one method of each such loop to prevent infinite recursion.

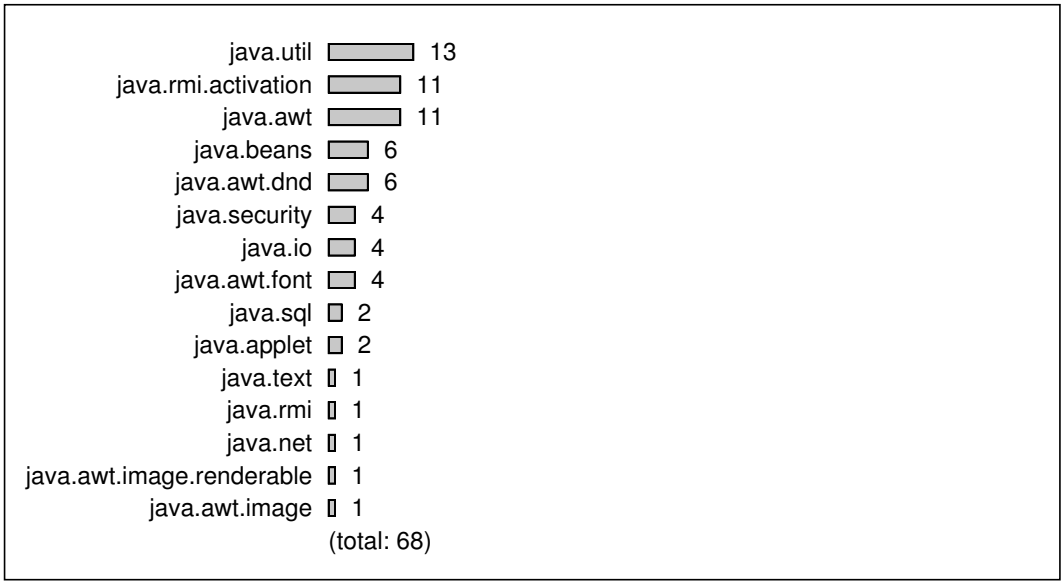

Figure 2.2: Static type definition constraints per package

We now describe an interesting static type definition constraint in detail that has been found in package `java.io`.

In Java, objects that implement the empty marker interface `Serializable` may be saved into a stream for later retrieval. Normally, marking a class as implementing `Serializable` is all a programmer needs to do for making it a serializable class. However, there is an

additional constraint that the programmer must be aware of. The "Java Object Serialization Specification" [Sun Microsystems, Inc. 1999b] states (emphasis added):

> "A `Serializable` class must do the following:
>
> - Implement the `java.io.Serializable` interface
> - Identify the fields that should be serializable (Use the `serialPersistentFields` member to explicitly declare them serializable or use the `transient` keyword to denote nonserializable fields.)
> - *Have access to the no-arg constructor of its first nonserializable superclass*"

Consider an object b of class B, which is serializable, and the non-serializable superclass A of B. During serialization of b, only the values of fields declared in B are written to the stream[4]. Then, if the serialized object is retrieved again, a new object of class B will be created, without calling B's constructor (since B's fields will be set to their saved values); but for initializing the fields declared in A, the constructor with no arguments (no-arg constructor) of A will be called. Since no constructor of B is called, there is no sensible way of calling a constructor of A that needs an argument.

Unfortunately, this constraint is not enforced statically; even worse, a constraint violation becomes apparent only when already serialized objects are retrieved from a stream. As the API documentation for `java.io.Serializable` puts it,

> "To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields. The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state. It is an error to declare a class Serializable [if there is no such constructor][5]. The error will be detected at runtime."

It would be easy to detect the error at compile-time, by checking that a no-arg constructor exists in the superclass. However, one might be reluctant to incorporate such checks in the Java compiler, because object serialization is not part of the Java programming language; it is realized as a standard library.

2.2.2 Method definition constraints

The most static method definition constraints have been found in the packages `java.awt` and `java.lang` (see Figure 2.3):

[4] It is possible to write the values of A's fields as well if B has access to A's fields and if it implements a serialization method.

[5] This sentence is incomprehensible in the original documentation; it reads "It is an error to declare a class Serializable in this case".

- The package `java.awt` contains three kinds of static method definition constraints: First, some methods should never be declared `synchronized` when overridden in subclasses. Second, for a number of methods, overriding methods are required to call the overridden method (see below for an extended example). Third, there are abstract methods whose implementations are expected to call the Java security manager for checking whether access to these methods can be granted.
- The static method definition constraints in package `java.lang` apply to the class `SecurityManager`. When subclasses override methods of this class, the overridden method in the superclass needs to be called because otherwise security could be compromised.

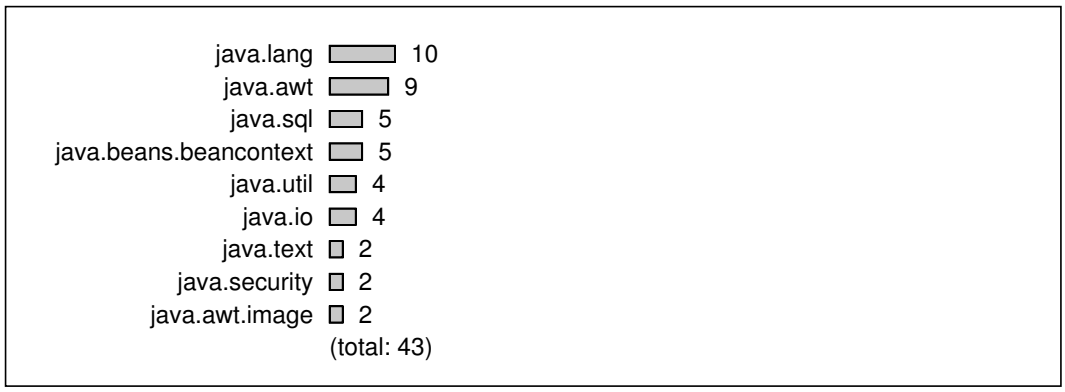

Figure 2.3: Static method definition constraints per package

We now present a typical example for the common constraint that an overriding method should call the overridden method.

The class `java.awt.Container`, responsible for managing graphical user interface components that contain embedded components, contains several `add` methods for adding embedded components. All of these methods forward the call to a generic method called `addImpl` which performs the addition and – if necessary – notifies the layout manager registered for the container object. For certain purposes, e.g. for tracking all add requests to a container, this method may be overridden in subclasses. However, as the documentation states,

> "an overriding method should usually include a call to the superclass's version of the method".

This rule is not very precise and leaves room for several possible interpretations. One interpretation could be the following: the first statement in overriding methods must be a call to the overridden method. Only after this call to `super`, additional actions are allowed.

2.2.3 Type usage constraints

Static type usage constraints, shown in Figure 2.4, are not as common as static type definition constraints. We will first examine the three packages that contain the majority of static type

usage constraints:

- In package `java.lang`, constraints apply to certain types of exceptions that should not be caught by application code. Most notably, `ThreadDeath`, the exception used for aborting threads preemptively, should not be caught as this prevents aborting threads. In a context where security is important, e.g. in a web browser that displays Java applets, this constraint is important because some of the security properties rely on the fact that threads that an applet has started cannot continue to run when the web page containing the applet is no longer displayed.
- The package `java.security`, which provides the classes and interfaces for the Java security framework, contains a number of classes each of which have two kinds of clients, *providers* of security infrastructure on one hand and *users* of security services on the other hand. The constraints in this package state that the different clients of these classes should call only methods from their respective method subsets because these classes cannot be split into smaller classes for compatibility reasons with previous versions of the security framework.
- The three static type usage constraints in package `java.io` are synchronization constraints in the interfaces `Reader` and `Writer` (see below for a detailed explanation of this constraint), and a constraint in class `FileDescriptor` that states that applications should not create their own instances of class `FileDescriptor`. It seems that this constraint cannot be enforced using Java's access modifiers, because other Java standard classes in other packages need access to this class.

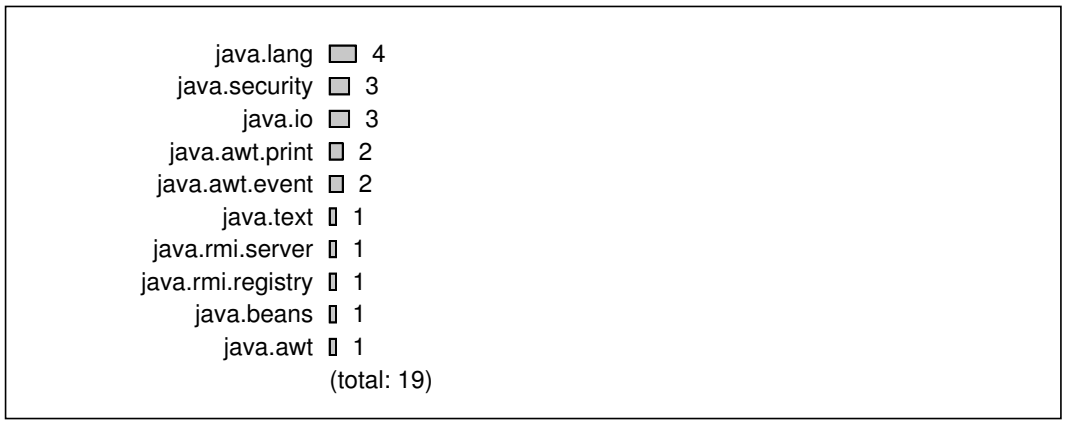

Figure 2.4: Static type usage constraints per package

We now describe a synchronization constraint found in package `java.io` in more detail.

The standard Java library contains a hierarchy of classes for writing to character-based streams. The common superclass of this hierarchy is the abstract class `java.io.Writer`, which provides methods for writing a single character, a character array, or a `String`. Concrete functionality, such as writing to a file, or writing to an array of characters, is provided by subclasses of `Writer`. Additionally, certain subclasses of `Writer` define character streams that may form a chain. For example, normal writers do not perform buffering, but

when buffering is desired, a `BufferedWriter` may be used that is based on the unbuffered `Writer`. Another example is the class `PrintWriter`, which provides convenient methods for writing `String` representations of objects and values of various types based on the simpler `String`-writing methods of an underlying `Writer`. For example, the following chain of `Writer` objects is a common idiom in Java for character-based writing to a file:

```
PrintWriter out = new PrintWriter(
                        new BufferedWriter(
                              new FileWriter("foo.out")));
```

Accesses to `Writer` objects need to be synchronized, such that if several threads use the same `Writer` object concurrently, the internal representation of the character stream remains consistent, and calls to `Writer` objects are properly serialized. One solution for this would be to define all methods on `Writer` objects as synchronized methods. Unfortunately, this simple solution would result in duplicate synchronization in all of the chained `Writer` objects if there is more than one `Writer` object in a chain. For this common case, it would be more efficient to only synchronize once on the front-end `Writer` object. Determining the front-end object, however, is not possible, because this object itself might be wrapped in another `Writer` object, and it is possible that there is no single front-end object if some client calls are issued directly to `Writer` objects in the middle of the chain.

Based on the assumption that re-acquiring a lock that is already held is cheaper than the costly operation of acquiring a lock for the first time, the implementation of all `Writer` classes is based on a single lock object per chain, using `synchronized` blocks rather than `synchronized` methods. For this purpose, class `Writer` contains a protected field `lock` referencing the object which should be used in all subclasses of `Writer` for synchronization purposes.

The API documentation of `Writer` contains the following description of the field `lock`:

> "The object used to synchronize operations on this stream. For efficiency, a character-stream object may use an object other than itself to protect critical sections. A subclass should therefore use the object in this field rather than `this` or a `synchronized` method."

There is no simple way to enforce that synchronization in subclasses of `Writer` is performed properly, because checking safety and liveness properties is complex and highly application-specific. However, even if one cannot check *sufficient* conditions, it is already very useful to check *necessary* conditions for proper synchronization: Certain synchronization schemes clearly do not fit in the scheme that is defined by class `Writer`. For example, as a first step, it is possible to disallow `synchronized` methods in subclasses of `Writer` (this is a static type definition constraint). Adding a static type usage constraint, one can do even better, and disallow `synchronized` statements that operate on objects of type `Writer`.

2.2.4 Method usage constraints

Many static method usage constraints have been found; most of them are in two packages, as shown in Figure 2.5.

- In package `java.awt.image`, there are numerous methods that should only be called by other classes of the AWT framework and not by application code. Additionally, there are precondition-like constraints that can be enforced statically, such as several methods that expect certain parameters to be `null` when called by application code.

- The package `java.awt` contains similar constraints, most of which state that methods that need to be `public` should not be called from application code.

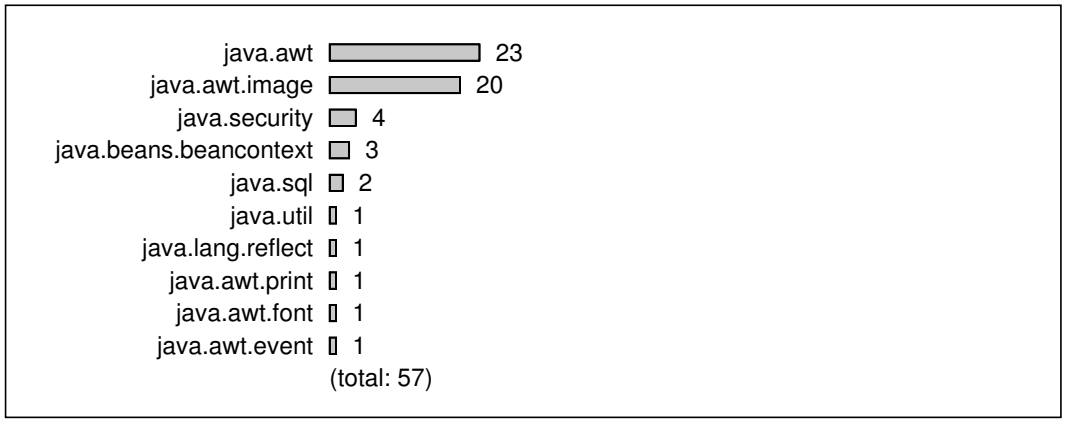

Figure 2.5: Static method usage constraints per package

In the remainder of this section, we will give a detailed example of such a constraint.

In Java, *access modifiers* determine whether or not a class or an interface has access to other entities, such as for example classes, interfaces, fields, and methods. No restrictions — global accessibility — is denoted by the modifier `public`. The modifier `protected` specifies that access is only possible from within the same package or from subtypes. No access modifier implies package-only access, i.e. access is permitted only from within the same package. Finally, the access modifier `private` allows to access an entity only from within the same class or interface.

For some purposes, this scheme is too coarse-grained: For example, two packages might be closely related, such that classes in one package need access to methods in the other package even if they are not subclasses. This already implies that the called methods need to be declared as `public`, making them accessible not only to the closely related package, but to all other packages as well[6]. There are a number of methods and constructors in classes of the abstract window toolkit (AWT), which should not be called by application-specific classes.

One such situation occurs in class `java.awt.Component`, the abstract superclass of all graphical objects: It contains two methods (`addNotify` and `removeNotify`) that are to be called only by classes in packages that form the platform-dependent implementations of the AWT classes. Application-specific classes, however, should not call these methods although they are declared as `public`. The documentation for both methods includes the following:

[6]This problem is due to a language feature which Java is lacking. In C++, it is possible to declare external classes as `friend` classes which makes access-proteced fields and methods accessible to them. In Eiffel, this can be accomplished by selectively exporting fields or methods to other classes.

> "This method is called internally by the toolkit and should not be called directly by programs."

With a static method usage constraint, calls of methods that are public can be restricted to specific packages, barring application-specific classes from accessing these methods.

2.2.5 Field usage constraints

Only two field constraints have been found, both of which are field usage constraints:

The class `java.awt.AWTEvent` defines static (constant) fields which are used as event mask values. The class documentation states that

> "The event masks defined in this class are needed ONLY by component subclasses which are using `Component.enableEvents()` to select for event types not selected by registered listeners."

This constraint adresses usages of the static fields, which should only occur in certain subclasses of `java.awt.AWTEvent`.

In the class `java.beans.beancontext.BeanContextSupport`, the documentation for the field `services` states that

> "all accesses to the `protected transient HashMap services` field should be synchronized on that object."

A sufficient condition for this constraint can be checked statically by requiring that all usages of this field occur either in `synchronized` methods of the class `java.beans.beancontext.BeanContextSupport` itself, or in other places where the field access to an object is within a `synchronized` block which synchronizes on that object.

2.3 Dynamic constraints

Dynamic constraints are quite common; they make up the largest part of all constraints that have been found in the Java standard classes. As has already been noted in Section 2.1.1, it was difficult to decide for 167 of the 401 dynamic constraints whether they are static or dynamic constraints. These constraints, which are concerned with *genericity*, *aliasing*, and *sequencing of method calls*, are described in Section 2.3.1. Interestingly, all remaining dynamic constraints, which are discussed in Section 2.3.2, can be classified as class invariants (dynamic type definition constraints), method preconditions (dynamic method usage constraints), and method postconditions (dynamic method definition constraints) as known from Eiffel [Meyer 1992].

2.3.1 Accidentally dynamic constraints

For the constraints in this section, it is unclear whether they are dynamic or static constraints. In principle, genericity constraints could be checked statically. In practice, this requires annotations on the type level which are not supported by Java[7]. Thus, we have decided to classify them as dynamic constraints in this context. For aliasing and sequencing constraints, it is still an open research issue whether they can be enforced statically without restricting common programming practices. For a detailed discussion of the four constraint categories, see the following subsections.

Genericity constraints

There are 72 constraints that deal with type compatibility issues which could be checked statically if Java had parameterized types [Bracha et al. 1998]. Although it is possible to employ type inference to find out type parameters and bounds for these parameters [Duggan 1999], explicit declarations of type parameters are required to detect constraint violations — using type inference, these would only lead to more general bounds and not to type errors. Thus, we have classified these 72 constraint as dynamic constraints. See Figure 2.6 for an overview on where genericity constraints have been found. Interestingly, most genericity constraints appear in only two packages:

- The package `java.awt.image` provides 40 classes and interfaces for creating and modifying images. Its documentation states:

  > "Images are processed using a streaming framework that involves an image producer, optional image filters, and an image consumer. This framework makes it possible to progressively render an image while it is being fetched and generated. Moreover, the framework allows an application to discard the storage used by an image and to regenerate it at any time. This package provides a number of image producers, consumers, and filters that you can configure for your image processing needs."

  The package `java.awt.image` deals with pixels, arrays of pixels, colors in different color models, and image transformations. The genericity constraints in this package mostly concern the correct initialization of the streams from image producers to image consumers. For example, many constraints require the color model of certain objects to be compatible. These compatibility requirements seem to be type-based and thus could be expressed on the type level if Java had parameterized classes.

- The package `java.util` contains "the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array)" (quote from the documentation). It is not surprising that there are many genericity constraints in this package. For example, the classes and methods for sorting and searching require objects to implement the interface `Comparable`.

[7] Other object-oriented languages have a type system in which genericity constraints can be checked statically. For example, the language Eiffel [Meyer 1992] supports parameterized types for expressing genericity constraints.

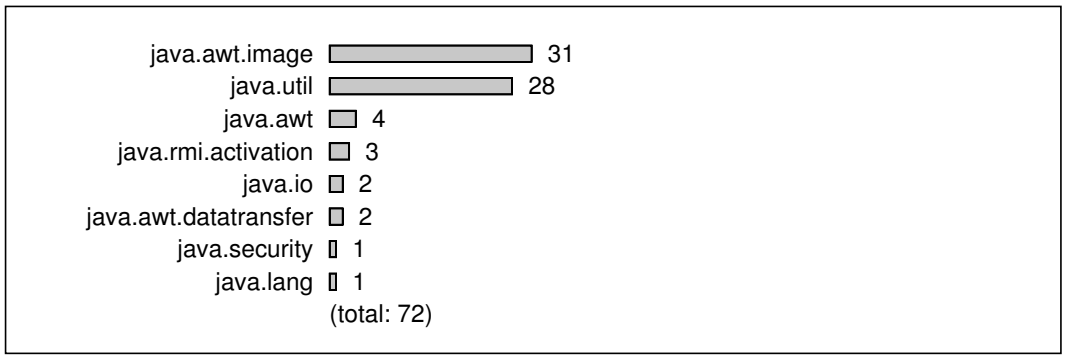

Figure 2.6: Genericity constraints per package

Aliasing constraints

A total of 24 constraints specify aliasing properties. For example, the documentation of class `java.util.WeakHashMap` states that values of

> "`WeakHashMap` are held by ordinary strong references. Thus care should be taken to ensure that value objects do not strongly refer to their own keys, either directly or indirectly, since that will prevent the keys from being discarded."

Checking this constraint statically would require points-to analysis as proposed, for example, in [Yur et al. 1999] and [Steensgaard 1996]. However, this analysis usually requires global program analysis which makes it difficult to use for large systems. A related approach is research on aliasing declarations [Noble et al. 1998] which can be checked statically and do not require global analysis. Finally, it is possible to check aliasing constraints dynamically, but this would probably require an instrumented JVM. See Figure 2.7 for an overview of the packages that contain aliasing constraints.

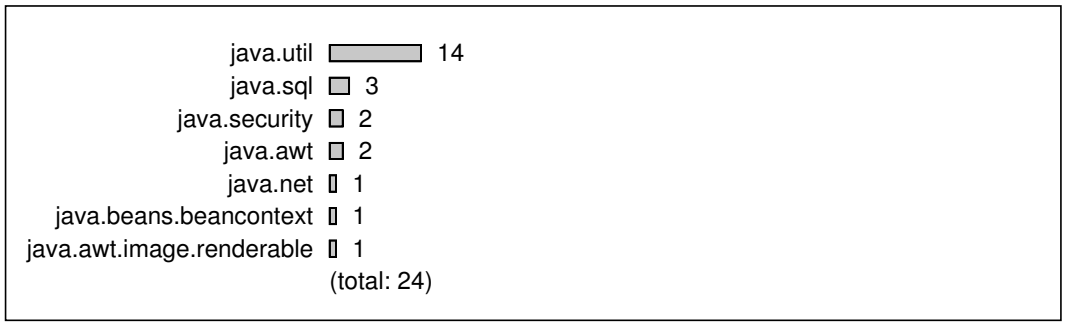

Figure 2.7: Aliasing constraints per package

Sequencing constraints

We have found 71 sequencing constraints; they state that certain events, usually method calls, should be performed in a specific order or according to a specific protocol. As can be

seen in Figure 2.8, the largest number of examples of this class of constraints can be found in the package `java.sql`, a stateful interface to relational database systems, which contains 30 sequencing constraints. For example, the documentation of the method `wasNull` in `java.sql.CallableStatement` states:

> "Note that this method should be called only after calling the `get` method; otherwise, there is no value to use in determining whether it is `null` or not."

Some of these constraints seem to be a sign of bad design and could be removed if the package could be restructured. In the above example, it would be possible to wrap the result of the `get` method in an object which can be used to retrieve the result and to query whether or not the result is the SQL `null` value. Note that most of the other constraints found in the Java standard classes do not result from bad design. See Chapter 8.2 for a more detailed discussion of this point.

Sequencing constraints sometimes can be checked dynamically using special variables that keep track of the current state of an object, and checking the correct order of method calls in method preconditions. Whether sequencing constraints can be checked statically is still an open research question that so far has been tackled only for idealized languages or calculi [Vasconcelos 1994, Nierstrasz 1995, Bokowski 1996, Yellin, Strom 1997, Ravara et al. 1998].

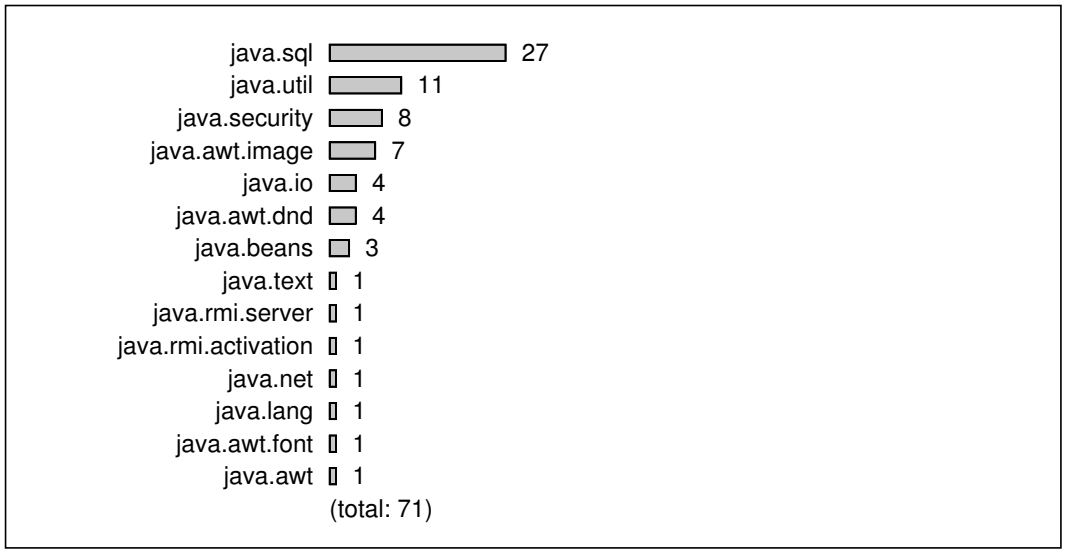

Figure 2.8: Sequencing constraints per package

2.3.2 Inherently dynamic constraints

Figure 2.9 shows the package names and corresponding numbers of dynamic constraints which are neither genericity, aliasing, nor sequencing constraints. It is interesting to note that about 80% of these dynamic constraints belong to just five of the 35 packages that make up the Java standard classes. In this section, we will have a closer look at these five packages and give an overview of the types of dynamic constraints that occur in these packages.

- The package with the most of these dynamic constraints, `java.awt.image`, deals with pixels, arrays of pixels, colors in different color models, and image transformations. Most of the dynamic constraints are method preconditions, requiring, e.g., that pixel array parameters be of a specific shape corresponding to width and height parameters, or that certain parameters be non-null. It seems that the problem domain of image creation, transformation and modification requires many of these preconditions.

- In the package `java.util`, the collections framework, many constraints are preconditions related to different ways of searching; they require that the collection be sorted for calling methods that perform searches on it.

- Package `java.awt`, the abstract window toolkit, contains classes and interfaces for creating user interfaces and for painting graphics and images. The majority of dynamic constraints in this package are simple method preconditions which, e.g., require alignment or placement parameters to be within a certain range of values, or parameters that specify certain layout choices to be one of a certain set of string or numerical constants. Many of these constraints, however, could be handled on the type level if Java included enumeration types and range types.

- The package `java.lang` provides classes and interfaces that are fundamental to the design of the Java programming language. Apart from a few simple method preconditions, it repeats the dynamic constraints regarding objects implementing `Comparable` and defines additional properties that comparison methods must have. Most importantly, any class implementing the methods `equals` and `hashCode` must ensure that `equals` is reflexive, symmetric, and that comparing objects with `null` using `equals` yields `false`. Furthermore, implementations of `hashCode` must be consistent with `equals`, i.e., objects for which `equals` returns `true` must have the same `hashCode`.

- Finally, the package `java.sql`, which contains the JDBC package, an interface to SQL database management systems, imposes constraints regarding type compatibility between Java and SQL. Most of these constraints can be expressed as method preconditions that compare the actual type of the provided or queried Java object with dynamic type information retrieved from the database system. Because of the interface nature of the package (bridging between two different programming languages), parameterized classes would not help in this case.

A typical example for a precondition can be found in the class `java.awt.Color` for the method `getHSBColor`, which creates a `Color` object based on values supplied for the HSB color model: The method's documentation states that

> "Each of the three components should be a floating-point value between zero and one (a number in the range 0.0 =< h, s, b =< 1.0)."

A typical postcondition is documented for the method `getAlignmentX` in class `java.awt.Component`. The method, which can be overridden in subclasses, returns the component's alignment along the x axis. The documentation states with regard to the value returned by the method:

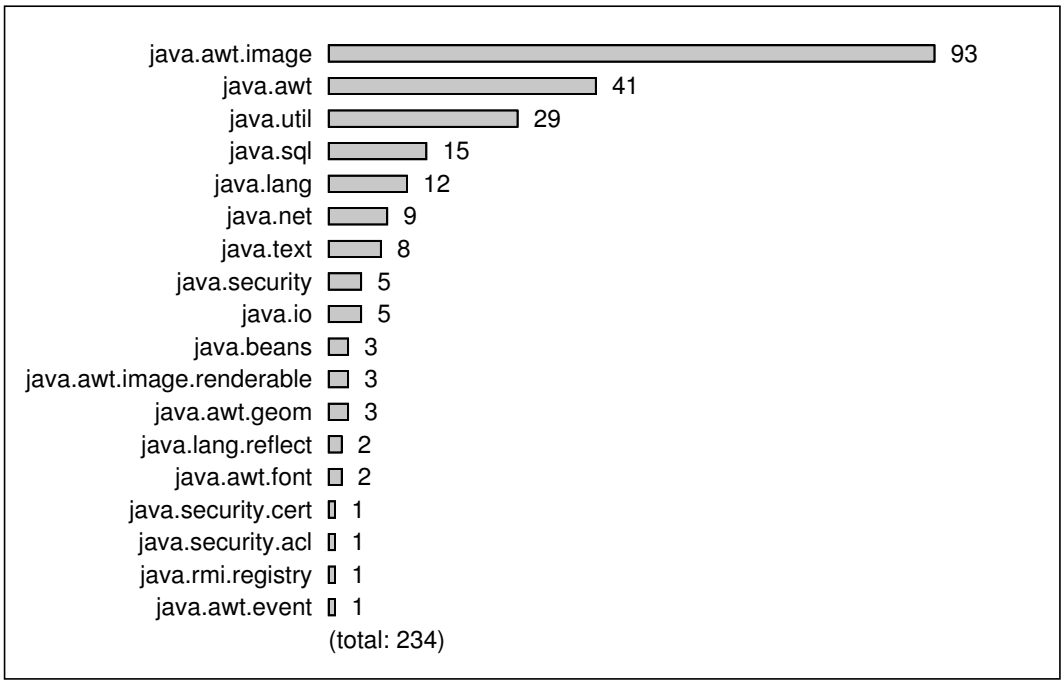

Figure 2.9: Other dynamic constraints per package

> "This specifies how the component would like to be aligned relative to other components. The value should be a number between 0 and 1 where 0 represents alignment along the origin, 1 is aligned the furthest away from the origin, 0.5 is centered, etc."

Only four occurrences of class invariants have been found, two of which are mentioned in the documentation of the interfaces `java.util.SortedSet` and `java.util.SortedMap`; the other two occurrences are in the two implementations of these interfaces, `java.util.TreeSet` and `java.util.TreeMap`. The four constraints are very similar, for example, the constraint in `java.util.SortedSet` reads:

> "Note that the ordering maintained by a sorted set (whether or not an explicit comparator is provided) must be consistent with `equals` if the sorted set is to correctly implement the `Set` interface."

2.4 Alternative categorizations

The categorization that has been presented is only one possible way of categorizing constraints. In this section, we discuss two obvious alternative categorizations and why these are not suitable for our purposes.

2.4.1 Categorization based on degree of abstraction

One possible categorization focuses on the degree of abstraction, distinguishing between *stylistic constraints* that are concerned with aspects of a program that, when changed, do not affect its semantics — for instance, naming issues; *implementation constraints* that deal with problematic language constructs or cover common traps and pitfalls that may easily lead to subtle programming errors; and *design constraints* that reflect programming rules for the correct use of a framework, or coding conventions resulting from the use of design patterns. This categorization has been chosen in [Chowdhury, Meyers 1993], which describes a system for checking constraints on C++ programs.

Examples for stylistic constraints could be to require that that package names are lowercase only, or that in a class definition, the declarations of public variables, public constructors, and public methods precede the private declarations, or that the scope of local variables is minimal, i.e. each variable declaration is in the smallest block that contains all uses of the variable.

Implementation constraints might require, for example, that a class that provides its own implementation of the method `public boolean equals(Object other)` also implements the method `public int hashCode()` and vice-versa, because equal objects must have the same hash code to be correctly added to and removed from hash-based collections, that branches of an `if`-statement are blocks rather than single statements, because when adding a new statement to a single-statement branch, programmers often forget to correctly group both statements in a block, or that `String` objects are compared using the method `equals()` rather than the identity operator "`==`".

Thus, while this categorization is useful to convey the broad scope of constraints that occur in object-oriented software development projects, it is not appropriate for our purpose of developing a system for automatic checking of constraints. First, it differentiates between constraints based on the intentions of the constraint designers rather than based on structural differences inherent to the constraints. Second, it seems no good fit for the constraints that have been found in the Java standard classes because they would all fall into a single category, design constraints. It appears that "degree of abstraction" is a dimension for classifying constraints which is independent from the dimensions presented in the main part of this chapter. Thus, CoffeeStrainer, the system presented by this thesis, can be used to check implementation, stylistic, and design constraints as described above.

2.4.2 Categorization based on programming language concepts

Rather than identifying orthogonal dimensions for distinguishing categories, one could use concepts known from programming language research as categories: *aliasing*, *parameterized types*, *sequencing*, *assertions*, *preconditions*, *postconditions*, *class invariants*, *immutability*, *access control*, etc. Some of these research areas have been mentioned already in Sections 2.3.1 and 2.3.2.

While this categorization is certainly useful and helps to focus research interests, it does not cover all constraints that have been found. It would be unsatisfactory to have a category "other constraints" for the remaining constraints that do not fall into one of the identified

research areas. Moreover, this categorization does not help structuring the problem domain as nicely as the independent classification dimensions of Section 2.1.