Fault Injection
Introduction to Programming
Summative Assignment
Michælmas 2010

Jan Minář <rdancer@rdancer.org>

December 1, 2010

Chapter 1

Abstract

In this exercise,[1] we are going to test two commonly understood positions: (1) That strongly typed languages such as *Java* fare better than weakly typed languages such as *C* when it comes to programming error detection, and (2) that compilers producing unhelpful inaccurate messages due to inherent theoretical difficulty of doing otherwise. We are going to show that while the former is likely true, the latter is dubious at best.

The source code for the programs used in this exercise accompanies this report, and will be published on the author's web site after the moratorium imposed by (*University Guidance on Procedures in respect of Assessment Irregularities* 2010) will have passed.

[1] This is a summative work for the Introduction to Programming module taught at Durham University, UK in Michælmas 2010.

Chapter 2

Introduction

Java (Gosling, Joy, Steele & Bracha 2005) is a strongly typed language. This means that whenever the compiler encounters an assignment or comparison, it checks and enforces that the variables and values in that statement are of the same (or compatible) data type. The main advantage strong typing has over weak typing is that it ensures that programming mistakes cause an error early during the development process, and that way resources are saved that would have to be spent on testing and debugging.

Error messages generated by computer programs can sometimes be perceived as cryptic and not entirely helpful. It is a common experience that for any given program, finding and correcting design and implementation errors will take vastly more time than actually writing the source code. If the strong type system identifies most errors during compile-time, surely if the compiler would pinpoint the error accurately, the error would be easy to correct?

In this experiment, we are going to test the following two hypothesis (quoted verbatim from (Bradley 2010, p1)):

1. "The Java type system helps to identify programming errors at compile-time instead of run-time"

2. "It is difficult for compilers to identify correctly what programming errors have been made and where"

Chapter 3

Method

In order to test the hypotheses, we used fault injection (Bradley 2010, p1) to introduce random errors into the source code of a Java program. Following the instructions in (Bradley 2010), we have created three small data sets with ten data points each, for a grand total of 30 data points.

The flowchart 3.1 shows an overview of or workflow in generating the three data sets.

3.1 Generating data sets

The assignment suggests using the *BlueJ* environment, however, it doesn't require it. We have opted to use our normal Java build environment for compiling and running the code. This is a *GNU Make*-based environment developed for our coursework, in part for the purposes of (Minář 2009). This allows us to automate the experiment, and in doing that, improve the quality of the generated data.

We are going to be using *OpenJDK*, Java version 1.6, running on *Ubuntu*.

```
$ java -version
java version "1.6.0_18"
OpenJDK Runtime Environment (IcedTea6 1.8.2) (6b18-1.8.2-4ubuntu2)
OpenJDK 64-Bit Server VM (build 16.0-b13, mixed mode)
```

We have written a Java program `MyRandom.java`, which generates raw random data sets. The program is executed by running the `random` wrapper script. We have decided that in order to better test the hypotheses, that we would not perform source code changes in this experiment that would obviously result in no *semantic* change (e.g. removing an empty line or changing the contents of a comment). The raw data sets will therefore need to be evaluated by hand. This necessity allows us to simplify `MyRandom.java` and allow it to generate duplicit entries.

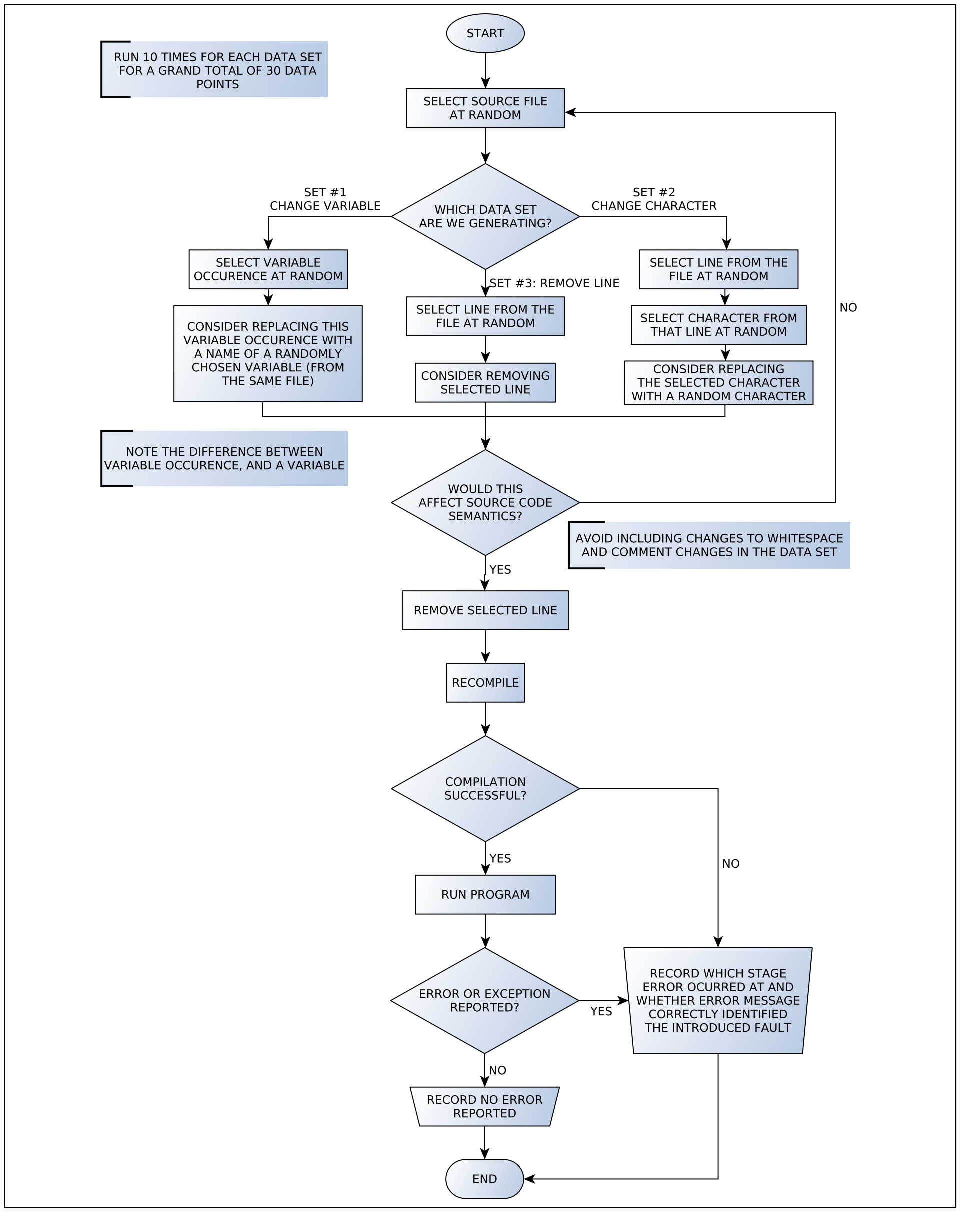


Figure 3.1: Testing method

Once we had generated the random data, thirty copies of the source tree supplied with the assignment were made, on which alterations would be made. We would alter the source tree as per the instruction in the random data, let the Make environment perform the compilation and runtime test, and note down the results.

3.2 Extra data sets

Apart from the set generated by us, we have used (Hoad 2010) and (Nugee 2010). Thanks go to their authors for allowing us to use their high-quality data sets.

3.3 Limitations

Closer critical scrutiny could concievably yield several objections as to the limitations of the method.

1. The data set is rather small. It is the hope of the author that when the results of all the students in the year are collated, meaningful conclusions could be made.

2. It is dificult to infer from a particular product behaviour on the theoretical limits of the technology. We are using two compilers in this experiment (*BlueJ* and *OpenJDK*), and we are going to show that they do not go to their theoretical limits when identifying fault locations. It is impossible to infer whether it is "difficult for compilers" in general to identify what fault caused an error and where it was located.

3. We may be using *pseudo-random* in place of *truly random* data, at least on most contemporary personal computers. This is a limitation set explicitly in (Bradley 2010), as we are asked to use `java.util.Random`, this class may use less-then-true randomness on a particular platform.

4. The data in (Nugee 2010) and (Hoad 2010) suffers from the same problems as the data generated by ourselves, compounded by the fact that the method may differ slightly, and we have not had the opportunity to study their method of generating the data in detail, as that would mean reading their report, clearly a breach of (*University Guidance on Procedures in respect of Assessment Irregularities* 2010, §6.3.5)

3.4 Runtime Testing

The program is rather small (602 source code lines), and so one of the options would have been to perform white box testing, run the program in a debugger, and note how the injected fault affected the program behaviour. However, it was felt that this was only tangentially related to the main aim of this experiment, and so a much simpler black box test was employed. The sample data file which was supplied with the program was read in, and the subsequent console output of the program was compared to the console output of an unaltered program. If the two did not differ, the main method of the program was deemed to have "execute[d] as before" (Bradley 2010, p2).

Chapter 4

Results

We presents our results in several tables and with graphs.

Note: In (Nugee 2010), there are eleven data points in each group instead of ten. This is convenient, as in our method, we do not include data points that do not affect the semantics, and there are three such data points in the first two groups.

Modification				Fault detection			Error reported			
Data point number	File base name	Line number	Original line text [sans semantically irrelevant white space]	Compile-time	Run-time	Undetected	Error message / details	reporeted line number	Problem identified	Line distance from injected fault
1	Album	22	{	x			';' expected public Album(String aName) ^	21	no	1
2	Album	30	artist = a;			x	n/a	n/a	n/a	n/a
3	Artist	45	members.add(member);			x	n/a	n/a	n/a	n/a
4	Library	7	import java.util.Random;	x			cannot find symbol symbol : class Random location: class Library Random random = new Random(); ^	53	yes	46
5	Library	14	/**	x			illegal start of type * Constructor for objects of class Library ^	14	no	0
6	Library	78	randomTrackList.get(random.nextInt(tempTrackList.size()));			x	n/a	n/a	n/a	n/a
7	Library	82	}	x			reached end of file while parsing } ^	82	yes	0
8	Main	168	BufferedReader buffr = new BufferedReader(new InputStreamReader(System.in));	x			cannot find symbol symbol : variable buffr location: class Main str = buffr.readLine(); ^	172	yes	4
9	Track	36	rating=r;			x	n/a	n/a	no	n/a
10	Track	80	public double getLength()	x			return outside method return length; ^	81	yes	1
11	Main	194	*/	x			Unclosed comment	190	yes	4
12	Main	201	return library;	x			Missing return statement	202	yes	1
13	Artist	56	}	x			Illegal start of expression	58	no	2
14	Album	14	/**	x			Illegal start of type	15	no	1
15	Track	66	public void setDate(String d)	x			Cannot find symbol – variable d	68	yes	2
16	Artist	64	for (Artist m:members)	x			Cannot find symbol – variable m	65	yes	1
17	Track	81	{	x			';' expected	80	no	1
18	Main	16	public static void main(String []args) throws IOException	x			Unreported exception java.io.IOException; must be caught or declard to be thrown	28	yes	12
19	Library	6	import java.util.ArrayList;	x			Cannot find symbol – class ArrayList	11	yes	5
20	Main	86	Artist artist = new Artist();	x			Cannot find symbol – variable artist	92	yes	6
21	Artist	24	public void setName(String n)	x			cannot find symbol - variable n	26	yes	2
22	Artist	58	public String toString()	x			return outside method	67	yes	9
23	Album	103	for (Track t: tracks)	x			cannot find symbol - variable t	104	yes	1
24	Artist	22	}	x			illegal start of expression	24	yes	2
25	Main	187	(blank line)			x	n/a	n/a	n/a	n/a
26	Artist	44	member.setIsSoloist(false);			x	n/a	n/a	no	n/a
27	Album	18	{	x			';' expected	17	yes	1
28	Artist	43	member.setName(n);			x	n/a	n/a	n/a	n/a
29	Track	127	public void addPlayCount()			x	n/a	n/a	n/a	n/a
30	Main	54	BufferedReader reader = new BufferedReader(new FileReader("MusicLib.txt"));	x			cannot find symbol - variable reader	55	yes	1
			totals	*22*	*0*	*8*		*yes*	*17*	*4.7*
					total	*30*		*no*	*7*	average
								n/a	*6*	
								total	*30*	

Figure 4.1: Data set #1: Remove line

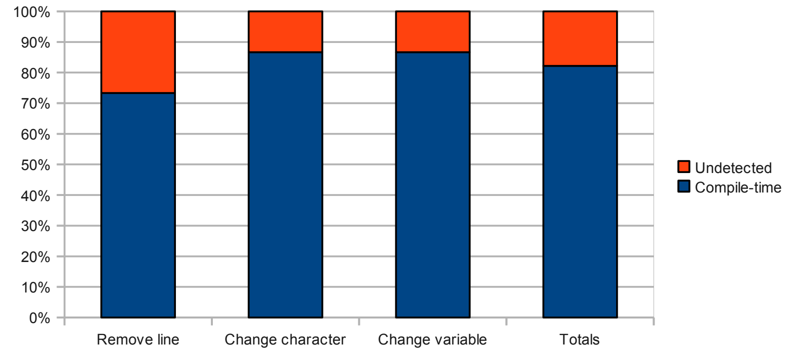
Modification							Fault detection			Error reported			
Data point number	File base name	Line number	Character position	To be replaced	Replace with	Original line text [sans semantically irrelevant white space]	Compile-time	Run-time	Undetected	Error message / details	reporeted line number	Problem identified	Line distance from injected fault
1	Album	54	26	T	g	public void addTrack(Track t)	x			cannot find symbol symbol : class grack location: class Album public void addTrack(grack t) ^	54	yes	0
2	Artist	64	29	m	4	for (Artist m:members)	x			cannot find symbol symbol : variable me4bers location: class Artist for (Artist m:me4bers) ^	64	yes	0
3	Library	36	11	' '	Q	{	x			not a statement { Q ^	36	yes	0
4	Library	41	15	' '	}	tracks.add(t);	x			illegal start of expression } tracks.add(t); ^	41	yes	0
5	Library	57	16	s	)	ArrayList<Track> randomTrackList = new A	x			not a statement ArrayLi)t<Track> randomTrackList = new ArrayList<Track>(); ^	57	yes	0
6	Main	112	6	' '	E	}	x			not a statement E } ^	112	yes	0
7	Main	155	22	.	y	track.setGuestArtist(addArtist());	x			cannot find symbol symbol : method trackysetGuestArtist(Artist) location: class Main trackysetGuestArtist(addArtist()); ^	155	yes	0
8	Main	203	10	' '	5	track.setGuestArtist(addArtist());	x			not a statement 5 track.setGuestArtist(addArtist()); ^	203	yes	0
9	Track	94	2	' '	)	public void setRating(int r)	x			illegal start of type) public void setRating(int r) ^	94	yes	0
10	Track	111	4	' '	:	}	x			illegal start of expression :} ^	111	yes	0
11	Main	100	23	t	M	System.out.println("Enter Band Name");			x	n/a	n/a	n/a	n/a
12	Main	177	12	.	P	str = buffr.readLine();	x			Cannot find symbol – method buffrPreadLine()	177	yes	0
13	Track	46	17	T	U	public void setUitle(String t)	x			Cannot find symbol – method setTitle(java.lang.String); maybe you meant: getTitle() or setUiule(String)		yes	another source file
14	Main	105	1	{	C	{	x			Not a statement	105	yes	0
15	Album	3	28	' '	=	* size and the guest Artist - if applicable-			x	n/a	n/a	n/a	n/a
16	Library	49	9	m	O	* @param rating the minimum rating			x	n/a	n/a	n/a	n/a
17	Artist	11	23	s	5	private ArrayList<Artist> members;	x			Cannot find symbol – class Arti5t	11	yes	0
18	Artist	10	15	' '	s	private String name;	x			<identifier> expected	10	yes	0
19	Main	129	27	T	Z	System.out.println("Enter Track Title");			x	n/a	n/a	n/a	n/a
20	Artist	18	1	{	k	{	x			';' expected	17	no	1
21	Track	113	2		n	public String getLocation()	x			<identifier> expected	113	yes	0
22	Album	28	24		v	albumName = aName;	x			cannot find symbol - variable albumvame	28	yes	0
23	Album	53	20		1	}	x			not a statement	53	yes	0
24	Artist	30	24		O	{	x			';' expected	29	yes	1
25	Track	59	12		(	}	x			illegal start of expression	59	yes	0
26	Track	51	32		s	public String getTitle()	x			cannot find symbol - class string	51	yes	0
27	Main	191	7		v	/**	x			<identifier> expected	191	yes	0
28	Album	24	17		B	tracks = new ArrayList<Track>();	x			cannot find symbol - class ArrByList	24	yes	0
29	Artist	67	4		7	return str;	x			cannot find symbol - class ret7rn	67	yes	0
30	Track	63	13		n	return artist;	x			cannot find symbol - variable artisn	63	yes	0
						totals	*26*	*0*	*4*		*yes*	*25*	*0.1*
								total	*30*		*no*	*1*	average
											n/a	*4*	
											total	*30*	

Figure 4.2: Data set #2: Change character

	Modification					Fault detection			Error reported			
Data point number	File base name	Line number	To be replaced	Replace with	Original line text [sans semantically irrelevant white space]	Compile-time	Run-time	Undetected	Error message / details	reporeted line number	Problem identified	Line distance from in-jected fault
1	Library	54	random	albums	Random random = new Random();	x			cannot find symbol symbol : variable random location: class Library int i = random.nextInt(); ^	67	yes	13
2	Library	78	random	rating	randomTrackList.get(random.nextInt(tempTrackList.size()));	x			int cannot be dereferenced randomTrackList.get(rating.nextInt(tempTrackList.size())); ^	78	yes	0
3	Artist	10	name	n	private String name;	x			cannot find symbol symbol : variable name location: class Artist Name = ""; ^	19	yes	9
4	Track	32	title	s	Title =t;	x			incompatible types found : java.lang.String required: double s =t; ^	32	yes	0
5	Album	37	albumName	a	albumName = n;	x			cannot find symbol symbol : variable a location: class Album a = n; ^	37	yes	0
6	Main	130	buffr	str	str = buffr.readLine();	x			cannot find symbol symbol : method readLine() location: class java.lang.String str = str.readLine(); ^	130	yes	0
7	Track	78	length	artist	length = l;	x			incompatible types found : double required: Artist artist = l; ^	78	yes	0
8	Track	33	a	artist	artist = a;			x	n/a	n/a	n/a	n/a
9	Track	56	a	date	public void setArtist(Artist a)	x			cannot find symbol symbol : variable a location: class Track artist = a; ^	58	yes	2
10	Track	46	t	location	public void setTitle(String t)	x			cannot find symbol symbol : variable t location: class Track title = t; ^	48	yes	2
11	Album	28	albumName	artist	albumName = aName;	x			Incompatible types – found java.lang.String but expected Artist	28	yes	0
12	Track	87	size	title	size=z	x			Incompatible types – found double but expected java.lang.String	87	yes	0
13	Track	100	rating	size	return rating;	x			Possible loss of precision found : double required: int	100	yes	0
14	Artist	20	members	name	members = new ArrayList<Artist>()	x			Incompatible types – found java.util.ArrayList<Artist> but expected java.lang.String	20	yes	0
15	Album	29	albumType	albumName	albumType=aType;			x	n/a	n/a	n/a	n/a
16	Artist	55	isSoloist	members	return isSoloist;	x			Incompatible types – found java.util.ArrayList<Artist>but expected boolean	55	yes	0
17	Track	48	title	rating	title = t;	x			Incompatible types – found java.lang.String but expected int	48	yes	0
18	Track	33	artist	location	artist = a	x			Incompatible types – found artist but expected java.lang.String	33	yes	0
19	Track	105	location	date	location = l;			x	n/a	n/a	n/a	n/a
20	Album	30	artist	tracks	artist = a;	x			Incompatible types – found artist but expected java.util.ArrayList <Track>	30	yes	0
21	Album	46	albumType	albumName	albumType = t;			x	n/a	n/a	n/a	n/a
22	Artist	50	isSoloist	members	isSoloist = s;	x			incompatible types - found boolean but expected java.util.ArrayList<Artist>	50	yes	0
23	Album	75	length	albumType	double length = 0.0;	x			cannot find symbol - variable length	77	yes	2
24	Album	77	length	albumType	length = length + t.getLength();	x			incompatible types - found double but expected java.lang.String	77	yes	0
25	Track	48	title	guestArtist	title = t;	x			incompatible types - found java.lang.String but expected Artist	48	yes	0
26	Artist	20	members	name	members = new ArrayList<Artist>();	x			incompatible types - found java.util.ArrayList<Artist> but expected java.lang.String	20	yes	0
27	Track	82	length	date	return length;	x			incompatible types - found java.lang.String but expected doublt	82	yes	0
28	Track	33	artist	location	artist = a;	x			incompatible types - found Artist but expected java.lang.String	33	yes	0
29	Album	105	rating	albumName	return rating/tracks.size();	x			operator / cannot be applied to java.lang.String,int	105	yes	0
30	Album	9	albumName	tracks	private String tracks;	x			tracks is already defined in Album	12	yes	3
					totals	*26*	*0*	*4*		*yes*	*26*	*1.2*
							total	*30*		*no*	*0*	average
										n/a	*4*	
										total	*30*	

Figure 4.3: Data set #3: Change variable

Fault detection			Problem identified			
	Compile-time	Undetected	yes	no	n/a	Line distance from injected fault
Remove line	22	8	17	7	6	1.2
Change character	26	4	25	1	4	0.1
Change variable	26	4	26	0	4	1.2
Totals	*74*	*16*	*68*	*8*	*14*	*0.8*
	82%	*18%*	*76%*	*9%*	*16%*	



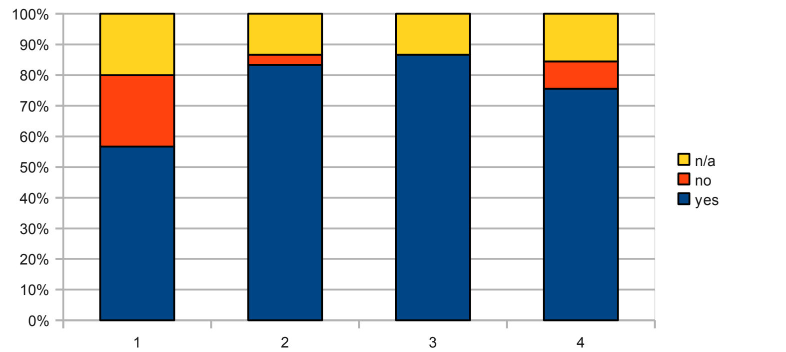


Figure 4.4: Summary of the data sets

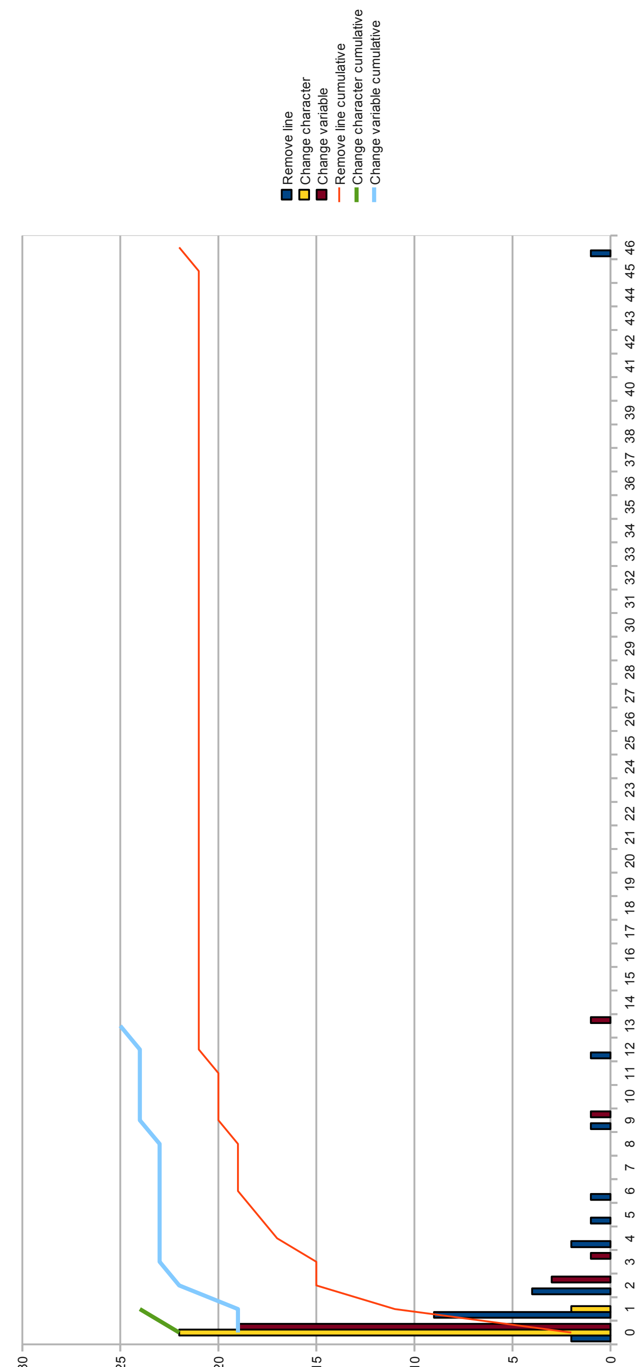


Figure 4.5: Distance of reported error from the injected fault, axis x: distance between actual and reported error [lines], axis y: number of data points

Chapter 5

Discussion

5.1 Typing system

Clearly, the strong typing system of Java helped to prevent some of the faults we introduced when changing a single character (4.2), as 18 out of the 26 compile-time errors involved a variable type mismatch. However, in the remaining two tests, not a single error involved the type system, and it is therefore dubious whether the typing system would have helped.

Runtime errors are conspicuously absent from our data — out of 30 tests run, there were 26 errors, all of them during compile time.

This seems to suggest that the Java typing system indeed prevents certain types of errors from getting past the compile-time error checking. The first hypothesis stands.

5.2 Error location

As can be seen in 4.5, most errors were identified to be within a small distance of the fault we had introduced. Removing lines resulted in the errors reported at a greater distance from the fault then replacing a character or a variable.

It is obvious that the compilers could do better when identifying error location. For example, if the source code is well indented, the compiler could report precisely which opening or closing curly bracket has been left out.

5.2.1 Lowering distance

We are going to show now that it is indeed possible to substantially lower the distance between where the fault was injected and which line number was reported to contain the error.

Let us take the three most distant data points; we have put them together in (?).

The first one is an obvious misspelling. The compiler finds out what the problem is. Using statistical analysis and a spellchecker, the compiler could be extended to suggest with confidence where the misspelling took place. The distance could be lowered to zero.

Second and third example is also easily solvable. It is much easier for a computer program then for a human to check what class *Random* is supposed to be. If the compiler checked all the subsequend methods `random` is supposed to contain, it could again confidence suggest that this is a case of a missing import statement. Import statements are customarily put near the top of the file, perhaps with other import statements. A suggestion could be made to automatically include the import statement (some IDEs have this functionality). Again, the distance could be lowered to zero or close to zero.

Third example is a variation on the second example. The instantiation of `random` is missing. This is a simple suggestion that the compiler could make, albeit with not such great confidence. The distance could again be lowered to close to zero.

The error messages could be made a lot more accurate and helpful. We are not convinced that there is a theoretical barrier that would prevent compilers from improving radically. This casts grave doubt on the second hypothesis.

5.3 Conclusion

We have shown that our limited data set doesn't support the hypothesis that "the Java type system helps to identify programming errors at compile-time instead of run-time" (Bradley 2010).

Appendix A

Bugs

A bug has been spotted in the supplied source code, which was corrected (the extraneous semicolon removed) before the testing begun

```
$ javac -Xlint Main.java
Main.java:51: warning: [empty] empty statement after if
          else if (Integer.parseInt(str) == 2);
                                              ^
1 warning
```

Bibliography

Bradley, S. (2010), *IP: Summative Assignment: Fault Injection - Lecture fi*, Durham University. Summative assignment for Introduction to Programming.

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2005), *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*, Addison-Wesley Professional.

Hoad, A. (2010), *Fault injection data set*, Durham University. Summative assignment for Introduction to Programming.

Minář, J. (2009), *Computer Systems — Introduction to Networks*, Durham University. Summative assignment.

Nugee, E. (2010), *Fault injection data set*, Durham University. Summative assignment for Introduction to Programming.

University Guidance on Procedures in respect of Assessment Irregularities (2010), *Learning and Teaching Handbook* .
URL: *http://www.dur.ac.uk/learningandteaching.handbook/6/3/5/*