Spring 4-12-2018

Proactive Empirical Assessment of New Language Feature Adoption via Automated Refactoring: The Case of Java 8 Default Methods

Raffi T. Khatchadourian
CUNY Hunter College

Hidehiko Masuhara
Tokyo Institute of Technology

Proactive Empirical Assessment of New Language Feature Adoption via Automated Refactoring: The Case of Java 8 Default Methods

Raffi Khatchadourian[1,2] Hidehiko Masuhara[3]

International Conference on the Art, Science, and Engineering of Programming
April 2018, Nice, France

[1]Computer Science, Hunter College, City University of New York, USA

[2]Computer Science, The Graduate Center, City University of New York, USA

[3]Mathematical and Computing Science, Tokyo Institute of Technology, Japan

Outline

Introduction

Background

Contributions

Methodology

Research Questions and Results

Conclusion

Introduction

New Programming Languages Features

- Programming languages change for a variety of reasons.

New Programming Languages Features

- Programming languages change for a variety of reasons.
- To benefit from new language features, developers must be willing to adopt them.

Empirical Study on Usage of Default Methods

- An empirical study assessing the adoption of a new language feature: default methods.

Empirical Study on Usage of Default Methods

- An empirical study assessing the adoption of a new language feature: default methods.
- Default methods are part of Java 8's *enhanced* interfaces.

Background

Java 8 Default Methods

- Allow both method declarations and *definitions*.

```
interface Collection<E> {
  default void add(E elem) { // optional.
    throw new UnsupportedOperationException();}}
```

Java 8 Default Methods

- Allow both method declarations and *definitions*.
- Implementers inherit the (**`default`**) implementation if none provided.

```
interface Collection<E> {
  default void add(E elem) { // optional.
    throw new UnsupportedOperationException();}}
```

```
class ImmutableList<E> implements Collection<E> {}
```

Java 8 Default Methods

- Allow both method declarations and *definitions*.
- Implementers inherit the (**`default`**) implementation if none provided.
- Original motivation to facilitate interface *evolution*.

```
interface Collection<E> {
  default void add(E elem) { // optional.
    throw new UnsupportedOperationException();}}
```

```
class ImmutableList<E> implements Collection<E> {}
```

Java 8 Default Methods

- Allow both method declarations and *definitions*.
- Implementers inherit the (**`default`**) implementation if none provided.
- Original motivation to facilitate interface *evolution*.
- Can also be used as a replacement of the *skeletal implementation pattern* (Goetz 2011).

```
interface Collection<E> {
  default void add(E elem) { // optional.
    throw new UnsupportedOperationException();}}
```

```
class ImmutableList<E> implements Collection<E> {}
```

```
abstract class AbstractImmutableList<E> implements
    Collection<E> {
  @Override public void add(E elem) {
    throw new UnsupportedOperationException();}}
```

Java 8 Default Methods

- Allow both method declarations and *definitions*.
- Implementers inherit the (**`default`**) implementation if none provided.
- Original motivation to facilitate interface *evolution*.
- Can also be used as a replacement of the *skeletal implementation pattern* (Goetz 2011).
  - Uses abstract class that interface implementers extend.

```
interface Collection<E> {
  default void add(E elem) { // optional.
    throw new UnsupportedOperationException();}}
```

```
class ImmutableList<E> implements Collection<E> {}
```

```
abstract class AbstractImmutableList<E> implements
    Collection<E> {
  @Override public void add(E elem) {
    throw new UnsupportedOperationException();}}
```

Java 8 Default Methods

- Allow both method declarations and *definitions*.
- Implementers inherit the (**`default`**) implementation if none provided.
- Original motivation to facilitate interface *evolution*.
- Can also be used as a replacement of the *skeletal implementation pattern* (Goetz 2011).
  - Uses abstract class that interface implementers extend.
  - Makes interfaces easier to implement (Bloch 2008, Item 18).

```
interface Collection<E> {
  default void add(E elem) { // optional.
    throw new UnsupportedOperationException();}}
```

```
class ImmutableList<E> implements Collection<E> {}
```

```
abstract class AbstractImmutableList<E> implements
    Collection<E> {
  @Override public void add(E elem) {
    throw new UnsupportedOperationException();}}
```

Contributions

Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.

Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.
- Pull requests (patches) issued that contained particular interface method implementations migrated to interfaces as default methods in a semantics-preserving fashion.

Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.
- Pull requests (patches) issued that contained particular interface method implementations migrated to interfaces as default methods in a semantics-preserving fashion.
- Found that there are non-obvious trade-offs to using default methods.

Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.
- Pull requests (patches) issued that contained particular interface method implementations migrated to interfaces as default methods in a semantics-preserving fashion.
- Found that there are non-obvious trade-offs to using default methods.
- Detail reactions of developers in adopting default methods in their projects.

Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.
- Pull requests (patches) issued that contained particular interface method implementations migrated to interfaces as default methods in a semantics-preserving fashion.
- Found that there are non-obvious trade-offs to using default methods.
- Detail reactions of developers in adopting default methods in their projects.
- Extract best practices of their uses.

Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.
- Pull requests (patches) issued that contained particular interface method implementations migrated to interfaces as default methods in a semantics-preserving fashion.
- Found that there are non-obvious trade-offs to using default methods.
- Detail reactions of developers in adopting default methods in their projects.
- Extract best practices of their uses.
- Situations where these new constructs work well and where trade-offs must be made.

Traditional Approaches to Assessing New Languages Features

- A popular approach for assessing language features involves a *postmortem* analysis.

Traditional Approaches to Assessing New Languages Features

- A popular approach for assessing language features involves a *postmortem* analysis.
- *Past* data of source repositories are analyzed

Traditional Approaches to Assessing New Languages Features

- A popular approach for assessing language features involves a *postmortem* analysis.
- *Past* data of source repositories are analyzed
- Surveys of previous coding activities are taken.

Traditional Approaches to Assessing New Languages Features

- A popular approach for assessing language features involves a *postmortem* analysis.
- *Past* data of source repositories are analyzed
- Surveys of previous coding activities are taken.
- Developers must discover new language features and integrate them themselves before any analysis of the construct can be done.

Traditional Approaches to Assessing New Languages Features

- A popular approach for assessing language features involves a *postmortem* analysis.
- *Past* data of source repositories are analyzed
- Surveys of previous coding activities are taken.
- Developers must discover new language features and integrate them themselves before any analysis of the construct can be done.
- Best practices and patterns that can normally be extracted from these studies are delayed.

Traditional Approaches to Assessing New Languages Features

- A popular approach for assessing language features involves a *postmortem* analysis.
- *Past* data of source repositories are analyzed
- Surveys of previous coding activities are taken.
- Developers must discover new language features and integrate them themselves before any analysis of the construct can be done.
- Best practices and patterns that can normally be extracted from these studies are delayed.
- Developers may be unable to *manually* identify *all* opportunities where the new language construct can be utilized.

Traditional Approaches to Assessing New Languages Features

- A popular approach for assessing language features involves a *postmortem* analysis.
- *Past* data of source repositories are analyzed
- Surveys of previous coding activities are taken.
- Developers must discover new language features and integrate them themselves before any analysis of the construct can be done.
- Best practices and patterns that can normally be extracted from these studies are delayed.
- Developers may be unable to *manually* identify *all* opportunities where the new language construct can be utilized.
- Observing software histories may discover cases where new language features are *adopted* but may not easily identify those where they were *rejected* as these may not have been adequately documented.

Our Proactive Approach

- A novel technique for assessing new language constructs *proactively*.

Our Proactive Approach

- A novel technique for assessing new language constructs *proactively*.
- The pull request changes in our study consist of transformations performed via an automated refactoring tool.

Our Proactive Approach

- A novel technique for assessing new language constructs *proactively*.
- The pull request changes in our study consist of transformations performed via an automated refactoring tool.
- Developers are immediately introduced to the new construct via a semantically equivalent transformation that they can either accept or reject.

Our Proactive Approach

- A novel technique for assessing new language constructs *proactively*.
- The pull request changes in our study consist of transformations performed via an automated refactoring tool.
- Developers are immediately introduced to the new construct via a semantically equivalent transformation that they can either accept or reject.
- Their decisions can be studied early to assess the feature's effectiveness, extracting best practices.

Methodology

Study Methodology

- The use of *conservative*, *theoretically sound*, and *minimally invasive* refactoring automation is key in minimizing human bias.

Study Methodology

- The use of *conservative*, *theoretically sound*, and *minimally invasive* refactoring automation is key in minimizing human bias.
- We use the Migrate Skeletal Implementation to Interface refactoring tool (Khatchadourian and Masuhara 2017), based on type constraints (Palsberg and Schwartzbach 1994; Tip et al. 2011).

Study Methodology

- The use of *conservative*, *theoretically sound*, and *minimally invasive* refactoring automation is key in minimizing human bias.
- We use the Migrate Skeletal Implementation to Interface refactoring tool (Khatchadourian and Masuhara 2017), based on type constraints (Palsberg and Schwartzbach 1994; Tip et al. 2011).
- Discover opportunities and semantics-preserving transformations for migrating methods possibly participating in the skeletal implementation pattern to interfaces as default methods.

Study Methodology

- The use of *conservative*, *theoretically sound*, and *minimally invasive* refactoring automation is key in minimizing human bias.
- We use the MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE refactoring tool (Khatchadourian and Masuhara 2017), based on type constraints (Palsberg and Schwartzbach 1994; Tip et al. 2011).
- Discover opportunities and semantics-preserving transformations for migrating methods possibly participating in the skeletal implementation pattern to interfaces as default methods.
- Assess the use of default methods in *existing* code.

Study Methodology

- The use of *conservative*, *theoretically sound*, and *minimally invasive* refactoring automation is key in minimizing human bias.
- We use the Migrate Skeletal Implementation to Interface refactoring tool (Khatchadourian and Masuhara 2017), based on type constraints (Palsberg and Schwartzbach 1994; Tip et al. 2011).
- Discover opportunities and semantics-preserving transformations for migrating methods possibly participating in the skeletal implementation pattern to interfaces as default methods.
- Assess the use of default methods in *existing* code.
- Substituting the skeletal implementation pattern is the only sensible use of default methods when not introducing *new* functionality.

Study Methodology

- The use of *conservative*, *theoretically sound*, and *minimally invasive* refactoring automation is key in minimizing human bias.
- We use the Migrate Skeletal Implementation to Interface refactoring tool (Khatchadourian and Masuhara 2017), based on type constraints (Palsberg and Schwartzbach 1994; Tip et al. 2011).
- Discover opportunities and semantics-preserving transformations for migrating methods possibly participating in the skeletal implementation pattern to interfaces as default methods.
- Assess the use of default methods in *existing* code.
- Substituting the skeletal implementation pattern is the only sensible use of default methods when not introducing *new* functionality.
- An acceptance of the refactoring is equivalent to acceptance of using default methods as a programming construct for existing code and vice-versa.

	subject	pull ID	KLOC*	watches†	stars†	forks†	contribs†	+LOC	-LOC	δ files	concrete?
merged	aalmiray/jsilhouette	1	2	2	4	1	2	147	294	4	*false*
	aol/cyclops-react	258	99	68	554	54	21	8	15	2	*false*
	eclipse/eclipse-collections	128	1,266	40	258	63	18	172	307	21	*false*
	nhl/bootique	79	5	103	744	183	5	22	31	4	*true*
rejected	iluwatar/java-design-patterns	472	20	1,783	17,234	5,808	71	24	38	6	*false*
	jOOQ/jOOQ	5469	136	127	1,614	411	40	93	187	22	*false*
	google/guava	2519	244	1,568	14,721	3,502	98	241	427	16	*false*
	google/binnavi	99	309	215	2,048	373	16	244	469	16	*false*
	eclipse/jetty.project	773	329	196	1,225	811	61	140	263	29	*false*
	spring-projects/spring-framework	1113	506	2,299	12,463	9,575	200	770	1,674	135	*false*
	elastic/elasticsearch	19168	1,266	1,928	21,063	7,275	784	297	544	51	*false*
	jenkinsci/blueocean-plugin	296	7	114	1,688	173	28	8	19	5	*true*
	junit-team/junit5	5365	25	146	865	215	41	4	18	1	*true*
	ReactiveX/RxJava	4143	154	1,677	21,792	3,819	142	29	131	23	*true*
pending	perfectsense/dari	218	66	111	48	31	28	39	58	7	*false*
	eclipse/jgit	34	172	57	429	247	121	35	127	10	*false*
	rinfield/java8-commons	81	2	1	0	2	1	26	48	3	*true*
	criscris/koral	1	7	1	1	1	1	169	197	6	*true*
	advantageous/qbit	767	52	82	534	115	12	80	202	29	*true*
	Totals:		4,665	10,518	97,285	32,659	1,690	2,548	5,049	390	

* At time of analysis.

† As of February 27, 2017.

Table 1: Pull requests. More info at `http://cuny.is/interefact`.

Research Questions and Results

Default Method Adoption

Question

In which situations do developers adopt default methods in their projects? What are the reasons?

Default Method Adoption

Question
In which situations do developers adopt default methods in their projects? What are the reasons?

Answers

Interface Locality Default implementation was mostly in terms of *both* methods and constant fields declared either within the same interface or one up its hierarchy.

Default Method Adoption

Question
In which situations do developers adopt default methods in their projects? What are the reasons?

Answers

Interface Locality Default implementation was mostly in terms of *both* methods and constant fields declared either within the same interface or one up its hierarchy.

Parameter Locality No new dependencies introduced by the default method by referencing only parameters.

Default Method Adoption

Question
In which situations do developers adopt default methods in their projects? What are the reasons?

Answers

Interface Locality Default implementation was mostly in terms of *both* methods and constant fields declared either within the same interface or one up its hierarchy.

Parameter Locality No new dependencies introduced by the default method by referencing only parameters.

Optional Methods Default implementation threw `UnsupportedOperationException`s (self-documenting).

Default Method Adoption

Question
In which situations do developers adopt default methods in their projects? What are the reasons?

Answers

Interface Locality Default implementation was mostly in terms of *both* methods and constant fields declared either within the same interface or one up its hierarchy.

Parameter Locality No new dependencies introduced by the default method by referencing only parameters.

Optional Methods Default implementation threw `UnsupportedOperationException`s (self-documenting).

Static Methods as Instance Methods Allowed static methods to be called as instance methods via forwarding.

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

JDK Versions

- Needed to maintain compatibility with legacy clients (e.g., Android).

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

JDK Versions

- Needed to maintain compatibility with legacy clients (e.g., Android).
- Developers must not only consider the language construct itself but also substantial reliance on platform backwards compatibility.

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

JDK Versions

- Needed to maintain compatibility with legacy clients (e.g., Android).
- Developers must not only consider the language construct itself but also substantial reliance on platform backwards compatibility.

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

JDK Versions

- Needed to maintain compatibility with legacy clients (e.g., Android).
- Developers must not only consider the language construct itself but also substantial reliance on platform backwards compatibility.

Architecture

- Developers did not always want to introduce new external dependencies into interfaces as some default methods required.

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

JDK Versions

- Needed to maintain compatibility with legacy clients (e.g., Android).
- Developers must not only consider the language construct itself but also substantial reliance on platform backwards compatibility.

Architecture

- Developers did not always want to introduce new external dependencies into interfaces as some default methods required.
- Projects separated their APIs (interfaces) and an implementation of that API into separate modules.

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Default Method Rejection

Question

Are there situations where developers do not favor default methods?

Answers

Clients

- Anxious about "inlining" skeletal implementations directly into interfaces, particular frameworks.

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

Clients

- Anxious about "inlining" skeletal implementations directly into interfaces, particular frameworks.
- Desired **forcing** clients to implement interfaces directly despite *providing* skeletal implementations in a separate classes.

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

Clients

- Anxious about "inlining" skeletal implementations directly into interfaces, particular frameworks.
- Desired **forcing** clients to implement interfaces directly despite *providing* skeletal implementations in a separate classes.

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

Clients
- Anxious about "inlining" skeletal implementations directly into interfaces, particular frameworks.
- Desired **forcing** clients to implement interfaces directly despite *providing* skeletal implementations in a separate classes.

Generality
- Skeletal implementations too *narrow* to be the "de facto."

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

Clients

- Anxious about "inlining" skeletal implementations directly into interfaces, particular frameworks.
- Desired **forcing** clients to implement interfaces directly despite *providing* skeletal implementations in a separate classes.

Generality

- Skeletal implementations too *narrow* to be the "de facto."
- Pattern allows for multiple implementations per method, enhanced interfaces do not.

Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

Clients

- Anxious about "inlining" skeletal implementations directly into interfaces, particular frameworks.
- Desired **forcing** clients to implement interfaces directly despite *providing* skeletal implementations in a separate classes.

Generality

- Skeletal implementations too *narrow* to be the "de facto."
- Pattern allows for multiple implementations per method, enhanced interfaces do not.
- Skeletal implementations from tests were too **specific**.

Default Method Trade-offs

Question
What are the trade-offs of using default methods over the skeletal implementation pattern?

Default Method Trade-offs

Question
What are the trade-offs of using default methods over the skeletal implementation pattern?

Answers

Control
- Contrary to pattern, default methods are available to *all* interface implementers.

Default Method Trade-offs

Question
What are the trade-offs of using default methods over the skeletal implementation pattern?

Answers

Control

- Contrary to pattern, default methods are available to *all* interface implementers.
- Explicitly presents implementers with a skeletal implementation.

Default Method Trade-offs

Question
What are the trade-offs of using default methods over the skeletal implementation pattern?

Answers

Control

- Contrary to pattern, default methods are available to *all* interface implementers.
- Explicitly presents implementers with a skeletal implementation.
- Implementers may or may not choose to override with their own.

Default Method Trade-offs

Question
What are the trade-offs of using default methods over the skeletal implementation pattern?

Answers

Control

- Contrary to pattern, default methods are available to *all* interface implementers.
- Explicitly presents implementers with a skeletal implementation.
- Implementers may or may not choose to override with their own.
- May have a **negative** effect if not applicable to implementer but choose *not* to override.

External Factors

Question
Which external factors, if any, influence developer's decisions in adopting default methods?

External Factors

Question
Which external factors, if any, influence developer's decisions in adopting default methods?

Answers

Java 8 Projects that *previously* used (other) Java 8 features were more likely to accept.

External Factors

Question
Which external factors, if any, influence developer's decisions in adopting default methods?

Answers

Java 8 Projects that *previously* used (other) Java 8 features were more likely to accept.

Size Smaller change sets were **more** likely to be accepted.

External Factors

Question
Which external factors, if any, influence developer's decisions in adopting default methods?

Answers

Java 8 Projects that *previously* used (other) Java 8 features were more likely to accept.

Size Smaller change sets were **more** likely to be accepted.

Span Change sets spanning **multiple files** across **module boundaries** were **less** likely.

External Factors

Question
Which external factors, if any, influence developer's decisions in adopting default methods?

Answers

Java 8	Projects that *previously* used (other) Java 8 features were more likely to accept.
Size	Smaller change sets were **more** likely to be accepted.
Span	Change sets spanning **multiple files** across **module boundaries** were **less** likely.
Abstractness	Implementations originating from *abstract* classes **more** likely (more general).

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Default methods should be simple.

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Default methods should be simple.
  - Reduces likelihood of complex dependencies in interfaces.

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Default methods should be simple.
  - Reduces likelihood of complex dependencies in interfaces.
  - Promote self-containment.

Best Practices for Default Methods

Question

Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Default methods should be simple.
  - Reduces likelihood of complex dependencies in interfaces.
  - Promote self-containment.
  - Enhancement to the interface documentation.

Best Practices for Default Methods

Question

Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Default methods should be simple.
  - Reduces likelihood of complex dependencies in interfaces.
  - Promote self-containment.
  - Enhancement to the interface documentation.
    - What optional methods do when called if they are not implemented?

Best Practices for Default Methods

Question

Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Default methods should be simple.
  - Reduces likelihood of complex dependencies in interfaces.
  - Promote self-containment.
  - Enhancement to the interface documentation.
    - What optional methods do when called if they are not implemented?
- Take care in using default methods for *new* methods that interface implementers should override.

Best Practices for Default Methods

Question

Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Default methods should be simple.
  - Reduces likelihood of complex dependencies in interfaces.
  - Promote self-containment.
  - Enhancement to the interface documentation.
    - What optional methods do when called if they are not implemented?
- Take care in using default methods for *new* methods that interface implementers should override.
  - May **inadvertently mask** interface evolution if the developers' intention is to break *existing* implementers.

Best Practices for Default Methods

Question

Are there best practices and/or patterns that can be extracted from these situations?

Best Practices for Default Methods

Question

Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.
  - More self-contained.

Best Practices for Default Methods

Question

Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.
  - More self-contained.
  - Reduces external dependencies.

Best Practices for Default Methods

Question

Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.
  - More self-contained.
  - Reduces external dependencies.
- Consider architectural implications.

Best Practices for Default Methods

Question

Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.
  - More self-contained.
  - Reduces external dependencies.
- Consider architectural implications.
  - Rethink separating interface declarations and interface implementations into separate modules.

Best Practices for Default Methods

Question

Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.
  - More self-contained.
  - Reduces external dependencies.
- Consider architectural implications.
  - Rethink separating interface declarations and interface implementations into separate modules.
  - Default methods may contain references to implementation modules.

Best Practices for Default Methods

Question

Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.
  - More self-contained.
  - Reduces external dependencies.
- Consider architectural implications.
  - Rethink separating interface declarations and interface implementations into separate modules.
  - Default methods may contain references to implementation modules.
  - Typically not available to interface modules.

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Call forwarding for **deprecated** interface methods.

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Call forwarding for **deprecated** interface methods.
  - Forward to replacement API, if applicable.

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Call forwarding for **deprecated** interface methods.
  - Forward to replacement API, if applicable.
  - Self-documenting.

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Call forwarding for **deprecated** interface methods.
  - Forward to replacement API, if applicable.
  - Self-documenting.
  - Eliminates any confusion over deprecation between interface and skeletal implementation class.

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Call forwarding for **deprecated** interface methods.
  - Forward to replacement API, if applicable.
  - Self-documenting.
  - Eliminates any confusion over deprecation between interface and skeletal implementation class.
- Choose **general** default implementations.

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Call forwarding for **deprecated** interface methods.
  - Forward to replacement API, if applicable.
  - Self-documenting.
  - Eliminates any confusion over deprecation between interface and skeletal implementation class.
- Choose **general** default implementations.
  - General enough for *all* potential implementers.

Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Call forwarding for **deprecated** interface methods.
  - Forward to replacement API, if applicable.
  - Self-documenting.
  - Eliminates any confusion over deprecation between interface and skeletal implementation class.
- Choose **general** default implementations.
  - General enough for *all* potential implementers.
  - If too narrow, use skeletal implementation pattern instead.

Conclusion

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
  - Can possibly allude traditional postmortem approaches.

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
  - Can possibly allude traditional postmortem approaches.
- *Experienced* project committers provide valuable feedback.

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
  - Can possibly allude traditional postmortem approaches.
- *Experienced* project committers provide valuable feedback.
- Approach was applied to 19 open source projects to assess Java 8 default methods.

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
  - Can possibly allude traditional postmortem approaches.
- *Experienced* project committers provide valuable feedback.
- Approach was applied to 19 open source projects to assess Java 8 default methods.
- Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
  - Can possibly allude traditional postmortem approaches.
- *Experienced* project committers provide valuable feedback.
- Approach was applied to 19 open source projects to assess Java 8 default methods.
- Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.
- Best practices extracted.

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
  - Can possibly allude traditional postmortem approaches.
- *Experienced* project committers provide valuable feedback.
- Approach was applied to 19 open source projects to assess Java 8 default methods.
- Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.
- Best practices extracted.
- Can benefit developers and language designers, especially those considering similar constructs for other languages.

Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
  - Can possibly allude traditional postmortem approaches.
- *Experienced* project committers provide valuable feedback.
- Approach was applied to 19 open source projects to assess Java 8 default methods.
- Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.
- Best practices extracted.
- Can benefit developers and language designers, especially those considering similar constructs for other languages.
- More info at `http://cuny.is/interefact`.

For Further Reading

- Bloch, Joshua (2008). *Effective Java*. Prentice Hall.
- Goetz, Brian (June 2011). *Interface evolution via virtual extensions methods*. Tech. rep. Oracle Corporation. URL: http://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v4.pdf (visited on 08/03/2017).
- Khatchadourian, Raffi and Hidehiko Masuhara (2017). "Automated Refactoring of Legacy Java Software to Default Methods". In: *International Conference on Software Engineering*. ICSE '17. ACM/IEEE. Buenos Aires, Argentina: IEEE Press, pp. 82–93. ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.16.
- Palsberg, Jens and Michael I. Schwartzbach (1994). *Object-oriented type systems*. John Wiley and Sons Ltd. ISBN: 0-471-94128-X.
- Tip, Frank et al. (May 2011). "Refactoring Using Type Constraints". In: *ACM Transactions on Programming Languages and Systems* 33.3, pp. 91–947. ISSN: 0164-0925. DOI: 10.1145/1961204.1961205.