the fix patterns for warning types, with the columns having the same meaning as in Table 2. We only detail the top five error types and warning types here. The complete list is available on our website[1]. In the following, we first explain the fix patterns for the most common error types, and then the most common warning types.

cant.resolve. We analyzed 96 broken builds to have stable fix patterns for cant.resolve. This compiler error indicates that the compiler cannot find a symbol in the code. This symbol could be a variable, method, or class that hasn't been properly declared. The most common fix is to correct mistypes, e.g., change package, class, method or variable name. The error is usually due to misspelling, incomplete refactoring or code change, usage of snapshot libraries, and different Java versions in build environments. Another common fix is to remove all the relevant code about the symbol not found, which is often caused by incomplete refactoring or code change. An interesting pattern is to add a class, meaning that developers seem to use a class that does not exist. A close look at the commit messages shows that developers often miss some code files in commits. Adding an import statement or a dependency library is also quite common. Less common patterns include adding a variable declaration, updating the dependency library version, moving a class from a package to another or casting the object type.

doesnt.exist. From 96 broken builds, we summarized four patterns to fix doesnt.exist. This error indicates the referenced package is not found in the classpath. In particular, adding a dependency library is the most common pattern, which resolves errors caused by a missed dependency declaration in automated building tools. Changing an imported package fixed eight broken builds. These errors are often due to misspelling, incomplete code change, or usage of snapshot libraries. Removing the package import statement fixed another 25 broken builds, which were typically caused by redundant imports. Similar to the case in cant.resolve, developers may miss to commit a whole package. Thus, a corresponding fix pattern is to add the package and the classes in it. It is interesting that in one of the broken builds, a cant.resolve error was falsely reported as a doesnt.exist error due to static method invocation, where the compiler falsely considered a class as a package since it is possible that a class is used through its fully qualified name without an import statement.

expected. We investigated 96 broken builds of expected. As this error type is generally related to syntax violations, many fixes are hard to summarize, e.g., changing a token from "," to ";", add "{", or remove ")". Therefore, we grouped them together as a pattern to change, add or remove a token. Besides, a common pattern is to remove illegal tokens, which occurred 16 times. Most of the illegal tokens were resulted from merge conflicts, which developers did not resolve. The other two patterns are to change misspelled keywords and move code (e.g., moving a method declaration out of a method declaration).

[1] https://cicompilation.github.io/

Table 3: Fix Patterns of the Common Compiler Warning Types

Warning Type	Warning Description	Fix Pattern	Eclipse		IntelliJ IDEA		Imprecise
			Sug.?	Aut.?	Sug.?	Aut.?	
has.been.deprecated (25)	obsolete API is used	update API (16)	×	×	×	×	0
		remove relevant code (5)	×	×	×	×	
		suppress warning (2)	✓	✓	✓	✓	
		ignore warning (1)	×	×	×	×	
		revert dependency library (1)	×	×	×	×	
unchecked.assign (15)	raw type is assigned to generic type	specify type parameter (7)	✓	✓	✓	✓	0
		suppress warning (6)	✓	✓	✓	✓	
		ignore warning (2)	×	×	×	×	
unchecked.cast.to.type (20)	raw type is cast to generic type	suppress warning (11)	✓	✓	✓	✓	0
		ignore warning (4)	×	×	×	×	
		skip or remove command (3)	×	×	×	×	
		remove relevant code (2)	×	×	×	×	
unchecked.call.mbr.of.raw.type (15)	member of raw types is used	specify type parameter (5)	✓	✓	✓	✓	0
		suppress warning (4)	✓	✓	✓	✓	
		ignore warning (4)	×	×	×	×	
		skip or remove relevant code (2)	×	×	×	×	
sun.proprietary (15)	internal API is used	remove relevant code (10)	×	×	×	×	0
		use alternative API (5)	×	×	×	×	

Surprisingly, 29 (82.9%) broken builds had imprecise error messages that failed to locate the position of an error.

method.does.not.override.superclass. Via analyzing 96 broken builds, we summarized eight fix patterns. This error occurs when the @Override annotation is used on a method that does not actually override a method in a superclass. Most commonly, it is fixed by changing the method signature in the subclass to match the already changed method in its superclass. Another common fix is to remove the @Override annotation and method in the subclass, which is usually caused by the removed method in its superclass (i.e., incomplete code changes). Adding the superclass fixed two broken builds, where the cant.resolve error was reported together. Updating dependency library version fixed 11 broken builds. The libraries were developed by developers in the same team, but violated the original design. Thus, the libraries were updated to follow the original design. Importing the missed superclass fixed 7 broken builds, as the absent import statements prevented the compiler from recognizing the intended parent class. Other patterns are to add the dependency library, the missed superclass, or the method in the superclass, and to remove the @Override annotation.

cant.apply.symbol(s). This error occurs when a method is called with incorrect arguments. For the 96 broken builds, the common fix patterns are all about changing arguments in a method call, i.e., change the type of an argument, add an argument, remove an argument, and change the order of two arguments. These errors are caused by incomplete refactoring or code changes, or method misuses due to carelessness or unfamiliarity. Another common fix is to add missed method declaration, which is usually caused by new features. A common pattern is to change the method name, which means that developers called the wrong method. A less common pattern is to remove the method call. Interestingly, one broken build was fixed by updating the dependency library to update the method signatures. In addition, we got one false report due to a bug in type inference in Java generics, and it was resolved by updating Java version.

illegal.start. Similar to expected, illegal.start is related to syntax violations, and often occurs simultaneously with expected. Its fix patterns are almost the same to expected except that it has a pattern that adds the generic type in constructor or method invocations. Actually, generic types can be omitted in Java 7 and beyond. This error occurs because of a lower Java version in CI build environment.

incompatible.types. This error type usually occurs in assignments, where the right hand side can be a literal, a variable, a method invocation, etc. Among 96 broken builds, the most common fix pattern is to change the assignee's declared type to match the previously-changed type of the right hand side. Correspondingly, some fix patterns are to change the type of right hand side, e.g., change method's return type and its implementation, assign a new value, and convert the type of a method call's return value through a chained call (e.g., a chained call to toString()). Other fix pattern are to add a generic type when the generic type inference fails to work, or to simply remove the assignment code

resulted from incomplete code changes. Similar to cant.apply.symbol(s), we got two false reports due to the same bug in type inference in Java generics.

does.not.override.abstract. Analyzing 96 broken builds, we derived three fix patterns for does.not.override.abstract. This error occurs when a class declared as non-abstract fails to provide concrete implementations for all inherited abstract methods. The most common one is to implement the abstract method in the subclass. Such errors are mostly caused by the new features added in the superclass. Because of incomplete refactoring or code changes, the method signature in the superclass is changed without timely changing the method signature in the subclass. Therefore, the corresponding fix is to accordingly change the method signature in the subclass. Another common pattern is to remove the method in the superclass, which occurs due to the redesign of interfaces.

duplicate.class. We analyzed 88 broken builds of duplicate.class. This error occurs when there are two or more source files that define a class with the same fully qualified name. The most common fix pattern is to remove the duplicated class. Such errors are usually caused by branch merges or a lack of coordination among developers. Additional fix patterns include changing the class or package name to resolve naming conflicts among developers, and changing the project configuration to eliminate duplicate class definitions.

has.been.deprecated. We examined 97 builds to identify five fix patterns for has.been.deprecated. This warning occurs when an obsolete API is used. The predominant pattern is to update the API by replacing the API with a newer version. Less frequent fixes include removing the code using the obsolete API and ignoring the warning by editing configurations to prevent this type of warning from being reported. Another pattern is to suppress the warning using the @SuppressWarnings annotation. An interesting fix pattern is to revert the dependency library to an older version where the API has not been marked as deprecated.

unchecked.assign. After analyzing 97 builds, we determined five fix patterns for unchecked.assign. This warning occurs when a raw type is assigned to a generic type without proper type checking. The most common fix pattern is to specify type parameters, thus converting raw types to generic types. Instead of fixing this warning, 34 cases were addresses by using either the @SuppressWarnings annotation to suppress the warning, or setting the Xlint parameter to ignore this warning. The other two less common patterns are skipping or removing relevant code and updating the API.

unchecked.cast.to.type. We examined 97 builds to identify four fix patterns for unchecked.cast.to.type. This warning is triggered by casting a raw type to a generic type without type checking. The most common fix is to suppress the warning. In 16 cases, this warning is resolved by skipping or removing the corresponding commands from the configuration file of Travis CI. Removing the code that triggers this warning fixes 14 cases. Ignoring warnings fixes nine cases.

unchecked.call.mbr.of.raw.type. We examined 97 builds to derive four fix patterns for unchecked.call.mbr.of.raw.type. This warning arises when a mem-

ber of a raw type is accessed. The predominant fix is to specify information of type parameters explicitly. Less common fix patterns include suppressing warnings, skipping or removing relevant code, and ignoring warnings.

sun.proprietary. We analyzed 97 builds to identify two fix patterns for sun.proprietary. This warning is triggered by the use of internal APIs from the sun.* packages. These APIs are not part of the public JDK APIs and are not guaranteed to work on all Java-compatible platforms. Relying on these APIs can lead to portability issues and maintenance challenges. The two fix patterns involve either removing the relevant code or refactoring it to use alternative APIs.

unchecked.meth.invocation.applied. Analysis 97 builds revealed five fix patterns for unchecked.meth.invocation.applied. When a method designed for generic type parameters receives a raw type instead, this warning occurs. The most common fix is to change the type of arguments, i.e., convert the raw type to a generic type by specifying type parameter. The other fix patterns include suppressing warnings, removing the relevant code, skipping the command that generates this warning or just ignoring warnings.

annotation.method.not.found. After analyzing 96 builds, we determined three fix patterns for annotation.method.not.found. This warning typically appears when the compiler cannot find a method declared in an annotation, often due to issues with dependencies. The most common fix is to add or change the dependency library to include the missing method. The less common fix patterns include removing the relevant code or ignoring the warning.

unchecked.generic.array.creation. We analyzed 96 builds to identify four fix patterns for unchecked.generic.array.creation. This warning occurs when an array with generic types is created, which Java does not guarantee type safety at runtime. The most common fix is to change the API (e.g., change method name, argument types, and the number of arguments) to avoid generating generic type arrays. In 30 cases, warnings are suppressed, while 12 cases simply ignore the warning. Lastly, eight instances remove the relevant code.

source.no.bootclasspath. We analyzed 97 builds to derive three fix patterns. This warning arises when using an outdated configuration setting to compile code against a newer JDK version. The primary fix is to change the outdated configurations to align with the JDK version in use. The other fix patterns include changing the JDK version and ignoring this warning.

unchecked.varargs.non.reifiable.type. Analysis of 97 builds established four fix patterns for unchecked.varargs.non.reifiable.type. This warning pertains to the use of non-reifiable types with varargs (i.e., variable argument) methods. Non-reifiable types are types that lose specific type information due to type erasure, such as generic types. The most common fix is to suppress the warning. 23 cases involve refactoring the varargs to use a fixed number of parameters or changing the non-reifiable type to a reifiable type.Another 23 cases were resolved by ignoring warnings. The last pattern is to remove the relevant code.

From this manual analysis, we found that while fix patterns often exist for most of the common issue types, the root causes of the issues are of-