ISSN 1744-1986

Technical Report N° 2010/ 22

Designing and evaluating an intention-based comment enforcement scheme for Java

Kevin Matz

15 September, 2010

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk

Designing and evaluating an intention-based comment enforcement scheme for Java

A dissertation submitted in partial fulfilment
of the requirements for the Open University's
Master of Science Degree
in Software Development

Kevin Matz
X9786928

5 March 2011

Word count: 15,872

Preface

I would like to thank my supervisor, Dr. Les Neal, for his extremely helpful guidance and advice throughout this research project. His constant support and encouragement are very much appreciated. I would also like to thank my specialist adviser, Dr. Yijun Yu, for his very useful feedback on early drafts.

I extend my appreciation to Dr. Frank Martin, who kindly agreed to distribute the survey to members of the BCS Advanced Programming Specialist Group, which greatly increased the number of survey responses.

I am also very grateful to all of the participants who completed the survey. I am pleased by the insightful and constructive feedback that was provided on the proposed approach described in this dissertation.

Table of contents

List of figures

List of tables

Abstract

Software maintenance forms a significant portion of the cost of large-scale software projects. A time-consuming part of maintenance is program comprehension – reading legacy code to understand how and where to make changes. The process of understanding existing code involves reconstructing the design intentions and rationale of the original developers. This dissertation argues that explicitly recording intention and rationale information during design and construction eases program comprehension during maintenance.

This dissertation conducts a survey of practicing software developers to understand difficulties in software maintenance and opinions on software documentation. The results and a literature survey are then used to argue that significant problems exist which can best be dealt with by designing a new technology-based solution.

By reviewing the program comprehension literature and examining past solutions, requirements are formulated for an "ideal" solution for recording intention and rationale documentation.

A partial solution, *Design Intention Driven Programming*, is proposed, which encourages developers to record design intentions before writing code. The process is supported by a language, *Java with Intentions*, which adds *intention comments*, first-class documentation constructs, to the Java language. The compiler flags as errors any artefacts (e.g., classes) not described by intention comments and uses complexity metrics to detect "empty" comments. A rudimentary prototype of a precompiler for the language and a sample application are constructed as proofs of concept.

The solution is evaluated using several analyses and by surveying developers for feedback on its practicality. Respondents' opinions are divided on the solution's feasibility and utility. Numerous problematic issues are identified, including resistance of developers to write documentation, limitations of the documentation enforcement mechanism, and the lack of concrete evidence of long-term cost savings. The evaluation suggests that, while the approach may be promising for some projects and teams, its unpopularity with most developers renders it impractical for typical commercial projects.

1 Introduction

Large-scale enterprise software projects are costly endeavours. While the initial analysis, design, and construction of a software system can often take dozens or even hundreds of person-years of effort, over the system's entire operational life, the majority of the cost and labour will be spent during the *software maintenance* phase (Pressman, 2010, p. 763; van Vliet, 2008, p. 469).

Software maintenance, also known as *software evolution*, refers to the phase after a system has first gone into productive operation, during which the system undergoes adaptations to meet changing requirements, and corrections to fix defects (Pfleeger, 1998, p. 412).

Maintenance is considered one of the most problematic issues in software development:

- It has been estimated that approximately 60 to 70 percent of a typical software organisation's resources are spent on software maintenance (Pressman, 2010, p. 763; Yip *et al.*, 1994, p. 71).
- According to a survey, 70.2% of project managers regard the software maintenance process as "inefficient" (Sousa and Moreira, 1998, p. 270).
- Software developers in maintenance projects tend to report low morale and high levels of job frustration (van Vliet, 2008, p. 474).
- The percentage of resources spent on maintenance as opposed to new development tends to increase over time; in 2005, an estimated 76 percent of developers in the U.S. were considered "maintenance developers" as opposed to developers[1] building new systems; this has increased from 17 percent in 1975 and 47 percent in 1990 (Jones, 2006, p. 4).

Because of the quantity of time, money, and effort spent on software maintenance, any potential solutions that even partially alleviate fundamental problems in software maintenance would be welcomed as cost-saving measures in software organisations.

[1] In this dissertation, we differentiate between "maintenance developers" and developers involved in constructing new systems. Used alone, the general term "developer" refers to both groups, i.e., the general population of all software developers.

1.1 Program comprehension as a major cost factor in software maintenance

Software maintenance consists of the following types of activities (Swanson, 1976):

1. *Corrective maintenance:* correcting reported defects
2. *Adaptive maintenance:* performing changes necessitated by modifications to the system's environment
3. *Perfective maintenance:* adding functionality or improving performance, maintainability, test coverage, etc.
4. *Preventive maintenance:* refactoring and proactive correction of faults identified by developers but not yet reported by users

Knowing where and how to make a change requires an understanding of the structure and functioning of either parts of the system, or the entire system. The process of gaining an understanding a system's source code is referred to as *program understanding* or *program comprehension* (Corbi, 1989). It can also be called *reverse engineering*, though this term tends to imply that higher-level abstractions and models are being derived and documented (Pressman, 2010, p. 771).

Program comprehension is one of the largest factors contributing to the costs of software maintenance. Estimates of the percentage of time developers spend on this activity range from 30 to 60 percent (Devanbu, 1990, p. 250), to 40 percent (Sousa and Moreira, 1998, p. 269), to as high as 50 to 90 percent (Standish, 1984).[2]

To understand the behaviour and structure of a program, developers can read the source code, run and trace the program, or read documentation about the program (Corbi, 1989). Although analysis and design documentation can be useful, documentation is not always available, up-to-date, or relevant, and as a result, source code and comments are the most trusted and most used artefacts used by developers during program comprehension (LaToza et al., 2006, p. 499).

[2] No more recent studies on this topic could be located. The percentage of time spent on program comprehension might have decreased since these studies due to improvements in software engineering techniques, or might have increased due to increases in the size and complexity of systems.

Program comprehension is characterised as a difficult task. The size and complexity of modern systems is a major factor. Another primary reason for the difficulty is the fact that *intentions* and *rationale* are not explicitly expressed in source code, unless explicitly stated in program comments.

1.2 Intention and rationale in software development

To understand what is meant by *intention* and *rationale*, let us examine the *Seven Stages of Action* model (Norman, 1998, p. 46), which argues that humans perform the following steps when interacting with a device:

1. Forming a goal
2. Forming an intention
3. Specifying an action
4. Executing an action
5. Perceiving the state of the world
6. Interpreting the state of the world
7. Evaluating the outcome

Applying this model to computer programming, we can imagine that a developer implementing a module or making a change first formulates a high-level goal, such as "fulfil the requirement that the user shall be able to sort the song titles in a playlist". There may be many ways to fulfil this goal. The developer formulates an intention – a way of reaching the goal – which in this case may be to implement the Quicksort algorithm in order to sort records. The developer then formulates a *plan*, consisting of one or more actions, to fulfil the intention (Bratman, 1987, p. 29). The developer then executes the actions, which involve writing or changing pieces of code according to the intention. "Perceiving and interpreting the state of the world" and "evaluating the outcome" correspond to unit testing activities: the developer verifies whether the goal and intentions were carried out correctly. Iterations will take place if the evaluation finds that the goal and intentions have not yet been fulfilled correctly.

A developer's *intention,* then, is *what* he or she wants a component or aspect of the system to perform, and *how* the system should do it. The intention is a

"characterisation of a desired action" (Bratman, 1987, p. 1), or "a desire that something be accomplished" (Simonyi, 1995). A developer carries out an intention by means of actions involving the writing of source code.

A major category of software defects consists of cases where the developer has mistakenly written code that does not match his or her original intention. For example, the developer may intend to implement the Quicksort algorithm, but he or she makes an error such that the sort order is sometimes incorrect. What was implemented is thus not really the Quicksort algorithm at all; the implementation does not match the intention.

Rationale is the reasoning behind the intention. Rationale is an explanation of why something is implemented in a particular way (LaToza *et al*., 2006, p. 499), or why one particular alternative was chosen over other alternatives. For example, why was Quicksort and not some other algorithm chosen in this particular instance?

Lengthy descriptions of rationale are unimportant for many trivial implementation details, but at higher levels of design and architecture, understanding the reason why the system was designed in a particular way can prevent later developers from choosing alternatives that the original designers have already determined to be problematic or unsuitable.

In summary, an intention is *what* the designer or developer wants some element of the system to do and *how* it should do it, and the rationale is *why* the designer believes it should be that way. Simonyi (1995) asserts that "the forming of some intention in the programmer's mind" is "a key element in programming".

1.3 The loss of intention and rationale in the transformation from requirements to code

In a typical "ideal" document-driven software design process, designing and building a system involves a series of documentation artefacts. There are many variations, but typically, goals for the system are first determined (van Lamsweerde, 2001), from which a requirements specification document is produced. From the requirements,

architectural designs and functional specifications may be produced. Technical design specifications may then be written to concretise how functionality is to be implemented. Source code is then constructed on the basis of the functional and technical specifications.

At each stage of this process, design decisions are made, by which high-level abstractions are translated into lower-level, more concrete details. For example, a requirement might state that the "user interface shall conform to the organisation's standard conventions". Based on this, the functional design might specify that the application must use pull-down menus, with a specific wording and ordering for consistency across the organisation's systems. The technical design might then specify that pull-down menus are to be implemented using a particular GUI component, and that menu items are to be defined in an XML file. The developer might then need to devise a format for the XML file, and write code to populate the GUI component with data loaded from the file.

At each stage in the process, each respective document tends to focus on presenting the design that was arrived at by a series of decisions that took place at that stage (van Vliet, 2008, p. 474). Ideally, there will be a cross-reference to preceding documents, and there will be a discussion or a justification of why one particular design decision was chosen. Very frequently, however, this information is missing.

This means that information about higher-level abstractions can be "lost" at each stage (van Vliet, 2008, p. 481). Translation from a higher-level, abstract conceptualisation to a more fine-grained specification is, in general, a lossy process, as the architectural principles and design decisions, and rationale behind those choices, are usually not explicitly stated and transferred to the new document. This is especially true at the transition from natural-language specifications to programming-language source code.

Also, this discussion so far has assumed that an ideal, rational design process has been followed; in many organisations, software development is a more disorganised process that does not involve formal stages and documentation artefacts, and even teams that attempt to follow a formal document-driven process are rarely able to

accomplish it as planned (Parnas, 1986). Whether design documents have been produced and have become outdated or are of poor quality, or no design documents exist at all, the result is that maintenance developers must rely on the source code and comments in the source code as the primary source of understanding of the system (LaToza *et al.,* 2006, p. 499; de Souza *et al.,* 2005, p. 74).

Visualisation tools (discussed later in this dissertation) can aid somewhat in uncovering structures in software systems. Reverse-engineering techniques such as processes for generating goal models from existing code (Yu *et al.*, 2005) are also promising. Visualisation tools and reverse-engineering techniques are still no "silver bullet"; they are labour-intensive to use (they generally cannot automatically identify relevant structures or underlying architectural principles, and even if they could, the human operator must still inspect, comprehend, and verify them and synthesise models), and they depend on the quality and consistency of naming and the presence and accuracy of comments.

When source code contains little or no discussion of the original developers' intentions and rationale, maintenance developers need to painstakingly parse the source code to figure out what each part of the program does, how it works, and how it interacts with other components of the program. In effect, when intentions and rationale are not available, maintenance developers must reconstruct that understanding, which is time-consuming, error-prone, and sometimes simply impossible because the information about the original higher-level abstractions is simply no longer present. Respondents to a survey "most often said that understanding the original programmer's intent was the most difficult problem facing the person asked to change the function of [a] program" (Fjeldstad and Hamlen, 1979, p. 22).

This dissertation argues that, if designers and developers were to make a systematic effort to record intention and rationale information in a structured way when designing and writing a program, whether in documents, comments, or some new form, it would relieve maintenance developers of much of the effort of trying to reconstruct that information, thus reducing the time spent on program comprehension and potentially leading to savings in long-term maintenance costs.

Because current software development practices and technologies do not do a sufficient job of recording intention and rationale information, it is the goal of this dissertation to either identify or formulate some form of approach or "solution" that will both encourage and enable the proper recording of such information during the construction of software programs. This solution could take the form of a technique or process, a new or improved tool or technology, or some combination of these.

This dissertation will thus address the following multi-part research question:

PART 1	What evidence can be found to justify the design of a new solution to aid the recording of intention and rationale information during software development?
PART 2	What are the requirements for an "ideal" solution?
PART 3	Given the requirements for an ideal solution, can a design for a solution be developed that is feasible, practical, and effective?

As was discussed above, a fundamental motivation underlying this investigation is the desire to reduce the long-term cost of software maintenance projects. Given a specification and implementation of a proposed solution, it would be interesting to test, by means of some form of empirical investigation, whether the solution actually leads to long-term cost savings in software projects. However, as will be explained later in this dissertation, an investigation of this type is not feasible within the time limits imposed by this research project. Because it cannot be adequately addressed, it is not explicitly included in the research question.

1.4 Roadmap for this dissertation

In the present chapter, with reference to the research literature, we have explored difficulties posed by software maintenance and program comprehension, identified the role of intention and rationale in software development, and stated the research question.

Evidence to justify a solution will be gathered from the research literature and from a survey of practicing software developers. Chapter 2 describes the research methods involved in these tasks, and Chapter 3 carries out the research methods in order to make the case for the need for a solution. (This will address Part 1 of the research question.)

Chapter 4 explores general categories of solution and chooses the most promising category for investigation.

Chapter 5 draws again upon the research literature and the results of the survey in order to formulate a list of requirements for an "ideal" solution. (This will address Part 2 of the research question.)

Chapter 6 describes research methods involved in designing and evaluating a solution.

Chapter 7 presents a design for a novel solution intended to meet most of the identified requirements, and in Chapter 8, this solution will be critically evaluated by several means. Chapter 9 then critically examines the research methods used and evaluates the reliability and validity of the evaluation of the solution.

Chapter 10 concludes the dissertation with a summary and interpretation of the evaluation in order to answer Part 3 of the research question. This chapter also reflects upon the research project, states the dissertation's contribution to knowledge, and discusses opportunities for further research.

2 Research methods for investigating the problem and justifying a new solution

In order to justify the investigation and design of a new solution, we shall examine problems experienced by software developers in maintenance projects, and collect requirements for a solution that will address these problems. This chapter explains and justifies the research methods chosen for these tasks.

2.1 Investigating the need for a solution

A new solution can be justified if it can be shown that problems exist in the current practice of software maintenance and that there is some potential to do things in an improved way. The problems will shape the solution and its requirements.

As the first research method, an examination of the literature will be conducted to find evidence of problems in software maintenance and program comprehension.

Referring to research conducted by others ("secondary sources") is useful, but for more credibility, a second research method shall be used to collect and analyse data directly from practicing software development professionals. The results can be checked against existing research.

2.1.1 Choosing a research method

Interviews with developers would yield useful qualitative data and would allow in-depth explorations of difficulties encountered. However, interviews are time-consuming and difficult to schedule; face-to-face interviews would limit participants to the interviewer's immediate geographical area, and strangers are unlikely to volunteer for telephone interviews.

Questionnaire surveys are more promising; they can potentially reach a much larger number of participants, and can yield useful quantitative and qualitative data.

However, questionnaires with fixed lists of questions cannot probe individual circumstances.

Web-based questionnaires are preferable to paper questionnaires as a URL can be easily disseminated to participants (costs are negligible), a wider geographic base can be reached, response rates tend to be higher, and responses require no manual rekeying.

For these reasons, a web-based survey has been chosen.

2.1.2 Planning, designing, and carrying out the survey

The questionnaire actually serves two functions and thus has two parts:

- Part A asks participants about practices at their current organisation and difficulties related to software maintenance and program comprehension, and asks for personal opinions regarding software documentation;
- Part B is concerned with evaluating the solution (which will be presented in Chapter 7). Part B of the survey will be described in Chapter 6.

The questionnaire is aimed primarily at practicing software developers. While academic experts could be consulted instead, those who work hands-on in the software industry will be best able to describe day-to-day problems in software maintenance, and it is this group who would potentially use any proposed new solution.

The survey is anonymous to encourage respondents to report frankly on issues and problems in their organisations.

During the design of the questionnaire, advice was drawn from Sapsford (2007), Weisberg *et al.* (1996), Jackson (1988), and Gray and Guppy (1994). Fowler (1995) provided valuable guidance on writing and evaluating survey questions. Yip *et al.* (1994), Sousa and Moreira (1998), Kajko-Mattsson (2005), de Souza *et al.* (2005), LaToza *et al.* (2006), and Babar *et al.* (2006) are examples of surveys querying

software developers and/or organisations about software maintenance and documentation topics. Root and Draper (1983) address using questionnaires as a software evaluation tool.

To analyse the data, simple summary statistics and analysis techniques will suffice; some rudimentary hypothesis testing will be undertaken to identify relationships in the data. Schlotzhauer (2009), Sapsford (2007), and Weisberg *et al.* (1996) provided useful instruction on summarising quantitative data, analysing relationships, and testing hypotheses. Qualitative coding as described by Richards (2005) will be applied to open-ended textual questions.

The questionnaire consisting of Parts A and B is presented in Appendix E, together with summary statistics. The raw survey response data is given in Appendix F.

The survey was hosted on-line using the service www.surveymonkey.com, chosen amongst several similar services for its usability and cost. The survey's URL was distributed to participants; the first page was a welcome page with instructions.

A trial run of the survey was conducted with three colleagues. As a result of the feedback, the questionnaire was shortened to remove several questions perceived as redundant. Other questions were reworded for clarity.

The selection of participants represents a convenience sample, and this has implications for validity that will be discussed in Chapter 9. Table 1 lists the groups invited to take the survey[3].

[3] Due to the survey's anonymity, it is not possible to reliably count the number of participants from any particular category.

Table 1: Groups invited to participate in the survey

Category	Group	
Group invitations	1	Software developer colleagues at the author's (now previous) employer, JEA Pension System Solutions in Victoria, BC, Canada
	2	Fellow Open University Computing MSc students (via the M801 Chat forum)
	3	Members of the Advanced Programming Specialist Group of the British Computer Society
	4	Attendees of a talk given by the author at the Vancouver Island Java Users' Group
Personal invitations	5	Friends and associates in the author's personal network who are software developers
Web visitors	6	Google AdWords text advertisements
	7	Visitors who located the survey via a web search
Referrals	8	Referrals from participants who forwarded the survey link to their friends and colleagues

Incentives (free lunches and Amazon.co.uk gift vouchers) were offered to some groups.

The survey collection was regularly monitored for suspicious activity. "Prank" and empty submissions were rejected. Two partial submissions were retained where participants had completed significant portions of the survey.

The examination of the literature and the results of the survey will be presented in Chapter 3, where a case will be made to justify a new solution.

2.2 Formulating requirements for a solution

Goals and requirements for an ideal solution will be generated by the following means:

- Surveying the research literature and analysing past attempts at solutions;
- Brainstorming and reflecting on personal experiences in software development; and
- Interpreting responses of survey participants.

These methods will be put into practice in Chapter 5.

3 Making the case for the need for a solution

In this chapter, we will first examine the literature and then present and interpret the data from Part A of the questionnaire to show that problems exist in software maintenance that justify a new solution.

3.1 Evidence from the literature

In long-running software projects, the majority of labour is expended in the maintenance phase. The amount of resources spent on maintenance by software organisations varies, but past estimates have ranged anywhere from 48 percent (Lientz and Swanson, 1980, p. 9), to 51 percent (Fjeldstad and Hamlen, 1979, p. 14), to approximately 60 to 70 percent (Pressman, 2010, p. 763; Yip *et al.*, 1994, p. 71; Boehm, 1976, p. 1236).[4]

One of the most time-consuming and difficult aspects of software maintenance is program comprehension (LaToza *et al.*, 2006, p. 496) – that is, reading and tracing through source code to understand its function and behaviour and deducing the underlying intention and rationale. LaToza *et al.* state that "understanding the rationale behind code is the most serious problem developers face," with 82 percent of survey respondents agreeing that "it takes a lot of effort to understand why the code is implemented the way it is" (*ibid.*, p. 499). Maintenance developers may spend anywhere from 23 to 33 percent (Fjeldstad and Hamlen, 1979, p. 20) to as high as "50 to 90 percent" (Standish, 1984) of their time on reading and understanding source code.

Specifications and other forms of documents are useful but are not always available, accurate, complete, or current. Sousa and Moreira (1998, p. 269) identify missing documentation and insufficient time for keeping documentation current as two of the primary causes of software maintenance difficulties. Poor-quality documentation

[4] It is recognised that some of sources cited in this section are quite old and may no longer be entirely accurate in modern software development environments, but are included for completeness in cases where more recent statistics could not be found.

contributes to developers' dissatisfaction with maintenance work (Kajko-Mattson, 2005, p. 31). Because of insufficient or poor-quality documentation, source code and comments are the preferred artefacts studied by developers owing to their relevancy (LaToza *et al.*, 2006, p. 499); one study found that developers spent approximately four times as long studying source code as they did reading any corresponding documentation (Fjeldstad and Hamlen, 1979, p. 20).

These figures indicate that software maintenance is costly, that a major portion of software maintenance effort is spent on program comprehension, and that much program comprehension effort involves reading and tracing source code because other potentially useful sources of documentation are insufficient or irrelevant. Seeking a new solution to improve the way developers deal with documentation is worthwhile, as this could reduce time spent on program comprehension and thus reduce the total resources spent on software maintenance.

3.2 Evidence from the survey of practitioners

Part A of the questionnaire asked participants about their experiences and difficulties with software maintenance. The full questionnaire is reproduced in Appendix E.

A total of 38 legitimate responses were received. Care must be taken when drawing conclusions from such a small sample (discussed further in Chapter 9).

The survey contains multiple-choice and open-ended free-text questions, numbered Q01 to Q78. The majority of questions solicit opinions and responses using a seven-point Likert scale (see Table 2).

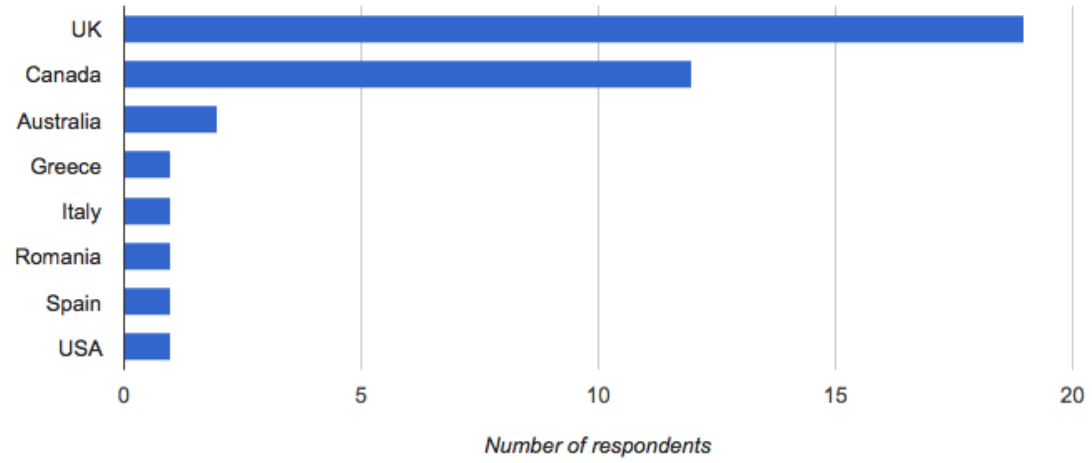
Table 2: Seven-point Likert scale

1	2	3	4	5	6	7
Strongly disagree	Disagree	Slightly disagree	Neutral	Slightly agree	Agree	Strongly agree

3.2.1 Characteristics of respondents

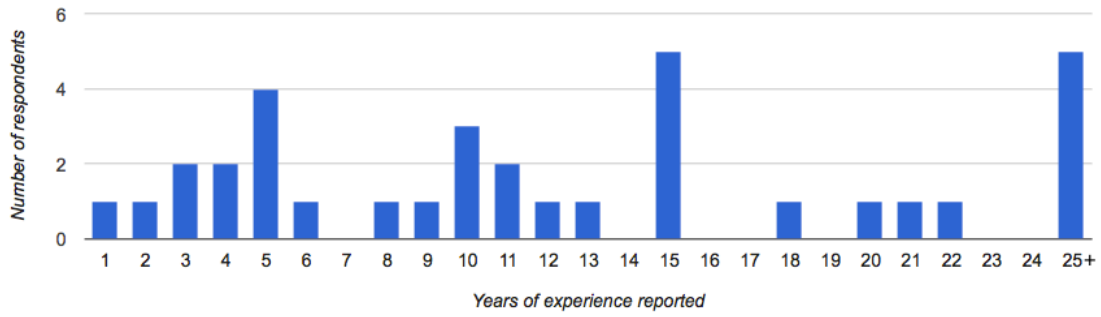
Figure 1 shows the geographic distribution of respondents.

Figure 1: Geographic distribution of survey respondents (data obtained from reverse IP lookup)



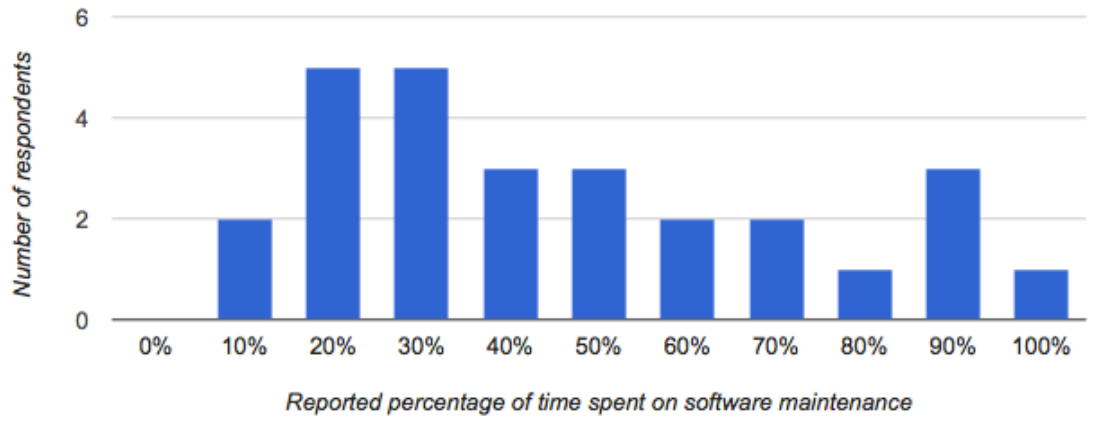
On average, respondents report having 13.2 years of software development experience (question Q64); see Figure 2.

Figure 2: Years of software development experience reported by survey respondents



All respondents reported some software maintenance activity as part of their job duties (question Q63). On average, respondents spend 46% of their work hours on maintenance development activities such as reading or modifying existing code; see Figure 3.

Figure 3: Reported percentage of developers' time spent on software maintenance



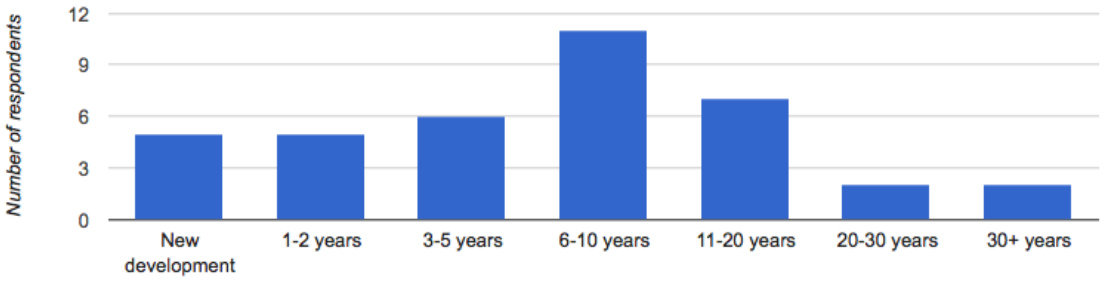
Note that other surveys have attempted to measure the total costs expended by organisations on software maintenance rather than the time spent by individual developers. Yip *et al.* (1994), for instance, found that maintenance consumes 66% of the total life cycle cost.

3.2.2 Survey results and interpretation

Project characteristics

13.2% of respondents report working on new development projects; the remainder work on existing systems of varying ages, as shown in Figure 4 (question Q01).

Figure 4: Age of systems that respondents primarily work on



51.3% indicate that the systems they work on use modern technologies and techniques, while 27.0% disagree (Q06). 48.6% describe their systems as object-oriented (Q07). 60.5% report working on "large" systems (Q02), and 71.1% consider the application domain to be complex and specialized (Q04).

36.8% report working in projects using formal, structured approaches with strict processes (Q11), while 47.3% report using agile approaches (Q12).

Perceived quality of systems and source code

42.2% consider their systems to have a well-designed architecture (Q09), but 72.9% feel the architecture has decayed over time (Q10). 62.8% report maintaining code written by developers who have long left the organisation (Q56).

40.5% consider the source code in their projects to be of good quality (though no respondents "strongly agree" with that statement); 37.8% disagree (Q08). 47.2% are dissatisfied with the defect rate in their projects (Q61), and 70.4% report that quality issues have led to deadline or budget overruns in their projects (Q62).

Use of documentation in projects

31.6% report that formal documentation plays a major role in their projects, while 55.2% disagree with that assessment (Q13). However, somewhat contradictory is the fact that 55.6% report regularly referring to requirements specifications (Q14) and 59.5% report referring to functional specifications (Q16).

Table 3 lists documentation artefacts that participants report using most frequently.

Table 3: Documentation artefacts most frequently used by survey respondents

Documentation artefact type	Reading	Writing/ updating	Combined score (sum of percentages)
Comments in code	78.4%	91.7%	170.1
Bug/defect reports	83.8%	83.8%	167.6
Test cases and test data	54.0%	75.7%	129.7
Informal documentation such as wiki pages	62.2%	64.9%	127.1
Functional specifications	59.5%	41.7%	101.2
Requirements specifications	55.6%	36.1%	91.7
Architectural design documentation	37.8%	52.8%	90.6

That source code and comments rate so highly as a useful form of documentation confirms the findings of surveys by de Souza *et al.* (2005, p. 72) and Sousa and Moreira (1998, p. 269). It is interesting to note that 91.7% of respondents claim to write or update comments, but only 78.4% report reading comments.

Table 4 lists the least frequently used documentation artefacts.

Table 4: Documentation artefacts least frequently used by survey respondents

Documentation artefact type	Reading	Writing/ updating	Combined score (sum of percentages)
User story cards	13.5%	19.4%	32.9
Data dictionaries	24.3%	19.4%	43.7
UML diagrams	19.4%	30.6%	50.0

It should be noted that an observational study found that developers actually referred to documents even *less* frequently than they reported in questionnaires (Lethbridge *et al.,* 2003, p. 38). Also, the current survey unfortunately neglected to ask about communication with other developers; LaToza *et al.* (2005, p. 495) identify face-to-face and e-mail communication as primary information sources when questions arise.

38.9% report regularly using a code-level documentation system such as Javadoc or Doxygen (Q38).

72.2% agree that comments appear "frequently" in their systems' source code (Q39), while 27.8% report the opposite. 64.0% report that comments tend to accurately match the corresponding source code (Q40). 44.5% feel that the comments are often out-of-date (Q41). 69.4% find that the comments are not written consistently across the source code (Q43). 47.2% expressed dissatisfaction with the quality of comments (Q44), but another 38.9% found the general quality of comments to be high.

When asked whether "the existing comments in the source code [are] very helpful in understanding what the code does and how it does it", 52.7% agreed and 27.9% disagreed (Q42).

Personal opinions on comments

As Table 5 shows, most respondents disagreed with general statements that were dismissive of the use of comments in source code, though a substantial minority consistently agreed with the statements.

Table 5: Respondents generally disagreed with statements critical of commenting

Question no.	Negatively-formulated prompt	Mean response (1 = strongly disagree; 4 = neutral; 7 = strongly agree)	Percentage of "disagree" responses (1, 2, or 3)	Percentage of "agree" responses (5, 6, or 7)
Q45	"Program comments are a form of heavy documentation which violate agile principles."	2.69	72.2%	19.5%
Q46	"If code is written properly, it is self-documenting and doesn't need any comments."	3.22	63.9%	27.8%
Q47	"Documenting or commenting code is a waste of time because the documentation and code will drift out of sync as the code is changed."	2.39	80.5%	14.0%
Q49	"I find that comments get in my way."	2.64	66.6%	19.5%

Positive statements about comments tended to garner strong agreement, as shown in Table 6.

Table 6: Positively-phrased statements about comments tended to garner strong agreement

Question no.	Positively-formulated prompt	Mean response (1 = strongly disagree; 4 = neutral; 7 = strongly agree)	Percentage of "disagree" responses (1, 2, or 3)	Percentage of "agree" responses (5, 6, or 7)
Q50	"Having better comments and documentation in the existing code would make my job easier."	5.36	11.2%	75.0%
Q53	"I find comments at the top of classes or files are useful."	5.17	17.1%	77.1%
Q54	"I find comments within methods or functions useful."	5.37	11.4%	82.8%
Q55	"I find comments within methods or functions useful."	5.06	17.2%	74.3%

The most useful types of comments are those describing methods/functions, followed by comments describing classes or files, and followed lastly by "in-line" comments within methods (Q53-Q55).

Self-evaluation of commenting habits

68.5% of respondents considered themselves diligent about writing comments (Q51), and 48.5% considered themselves more diligent than their peers or colleagues in consistently documenting their code (Q52), suggesting that some respondents are somewhat dissatisfied with the documentation habits of team members. 40.0% say they would like to document their code better but are constrained by deadline pressure; 42.9% disagree with this statement (Q48)[5].

[5] This is a poorly worded question: it presumes that deadline pressure is the only reason preventing developers from documenting their code better.

Reported difficulties in program comprehension and maintenance

75.1% of respondents reported sometimes having difficulty understanding what the code does or how it works (Q57), and 77.8% reported difficulties in gaining a "big picture" understanding of the system from reading source code (Q58). These figures match the results of another survey (LaToza *et al.*, 2006, p. 499) which found that 82% agreed that "it takes a lot of effort to understand why the code is implemented the way it is".

61.1% agreed with the statement "I spend more time reading and debugging code than I feel I should have to", while 27.8% disagreed (Q60).

3.2.3 Hypothesis testing

To gain more insight into developers' preferences and habits, hypothesis testing was conducted to determine whether evidence could be found to support several suspected relationships. This analysis is presented in Table 7. For this analysis, indices – aggregates of scores from multiple related questions (Weisberg *et al.,* 1996, p. 210) – were constructed, and these are explained in Appendix I, together with the statistical testing procedure.

For these hypothesis tests, a significance level (*alpha*) of 0.10 is used. In the social sciences, 0.05 is generally accepted, and 0.10 is tolerable in cases of limited responses (Weisberg *et al.*, 1996, p. 186). This level indicates acceptance of a ten-percent chance that a true null hypothesis will be incorrectly rejected (a *Type I* error).

Table 7: Hypotheses about respondents' preferences and practices relating to commenting

No.	Hypothesis	Represented in the data by (see Appendix I)	Evidence (see Appendix I)	Interpretation
H1	*Those respondents who generally express support for commenting and documentation will be more likely to report writing and updating comments*	Association between index IND01 and question Q51	Pearson chi-square test: $p = 0.0839 < 0.10$ (OK) Fisher's exact test: $p = 2.5 \times 10^{-16} < 0.10$ (OK) Kendall's Tau-b: $p = 0.0025 < 0.10$ (OK) → Tau-b = 0.40 Spearman's correlation coefficient: $p = 0.0014 < 0.10$ (OK) → r = 0.50	A statistically significant association of 0.40 (using Kendall's Tau-b) or 0.50 (using Spearman's correlation coefficient) exists.
H2	*Those with more reported experience will be more likely to express support for commenting and documentation*	Association between question Q64 and index IND01	Pearson chi-square test: $p = 0.44 > 0.10$ (NOK) Fisher's exact test: $p = 4.7 \times 10^{-26} < 0.10$ (OK) Kendall's Tau-b: $p = 0.79 > 0.10$ (NOK) Spearman's correlation coefficient: $p = 0.74 > 0.10$ (NOK)	The evidence does not support this hypothesis.
H3	*Those who express higher job frustration will be more likely to express support for commenting and documentation*	Association between index IND03 and index IND01	Pearson chi-square test: $p = 0.46 > 0.10$ (NOK) Fisher's exact test: $p = 4.1 \times 10^{-28} < 0.10$ (OK) Kendall's Tau-b: $p = 0.27 > 0.10$ (NOK) Spearman's correlation coefficient: $p = 0.30 > 0.10$ (NOK)	The evidence does not support this hypothesis.

3.2.4 Concluding interpretation

The survey reached a fairly wide spectrum of respondents in terms of experience levels, geographical location, and focus on maintenance versus new development. Most respondents reported some difficulties with program comprehension and maintenance in their current projects; for example, 75.1% reported sometimes having difficulty understanding existing code. The average responses to some of the

questions suggest a mild level of frustration with the tasks and problems involved with software maintenance activities.

In general, respondents find comments valuable: comments were found to be the most frequently used type of software documentation in respondents' systems, despite the fact that many respondents described the comments as often being out-of-date, inconsistent, or of poor quality. 68.5% of participants consider themselves "diligent" in writing comments, and 75.0% indicated that having better comments would make their jobs easier. At the same time, there appears to be a fairly consistent group of about 15 to 20 percent of the respondents who find commenting of little value.

Developers who experience difficulties and frustration, and who are diligent about writing comments, would likely be willing to consider an approach or tool that would help them to write and maintain better software documentation. Efforts to seek some new form of solution can thus be justified on the basis that (1) there is a problem, and (2) there exists a community of people who would likely be receptive to some solution that would help resolve that problem.

Having argued that a solution can be justified, we have addressed Part 1 of the research question. In the next chapter, we will determine what particular category of solution is most suitable for investigation.

4 Making the case for a specific category of solution

We have argued that explicitly recording design intention and rationale information during development would ease later program comprehension and theoretically reduce the time and expense required for software maintenance.

The general idea of capturing intention and rationale information needs to be concretised and systematised into some structure (e.g., a formal approach, a tool, a language, a product, or some combination of these or others) to be considered a "solution". In order to formulate requirements for an "ideal" solution, we need to narrow the scope to a specific category of solution.

If we consider efforts to improve the efficiency of software maintenance in general, we might consider two broad classes of solutions:

1. *General managerial policies* intended to influence *macroscopic* variables such as system size, quality, staff workload, and morale. Examples include managing demands for enhancements (Lientz and Swanson, 1980, p. 9); employing skilled, experienced developers in the maintenance effort; instituting peer-review processes (Fagan, 1976), etc.

2. *Targeted solutions* intended to address a specific problem at the *microscopic* level (e.g., at the level of source code).

General managerial policies do have an impact on costs, but the effect comes about from restricting coarse factors in order to reduce the general workload rather than leveraging any particular insight into deeper causative factors. For example, keeping the size of the system small reduces maintenance costs because, all other things being equal, smaller systems are less complex and thus easier to maintain. But once the features or change requests have been reduced to the minimum, there are no further opportunities for improvement along these lines. To achieve further reductions in maintenance effort, we must explore solutions specifically designed on

the basis of insight into the fundamental causative issues of software maintenance problems and informed by an understanding of software structure and processes[6].

Considering "targeted" solutions addressing specific issues, we may again find two categories:

1. *Approaches*, i.e., primarily people-driven processes
2. *Tools*, i.e., primarily technology-based solutions

Effective people-driven processes can have a definite impact on software maintenance costs. The effectiveness depends greatly upon the skills of the developers carrying out the process, however, and processes are often rarely followed unless enforced by a tool.

A tool which supports developers in their work offers more potential for greater impact; we might be able to fundamentally change the way developers do their work. (Of course, a tool is best used together with an effective process.)

Tools relevant to software maintenance tasks are classifiable into two further groups:

1. Tools and techniques that aid the reading and interpretation of source code by supporting the visualisation, navigation, and reverse-engineering of existing programs. Examples of such tools include Rigi (Müller *et al.*, 1994), CARE (Linos *et al.*, 1994), and Imagix 4D (Imagix Corporation, 2010). Let us call these *interpretative* approaches.

2. Tools and techniques that aid in the structuring of artefacts and the recording of information (intentions and rationale) during construction and maintenance phases in such a way as to reduce the need to reverse-engineer the system later. Let us call these *constructive* approaches.

Naturally, the two aspects are linked – that which is written is meant to be read – and a solution may involve both constructive and interpretative aspects. Most current

[6] Of course, a combination of both general organisational policies and some more specific solution will give the best results.

approaches, however, deal with only one side of the equation; visualisation tools for program comprehension fit into the interpretative category, while Literate Programming tools (Knuth, 1984) fit into the constructive category.

For systems that have already been built, only interpretative approaches can be practically applied, though the findings of reverse-engineering activities could be recorded for future use using constructive approaches.

For new systems being built now, and which will need to be maintained in the future, judicious use of effective constructive approaches could make systems more understandable and maintainable, reducing the need for interpretative-type solutions later, and potentially leading to long-term cost savings.

A “constructive” approach thus appears to be the best way to attack the fundamental problem, so this dissertation will focus on identifying or designing a constructive solution. In the next chapter, we will formulate requirements for a solution of this type.

5 Formulating requirements for a solution

This chapter surveys the literature on program comprehension and evaluates past attempts at solutions. Throughout the discussion, requirements will be collected for an "ideal" solution of the constructive type defined in the previous chapter. Requirements are presented in tables as they are identified. At the end of the chapter, the requirements will be summarised and categorised, and it will be determined whether any past attempts at solutions meet the requirements sufficiently.

5.1 Exploring program comprehension

Program comprehension refers to the task of actively reading software artefacts (primarily source code) in order to form an understanding of the system as a whole, or to identify the locations in the system that must be modified in order to correct a defect or implement a change request (Corbi, 1989, p. 300). Many developers distrust external documentation (van Vliet, 2008, p. 474) as it often no longer matches the implementation; 68 percent of developers feel that "documentation is always outdated" (Forward and Lethbridge, 2002, p. 29).

Table 8: Requirement R1

Requirement No.	Description
R1	The solution shall ensure that documentation elements can be tightly linked to, or embedded within, the source code (as developers tend to prefer and trust the source code over external documentation as an information source).

Program comprehension involves *idea processing*: developers "[move] from a chaotic collection of unrelated ideas to an integrated, orderly interpretation of the ideas and their interconnections" (Halasz *et al.,* 1987, p. 45). Developers may employ a *systematic* approach to gain a global understanding of the system, or an *as-needed* approach, investigating only parts of the system related to the task at hand (Storey *et al.*, 1997, p. 19).

Developers need to understand data and data structures, algorithms, and control flow (Pressman, 2010, p. 773; Pfleeger, 1998, p. 267), and in object-oriented systems, class hierarchies and interactions. Developers depend on *beacons* in the code – identifier names of variables, methods, classes, modules, etc., as well as idioms, patterns, and comments – for clues in forming an accurate mental model (Brooks, 1982, p. 128; Storey *et al.,* 1997, p. 18). The clarity and descriptiveness of identifier names has a particularly significant impact on the readability and understandability of programs (Butler *et al.*, 2010).

Several cognitive models of program understanding have been proposed: the top-down, the bottom-up, and opportunistic models.

5.1.1 Top-down model

Top-down models (Brooks, 1982 and 1983) suggest that developers first attempt to grasp high-level structural aspects of a program, and then systematically "[work] towards understanding the low-level details such as data types, control and data flows and algorithmic patterns in a top-down fashion" (Grubb and Takang, 2003, p. 111). Beacons provide clues for formulating, confirming, and refining hypotheses (*ibid*., Brooks, 1982, p. 128; Storey *et al.,* 1997, p. 18).

Upon reaching a "complete" understanding of a program, a developer will have formed in his or her mind "a hierarchical structure with the primary hypothesis as the top, subsidiary hypotheses below, and each segment of program bound to a subsidiary hypothesis, with no unbound parts of the program" (Brooks, 1982, p. 128).

Table 9: Requirement R2

Requirement No.	Description
R2	A developer attempting to understand an existing program using a top-down approach should be able to use the solution to record his/her understanding as a hierarchical (or otherwise interlinked) structure of textual descriptions of program elements.

5.1.2 Bottom-up model

Bottom-up models propose that developers identify groupings of statements in code, and "chunk" them together into higher-level abstractions, repeatedly aggregating them until a satisfactory understanding of the system is reached (Storey *et al.,* 1997, p. 18; Grubb and Takang, 2003, p. 113). The psychological process of "chunking" was first described by Miller (1956).

Table 10: Requirement R3

Requirement No.	Description
R3	The solution shall allow a developer to associate a description with a section of code, giving an abstract interpretation of that section of code for use in a written representation of a bottom-up reconstruction of the program's structure.

5.1.3 Opportunistic model

Opportunistic models suggest that developers exploit both bottom-up and top-down strategies, making use of cues as they become available (Letovsky, 1986, p. 69; Grubb and Takang, 2003, p. 115).

5.2 General software engineering advances that have improved maintenance

Maintenance has been improved by the adoption of general advances in software development techniques such as structured programming, object-oriented programming, design patterns, and frameworks.

Table 11: Requirement R4

Requirement No.	Description
R4	The solution shall be suitable for use with object-oriented systems.

5.3 Evaluating past "constructive" solutions

This section examines existing "constructive" approaches for documenting software systems. We differentiate between *internal documentation* schemes involving comments written directly within source code, and *external documentation* schemes involving documents separate from source code.

5.3.1 Approaches focusing on internal documentation

Comments in program code

Comments – remarks written within source code files – are a facility provided in all high-level programming languages dating from FORTRAN (IBM Corporation, 1956, p. 8) and even Zuse's 1945 *Plankalkül* (Bauer, 1972, p. 681).

Comments can be used to record intentions and rationale. McConnell (2004) argues that the purpose of comments is exactly that: "Good comments don't repeat the code or explain it. They clarify its intent. Comments should explain, at a higher level of abstraction than the code, what you're trying to do" (p. 638).

Storing explanatory text in comments within the source code, as opposed to in external documentation, increases its *visibility*: it makes it likelier that it will be found and read, because it is in close proximity to the corresponding code. Rüping (2003, p. 126) writes: "Documentation of the code… is best done through source code comments." However, comments alone are typically insufficient for capturing and explaining higher-level material such as requirements, overviews, and architectural designs (*ibid.,* p. 127).

Simonyi (2005) argues that "programming languages … were not designed with the express purpose of retaining the intentions... In fact, the best means most languages offer for preserving intentions is the trivial 'comment' facility with its well-known problems." This is a key insight, and raises the question: might developers give more consideration to recording intentions and rationale if a programming language construct other than the traditional comment were available? Some improved form of

commenting construct could potentially even provide opportunities for *enforcing* the use of comments, and templating features could help ensure consistency.

Table 12: Requirements R5 to R7

Requirement No.	Description
R5	The solution shall include a mechanism to enforce the presence of comments or other forms of internal documentation.
R6	The solution shall allow some form of reusable templates to be defined for comments or other forms of internal documentation, so that similar comments share a recognisable form.
R7	The design of the solution shall attempt to increase the importance and visibility of comments or other forms of internal documentation through the provision of some process, mechanism, artefact, or construct.

Explicit documentation of instances of design patterns in code

Design patterns (Gamma *et al.*, 1995) have become a major technique for structuring object-oriented systems. While instances of patterns (also called *pattern applications*) can be identified in UML class diagrams using a dashed ellipse and connecting line notation (Schauer and Keller, 1998), there still remains no generally accepted way to explicitly document instances of patterns within code.

Comments *can* be used to document pattern instances, but this is rarely done. Prechelt *et al.* (2002) found that explicit documentation of patterns can aid comprehensibility; without explicit documentation of pattern instances, readers must piece out what pattern is being used and which classes and objects play which roles. Crucially, developers may remain unaware that a particular pattern is in use if they are unfamiliar with it.

Prechelt *et al.* (2002) suggest using "Pattern Comment Lines" (PCL) to document pattern instances by filling out a template in comments attached to each participating class or object. Torchiano (2002) has exploited Javadoc's "taglet" functionality to systematically record roles in pattern instances using Javadoc comments. Unfortunately, neither scheme has become widespread, and neither scheme offers any means of enforcement.

Table 13: Requirement R8

Requirement No.	Description
R8	The solution shall provide a means by which pattern instances can be explicitly documented.

Java annotations

Annotations, introduced with Java 5, are a form of user-definable metadata that can "decorate" classes, methods, variable declarations, and parameters (Sun Microsystems, 2004). Annotations primarily serve to allow tools to extract metadata about program elements (*ibid.*), but annotations can also be used as documentation for human readers; they can mark program elements as having particular properties[7].

Because annotations can be defined with parameters, they could be used as a structured way of documenting program elements, ensuring that certain fields are provided. There is no way to enforce the use of annotations, however.

5.3.2 Approaches focusing on external documentation

Object-oriented documentation

Sametinger (1994) identifies similarities between documentation and source code and recommends applying object-oriented concepts to documentation. Documenting object-oriented systems with object-oriented documentation tools and techniques allows specialized documentation sets, suitable for different audiences, to be generated from a single source, using a scheme modelled after the visibility modifiers (e.g., *public*, *private*, *protected*) of object-oriented languages.

[7] For example, Goetz *et al.* (2006) suggest applying annotation `@ThreadSafe` to thread-safe classes and methods.

Table 14: Requirement R9

Requirement No.	Description
R9	The design of the solution shall allow comments or other documentation elements to be structured in an object-oriented fashion and to use object-oriented features such as inheritance where reasonable.

Diagrammatic notation for object-oriented programming

Diagrams such as flowcharts, statecharts, and call graphs are useful aids for modelling and communicating the designs of new or existing systems. Static and dynamic aspects of modern object-oriented systems are usually modelled and communicated using UML (Booch *et al.,* 2005).

As mainstream programming languages use plain-text source code files, UML diagrams cannot be embedded directly into source code comments. UML diagrams are thus a form of external documentation, and frequently, diagrams are not synchronised to match changes to the source code. Model-driven CASE tools incorporate *round-trip engineering*, whereby changes in model diagrams are reflected in the source code and vice versa (*ibid.*, p. 10).

Table 15: Requirements R10 and R11

Requirement No.	Description
R10	The solution shall allow the embedding of diagrams in documentation; if references to external diagrams must be used, the solution should check the "relational integrity" of references so that "dead links" do not arise.
R11	The solution shall either aid in keeping code and diagrams or models in synchronisation, or shall provide alerts when changes to the code are made that may necessitate the updating of diagrams or models.

Intent specifications

Intent specifications (Leveson, 2000) are a form of structured external documentation for specifying safety-critical systems. Intent specification documents follow a specific format, based on an underlying three-dimensional model of *intent, decomposition,* and *levels of refinement. Means-end hierarchies* present design rationale at different levels of abstraction and provide traceability to goals and requirements.

Table 16: Requirement R12

Requirement No.	Description
R12	The solution shall allow intention and rationale information to be represented at different levels of abstraction.

5.3.3 Approaches blending internal and external documentation

Literate programming

One of the earliest attempts to encourage and support the explicit documentation of design intentions in programs is Knuth's *Literate Programming* approach (1984). A literate program consists of a prose document explaining the structure and flow of the program and the reasoning behind it. Code fragments are then embedded within this prose document. A *tangle* tool extracts the code for compilation, while a *weave* tool generates a typeset document suitable for printing.

Blocks of code can be summarised with a descriptive textual label using angle bracket notation (e.g., "<Initialize the data structures>") and such blocks can be reused elsewhere in the program by referencing the angle bracket notation (*ibid.*). The label essentially summarises the intention behind that block of code. Programs can thus be broken down into a nested, hierarchical tree of intentions, mirroring the typical top-down strategy of decomposing problems.

Though the literate programming approach sounds promising, it appears rather unmanageable for very large systems maintained by multiple developers. The

original WEB literate programming system (Knuth, 1984) and its successors – Ryman (1993), Knuth and Levy (1994), Ramsey (1994), and Morales-Germán (1994) – have not gained any widespread adoption in industry.

Table 17: Requirements R13 to R15

Requirement No.	Description
R13	The solution shall foster a literate style of programming, encouraging explanatory text to be closely linked with source code, ideally within the same document.
R14	The solution shall allow code blocks to be associated with intention descriptions (i.e., a textual summary of what the code block is supposed to do), and this shall also apply to code blocks at any level of nesting.
R15	The solution must be practical for industrial-scale software projects involving multiple developers.

Javadoc and Checkstyle

Javadoc (Sun Microsystems, 1997) is a documentation system allied with the Java language. The *javadoc* tool extracts comments written in a particular syntax from source code and generates an interlinked set of HTML pages suitable for use as an API reference.

Pieterse *et al.* (2004) remark that Javadoc's widespread adoption shows that "the resistance of programmers to put enough emphasis on the documentation aspect of programming is no longer as severe as it was experienced when [Literate Programming] was first introduced" (p. 6). Forward and Lethbridge's survey (2002) found that 51% of respondents found Javadoc and similar tools as "useful" for creating, editing, and browsing internal documentation. Javadoc's success suggests that developers' programming habits *can* be changed by tools that bring concrete benefits (hypertext API references) and which are highly visible (all of the Java APIs are documented using Javadoc).

Javadoc is primarily intended for generating API documentation; the generated documentation shows only "public" classes, methods, and variables. Javadoc comments written for private methods and fields are suppressed in the generated hypertext documentation.

Javadoc comments are entirely voluntary; programs will compile in the complete absence of Javadoc comments. Tools such as Checkstyle (Checkstyle, n.d.) can be used to enforce that Javadoc comments are present, but programmers can effectively bypass the enforcement by entering a "." for the comment text. While no tool can parse natural language text to check comments for correctness, it would be desirable to be able to at least prevent obvious enforcement bypass attempts.

Table 18: Requirements R16 to R18

Requirement No.	Description
R16	The solution shall be capable of generating hypertext documentation similar to Javadoc.
R17	The solution shall support the documentation of both "public" (exposed API) and "private" (internal implementation) aspects of a system.
R18	The solution shall attempt to enforce that comments or internal documentation contain "reasonably sufficient" contents, to the extent possible by current technology.

Elucidative programming

Elucidative programming (Nørmark, 2000) resembles Literate Programming, but the documentation text is stored in artefacts separate from code, and bidirectional links between documentation and code are established. A viewer tool displays code and matching documentation side-by-side, reducing the need to repeatedly switch between source code and external documents.

This holds promise if, instead of using a separate viewer tool, the Integrated Development Environment (IDE) supported the seamless editing and viewing of code and documentation together, with graphical interlinking. As long as the documentation is stored separately from source code, however, there is always the risk of simply neglecting the documentation (e.g., by turning off the documentation window), and if the code is later migrated to another tool, likely only the source code would be transferred.

Table 19: Requirement R19

Requirement No.	Description
R19	The solution shall allow comments or other forms of documentation to be stored either 1. within source code files, or 2. externally, with a robust interlinking system.

5.3.4 Radically new programming systems

Intentional Programming

Intentional Programming (IP) is a programming paradigm with innovations that promise to aid program comprehension (Simonyi, 1995; Simonyi *et al.,* 1998; Simonyi *et al.*, 2008). IP separates structure from presentation by storing source code in databases rather than text files, allowing the IDE to "project" programs into various notations and syntaxes. A *domain workbench* tool allows the construction of domain models and domain-specific languages, theoretically enabling domain concepts to be manipulated at higher levels of abstraction than traditional source code.

"Intentions" in IP appear to be abstract and general program structuring constructs; the literature is vague but IP's intention constructs do not appear to include a textual description of the programmer's intention. Early demonstrations of IP (Microsoft Research, n.d.) are intriguing, but as a proprietary technology under development, information is sparse and no programming environments are yet available for evaluation.

Intent-First Design

Perry and Grisham (2006), asserting that the "core problem" of software development is "how to capture, express, and utilize intent" (*ibid.*), elucidate a vision for an IDE for constructing "rationale/intent models" which "represent the architect's intent in transforming requirements into system architectures" (*ibid.*). Unfortunately, the paper is very abstract and reveals little concrete detail of the proposed IDE or models.

Perry and Grisham introduce the "Intent-First Design" approach, which, somewhat

analogously to Test-Driven Development, encourages developers to record their intentions (in a process called *rationale reification*) before constructing software artefacts (*ibid.*).

Table 20: Requirement R20

Requirement No.	Description
R20	The solution shall encourage developers to follow a process of recording their design intentions before writing code.

5.3.5 Documentation enforcement systems

srcDoc

Shearer and Collard (2007) present srcDoc, a tool that can enforce certain constraints between Java source code elements and corresponding Javadoc comments, as a means of enforcing certain types of design decisions and policies. This is one of the few systems demonstrating a means of enforcing Javadoc comments.

5.4 Exploring "interpretative" tools

Although our focus is on "constructive" tools, we will briefly examine "interpretative" tools for reverse-engineering existing systems, as an ideal solution should allow developers to efficiently browse and navigate source code artefacts and improve and update existing documentation with any findings of their reverse-engineering activities. Storey *et al.* (1997) identify common features of such tools; features relevant to our discussion include:

- *Reducing the effect of delocalised plans*

  Plans are "program fragments that represent stereotypic action sequences in programming, e.g., a running total loop plan, an item search loop plan" (Soloway, 1986). *Delocalised plans* are plans that are fragmented across multiple software artefacts (Soloway *et al.*, 1988). "Without tool assistance, reading code belonging to a delocalised plan can be cumbersome as it may involve frequent switching between files which will quickly lead to a feeling of disorientation" (Storey *et al.,* 1997, p. 21).

Table 21: Requirement R21

Requirement No.	Description
R21	The solution shall aid in the documentation of delocalised plans.

- *Providing abstraction mechanisms*

  "Facilities should be available to allow the maintainer to create their own abstractions and label and document them to reflect their meaning. Abstraction can be supported by selecting lower level objects and aggregating them into higher level abstractions" (Storey *et al.*, 1997, p. 23).

Table 22: Requirement R22

Requirement No.	Description
R22	The solution shall allow maintainers to document their own abstractions separately from any structures or abstractions already used in the program.

- *Providing directional navigation*

  Modern IDEs like Eclipse (Eclipse Foundation, n.d.) allow rapid navigation between code elements using hypertext-style linking. Such support should also extend to any documentation embedded within or linked to the source code.

Table 23: Requirement R23

Requirement No.	Description
R23	The solution shall provide hypertext navigation between and within source code and documentation artefacts in the IDE.

5.5 Requirements derived from the survey

In the free-form questions of the survey, one respondent mentioned the importance of test-driven development; another suggested that if design information is being recorded, it should be possible to validate this design before beginning to write code.

Table 24: Requirements R24 and R25

Requirement No.	Description
R24	The solution shall be compatible with a test-driven development approach and shall permit the documentation of automated tests.
R25	The solution shall allow intention information to be validated (e.g., by another designer or developer) before code is written.

5.6 Additional requirements

This section addresses several requirements not easily categorised into any previous sections.

Improper change control procedures leading to out-of-date documentation

Ryman (1992, p. 134) defines *design entropy* as "a conceptual measure of the discrepancy between the design specification of a software system and its code". The design entropy of a system increases if proper change control procedures are not followed, i.e., the design documentation is not updated after an urgent "quick-fix" change to the code.

Table 25: Requirement R26

Requirement No.	Description
R26	The solution shall encourage the updating of design documentation when code is changed, or, if possible, structure the process so that the design documentation must be updated first.

Heterogeneous technology landscapes

Most enterprise systems involve multiple programming languages and technologies.

Table 26: Requirement R27

Requirement No.	Description
R27	The solution shall ideally support projects consisting of artefacts written in multiple languages.

Limitations of rational design processes

Parnas and Clements (1986) argue that completely rational design processes, such as the idealised "waterfall model" in which systems are completely specified before they are built, are inherently unrealistic for large, complex systems: "We believe that no system has ever been developed in that way, and probably none ever will" (*ibid.*). However, it is beneficial to maintainers if "we fake the process by producing the documents that would have produced if we had done things the ideal way. We attempt to produce the documents in the order that we have described… We do not show the way things actually happened; we show the way we wish they had happened and the way things are" (*ibid.*). Design documents are a critical deliverable of the project, and should be produced for use by future maintainers, even if the system was actually built without those documents.

Table 27: Requirement R28

Requirement No.	Description
R28	The solution shall encourage the recording of designs and design decisions before code is written (the ideal case), but shall also allow design information to be added to as-yet undocumented code.

5.7 Summarisation and categorisation of requirements

Table 29 summarises the 28 requirements collected in this chapter, grouped into categories. The table shows the degree to which some of the more promising approaches meet the requirements. Table 28 provides a legend of the codes used in Table 29.

Table 28: Legend of degree-of-fit codes used in Table 29

Code	Degree of fit
Y	*Yes*, solution meets requirement
P	*Partially* meets requirement
C	Solution *could* be used to meet requirements, but would require unusual discipline as it is not traditionally used in this manner
N	*No*, does not meet requirement
?	*Unknown* whether requirement can be met (due to lack of information about the solution)

Table 29: Summary and categorisation of requirements with degree of fit for potential solutions

Requirements			Existing approaches				
Category	**No.**	**Description**	Traditional comments	Pattern Comment Lines	Literate programming (WEB)	Javadoc + Checkstyle	Intentional Programming
Relationship to documentation other software artefacts	R1	The solution shall ensure that documentation elements can be tightly linked to, or embedded within, the source code (as developers tend to prefer and trust the source code over external documentation as an information source).	Y	Y	Y	Y	Y
	R3	The solution shall allow a developer to associate a description with a section of code, giving an abstract interpretation of that section of code for use in a written representation of a bottom-up reconstruction of the program's structure.	Y	Y	Y	Y	Y
Form and structure of documentation	R2	A developer attempting to understand an existing program using a top-down approach should be able to use the solution to record his/her understanding as a hierarchical (or otherwise interlinked) structure of textual descriptions of program elements.	C	C	P	C	?
	R6	The solution shall allow some form of reusable templates to be defined for comments or other forms of internal documentation, so that similar comments share a recognisable form.	N	N	N	N	N
	R9	The design of the solution shall allow comments or other documentation elements to be structured in an object-oriented fashion and to use object-oriented features such as inheritance where reasonable.	N	N	N	N	N
	R10	The solution shall allow the embedding of diagrams in documentation; if references to external diagrams must be used, the solution should check the "relational integrity" of references so that "dead links" do not arise.	C	C	C	C	Y
	R12	The solution shall allow intention and rationale information to be represented at different levels of abstraction.	C	C	Y	C	?
	R13	The solution shall foster a literate style of programming, encouraging explanatory text to be closely linked with source code, ideally within the same document.	C	C	Y	P	?
	R14	The solution shall allow code blocks to be associated with intention descriptions (i.e., a textual summary of what the code block is supposed to do), and this shall also apply to code blocks at any level of nesting.	C	C	Y	C	Y
	R17	The solution shall support the documentation of both "public" (exposed API) and "private" (internal implementation) aspects of a system.	C	C	C	Y	?
	R19	The solution shall allow comments or other forms of documentation to be stored either 1.	Y	Y	Y	Y	Y

Requirements			Existing approaches				
Category	**No.**	**Description**	Traditional comments	Pattern Comment Lines	Literate programming (WEB)	Javadoc + Checkstyle	Intentional Programming
		within source code files, or 2. externally, with a robust interlinking system.					
Applicability domain of solution	R4	The solution shall be suitable for use with object-oriented systems.	Y	Y	Y	Y	Y
	R15	The solution must be practical for industrial-scale software projects involving multiple developers.	Y	Y	N	Y	Y
	R27	The solution shall ideally support projects consisting of artefacts written in multiple languages.	Y	C	?	N	?
What can be documented	R8	The solution shall provide a means by which pattern instances can be explicitly documented.	C	Y	C	C	?
	R21	The solution shall aid in the documentation of delocalised plans.	C	P	Y	C	?
	R22	The solution shall allow maintainers to document their own abstractions separately from any structures or abstractions already used in the program.	C	C	C	C	?
	R24	The solution shall be compatible with a test-driven development approach and shall permit the documentation of automated tests.	Y	Y	?	Y	Y
Enforcement	R5	The solution shall include a mechanism to enforce the presence of comments or other forms of internal documentation.	N	N	N	P	N
	R18	The solution shall attempt to enforce that comments or internal documentation contain “reasonably sufficient” contents, to the extent possible by current technology.	N	N	N	N	N
Process	R20	The solution shall encourage developers to follow a process of recording their design intentions before writing code.	N	N	Y	N	?
	R28	The solution shall encourage the recording of designs and design decisions before code is written (the ideal case), but shall also allow design information to be added to as-yet undocumented code.	Y	Y	Y	Y	Y
	R25	The solution shall allow intention information to be validated (e.g., by another designer or developer) before code is written.	C	C	C	C	C
Changing mindset	R7	The design of the solution shall attempt to increase the importance and visibility of comments or other forms of internal documentation through the provision of some process, mechanism, artefact, or construct.	N	Y	Y	Y	?
Maintaining quality of documentation	R11	The solution shall either aid in keeping code and diagrams or models in synchronisation, or shall provide alerts when changes to the code are made that may necessitate the updating of diagrams or models.	N	N	N	N	?
	R26	The solution shall encourage the updating of	N	N	N	N	N

Requirements			Existing approaches				
Category	**No.**	**Description**	Traditional comments	Pattern Comment Lines	Literate programming (WEB)	Javadoc + Checkstyle	Intentional Programming
		design documentation when code is changed, or, if possible, structure the process so that the design documentation must be updated first.					
Aids to reading documentation	R16	The solution shall be capable of generating hypertext documentation similar to Javadoc.	N	P	Y	Y	?
	R23	The solution shall provide hypertext navigation between and within source code and documentation artefacts in the IDE.	N	N	N	N	N

This collection of requirements answers Part 2 of the research question.

Table 29 shows that none of the past attempts at solutions fulfil any substantial number of requirements. Thus it is worthwhile attempting to design a new solution that will meet as many of the requirements as possible, and the next chapter will address the methods involved in the task of designing and evaluating a new solution.

6 Research methods for designing and evaluating a solution

Having generated requirements for an "ideal" solution, and having determined that no past attempts at solutions are entirely suitable, a design for a solution that meets as many requirements as possible shall be devised. The solution should then be critically evaluated. This chapter explains the methods necessary to conduct this work.

6.1 Conceptualising, designing and elucidating a proposed solution

On the basis of the identified requirements, an idea for a solution will be generated, which will be developed into a detailed design. Design is a creative activity not easily formalised into a strict procedure, but involves inspiration (generating a spark of an idea) followed by iterative brainstorming, exploration, experimentation, problem solving and refinement while conceptualising, specifying, and constructing the product (Aspelund, 2010).

The idea and solution must be clearly communicated. The author's proposed solution will be outlined in Chapter 7 and elaborated in detail in the appendices.

The author's proposed solution is based on the idea of adding extensions to an existing programming language for the purpose of recording intention and rationale. The language will be described informally through a tutorial-style explanation. A more formal specification of the language will take the form of a grammar written for the ANTLR compiler construction tool (Parr, n.d.), employing the Extended Backus-Naur Form, combined with descriptions of contextual constraint rules and the behaviour of a tool to process the language. (The language extensions are purely documentation elements with no influence on program execution, so there are no "semantics" to define in the sense of run-time behaviour.)

Fundamental programming language design and implementation concepts are given by Watt and Brown (2000); guidelines for effective language design and

specification are given by Schünemann (2001) and Bjork (2009). Gosling *et al.* (2005) serves as an example of a well-written specification.

6.2 Implementing a prototype of the designed solution

To investigate the solution's feasibility, a limited prototype of a processing tool for the language will be constructed. Chapter 7 will explain the levels of support possible for the language; a complete IDE is the ideal, but constructing such a tool is unrealistic within the project time limits. As a compromise, a precompiler that translates programs written in the language into plain Java source code files will be constructed instead. This is a still a major task and scope limitations are essential. The scope is defined in Appendix C.

To speed the implementation, the compiler construction tools JavaCC (JavaCC, n.d.) and ANTLR (Parr, n.d.) were evaluated. ANTLR was chosen due to superior documentation (Parr, 2007) and because it has an existing Java grammar (Parr, 2008) that is relatively understandable and modifiable.

6.3 Constructing a sample project using the language

A small application project will be constructed using the new language, to demonstrate the approach, and to act as a test suite for the precompiler.

6.4 Evaluating the proposed solution

Evaluating the solution must involve determining how well it meets the stated requirements, and addressing Part 3 of the research question, which asks whether the solution is *feasible, practical,* and *effective*.

Some parts of the evaluation can be done by the author, but external evaluation is also needed.

6.4.1 Evaluation by the author

A critical analysis of the proposed solution will be conducted by the author using methods enumerated in Table 30.

Table 30: Methods to be used in the author's own evaluation of the proposed solution

Method no.	Description	Refer to section(s)
1	Evaluation of the proposed solution against the list of identified requirements	8.1.1
2	Enumeration and discussion of advantages and disadvantages	8.1.2, 8.1.5
3	Analysis of the literature for statements for or against the general approach	8.1.3
4	Comparison of the proposed solution with alternative solutions	8.1.4
5	Evaluation of the language design against language evaluation criteria such as those given by Bjork (2009)	8.1.6
6	Reflection on the experience of building the sample application using the language extensions	8.1.7
7	Consideration of ethical issues related to the proposed solution	8.1.5 and Appendix H

Additionally, the prototype precompiler and sample application serve as proofs of concept of the technical *feasibility* of the language and approach and *practicality* for use, at least in small projects. Note that purely technical evaluations of the precompiler such as performance measurements would give no insight into the solution's practicality or effectiveness and so will be omitted.

6.4.2 Evaluation involving outside evaluators

To address whether the solution is truly *effective*, it would be ideal to determine whether the solution can be shown to reduce long-term maintenance costs, or at least whether the solution can improve program comprehension.

A long-term case study or "quasi-experiment" involving multiple software projects could be run, comparing projects using the proposed solution against projects that do not. This is impractical as an effective study would have to span several years, and

while useful qualitative data might be obtained, no definite conclusions could be drawn due to the vast number of uncontrolled variables and differences between projects and teams.

A field experiment observing developers working with the solution could provide insight into the solution's practicality and impact on program comprehension. Usability engineering methods such as think-aloud protocols (Holzinger, 2005) could reveal work practices and thought patterns; examples of such studies are Wallace *et al.* (2002) and Chuntao (2009). Unfortunately, this exercise requires a production-quality implementation of the solution and requires time for developers to learn and become competent with the language and system. Finding volunteers would be difficult due to the time commitment needed. Extrapolating conclusions about long-term benefits from a relatively short investigation may not be reasonable.

Less time commitment from volunteers would be required for a comprehension quiz experiment like those of Prechelt *et al.* (2002) and Nurvitahdi *et al.* (2003). Participants would be randomly assigned to different groups. One group would receive a program listing without comments, another would receive the same listing but documented with a "standard" use of comments, and another would receive the listing documented using the proposed language extensions. All groups would be given the same comprehension quiz. If the latter group were to receive (statistically-significant) higher scores than the other groups, it might be considered as limited evidence that the solution improves comprehension. The experiment design would be difficult methodologically: it is impossible to properly control all variables in such experiments (Parnas, 2003, p. 4), and given enough time, all participants could study the listings and earn near-perfect scores. Variability of participant skill levels implies a need for hundreds of subjects (Brooks, 1980), and comprehension quiz scores, especially based on small programs, will not necessarily correspond to reduced maintenance costs (*ibid.*).

Interviews and questionnaire surveys of practicing software developers are feasible "compromise" approaches. The solution would be explained to participants, who would then provide feedback on its perceived effectiveness. Unfortunately, these approaches have a critical limitation: by merely soliciting opinions, they provide

little “direct” empirical evidence towards the underlying question of whether the solution truly improves comprehension or reduces long-term costs. Root and Draper (1983) warn that “asking users about the value of some proposed change without giving them experience of it is an essentially useless guide to their satisfaction with it in practice” (p. 86). Nevertheless, these techniques can still provide useful qualitative and quantitative data for analysis, and are feasible within the time limits.

As the questionnaire was chosen in Chapter 2, it was decided to add a Part B to the questionnaire to solicit evaluation of the solution[8].

Chapters 7 and 8 employ the research methods chosen in this chapter to design and evaluate a new solution.

[8] Please refer back to Chapter 2 for references to the literature on survey design and analysis.

7 Proposing, designing, and building a solution

This chapter introduces a solution intended to meet most of the requirements identified in Chapter 5.

7.1 The structure of the proposed solution

The proposed solution is a scheme consisting of a *process* and a *tool/technology*.

The process is an approach called *Design Intention Driven Programming* (DIDP), which *encourages* developers to record their intentions before writing code, and attempts to *enforce* this by means of the tool/technology.

The tool/technology is a set of language extensions that can be added to existing programming languages. The extensions are language constructs specially designed for recording intention information. A specialised compiler enforces the use of these constructs in program code by flagging absences as compiler errors. Adding the extensions to Java has produced a language tentatively called *Java with Intentions* (JWI).

A brief introduction to the scheme, referred to as DIDP/JWI for brevity, is presented in the following section[9]. Appendix B gives a more exhaustive description, detailing all features and addressing typical questions and objections.

[9] Due to strict word count limits, only a summary of the scheme can be explained in the body of the dissertation.

7.2 Introducing *Design Intention Driven Programming* and *Java with Intentions*

In the Design Intention Driven Programming (DIDP) approach, when developers implement a requirement or a feature, they first record their *design intentions* for the software component before they write the code for that component. This simply involves writing a brief description of what the component is planned to do, and how it will do it. The description may also include *rationale* – a justification of why one particular solution was chosen over alternative solutions.

Design intentions are written using specialised programming language constructs called *intention comments*. Like traditional comments, intention comments contain textual descriptions. But unlike a traditional comment, intention comments have a richer structure modelled after object-oriented classes. Intention comments:

- are named;
- can contain fields for storing text, allowing further structuring of explanations;
- can contain fields referencing other intention comments, goals, or requirements, which allows rich graph structures to be formed;
- can contain fields referencing program entities such as classes;
- can be declared *abstract*, allowing for templating;
- support inheritance using the `extends` keyword, allowing structured re-use.

A developer will first write an intention comment, and then will write the code to fulfil the intention. Upon completion, the classes and methods involved are linked to the intention comment.

In *Java with Intentions* (JWI), intention comments are declared using the `intention` keyword. Figure 5 illustrates the syntax of an intention comment and shows a class linking to it.

Figure 5: An intention comment and a class linking to it

```
intention QuizStateIntention {

    description {
        Class QuizState maintains the state of the current quiz, i.e., the current
        session in which all of the flashcards in a flashcard set will be presented
        once. This class is responsible for keeping track of the current flashcard,
        the user's score, and the application's mode (whether a game is in progress
        or is stopped).
    }

    requirementsreference[] satisficesRequirements = {
        EachFlashcardPresentedOncePerQuizSession,
        KeepScore
    };

    intentionreference playsRoleInPattern = FlashcardTrainerMVCPatternInstance;

}

public class QuizState implementsintention FlashcardTrainerMVCPatternInstance,
        QuizStateIntention {
    ...
}
```

Enforcing documentation

In JWI projects, all classes must refer to an intention comment; a compiler error will be raised if a class does not have an appropriate intention describing it. This is an attempt to enforce the *presence* of documentation.

Enforcing that descriptions are *correct* or *sufficient* is impossible as current technology cannot interpret and reason about natural-language prose. However, an imperfect and limited form of sufficiency enforcement can be conducted as follows. To prevent "empty" or "gibberish" comments, the compiler will calculate a complexity metric such as the McCabe Cyclomatic Complexity index (van Vliet, 2008, p. 342) for each section of code, and then using another algorithm, it will calculate a metric that quantifies the "information content" of the descriptive text associated with that code. If the ratio of the information content metric to the code complexity metric falls below a threshold, the description is deemed insufficient to describe the code, and a compiler error is generated.

Documenting design pattern instances

Abstract intention comments can describe general design patterns, as illustrated in Figure 6.

Figure 6: An abstract intention comment defining a general design pattern

```
abstract intention ModelViewControllerPattern {

    description {
        The Model-View-Controller pattern structures the user interface
        code into separate components. This separation of concerns helps
        improve understandability and modifiability.

        The model consists of a representation of the application's data.
        The model notifies listeners (typically, one or more view
        components) when the data changes.

        The view component presents the data to the user in the form of
        UI components. Multiple views based on the same model may exist.

        The controller acts upon input from the user and updates the
        model and/or interacts with the view.
    }

    classreference[] modelClasses;
    classreference[] viewClasses;
    classreference controllerClass;

}
```

Concrete instances of the design pattern can then be documented by extending the abstract intention comment to form a concrete intention comment for the instance; fields are filled in with references to the components playing the roles in the pattern, as shown in Figure 7.

Figure 7: A concrete intention extending the abstract pattern definition to specify a particular instance of the pattern

```
intention FlashcardTrainerMVCPatternInstance extends ModelViewControllerPattern {

    description {
        The flashcard trainer user interface is constructed according to the
        Model-View-Controller pattern.
    }

    modelClasses = { QuizState, FlashcardSet };
    viewClasses = { QuizFrame };
    controllerClass = QuizController;

}
```

Components taking part in the pattern can also link themselves to the intention comment (see Figure 8), so that new developers stumbling upon one of the components can follow the links to locate the other components of the pattern (and developers unaware of the pattern are guided by an explicit description). Prechelt *et al.* (2002) found that explicit documentation of pattern instances aids comprehensibility.

Figure 8: A component of the pattern instance links itself to the intention comment for the pattern instance

```
class QuizController implementsintention FlashcardTrainerMVCPatternInstance,
        QuizControllerIntention {
     ...
}
```

Goals and requirements

Intentions typically follow from requirements, and requirements typically follow from goals. Goals and requirements can be recorded in JWI projects using keywords `goal` and `requirement`. Figure 9 illustrates requirements recorded using JWI.

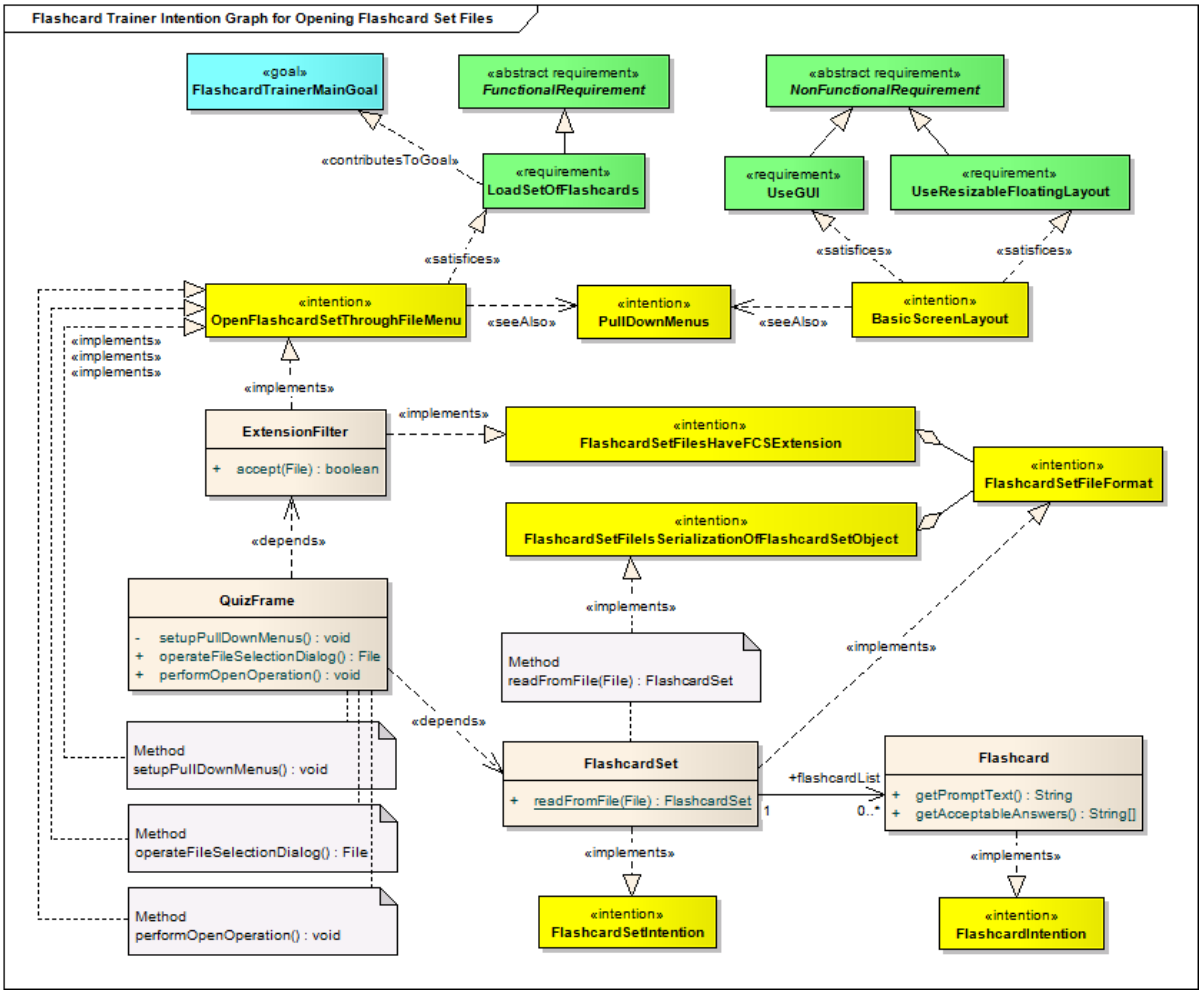
Figure 9: Requirements represented in code using JWI

```
abstract requirement FunctionalRequirement {
    description {
        Functional requirement.
    }
}

abstract requirement NonFunctionalRequirement {
    description {
        Non-functional requirement.
    }
}

requirement ShuffleFlashcards extends FunctionalRequirement {
    description {
        The application shall randomize (shuffle) the flashcards in the
        flashcard set so that the user is not presented with the same sequence
        of cards each time.
    }
}

requirement UseGUI extends NonFunctionalRequirement {
    description {
        The application shall use a graphical user interface.
    }
}
```

Intention graphs

The set of interlinked goals, requirements, and intentions for a project forms an *intention graph*. Representing entire intention graphs graphically is often impractical due to their size, but subsets can be useful. Figure 10 illustrates an intention graph subset using a modified UML class diagram notation.

Figure 10: Modified UML class diagram illustrating an intention graph subset (description texts for goals, intentions, and requirements omitted)



IDE navigability

An ideal DIDP/JWI implementation features an IDE (e.g., Eclipse) supporting syntax checking for the language extensions and offering rapid navigation between intention declarations and code by presenting references as hyperlinks. Fast navigation is important, as Ko *et al.* (2005) found that 35% of programming time involves navigating between dependencies and 46% involves inspecting code irrelevant to the task at hand.

This brief overview does not cover all features of the proposed scheme, nor does it address common questions and objections. Please refer to Appendix B for a more comprehensive discussion.

7.3 The prototype precompiler and the *Java with Intentions* language specification

Appendix C summarises the scope of the prototype precompiler implementation. Appendix D explains how to access the prototype and gives a walkthrough of its use.

The *Java with Intentions* language features are described in detail in Appendix B, and this serves as an informal language specification.

The ANTLR grammar developed for the prototype precompiler serves as a formal specification of the language syntax. Instructions for accessing the ANTLR grammar are provided in Appendix D. In addition to the syntax, there are contextual constraint rules that the precompiler must enforce; these rules (including scoping rules) are explained in section B.3 of Appendix B. Additionally, section C.1 of Appendix C defines the processing expected by the precompiler.

7.4 The sample application project

A sample application, *Vocabulary Trainer*, was constructed to demonstrate the use of JWI in an actual Java project. To inspect the source code, please refer to the instructions for accessing the application in Appendix D.

7.5 Summary

This chapter briefly introduced key points of the proposed solution. We will now evaluate the solution in the next chapter.

8 Evaluating the proposed solution

The previous chapter presented the *Design Intention Driven Programming* and the *Java with Intentions* scheme. We shall now critically evaluate this proposed solution.

8.1 Evaluation by the author

8.1.1 Degree of fit to requirements

Table 32 reiterates the requirements generated in Chapter 5, and for each requirement, the degree of fit achieved by the proposed solution is evaluated. Differentiation is made between the "ideal" solution with full IDE integration, and the "compromise" solution offering only precompiler support.

Table 31 lists the codes used in Table 32.

Table 31: Legend of degree-of-fit codes used in Table 32

Code	Degree of fit
Y	*Yes*, solution meets requirement
P	*Partially* meets requirement
N	*No*, does not meet requirement

Table 32: Summary and categorisation of requirements with degree of fit for potential solutions

Requirements			Proposed approach	
Category	**No.**	**Description**	**DIDP/JWI (ideal IDE-based solution)**	**DIDP/JWI (precompiler solution)**
Relationship to documentation other software artefacts	R1	The solution shall ensure that documentation elements can be tightly linked to, or embedded within, the source code (as developers tend to prefer and trust the source code over external documentation as an information source).	Y	Y
	R3	The solution shall allow a developer to associate a description with a section of code, giving an abstract interpretation of that section of code for use in a written representation of a bottom-up reconstruction of the program's structure.	Y	Y
Form and structure of documentation	R2	A developer attempting to understand an existing program using a top-down approach should be able to use the solution to record his/her understanding as a hierarchical (or otherwise interlinked) structure of textual descriptions of program elements.	Y	Y
	R6	The solution shall allow some form of reusable templates to be defined for comments or other forms of internal documentation, so that similar comments share a recognisable form.	Y	Y
	R9	The design of the solution shall allow comments or other documentation elements to be structured in an object-oriented fashion and to use object-oriented features such as inheritance where reasonable.	Y	Y
	R10	The solution shall allow the embedding of diagrams in documentation; if references to external diagrams must be used, the solution should check the "relational integrity" of references so that "dead links" do not arise.	P	P
	R12	The solution shall allow intention and rationale information to be represented at different levels of abstraction.	Y	Y
	R13	The solution shall foster a literate style of programming, encouraging explanatory text to be closely linked with source code, ideally within the same document.	Y	Y
	R14	The solution shall allow code blocks to be associated with intention descriptions (i.e., a textual summary of what the code block is supposed to do), and this shall also apply to code blocks at any level of nesting.	Y	Y
	R17	The solution shall support the documentation of both "public" (exposed API) and "private" (internal implementation) aspects of a system.	Y	Y
	R19	The solution shall allow comments or other forms of documentation to be stored either 1. within source code files, or 2. externally, with a robust interlinking system.	Y	Y
Applicability domain of solution	R4	The solution shall be suitable for use with object-oriented systems.	Y	Y
	R15	The solution must be practical for industrial-scale software projects involving multiple developers.	Y	Y
	R27	The solution shall ideally support projects consisting of	N	N

Requirements			Proposed approach	
Category	**No.**	**Description**	**DIDP/JWI (ideal IDE-based solution)**	**DIDP/JWI (precompiler solution)**
		artefacts written in multiple languages.		
What can be documented	R8	The solution shall provide a means by which pattern instances can be explicitly documented.	Y	Y
	R21	The solution shall aid in the documentation of delocalised plans.	Y	Y
	R22	The solution shall allow maintainers to document their own abstractions separately from any structures or abstractions already used in the program.	Y	Y
	R24	The solution shall be compatible with a test-driven development approach and shall permit the documentation of automated tests.	Y	Y
Enforcement	R5	The solution shall include a mechanism to enforce the presence of comments or other forms of internal documentation.	Y	Y
	R18	The solution shall attempt to enforce that comments or internal documentation contain "reasonably sufficient" contents, to the extent possible by current technology.	Y	Y
Process	R20	The solution shall encourage developers to follow a process of recording their design intentions before writing code.	Y	Y
	R28	The solution shall encourage the recording of designs and design decisions before code is written (the ideal case), but shall also allow design information to be added to as-yet undocumented code.	Y	Y
	R25	The solution shall allow intention information to be validated (e.g., by another designer or developer) before code is written.	Y	Y
Changing mindset	R7	The design of the solution shall attempt to increase the importance and visibility of comments or other forms of internal documentation through the provision of some process, mechanism, artefact, or construct.	Y	Y
Maintaining quality of documentation	R11	The solution shall either aid in keeping code and diagrams or models in synchronisation, or shall provide alerts when changes to the code are made that may necessitate the updating of diagrams or models.	N	N
	R26	The solution shall encourage the updating of design documentation when code is changed, or, if possible, structure the process so that the design documentation must be updated first.	N	N
Aids to reading documentation	R16	The solution shall be capable of generating hypertext documentation similar to Javadoc.	P	P
	R23	The solution shall provide hypertext navigation between and within source code and documentation artefacts in the IDE.	Y	N

The ideal variant meets 23 of the 28 requirements, with another two deemed to be "partially" met. The limited precompiler variant meets 22 requirements, and two "partially".

It should be noted that the requirements chosen in Chapter 5 are likely biased to predispose the DIDP/JWI solution. Chapter 9 addresses this concern in more depth.

8.1.2 Potential benefits of the scheme

The DIDP/JWI approach offers several benefits:

- Constructing graphs of intentions, goals, and requirements allows the designs of software systems to be structured in convenient ways (often mirroring architectural structures of the software itself) and at varying levels of abstraction;
- The application of inheritance to intention comments provides a templating mechanism that is convenient for explicitly documenting design pattern instances, something traditionally rarely done in practice;
- Similar to Javadoc, integrating documentation into source code increases the chances that it will be seen and updated by future developers;
- The mechanisms for enforcing the use of intention comments, while imperfect, provide some support for organisational policies mandating source code documentation.

More fundamentally, if maintenance developers have access to well-structured documentation describing design intentions and rationale, they will be able to more quickly and more effectively understand the structure and behaviour of program source code, theoretically reducing long-term software maintenance costs.

8.1.3 Attitudes in the literature towards the general approach

Documenting intentions before writing code is not a new idea; it is the basis of the familiar pseudocode method (McConnell, 2004, p. 176).

McConnell explains that intent descriptions are extremely valuable for later readers (*ibid.*, pp. 642-650). Raskin (2005) writes, “[t]he essential concept of writing the

documentation first, creating the methods in natural language, and describing the thinking behind them is a key to high-quality commercial programming… The use of internal documentation is one of the most-overlooked ways of improving software and speeding implementation" (p. 62).

However, such statements do not provide *empirical* evidence of the effectiveness of the approach. Lethbridge *et al.* (2003) remark that little evidence other than "opinion and conjecture" could be found that forcing developers to write verbose documentation and keep it meticulously up-to-date is effective (p. 38).

8.1.4 Comparison with alternative approaches

The merger of documentation and code in the DIDP/JWI approach is patterned after Literate Programming (Knuth, 1984), but the intention comment construct permits a somewhat richer and more structured modelling of the reasoning and design behind a program than hierarchically-organised free text.

DIDP is similar in nature to Perry and Grisham's (2006) Intent-First Design approach; both urge developers to record their intentions before constructing software code. Intent-First Design envisions an IDE for manipulating "intent models", which are not concretely described but which might presumably resemble intention graphs. Intent-First Design does not mention using programming language constructs to record intention descriptions.

As a means of documenting delocalised plans, intention graphs share properties with Robillard and Murphy's (2007) *concern graphs*, which allow scattered source code elements to be linked to *concerns*. Concerns are not represented as language constructs as in JWI; instead, developers link code fragments to concerns using wizards and views in the FEAT IDE (*ibid.*, p. 5). Concerns do not appear to be intended to be documentation artefacts; they take names but it is unclear whether further descriptions can be attached. Concerns are arrangeable hierarchically; richer interlinking and inheritance mechanisms do not appear to exist.

8.1.5 Criticisms of the approach

Table 33 summarises the most significant criticisms of the proposed approach.

Table 33: Major criticisms of the Design Intention Driven Programming approach

Category	Issue	Description
Limitations	*No guarantee of quality of comments*	The JWI system enforces the *presence* of intention comments, but cannot guarantee their *correctness*. It makes a crude attempt to enforce the *sufficiency* of descriptions for any given section of code by comparing information content metrics for the comment text with a complexity metric for the code section, and ensuring that the ratio meets a threshold. As it is infeasible to algorithmically interpret human-language text and determine its "correctness" or general quality, developers could still write nonsense descriptions that satisfy the "sufficiency" check.
	No guarantee that comments are current with regard to changes to the code	Developers frequently modify code without updating the corresponding comments, and the JWI system is not immune to this problem. A mechanism involving the source control system could be added to detect changes in the source code and flag intention comments potentially affected by those changes. This feature has not included in the proposed solution, but a similar capability is demonstrated by Robillard and Murphy's FEAT IDE (2007).
Cost	*Cost of training and familiarisation*	Developers need to learn the syntax and become proficient with the approach. Initial learning costs may be borne by the organisation (formal training) or by the developers (learning on one's own time). A period of lowered productivity and higher error rates can be expected as developers gain proficiency.
	Cost of slowing down work	While a future benefit (easier comprehension) is hoped for, in the present, it will take more time to write or modify code if strict documentation requirements are now enforced.
Ambiguity of benefits	*Cost/benefit ratio incalculable*	Measuring performance and productivity of software-related work is already notoriously difficult, and without any past track record to refer to, it is impossible to reliably estimate imagined future benefits, making the evaluation of the investment difficult.

Category	Issue	Description
	No guarantee of future benefits	There is no guarantee that this approach will in fact lead to long-term cost savings. Even if intention comments do in fact make a system easier to understand, it is possible that any potential time savings are cancelled out by the time spent maintaining the intention comments. Even if a case study were to show benefits in one project or organisation, that anecdotal evidence could not be extrapolated to suggest that similar benefits would be enjoyed by all projects or organisations.
	Benefits not measurable	Even if the approach does actually lead to cost savings in a project, it would be impossible to attribute the improved performance solely to the use of the approach, as many other factors (e.g., the skill and motivation of the team members) may have contributed to or caused the performance improvement effect.
Political	*Frustration of developers*	Documenting design intentions slows down the pace of development and may go against the "natural" way of writing programs that developers have grown accustomed to.
	Resistance of developers to learn yet another technology or syntax	Developers already spend enormous amounts of time keeping up-to-date with new technologies; adding another language to this burden only compounds the problem.
	Discord in teams	Project team members inevitably disagree on process issues. Being forced to write comments would likely stir up arguments between proponents and opponents of the scheme.
	Management expectations of productivity	Management may introduce this system with the expectation of improved productivity. Any productivity increases are more likely to occur in the long term, and productivity may suffer in the near term.
	Means of management control	Management, perceiving quality issues, may dictate the use of the system without first achieving buy-in from team members.
Ethical	*Burden on current developers for the benefit of future developers*	It can argued that it is an ethical violation to force current developers to sacrifice their productivity (and perceived job performance) to write documentation that will not immediately benefit them, but which will improve the productivity and job performance of other developers in the distant future. This violates the principle of *mutual benefit* and breeds ill will in project teams (Beck, 2005, p.

Category	Issue	Description
		14). (Ethical issues are explored in depth in Appendix H.)
	Misuses of metrics by management	If the system outputs quality metrics, or if the use of the system is accompanied by performance measurement metrics, there is the ethical concern that these metrics could be used by a manager to compare the performance of developers, and the metrics are likely not fair or reliable indicators of performance. (Ethical issues are explored in depth in Appendix H.)
Technical	*Difficulty of application to heterogeneous-technology projects*	Most software is constructed using multiple technologies (for example, a web application using HTML, SQL, and PHP and integrating with a legacy COBOL system). Intentions spanning code in multiple languages would require DIDP-compatible compilers for each language and some means of data exchange between them. The intention comment concept might be difficult to apply to declarative languages like SQL.

8.1.6 Evaluation of the Java with Intentions language design

Table 34 attempts to evaluate the design of the JWI extensions against the language evaluation criteria given by Bjork (2009)[10].

[10] This evaluation concerns only the language extensions and not the Java language on which the language extensions are based.

Table 34: Evaluation of *Java with Intentions* against language evaluation criteria (Bjork, 2009)

Category	Criteria	Evaluation and discussion
Ease of use	Well-defined syntax (lack of ambiguity)	The syntax for the language extensions is formally defined as EBNF production rules in an ANTLR grammar (see Appendix D).
	Well-defined semantics (lack of ambiguity)	The language extensions have no "semantics" in terms of program behaviour. The behaviour of the language processor (precompiler) is well-defined (see Appendix C).
	Consistency with commonly-used notation and conventions	The naming of keywords and the use of operators and symbols has been designed to follow the conventions of the Java language.
	Uniformity (similar constructs have similar meaning)	Yes, in the sense that the similar constructs `intention`, `goal`, and `requirement` have identical features and syntax.
	Orthogonality (limited number of features that can be combined)	Yes, in the sense that the Java `abstract` and `extends` keywords are applicable to intention comments and have similar semantics.
	General-purpose (suitable for any type of program/application domain)	Yes.
	Good pedagogy (easy to learn)	Difficult to judge without a study.
Software engineering	Supports development of correct programs	The language extensions do not explicitly support this other than to help developers to structure documentation, which may aid in validation efforts.
	Reliability (language makes it difficult to make careless errors)	Difficult to judge without a study.
	Support for modularity	Yes.
	Support for separate compilation	Yes.
	Provides/enables data types and data structuring	Yes, for documentation elements.
	Support for provability of correctness	No.
Performance	Lends itself to fast compilation	The precompiler's processing time is reasonable for small projects but could become an issue for extremely large projects.

8.1.7 Personal experiences constructing the sample project

Constructing the intention graph for the sample application was difficult and arduous at first, but partly this was due to doing this the first time, with the syntax still in flux. Eventually, things started to "click" and it was pleasing to see how well the scheme worked for documenting features that are implemented by means of a number of interrelated methods and variables and classes in different files (i.e., delocalised plans or cross-cutting concerns). Representing requirements and matching software structures to them also seemed to work well. Writing the documentation, however, consumed a lot of time and required sustained concentration. In real-world software projects, developers will have difficulty sparing the time to write intention comments, and a cost-benefit analysis would be necessary to justify the time and expense.

The sample application project reused code from a previous project. Converting existing code was painstaking and confirmed the author's intuition that the DIDP approach is most suitable for constructing new systems. Applying intention comments "after the fact" to existing code does force the developer to deeply engage with and understand the code, however, which helps identify refactoring opportunities.

In cases where new code was written, the author sometimes found himself writing the code first and then adding the intention comments afterwards, seemingly a "violation" of the suggested process. While this may simply be inexperience with executing the approach, it does suggest that the "write the intentions first" approach may be an unrealistic, unattainable ideal. However, as long as the documentation is written up in the end as if it had existed all along, it still has the same benefit to later readers (Parnas and Clements, 1986).

JWI is designed to co-exist with Javadoc comments; Javadoc comments are suitable for documenting a publicly-exposed API, while intention comments can better explain internal structures and rationale. However, replacing Javadoc comments for classes with intention comments would mean that the Javadoc generated documentation would be incomplete; not replacing them would lead to duplicated

documentation. This is an issue that the author had not fully considered until building the sample project was underway. Investigating a merger or "interoperability" of intention comments with Javadoc comments is recommended in a next-generation design.

Overall, the approach seems workable and useful, but not perfect, for describing the internal structures of the sample application. However, the effectiveness depends on the motivation and skill of the developer to construct a useful intention graph and write descriptive comments. Also, the sample application is quite small, and studies involving small-scale projects do not necessary scale up to larger, more complex production systems (von Mayrhauser and Vans, 1995, p. 54).

8.2 External evaluation: Results, analysis, and interpretation of Part B of the survey

Part B of the questionnaire asked participants for opinions and feedback after reading a brief article (reproduced in Appendix G) introducing the DIDP/JWI approach. This section summarises and analyses the results. Appendix E contains the questionnaire text and summary statistics, and Appendix F contains the raw survey data.

8.2.1 Survey results and interpretation

Receptivity to recording design intentions

Participants responded positively to the statement "The practice of recording design intentions before writing code is a sensible idea", with 85.7% registering agreement (question Q68). 74.3% agreed that "[i]nstances of design patterns should be documented for ease of understanding by later maintainers" (Q72).

Practicality

Asked whether "recording design intentions before writing code might be nice in theory, but is impractical for real-world projects," 34.4% agreed that it is impractical, but 51.4% believed that it is not necessarily impractical (Q69).

48.6% agreed that developers would resent being forced to write documentation, while 31.4% believed that this would not be the case (Q70). Several respondents suggested in written responses that initial resistance might be overcome if the system begins providing observable benefits to the developers (Q76, respondent 17; Q78, respondent 7).

Applicability domain

77.2% of respondents viewed the solution as being suited to formal, document-driven projects (Q74), while 40.0% viewed it as suitable for agile projects (Q73).

Perceived benefits

Some respondents see potential merit in the proposed solution: 51.4% of respondents agreed that "[s]oftware projects consistently documented in this way would be easier to understand than projects developed using traditional techniques" (Q75). But this is far from being overwhelming support: 25.8% disagree, and 22.9% are neutral. Respondents were polarised when asked if they "would be willing to give this approach a try, at least on a trial basis," with 48.6% agreeing and 37.2% disagreeing (Q66).

Coding analysis of free-text remarks

Respondents were given the opportunity to supply written remarks on the solution. A coding analysis of this qualitative data was performed by identifying unique issues raised in the remarks, counting the instances of each issue, categorising the issues, and sorting the issues within each category. Table 35 and Table 36 present issues that are critical and supportive of the proposed solution, respectively. Some problems raised in Table 35 match problems named in section 8.1.5, while others are new.

Table 35: "Critical" remarks in written survey responses evaluating the solution

Category	Issue	Count
Summary evaluations	Interesting idea but flawed	1
	Not practical for real-world projects	1
	Sceptical	1
Evaluations of applicability	Unsuitable for Java and OO languages but might help in functional languages	1
	Unsuitable for agile projects	1
Problems	Intention comments required to compile code would cause frustration	4
	Developers will resist (unless/until they see a clear benefit to themselves)	3
	Developers will bypass enforcement with bare-minimum comments or garbage/noise	3
	Forced comments add verbosity and clutter, hindering understanding	2
	Another syntax to learn is unwelcome	2
	Does not solve problem of out-of-date comments not matching code	2
	Creates more work, increases costs of development	2
	If people don't have time to write comments, how will they have time to write intentions?	2
	If people aren't disciplined enough to write decent comments, why would extra syntax help?	1
	Quality of comments not guaranteed	1
	Human peer review of intention comments still needed	1
	Mixing code and documentation problematic for releases	1
	Applying intention comments to existing code too time consuming	1
	Making comments look like code means comments will have bugs	1
	Unclear how to write intentions without knowing context of the classes	1
	No correctness checking	1
	"Language abuse" unwelcome	1
	Too much writing and duplication	1
Predictions	Would soon get switched off (projects will abandon the approach)	2
	Intention comments will not become a staple of any programming language	1
Total number of "critical" remarks or issues raised		**38**

Table 36: "Supportive" remarks in written survey responses evaluating the solution

Category	Issue	Count
Summary evaluations	Nice/good idea	4
Evaluations of applicability	Potentially useful for very complex systems	2
Benefits	Would benefit developers new to the team and junior developers	3
	Would make code easier to read and understand	2
	Would help ensure a minimum level of documentation	1
	Would improve quality	1
	Would improve maintainability	1
	Might actually encourage developers to write useful comments	1
	Useful for requirements traceability	1
	Generating documentation like Javadoc useful	1
	Use of solution would set standards and values for project	1
	Justification/rationale useful	1
	Highlighting relationships between pieces of code useful	1
	Reuse of intentions convenient (e.g., for pattern catalogues)	1
Total number of "supportive" remarks or issues raised		**21**

Table 37 lists alternatives to the proposed solution suggested by respondents.

Table 37: Alternatives to the proposed solution suggested by respondents

Category	Alternative	Count
Technological	Use Java annotations	3
	Use Javadoc and Checkstyle	1
	Use free-form comments	1
Process	Use Test-Driven Development	2
	Focus on achieving proper processes and workflow; use reviews	2
	Focus on applying structured methods, UML, OOAD	1
Team	Focus on building a good team and fostering good communication	1
Quality	Focus on good design	1
Training	Students/developers should study design patterns and APIs instead	1

8.2.2 Further analysis

To try to better understand possible reasons why respondents favour or disfavour the solution, hypothesis testing was conducted. As discussed in section 3.2.3, we use a significance level (alpha) of 0.10.

Table 38 presents several hypothesised relationships and indicates whether sufficient evidence could be found to support them. Appendix I explains the indices and statistical procedures used in these tests.

Table 38: Hypotheses about factors influencing respondents' support of the proposed solution

No.	Hypothesis	Represented in the data by (see Appendix I)	Evidence (see Appendix I)	Interpretation
H4	*Those respondents who generally express support for commenting and documentation will tend to be more likely to express support for the proposed solution*	Association between index IND01 and index IND02	Pearson chi-square test: $p = 0.0572 < 0.10$ (OK) Fisher's exact test: $p = 1.1 \times 10^{-29} < 0.10$ (OK) Kendall's Tau-b: $p = 0.0290 < 0.10$ (OK) → Tau-b = 0.29 Spearman's correlation coefficient: $p = 0.0295 < 0.10$ (OK) → r = 0.37	A very weak but statistically significant association of 0.26 (using Kendall's Tau-b) or 0.37 (using Spearman's correlation coefficient) exists.
H5	*Those with more reported experience will value the proposed solution more*	Association between question Q64 and IND02	Pearson chi-square test: $p = 0.56 > 0.10$ (NOK) Fisher's exact test: $p = 4.4 \times 10^{-29} < 0.10$ (OK) Kendall's Tau-b: $p = 0.54 > 0.10$ (NOK) Spearman's correlation coefficient: $p = 0.56 > 0.10$ (NOK)	The evidence does not support this hypothesis.
H6	*Those who report spending a larger percentage of time on maintenance will value the proposed solution more*	Association between question Q63 and IND02	Pearson chi-square test: $p = 0.48 > 0.10$ (NOK) Fisher's exact test: $p = 1.9 \times 10^{-18} < 0.10$ (OK) Kendall's Tau-b: $p = 0.32 > 0.10$ (NOK)	The evidence does not support this hypothesis.

No.	Hypothesis	Represented in the data by (see Appendix I)	Evidence (see Appendix I)	Interpretation
			Spearman's correlation coefficient: p = 0.33 > 0.10 (NOK)	
H7	*Those who express higher job frustration will value the proposed solution more*	Association between question IND03 and IND02	Pearson chi-square test: p = 0.70 (NOK) Fisher's exact test: p = 1.5×10^{-29} < 0.10 (OK) Kendall's Tau-b: p = 0.23 > 0.10 (NOK) Spearman's correlation coefficient: p = 0.21 > 0.10 (NOK)	The evidence does not support this hypothesis.

8.2.3 Interpretation

Many questions exhibit bipolar distributions, indicating that one group of respondents consistently opposes the solution and another group tends to support it. Partly this can be explained by the general observation in software organisations that there are some developers who find documentation useful and there are those who see documentation as an unnecessary burden, and the evidence for hypothesis H4 in Table 38 indicates that those who favour documentation have tended to show slightly more support for the solution. However, there may also be respondents who favour documentation but disfavour the proposed solution due to any number of flaws.

Overall, the responses to the Likert-scale questions tended to be more favourable toward the proposed solution than the author had expected, but this "support" for the solution cannot be considered overwhelming or even a majority opinion. Again, there was clearly a large group who did not find the solution acceptable. The written comments contained more statements critical of the solution than statements supporting the solution. The critical remarks were, uniformly, well-reasoned explanations of problems with the solution that are indeed reason for concern.

The generally favourable responses might be attributable to several bias factors that are discussed in Chapter 9.

8.3 Summary

While the survey found some support in favour of the proposed solution, and the sample project suggested that the approach is workable but labour-intensive, a substantial number of problems were identified which suggest that the proposed solution may be impractical for general use in software projects in industry.

In Chapter 9, we will critically examine the research methods used in this research project and investigate the validity of the evaluation performed in the present chapter.

9 Evaluating the research methods and the evaluation of the proposed solution

This chapter reflects upon and evaluates each of the research methods used. In particular, we are interested in investigating the reliability, validity, and trustworthiness of the evaluation of the solution.

9.1 Questionnaire survey

9.1.1 Evaluation of execution of method

Setting up the survey ran smoothly, but attracting participants and convincing them to take part was difficult. The number of responses was disappointing, but similar software maintenance-related surveys including Yip *et al.* (1994) and Sousa and Moreira (1998) have also reported very low participation rates. The duration of the activity (up to 30 minutes) and the bland topic (software documentation) likely discouraged many potential participants. Incentives boosted response rates, but the author could not afford to offer incentives to all groups.

E-mail invitations to large groups gave a predictably low response rate. The Google text ad was presented over 700,000 times but led to only two legitimate responses. The response rate even amongst my friends and coworkers was disappointingly low; repeated reminders were required and many who pledged to take the survey never actually did. Nevertheless, a reasonable number of participants was achieved, with a good diversity of geography, experience, and opinions, and the written responses were generally well-written and insightful.

9.1.2 Validity of survey results

Population validity

The survey sought responses from practicing software developers. Do the 38 responses accurately represent the opinions of the entire population of software developers in the world? The small sample size increases the risk of drawing inaccurate conclusions. Respondents were members of a *convenience sample* (Ruane, 2005, p. 117), i.e., people easy for me to reach, such as acquaintances, fellow students, and members of e-mail distribution lists. Approximately ten responses were from coworkers; having too many respondents from the same organisation could potentially skew the results away from those obtained from a perfect random sampling of the world's population of software developers.

Measurement validity

Measurement validity refers to how successful a research instrument is at measuring what it claims to measure.

Are the responses to the survey questions returning accurate measurements? The majority of the survey's questions involve opinions measured using a seven-point Likert scale. Fowler (1995) considers this a valid approach for quantifying "subjective states", and argues that the validity of such measurements is unquantifiable (and virtually irrelevant) as there are no "true", objective values of subjective opinions from which measurements can deviate due to bias or other factors.

The article explaining the solution to participants was intentionally kept very brief. It did not cover all aspects of the solution and did not address typical questions and objections. Respondents' opinions may have differed had they instead read Appendix B describing the solution, or had they experimented with the prototype precompiler or studied the sample application project. As was quoted earlier, "asking users about the value of some proposed change without giving them experience of it is an

essentially useless guide to their satisfaction with it in practice" (Draper, 1983, p. 86).

Bias

When interpreting the survey results and conclusions, the following sources of bias must be considered:

- Only a small percentage of those who received invitations actually participated in the survey. This is a case of *self-selection bias* (Weisberg *et al.*, 1996): only individuals interested in software documentation are likely to spend the approximately 30 minutes to answer the survey. As individuals with strong opinions in favour of documentation are most likely to participate, a disproportionately large percentage of respondents will hold such opinions, and hypothesis test H4 in Table 38 shows that those who favour documentation in general are more likely to express support for the solution. Thus the relatively high levels of support for the solution exhibited by the survey cannot be considered entirely representative of the population of software developers as a whole.

- Participants who know the author personally, or those who were offered incentives, may have intentionally or unintentionally answered questions in such a way as to give support to the proposed solution, in order to avoid offending the author or in the belief that positive responses would help the author personally.

- Questions asking about opinions on documentation may suffer from *social desirability bias* (Weisberg *et al.*, 1996, p. 86). Having been taught that it is good practice to write comments, participants may be inclined to say that they favour writing comments, when in fact they seldom do so.

Care was taken during the design of the questionnaire to avoid intentionally biasing the responses toward some preferred result. For example, an attempt was made to intersperse positively- and negatively-phrased questions to prevent temptations of

uniformly agreeing or disagreeing with all statements in a series. There is still the risk that a "leading question" early in the survey causes the respondent to take a position on some issue, which then shapes the later responses so that those responses are consistent with the original position (as the respondent does not want to appear inconsistent). Despite care taken to combat any intentional bias, bias may exist in the questionnaire and would skew the results.

Reliability

Reliability, a component of validity, assesses whether measurements are accurate and trustworthy (Sapsford, 2007, p. 15). For surveys, reliability can be tested by checking the *stability* of measures. If the same question is asked twice during the questionnaire, the responses should usually be similar if not identical; if they diverge greatly, the instability casts the reliability of the research instrument into doubt (*ibid.*).

In the first version of the questionnaire, redundancy was intentionally present in the form of question pairs (one positively-formulated and one negatively-formulated) that essentially asked the same question. Redundancy was eliminated after trial run participants complained about repetitive questions, however; this was a mistake, as reliability tests must now involve questions that are vaguely similar but not identical.

A positive association is expected between responses for questions Q08, "In general, I consider the quality of the existing code I work on to be very good" and Q44, "The general quality of comments [in the system you work on] is high"; with a confidence level (alpha) of 0.10, the association is 0.57, which is reasonable given that the questions discuss two different but related subjects (code and comments).

A *negative* association would be expected between mean responses for Q49, "I find that comments get in my way", and Q50, "Having better comments and documentation in the existing code would make my job easier", but with a confidence level (alpha) of 0.10, no statistically-significant association could be found (one would have to use an alpha of 0.33 to get a weak negative association of -0.12).

The reliability test must be judged inconclusive, and this is a failing of the survey design. The ultimate test of reliability is for another researcher to replicate the results with a similar but non-identical study.

9.2 Formulating requirements

The author had already conceived of the fundamental idea for the proposed solution before conducting the literature review. The literature review in Chapter 5 was therefore not only a familiarisation with the subject area and a search for requirements, but also a search for "prior art", i.e., seeking to determine whether the envisioned scheme had already been proposed. The presupposition of a potential solution unquestionably shaped the direction of the literature survey, and although the author attempted to impartially generate requirements for a *general* solution of the "constructive" category identified in Chapter 4, having had the conceptual idea already in mind, the requirements are undoubtedly preselected to favour the author's own proposed solution.

9.3 Conceptualising, designing, and elucidating the solution

The language design evolved as ideas were generated, as experiences were gained through constructing the sample application, and as problems were identified during the construction of the precompiler. Attempting to describe the DIDP/JWI approach in a clear and concise manner was particularly difficult and took several iterations.

9.4 Defining the language syntax and implementing the prototype

Not having previously designed a programming language formally and having little experience with compiler-construction tools like ANTLR, the author encountered challenges while designing the *Java with Intentions* syntax and implementing the precompiler. It took several iterations to discover and resolve inconsistencies and

problems in the planned syntax and approach. One lasting issue is that ANTLR's particular tokenisation strategy prevents the originally planned use of braces to surround free-text fields (such as used with the `description` keyword), as the lexer is unaware of the grammatical context in which tokens exist. (Of course, this is obvious in hindsight.) As a workaround, "double-braces" tokens ("`{{`" and "`}}`") are used for text fields in the prototype implementation and sample application, while in the text of this dissertation, single braces remain in the examples as this is still the "ideal" syntax. The problem could be avoided by writing a custom (non-ANTLR), context-aware lexer that can identify whether the `description` keyword has been processed immediately before an opening brace.

ANTLR's error messages are extremely obscure. Internet searches were often required to determine what they really meant and to find out how other users had overcome similar issues. Investigating errors consumed far more time than expected.

The originally-planned scope of the implementation of the precompiler was unfortunately not completed in the time available, but what was implemented (see the statement of scope in Appendix C) serves as an effective proof-of-concept that the language extensions can be stripped out and the resulting plain Java source code can be fed to the `javac` compiler.

9.5 Self-evaluation of the solution

Any evaluation that a designer performs on his or her own product is inherently subject to a "self-evaluation" bias, the desire to not find fault with one's own work. While this bias cannot be completely eliminated, the author has remained aware of its risk throughout the project and has strived to conduct the evaluation objectively and impartially, turning a critical and sceptical eye to the proposed solution and enumerating and frankly discussing the solution's problems and disadvantages.

The use of the survey to collect opinions from people other than the author is an important way to provide additional perspective.

9.6 General remarks on validity

Sapsford argues that, fundamentally, underlying "validity" is a question of trust in the honesty of the researcher. He writes, "All research depends ultimately on our trust that the researcher is at worst incompetent or 'short-sighted' but not positively mendacious" (2007, p. 16).

The author has attempted to conduct the survey and the evaluation impartially and without conscious bias, but this is no guarantee that biases or methodological errors have not affected the research. This dissertation has attempted to declare all potential biases, key choices, and weaknesses of the research methods. The reader is asked to note these and make allowances when interpreting and judging the results and conclusions.

9.7 Summary

Reflecting on each research method used, this chapter has identified potential biases and methodological weaknesses that may impact the research results. With these caveats in mind, the following and final chapter will formulate conclusions based on the results of the evaluation of the solution.

10 Conclusions

In this chapter, we interpret the results of the evaluation in order to answer Part 3 of the research question: having designed a solution, how *feasible*, *practical*, and *effective* is it? We summarise and conclude the dissertation with a discussion of the likelihood of the solution being adopted in real-world projects. Finally, we will examine the dissertation's contribution to knowledge, reflect on the research project, and discuss opportunities for further research.

10.1 Judging the feasibility, practicality, and effectiveness of the DIDP/JWI scheme

Feasibility

The construction of the prototype precompiler demonstrates that the language design and the required processing are technically feasible. Unfortunately, time constraints prevented the implementation of the sufficiency checking mechanism that would compare complexity and information content metrics, a key part of the approach.

The construction of the sample application (*Vocabulary Trainer*) using JWI demonstrated the general feasibility of documenting a small project with intention comments. However, it remains to be seen whether the complexity of interlinkages could become overwhelming in larger projects.

Practicality

The analysis and survey revealed a significant number of problematic issues that may render the scheme unsuitable for many projects and teams. While the author does not necessarily consider any of the problems to be severe or fatal flaws, a number of survey respondents disagree. Further studies investigating the advantages and disadvantages in long-term projects are required, and in any case, the judgement of whether the problems outweigh the benefits is largely subjective, depending heavily on the individual project situations and contexts in which the scheme would be used.

Despite the potential problems, 51.4 percent of survey respondents felt that the DIDP/JWI scheme could be practical in real-world projects, and 48.6 percent were willing to consider using it on a trial basis. 85.7 percent favour the general approach of recording design intentions before writing code. However, these figures are likely skewed upward by self-selection bias. At least 20 to 25 percent of respondents are highly sceptical of the scheme; the actual figure is likely higher in the larger population of software developers.

The resistance of developers to writing documentation, the cost (at least in the short term) of writing additional documentation, and the frustration that could emerge from documentation enforcement are three major objections. The scheme could only realistically be used in project teams having a unanimous opinion that intention documentation is worthwhile.

Effectiveness

The final decision of whether to employ the DIDP/JWI solution in a software project hinges largely on the question of whether the expected benefits outweigh the costs. The evaluation established that consistently documenting design intentions is very time-consuming. In most organisations, this expense would only be justifiable if evidence existed that intention documentation would make later program comprehension and maintenance easier and that this would lead to long-term cost savings.

Again, the underlying question that would have been most interesting to answer – whether a language-based comment enforcement scheme can really lead to long-term cost savings in large software development projects – unfortunately remains unanswered due to the infeasibility of the required research methods within the context of this research project. The cost/benefit question is a major part of evaluating the effectiveness of the proposed solution, and needs to be addressed by follow-up research.

The author, reflecting on the experience of constructing the sample application using JWI, found graphs of intention comments to be an effective way of representing

requirements, patterns, and software structures that spanned multiple files. Further research, however, is required to determine whether developers in general would find intention graphs more useful than alternative approaches during program comprehension.

10.2 On the likelihood of adoption of the scheme

Lethbridge *et al.* (2003) argue that developers are not necessarily lazy with respect to writing documentation, but rather, developers make cost-conscious value judgements, writing and updating only those forms of documentation perceived as relevant such as test cases and bug reports. They suggest that attempting to force discipline on developers tends to backfire, so instead developers should be empowered with "simple yet powerful documentation formats and tools". The DIDP/JWI approach was created out of a vision of empowering developers with a tool to help them build better software with less long-term frustration. But its simplicity is subject to debate (some comments in the questionnaire were critical of this), and, ironically, the comment sufficiency enforcement aspect of DIDP is itself a form of top-down discipline enforcement. Instead of empowering developers, this scheme may very well burden them.

There seem to be two factions in the developer community: those who like to read and write documentation, and those who do not. In general, the former group would probably find the DIDP/JWI solution a useful aid (unless they considered any of the solution's problems to be fatal flaws), and the latter group would probably vigorously reject it. Perhaps the following quote summarises it best:

> "People who like this sort of thing will find this the sort of thing they like."
>
> *– attributed to Abraham Lincoln*

It is unsatisfying to end a dissertation with such an indefinite conclusion, but the topic is fundamentally a "soft" and qualitative one; the practicality and usefulness of any potential new technology are dependent on the context in which it may be used and the personal preferences of those using it.

Let us make the following analogy. Had we presented an introduction to object-oriented programming (before it had become mainstream) and evaluated it using a survey similar to that used in this dissertation, we would likely get similar mixed results. Some would like it; some would hate it, pointing out critical problems. Object-oriented programming is useful in some situations and inappropriate in others, and the DIDP/JWI approach is probably similar – barring any fatal flaws, it could very well be useful for the right projects and teams, and yet it will be completely unsuitable for others. So although further studies are needed to overcome the limitations of the survey method and give deeper insights into the solution's potential effectiveness, those further studies won't change the fact that the solution is not a panacea suitable for use in every situation.

Realistically, even if DIDP/JWI may potentially have some merit, because its costs and benefits are uncertain, because its documentation enforcement aspect is imperfect, and because using it really involves quite a substantial amount of work which developers are generally reluctant to do, it probably has little chance of gaining widespread use in industry, and will likely remain a curiosity like the other Literate Programming systems (Knuth, 1984) surveyed in Chapter 5.

As a final practical matter, Table 39 provides a recommendation of a checklist of steps for software project teams considering adopting DIDP/JWI (assuming that a production-quality implementation is available).

Table 39: Checklist of recommended steps for project teams considering adopting DIDP/JWI

Step	Description
1	Identify whether the approach is potentially suitable for the project
2	Discuss with all team members their personal views on software documentation
3	Seek consensus amongst team members and management on whether to proceed further
4	Estimate costs and benefits
5	Have one or more developers conduct a trial of the approach and the tools using a small-scale prototype project
6	On the basis of the estimates and the results of the trial, decide whether to proceed
7	Plan the implementation strategy, budgeting sufficient time for training/learning/familiarisation and sufficient time for using the approach during the construction phase
8	Plan and implement a training programme
9	Identify a developer willing to become an expert on the approach and tools and who can serve as a mentor to other members of the team
10	Schedule regular code review sessions to help ensure that team members are using the approach correctly and efficiently
11	At regular intervals, review the process, make adjustments as necessary, and decide whether to continue using the approach

10.3 Contribution to knowledge

Part A of the survey provided confirmation that practicing software developers report difficulties with software maintenance, with many results confirming similar results reported by de Souza *et al.* (2005), Sousa and Moreira (1998), and others.

The *Design Intention Driven Programming* approach, the intention comment construct, intention graphs, and the comment enforcement scheme are novel contributions to knowledge. The approach can be seen as an extension or refinement of the Literate Programming concept (Knuth, 1984).

The evaluation, including Part B of the survey, contributed some evidence that the approach could potentially be useful in some circumstances, but has drawbacks that make its adoption by industry unlikely.

10.4 Project review

10.4.1 Addressing the research question

Table 40 revisits the multi-part research question from Chapter 1 and shows that the three parts of the question have been addressed.

Table 40: Evaluation of satisfaction of research question

Part	Research question portion	Evaluation
PART 1	What evidence can be found to justify the design of a new solution to aid the recording intention and rationale information during software development?	In Chapter 3, the evidence from the literature and from the results of Part A of the survey was deemed sufficient to justify seeking a new solution.
PART 2	What are the requirements for an "ideal" solution?	In Chapter 5, a list of 28 requirements was generated by surveying the literature and examining past attempts at solutions.
PART 3	Given the requirements for an ideal solution, can a design for a solution be developed that is feasible, practical, and effective?	In Chapter 7, a new solution was presented, and a rudimentary prototype of the solution was implemented. The present chapter (Chapter 10) has made conclusions on the feasibility, practicality, and effectiveness of the solution.

10.4.2 Reflecting on the project

Time constraints were the major difficulty in completing the project; given more time, or at least fewer work-related issues, the project likely might have taken a different shape.

More time was spent teaching myself statistical analysis techniques and learning to use the SAS software than was actually necessary. The simplest analysis techniques contributed the most value to the argument.

Acquiring participants for the survey was a challenge, and the low response rate weakens the dissertation.

Asking different questions on the survey would have possibly enabled better insights, but this is only recognisable in hindsight.

More interesting knowledge could have been generated if the long-term case study or comprehension quiz experiment methods could have been applied instead of the survey. These are suitable topics for follow-up studies, which are discussed in the following section.

10.5 Opportunities for future research

Table 41 discusses possibilities for follow-up research projects.

Table 41: Opportunities for further research

Category	No.	Project
Testing claims of effectiveness (comprehension, cost savings)	1	Investigate, using a comprehension quiz experiment such as those conducted by Prechelt *et al.* (2002) and Nurvitahdi *et al.* (2003), whether intention comments and intention graphs make software systems easier to understand
	2	Conduct a long-term comparative case study to study long-term advantages and disadvantages and to determine if evidence can be found that the DIDP/JWI approach leads to cost savings
Concretisation of design	3	Construct and compare algorithms for computing information content and code complexity metrics, and determine "acceptable" threshold ratios, perhaps by conducting a study or experiment involving practicing developers working with real-world source code
Refinements and new features	4	Design a scheme to detect changes in the source code and flag corresponding intention comments that may need updating; Robillard and Murphy (2007, p.11) discuss means for performing this type of *inconsistency management*
	5	Design a next-generation language/toolset merging intention comments and Javadoc into a single, unified scheme
	6	Design a scheme where requirements and external documents such as specifications and architectural designs can be stored in a Wiki or other repository and referenced from within the source code
Seeking synergies	7	Investigate how DIDP can be better tied in with the Test-Driven Development approach

References

American Heritage Dictionary (2009) *American Heritage Dictionary of the English Language (4e)*, Houghton-Mifflin.

Aspelund, K. (2010) *The design process*, New York: Fairchild Books.

Babar, M.A., Tang, A., Gorton, I., and Han, J. (2006) 'Industrial perspective on the usefulness of design rationale for software maintenance: A survey' in *Proceedings of the Sixth Annual Quality Software International Conference*, Beijing, Oct. 2006, pp. 201-208.

Bauer, F.L., and Wössner, H. (1972) 'The Plankalkül of Konrad Zuse: A forerunner of today's programming languages' in *Communications of the ACM*, Vol. 15, No. 7 (July 1972), pp. 678-685.

Beck, K. (2004) *Extreme programming explained: Embrace change*, Upper Saddle River, NJ: Addison-Wesley Professional (Pearson Education).

Bjork, R.G. (2009) *Language evaluation criteria* [online], a handout for course CPS 323 at Gordon College, Wenham, MA, http://www.math-cs.gordon.edu/courses/cs323/lectures-2009/LanguageEvaluationCriteria.pdf [accessed 2 August 2010].

Boehm, B. (1976) 'Software engineering' in *IEEE Transactions on Computers*, Vol. C-25, Issue 12, pp. 1226-1241.

Booch, G., Rumbaugh, J., and Jacobson, I. (2005) *The Unified Modeling Language user guide (2e)*, Upper Saddle River, NJ: Addison-Wesley Professional (Pearson Education).

Bratman, M.E. (1987) *Intention, plans, and practical reason*, Cambridge, MA: Harvard University Press.

Brooks, R. (1980) 'Studying programmer behaviour experimentally: The problems of proper methodology' in *Communications of the ACM*, Vol. 23, Number 4 (April 1980), pp. 207-213.

Brooks, R. (1982) 'A theoretical analysis of the role of documentation in the comprehension of computer programs' in *Proceedings of the 1982 Conference on Human Factors in Computing Systems*, Gaithersburg, MD, pp. 125-129.

Brooks, R. (1983) 'Towards a theory of the comprehension of computer programs', quoted in von Mayrhauser, A., and Vans, A.M. (1995) 'Program comprehension during software maintenance and evolution', *IEEE Computer*, Vol. 28, Issue 8 (August '95), pp. 44-55.

Butler, S., Wermelinger, M., Yu, Y., and Sharp, H. (2010) 'Exploring the influence of identifier names on code quality: An empirical study' in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, 15-18 March 2010, Madrid, Spain.

Checkstyle (n.d.) *Checkstyle 5.1 documentation* [online], http://checkstyle.sourceforge.net/config_javadoc.html [accessed 7 March 2010].

Chuntao, D. (2009) 'Empirical study on college students' debugging abilities in computer programming', in *Proceedings of the 1st International Conference on Information Science and Engineering (ICISE 2009),* Nanjing, pp. 3319-22.

Collins English Dictionary (2003) *Collins English Dictionary, Complete and Unabridged*, Harper-Collins, 2003.

Corbi, T. A. (1989) 'Program understanding: Challenge for the 1990s', in *IBM Systems Journal*, Vol. 28, No. 2, pp. 294-306.

De Souza, S.C.B., Anquetil, N, and de Oliveria, K.M. (2005) 'A study of the documentation essential to software maintenance', in *ACM Special Interest Group for Design of Communication: Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting and Designing for Pervasive Information*, pp. 68-75.

Devanbu, P.T., Brachman, R.J., Selfridge, P.G., and Ballard, B.W. (1990) 'LaSSIE – a knowledge-based software information system' in *Proceedings of the 12th International Conference on Software Engineering (ICSE'90)*, pp. 249-261.

Eclipse Foundation (n.d.) *Eclipse software development environment* [online], http://www.eclipse.org [accessed 27 June 2010].

Fagan, M.E. (1976) 'Design and code inspections to reduce errors in program development', in *IBM Systems Journal*, Vol. 15, No. 3, pp. 182-211.

Fjeldstad, R. K. and W. T. Hamlen (1979) 'Application program maintenance study: Report to our respondents' in *Proceedings GUIDE 48, Philadelphia, PA, Tutorial on Software Maintenance*, G. Parikh and N. Zvegintzov, eds., IEEE Computer Society.

Forward, A., and Lethbridge, T.C. (2002) 'The relevance of software documentation, tools and technologies: A survey', *Proceedings of the 2002 ACM Symposium on Document Engineering*, pp. 26-33.

Fowler, F.J. Jr. (1995) *Improving survey questions: Design and evaluation*, Thousand Oaks, CA: SAGE Publications.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) *Design patterns: elements of reusable object-oriented software*, Boston, MA: Addison-Wesley.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. (2006) *Java concurrency in practice*, Upper Saddle River, NJ: Addison-Wesley (Pearson Education), p. 7. See also http://www.javaconcurrencyinpractice.com/annotations/doc/index.html [accessed 4 May 2010].

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005) *The Java Language Specification, 3e*, Upper Saddle River, NJ: Prentice Hall.

Gray, G., and Guppy, N. (1994) *Successful surveys: Research methods and practice*, Toronto: Harcourt Brace & Company Canada.

Grubb, P. and Takang, A.A. (2003) *Software maintenance: concepts and practice (2e)*, Singapore: World Scientific Publishing.

Halasz, F.G., Moran, T.P., and Trigg, R.H. (1987) 'Notecards in a nutshell' in *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface*, Toronto, pp. 45-52.

Holzinger, A. (2005) 'Usability engineering methods for software developers' in *Communications of the ACM,* Vol. 48, Issue 1, January 2005.

IBM Corporation (1956) *The FORTRAN automatic coding system for the IBM 704 EPDM: Programmer's reference manual*, New York, NY: IBM Corporation.

Imagix Corporation (2010) *Imagix 4D (product website)*; http://www.imagix.com [accessed 30 December 2010].

Jackson, W. (1988) *Research methods: Rules for survey design and analysis*, Scarborough, Ontario: Prentice-Hall Canada.

JavaCC (n.d.) *Java Compiler Compiler (JavaCC) – The Java Parser Generator* [online], https://javacc.dev.java.net [accessed 24 October 2010].

Jones, C. (2006) *The economics of software maintenance in the twenty-first century (technical report), version 3 – February 14, 2006*, Hendersonville, NC: Software Productivity Research; http://www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf [accessed 17 May 2010].

Kajko-Mattsson, M. (2005) 'A survey of documentation practice within corrective maintenance' in *Empirical Software Engineering*, 10, 31-55, 2005, pp. 31-55.

Knuth, D.E. (1984) 'Literate programming' in *The Computer Journal (British Computer Society)*, Vol. 27, No. 2, pp. 97-111.

Knuth, D.E., and Levy, S. (1994) *The CWEB system of structured documentation*, Boston, MA: Addison-Wesley Longman.

Ko, A.J., Aung, H.H., and Myers, B.A. (2005) 'Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and preventative

maintenance tasks' in *27th International Conference on Software Engineering,* St. Louis, MO., May 2005, pp. 126-135.

LaToza, T.D., Venolia, G., and DeLine, R. (2006) 'Maintaining mental models: A study of developer work habits' in *Proceedings of the 28th International Conference on Software Engineering*, pp. 492-501.

Letovsky, S. (1986) 'Cognitive processes in program comprehension' in Empirical Studies of Programmers, pp. 58-79, New York, NY: Ablex Publishing; referenced in von Mayrhauser, A., and Vans, A.M. (1995) 'Program comprehension during software maintenance and evolution' in *IEEE Computer*, Vol. 28, Issue 8 (August '95), pp. 44-55.

Leveson, N. (2000) 'Intent specifications: An approach to building human-centered specifications' in *IEEE Transactions on Software Engineering*, Vol. 26, No. 1, January 2000, pp. 15-35.

Lientz, B.P., and Swanson E.B. (1980) *Software maintenance management*, Reading, MA: Addison-Wesley.

Linos, P., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P., and Tulula, P. (1994) 'Visualizing program dependencies: An experimental study', in *Journal of Software – Practice and Experience*, Vol. 24, Issue 4, April 1994.

M801 (2007) *M801 Research project and dissertation: Study guide*, Milton Keynes, Open University.

McConnell, S.C. (2004) *Code complete (2e)*, Redmond, WA: Microsoft Press.

Meyer, B. (1992) 'Applying "design by contract"' in *IEEE Computer*, Vol. 25, Issue 10 (Oct. 1992), pp. 40-51.

Microsoft Research (n.d.) *Intentional programming (demonstration videos)* [online]; Part 1: http://www.youtube.com/watch?v=tSnnfUj1XCQ; Part 2: http://www.youtube.com/watch?v=ZZDwB4-DPXE [accessed 4 May 2010].

Miller, G.A. (1956) 'The magical number seven, plus or minus two: Some limits on our capacity for processing information' in *Psychological Review*, Vol. 63, Issue 2, pp. 81–97.

Morales-Germán, D. (1994) 'An SGML-based programming environment for literate programming', in *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research (IBM Centre for Advanced Studies Conference)*, Toronto, pp. 47-

Müller, H.A., Wong, K., and Tilley, S.R. (1994) 'Understanding software systems using reverse engineering technology', in *Proceedings of the 62nd Congress of l'Association canadienne française pour l'avancement des sciences (ACFAS 1994)*, Montréal.

Norman, D.A. (1998) *The design of everyday things*. New York, NY: Basic Books, pp. 45-46.

Nørmark, K. (2000) 'Requirements for an elucidative programming environment' in *Proceedings of the 8th International Workshop on Program Comprehension*, pp. 119-129.

Nurvitadhi, E., Leung, W.W., and Cook, C. (2003) 'Do class comments aid Java program understanding?' in *33rd Annual ASEE/IEEE Frontiers in Education Conference, 2003*, Boulder CO, pp. 13-17.

Parnas, D.L., and Clements, P.C. (1986) 'A rational design process: how and why to fake it' in *IEEE Transactions on Software Engineering*, Vol. 12, Issue 2, pp. 251-257.

Parnas, D.L. (2003) 'The limits of empirical studies of software engineering' in *Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE'03)*, Rome, Italy, pp. 2-5.

Parr, T. (n.d.) *ANTLR: ANother Tool for Language Recognition* [online], http://antlr.org/ [accessed 24 October 2010].

Parr, T. (2007) *The definitive ANTLR reference: Building domain-specific languages*, Raleigh, NC: Pragmatic Bookshelf.

Parr, T. (2008) *A Java 1.5 grammar for ANTLR v3 derived from the spec* [online], http://www.antlr.org/grammar/1152141644268/Java.g [accessed 16 May 2010].

Perry, D.E., and Grisham, P. (2006) 'Architecture and design intent in component & COTS based systems' in *Proceedings of the Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS 2006)*, pp. 155-165.

Pfleeger, S.L. (1998) *Software engineering: Theory and practice*, Upper Saddle River, NJ: Prentice-Hall.

Pieterse, V., Kourie, D.G., Boake, A. (2004) 'A case for contemporary literate programming' in *Proceedings of the 2004 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries (SAICSIT)*, Stellenbosch, South Africa, Vol. 75, pp. 2-9.

Prechelt, L., Unger-Lamprecht, B., Phillipsen, M., and Tichy, W.F. (2002) 'Two controlled experiments assessing the usefulness of design program documentation in program maintenance' in *IEEE Transactions on Software Engineering*, Vol. 28, No. 6, June 2002.

Pressman, R.S. (2010) *Software engineering: a practitioner's approach*, New York: McGraw-Hill Higher Education.

Ramsey, N. (1994) 'Literate programming simplified' in *IEEE Software*, Vol. 11, Issue 5 (Sept. 1994), pp. 97-105.

Raskin, J. (2005) 'Comments are more important than code' in *ACM Queue*, Vol. 3, Issue 2 (March 2005), pp. 62-64.

Richards, L. (2005) *Handling qualitative data: A practical guide*, London: SAGE Publications.

Robillard, M.P., and Murphy, G.C. (2007) 'Representing concerns in source code' in *ACM Transactions of Software Engineering and Methodology,* Vol. 16, No. 1, Article 3, Feb. 2007.

Root, R.W. and Draper, S. (1983) 'Questionnaires as a software evaluation tool' in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (1983)*, Boston, MA, pp. 83-87.

Ruane, J.M. (2005) *Essentials of research methods: A guide to social science research,* Malden, MA: Blackwell Publishing.

Rüping, A. (2003) *Agile documentation: a pattern guide to producing lightweight documents for software projects*, Chichester: John Wiley & Sons.

Ryman, A. (1992) 'Foundations of 4Thought' in *Proceedings of the 1992 CAS Conference*, Volume I, pp. 133-155, Nov. 1992.

Sametinger, J. (1994) 'Object-oriented documentation' in *Journal of Computer Documentation*, Vol. 18, No. 1, pp. 3-14.

Sapsford, R. (2007) *Survey research (2e)*, London: SAGE Publications.

Schauer, R. and Keller, R.K. (1998) 'Pattern visualization for software comprehension' in *Proceedings of the 6th International Workshop on Program Comprehension (WPC '98)*, Ischia, pp. 4-12.

Schlotzhauer, S.D. (2009) *Elementary statistics using SAS*, Cary, NC: SAS Institute Inc.

Schünemann, U. (2001) *Defining programming languages* [online], http://web.cs.mun.ca/~ulf/pld/write.html [accessed 4 January 2011].

Shearer, C.D., and Collard, M.L. (2007) 'Enforcing constraints between documentary comments and source code' in *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pp. 271-280.

Simonyi, C. (1995) *The death of computer languages, the birth of intentional programming, Technical Report MSR-TR-95-52*, Redmond, WA: Microsoft Research. Available at: ftp://ftp.research.microsoft.com/pub/tr/tr-95-52.doc [accessed 9 May 2010].

Simonyi, C. (2005) 'Is programming a form of encryption?' in *The Intentional Software Corporation Blog*, http://blog.intentsoft.com/intentional_software/2005/04/dummy_post_1.html [accessed 9 May 2010].

Simonyi, C., Christerson, M., and Clifford, S. (2008) 'Intentional software' in *Proceedings of the 21st annual ACM SIGPLAN conference on object-oriented programming systems, languages, and applications*, Portland, OR, pp. 451-464.

Soloway, E. (1986) 'Learning to program = learning to construct mechanisms and explanations' in *Communications of the ACM*, Vol. 29, Number 6, pp. 850-858.

Soloway, E., Pinto, J., Letovsky, S., Littman, D., and Lampert, R. (1988) 'Designing documentation to compensate for delocalized plans' in *Communications of the ACM,* Vol. 31, Number 11, pp. 1259-1267.

Sousa, M.J.C., and Moreira, H.M. (1998) 'A survey on the software maintenance process' in *Proceedings of the 1998 International Conference on Software Maintenance*, Bethesda, MD, Nov. 1998, pp. 265-274.

Standish, T.A. (1984) 'An essay on software reuse' in *IEEE Transactions on Software Engineering*, Vol. SE-10, Issue 5, pp. 494-497, cited in Robson, D.J., Bennett, K.H., Cornelius, B.J, and Munro, M. (1991) 'Approaches to program comprehension' in *Journal of systems software*, Vol. 14, pp. 79-84.

Storey, M.-A.D., Fraccia, F.D., and Müller, H.A. (1997) 'Cognitive design elements to support the construction of a mental model during software visualization' in *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, pp. 17-28.

Sun Microsystems (1997, 2004) *Javadoc 5.0 Tool* [online], http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/index.html [accessed 7 March 2010].

Sun Microsystems (2004) *JDK 5.0 Developer's Guide: Annotations* [online], http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html [accessed 4 May 2010].

Swanson, E.F. (1976) 'The dimensions of maintenance' in *Proceedings of the 2nd International Conference on Software Engineering (IEEE)*, pp. 492-497, cited in Robson, D.J., Bennett, K.H., Cornelius, B.J, and Munro, M. (1991) 'Approaches to program comprehension' in *Journal of systems software*, Vol. 14, pp. 79-84.

van Lamsweerde, A. (2001) 'Goal-oriented requirements engineering: a guided tour' in *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, Toronto, pp. 249-262.

van Vliet, H. (2008) *Software engineering: Principles and practice (3e)*, Chichester: John Wiley & Sons.

von Mayrhauser, A., and Vans, A.M. (1995) 'Program comprehension during software maintenance and evolution', in *IEEE Computer*, Vol. 28, Issue 8 (August 1995), pp. 44-55.

Wallace, C., Cook, C., Summet, J., and Burnett, M. (2002) 'Assertions in end-user software engineering: A think-aloud study", in *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02),* pp. 63-65.

Watt, D.A., and Brown, D.F. (2000) *Programming language processors in Java: Compilers and interpreters*, Essex: Pearson Education Limited.

Weisberg, H.F., Krosnick, J.A., and Bowen, B.D. (1996) *An introduction to survey research, polling, and data analysis (3e)*, Thousand Oaks, CA: Sage Publications.

Yip, S.W.L, Lam, T., and Chan, S.M.K. (1994) 'A software maintenance survey' in *Proceedings of the First Asia-Pacific Software Engineering Conference*, Tokyo, Dec. 1994, pp. 70-79.

Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., and Cesar Sampaio do Prado Leite, J. (2005) 'Reverse engineering goal models from legacy code' in *Proceedings of the 13th IEEE International Requirements Engineering Conference (RE'05)*, Paris, pp. 363-372.

Bibliography

Lehman, M.M. (1980) 'Programs, life cycles, and laws of software evolution' in *Proceedings of the IEEE*, Vol. 68, No. 9 (Sept. 1980), pp. 1060-1076.

Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., and Turski, W.M. (1997) 'Metrics and laws of software evolution – the nineties view' in *Proceedings of the 4th International Symposium on Software Metrics*, Albuquerque, NM, pp. 20-32.

Parikh, G. and Zvegintzov, N. (1983) *Tutorial on software maintenance*, Silver Spring, MD: IEEE Computer Society.

Index

Appendix A: Extended abstract

Designing and evaluating an intention-based comment enforcement scheme for Java

Kevin Matz

Extended abstract of Open University MSc dissertation submitted 8 March 2011

Introduction

In large-scale enterprise software projects, a majority of the cost is expended during the *software maintenance* phase of the project lifecycle, during which a system undergoes continual adaptations to fix defects and meet changing requirements.

In order to understand where and how to make changes, maintenance software developers need to read and understand the existing source code. This is a time-consuming and error-prone activity, and much of the difficulty comes from trying to reconstruct the intentions and rationale that the original developers had in mind when they constructed the software.

Although reverse-engineering tools can help developers navigate and model structures in existing systems, this dissertation asserts that *explicitly* recording intention and rationale information during design and construction can reduce the time spent on such program understanding effort in the maintenance phase, and argues that a new technology-based solution is needed to better record intention and rationale.

Method

This dissertation makes use of a two-part survey aimed at practicing software developers involved in maintenance projects.

Part A of the survey collects data on participants' experiences and difficulties in software maintenance projects and solicits opinions on software documentation.

The results of the survey together with a literature review are then used to make the case that significant problems exist and that these problems are best dealt with by designing a new solution to aid in recording intention and rationale documentation.

A literature survey and examinations of past attempts at solutions are then used to formulate requirements for an "ideal" solution of this type.

Based on the requirements, a design was formulated for a solution (discussed under *Results* below), and a rudimentary prototype was created.

The design of the proposed solution was then evaluated by a number of analysis methods, including determining the degree of fit against the requirements, comparing it against other potential solutions, and discussing advantages and disadvantages.

Part B of the survey asked participants to read a short article explaining the proposed solution, and then solicited feedback on its perceived practicality and utility.

Results

The survey attracted 38 legitimate responses. Part A of the survey found that a majority of respondents do report having experienced difficulties in software maintenance, and, consistent with the results of similar surveys, it is established that developers rely on program comments more than other types of documentation. On this basis, a solution largely centred around internal documentation (program comments) is sought.

A list of 28 requirements is generated from the review of program comprehension literature and examination of past attempts at solutions.

A partial solution, an approach tentatively named *Design Intention Driven Programming*, is proposed. Under this approach, developers are encouraged to record their design intentions before writing code artefacts. The main feature of this approach is the addition of *intention comments* as specialised, first-class documentation constructs to programming languages; the compiler flags as errors any classes or other artefacts that are not described by intention comments.

To prevent empty or insufficient comments, the compiler computes a complexity metric for a section of code, and then computes an "information content metric" for the corresponding description text. An error is generated if the content of the description is deemed insufficient for the complexity of the code it describes, according to a threshold ratio that can be customised for each project.

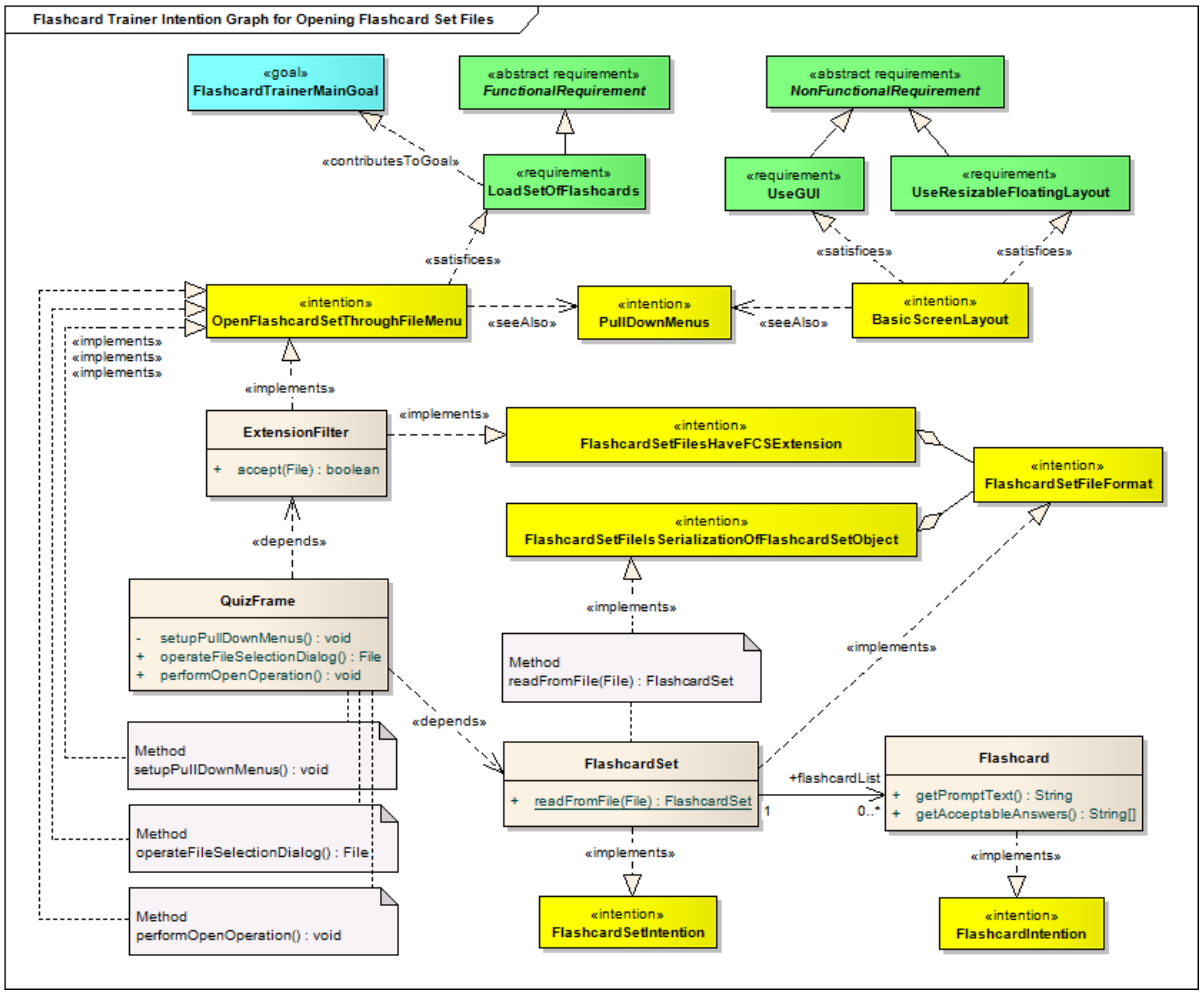
Intention comments have object-oriented features that allow templating opportunities that are particularly suitable for explicitly documenting instances of design patterns.

Applying the *Design Intention Driven Programming* approach to Java results in the tentatively-named *Java with Intentions* language. A rudimentary prototype of a precompiler supporting the language is constructed, and a sample application is developed using the language. These serve as proofs of concept for the approach.

An example of the intention comments syntax in *Java with Intentions* is given below.

```
intention QuizStateIntention {

    description {
        Class QuizState maintains the state of the current quiz, i.e., the current
        session in which all of the flashcards in a flashcard set will be presented
        once. This class is responsible for keeping track of the current flashcard,
        the user's score, and the application's mode (whether a game is in progress
        or is stopped).
    }

    requirementsreference[] satisficesRequirements = {
        EachFlashcardPresentedOncePerQuizSession,
        KeepScore
    };

    intentionreference playsRoleInPattern = FlashcardTrainerMVCPatternInstance;

}

public class QuizState implementsintention FlashcardTrainerMVCPatternInstance,
        QuizStateIntention {
    ...
}
```

The language also allows goals and requirements to be documented within code, and these can be interlinked with intention comments to form rich graphs describing the design of a software system at multiple levels of abstraction. A subset of such an "intention graph" is shown below using modified UML class diagram notation.



Analysis and discussion

Part B of the survey showed that respondents' opinions are divided on the perceived feasibility and utility of the proposed approach. While 86 percent of respondents agree with the general practice of recording design intentions before writing code, 51 percent view the solution as being potentially practical in real-world projects, and 49 percent indicate a willingness to use the solution on a trial basis, approximately 20 to 25 percent of respondents express strong dislike of the solution. An analysis shows that those favouring the solution are those who express strong support for documentation in general, and as self-selection bias means that such individuals likely comprise a disproportionately large percentage of the survey respondents, the opinions of this group cannot be considered entirely representative of the population of software developers as a whole.

A number of serious issues with the solution were identified, the most critical of which include:

- the general resistance of developers to write documentation;
- the increase in workload required to write and maintain the intention documentation;
- limitations of the documentation enforcement mechanism (it is unable to check the correctness of documentation, and the mechanism that attempts to enforce the sufficiency of descriptions can be easily bypassed); and
- the lack of concrete evidence of long-term cost savings.

The evaluation suggests that, while the approach may be promising for some projects and teams, the lack of evidence for the purported benefits, the cost of writing the documentation, and the unpopularity of writing documentation with most developers render it impractical for typical commercial software projects.

Follow-up studies are required to investigate the solution's impact on program comprehensibility and long-term cost savings; a survey is unable to provide direct empirical evidence to address these issues, and the research methods needed to investigate these issues were not considered feasible within the scope of this research project.

Appendix B: *Design Intention Driven Programming* and *Java with Intentions*

B.1 Introduction

Design Intention Driven Programming (DIDP) is an approach to programming that encourages software designers and developers to follow a process of producing a detailed technical design before constructing source code artefacts. This approach is supported by a tool: a small set of extensions that can be added to an existing programming language, and a compiler or other language processor which accepts programs written in the new language.

The approach permits the construction of a technical design as a structured network of units of documentation containing descriptions of the developer's design intentions and rationale. This network of documentation is stored in constructs in the program's source code files, and is intended to aid maintenance developers by reducing the amount of time and effort required to understand how the program works and why it was designed in that particular way.

In this dissertation, the Java programming language is extended to form a language tentatively named *Java with Intentions* (JWI).

DIDP and JWI together enable a pragmatic form of literate programming. Developers can record their design intentions within their program code by using special object-oriented constructs in the language designed specifically for that purpose[11], and these constructs can be interlinked to form rich structures, hierarchical or otherwise, representing the design of the program in terms of the relationships between goals, requirements, program structures and abstractions, and concrete code elements.

B.2 The role of intentions in programming and the case for special constructs to record intentions

When a programmer is about to write a component of a software program, he or she first makes a mental plan of *what* the component is intended to do and *how* it will work – in other words, the *design* of the component. Sometimes there is a *rationale* behind this plan – a reason *why* this design was chosen over any alternatives. The plan and rationale describe what the programmer is intending to do, and we can call this the programmer's *design intention*.

[11] It could be said that design intentions have been "reified" under this scheme. *Reification* is defined as "[considering] or [making] an abstract idea or concept real or concrete" (Collins English Dictionary, 2003), or "[regarding] or [treating] an abstraction as if it had concrete or material existence" (American Heritage Dictionary, 2009).

As a trivially simple example, the requirement for a simple program may be to calculate the sum of numbers between 1 and 10. The programmer's intention, then, may be to satisfy that requirement by writing a loop that iterates from 1 to 10 and increments a running total with the current number in the loop. If the programmer then writes Java code such as:

Figure 11: Correct implementation of the "sum of numbers between 1 and 10" intention

```
int runningTotal = 0;
for (int i = 1; i <= 10; i++) {
    runningTotal += i;
}
```

then the intention has been carried out correctly. If, however, the programmer had mistakenly written:

Figure 12: Incorrect implementation of the "sum of numbers between 1 and 10" intention

```
int runningTotal = 0;
for (int i = 1; i < 10; i++) {
    runningTotal += i;
}
```

then the intention has not been implemented correctly, as the loop will actually calculate the sum of numbers from 1 to 9. While the code at first glance may not appear to be incorrect, and it compiles correctly, it is nevertheless incorrect, in the sense that it does not match the requirements.

If the intention behind a piece of code is not explicitly documented, it is often not obvious that the code is incorrect. If a reader were to come across Figure 12 in a large and complex software system, the reader would probably have no way of knowing that the original intention was actually to sum the numbers from 1 to 10, unless he or she had an understanding of the requirements and the context of the application – not always possible for new members of a software team.

In complex systems, without an explicit recording of intentions, code that looks plausible but is in fact incorrect can remain in systems for long periods of time and cause incorrect results or side effects.

It is also time-consuming for developers to read and understand the meaning of code whose intention is not explicitly documented.

Ideally, then, the programmer would write a comment to record his or her intention, enabling a later reader to check whether the intention matches the implementation:

Figure 13: Source code section with intention documented via a simple comment

```
/* Calculate the sum of integers between 1 and 10: */
int runningTotal = 0;
for (int i = 1; i <= 10; i++) {
    runningTotal += i;
}
```

For this trivially simple example, the problem has been solved with a simple comment. There are a couple of problems with traditional comments in existing programming languages, however:

- Comments are optional, and many programmers simply don't use them.
- While small pieces of code can be documented with comments as in the example above, it doesn't scale well for documenting larger-scale design issues and architectural decisions that affect many parts of the system. "Delocalised plans" (Soloway, 1986) – structures or implementations of features that involve code spanning multiple files – are particularly difficult to document using comments.

In current programming languages, comments are a "second-class" linguistic element: comments are collapsed to single tokens and eliminated during lexical analysis. A program containing no comments will compile just as well as a program containing good explanatory comments. In the DIDP/JWI approach, the reasoning behind the design and construction of the program is considered to be just as important as the code itself. Hence, the idea is to add specialised constructs as an integral part of the language, elevating the documentation of intentions to the status of a "first-class citizen" in the programming language.

The constructs added to the language to aid in recording intention and rationale information are termed *intention comments*. Intention comments share much of the same purpose as traditional comments in programming language – communicating and explaining intent – but intention comments are a much richer construct that can contain multiple text fields as well as fields that can reference either other intention comments or other types of program elements.

The use of intention comments can enable the following advantages:

- The use of intention comments can be *enforced* by the compiler.
- Documentation can be structured according to object-oriented principles.
- Instances of design patterns can be explicitly documented.
- The technical design of a program can be formed and documented at multiple levels of abstraction as a graph of intentions.

We will explores these advantages briefly in the following subsections, after which we will explore the syntax of the *Java with Intentions* language extensions.

B.2.1 Documentation enforcement

Any attempts to enforce documentation would, ideally, encompass the following aspects:

1. *presence* of descriptions of intention and/or rationale
2. *sufficiency* of the descriptions
3. *correctness* of the descriptions

The introduction of the intention comment construct allows the compiler to enforce the *presence* of those constructs: if the language expects an intention comment to be

present for each class definition, for example, the compiler will simply generate a compiler error if the expected intention comment is missing.

Enforcing sufficiency and correctness of documentation using technological solutions is rather more difficult. It is impossible with current technology for a software program to comprehend and reason about natural-language (e.g., English) prose, so some automated checking of the *correctness* of descriptions in intention comments is unfortunately out of the question.

Testing for the *sufficiency* of descriptions is also difficult, not only because descriptions are written in natural-language prose, but also because the concept is completely subjective: the background knowledge of different readers will impact their judgements of whether a description is sufficient to explain some phenomena. However, we can make an imperfect attempt to address the issue of sufficiency by crude use of quantification and metrics. Using an algorithm, the compiler can calculate a complexity metric for some section of code, and then using another algorithm, the compiler can calculate a metric that quantifies the "information content" of the descriptive text associated with that section of code. If the ratio of the information content metric to the code complexity metric falls below a certain threshold, the description would be deemed insufficient to describe the code, and a compiler error would be generated. Threshold ratios would be configurable in each project.

This dissertation does not select or develop algorithms for complexity and information content metrics, nor does it make any recommendations for suitable threshold ratios (which may vary depending on the combinations of algorithms selected). These are topics suitable for a follow-up investigation. The McCabe Cyclomatic Complexity index is one example of a metric that could be used to measure the complexity of code within methods. The simplest information content metric is simply a count of characters. More advanced algorithms might compute scores by employing any number of techniques, such as parsing the grammatical structure of sentences, using dictionaries to recognise words, computing readability statistics, or using trainable neural networks or Bayesian statistical techniques to distinguish legitimate descriptions from garbage (similar to spam filtering).

B.2.2 Structuring documentation according to object-oriented principles

Intention comments resemble class definitions and can contain text fields as well as references to other intention comments or other code elements. The objected-oriented principle of inheritance can be applied to intention comments, and this allows developers to strategically re-use basic descriptions as templates, avoiding duplication of descriptions of similar but non-identical structures and aspects. An object-oriented style of documentation is a natural and effective way to describe object-oriented structures in programs.

In JWI, intention comments can use the `extends` keyword to inherit the descriptions and fields from another intention comment. Intention comments can also be declared `abstract`, which allows the specification of text or reference fields that then must be filled out by any derived intention comments that are not also `abstract`.

B.2.3 Explicitly documenting instances of design patterns

Design patterns are collections of collaborating classes and/or objects. Design patterns can be documented in a pattern catalogue, but *instances* (also called *applications*) of patterns in source code do not have any explicit "tangible" form beyond the particular collection of constituent classes or objects. Phrased differently, there is usually no easy way to search a source code collection to locate all instances of any particular pattern, because pattern instances are not named and don't have any artefact or construct of their own.

Intention comments allow the *reification* of design pattern instances: for each instance of a design pattern, an intention comment can be created and named, and all of the classes or objects participating in that design pattern instance can link to the intention comment. This allows a reader stumbling upon one of the constituent classes or objects firstly to understand that a particular design pattern is in use, and secondly to understand the role that the particular class or object plays in the design pattern instance.

The object-oriented nature of intention comments is particularly suitable for explicitly documenting instances of design patterns. An abstract intention comment can be created to describe a general design pattern, such as the Model-View-Controller pattern. Then, each instance of that design pattern can extend the abstract intention comment to form its own specific intention comment describing the context of that particular design pattern instance. An example will be presented in section B.4.

B.2.4 Formulating and documenting software designs as graphs of intentions

The ability of intention comments to contain references to other intention comments, and the ability to create inheritance hierarchies of intention comments, allow the construction of graphs of intention comments that can richly represent the design of a software system at multiple levels of abstraction.

This is a key difference between traditional comments and intention comments. Traditional comments are typically short remarks that usually relate only to the contents of the particular source code file they are located in. We might describe traditional comments as being "flat". Intention comments, in contrast, can be declared alongside program code in source code files, but they can also exist by themselves in source code files that contain no actual "executable" source code. This is suitable for describing higher-level structures and abstractions (such as design pattern instances) that span multiple source code files, as well as requirements, goals, and design intentions that apply to the entire system as a whole.

The Design Intention Driven Programming approach argues that, in principle, the design for a software system could be formed entirely of intention comments arranged in a series of layers, with interlinking between intention comments in each pair of adjacent layers:

- High-level goals
- Requirements
- Units of desired functionality (which may include references to external functional specification documents)
- High-level architectural design
- Fine-grained technical design (e.g., program structures and patterns)

Program code would then include references primarily to intention comments in the last layer, but could also refer to intention comments at higher levels. Elements in the code could then be easily traced back to design intentions and requirements.

This approach will be demonstrated with the sample Vocabulary Trainer application presented in Appendix D.

B.3 Introducing a syntax for intention comments in the JWI language

There are two basic forms of intention comments:

1. Free-standing
2. Inline

B.3.1 Free-standing intention comments

Free-standing intention comments are defined using the `intention` keyword, and must be named. Free-standing intention comment definitions are declared within Java source code files. They are usually situated within the file after any import statements, but before the first class definition. However, intention comments can also be included inside a class definition if the intention will be describing a method within that class.

Figure 14 gives an example of an intention comment named `FlashcardIntention`.

Figure 14: A simple intention comment

```
intention FlashcardIntention {

    description {
        The Flashcard class represents a flashcard for learning foreign-language
        vocabulary. A flashcard has a cue on one side of the card, and a list of
        one or more acceptable answers on the other side of the card.
    }

}
```

In general, intention names do not need the suffix “Intention”, but it is used in this case because the identifier `Flashcard` will be used by the class of that name.

The keywords `goal` and `requirement` can be used in place of `intention` for intentions that are better classified as one of those types, as shown in Figure 15.

Figure 15: Intention comments declared using keywords `goal` and `requirement`

```
goal FlashcardTrainerMainGoal {

    description {
        To provide an application to help a user learn foreign language vocabulary
        using flashcards.
    }

}

requirement WindowsCompatibility {

    description {
        The application must be able to operate under the Microsoft Windows
        operating system.
    }

}
```

The general term *intention comment* refers to elements declared using the `goal`, `requirement`, or `intention` keywords.

Text fields

Within a `goal`, `requirement`, or `intention` definition, one or more text fields can be included.

The `description` field is mandatory for all intention comments, and within braces, a textual explanation of the goal, requirement, or intention should be provided. The explanation is written in plain text; HTML tags can be used for markup.

Other ad-hoc text fields can be declared by specifying an identifier name (according to the standard Java language rules) and then placing text in braces after it. Figure 16 illustrates the use of a `fitCriteria` text field.

Figure 16: Adding a text field to an intention comment

```
requirement WindowsCompatibility {

    description {
        The application must be able to operate under the Microsoft Windows
        operating system.
    }

    fitCriteria {
        1. A Windows-compatible installer is provided.
        2. The product can be installed and the product will start on machines
           running English-language Windows 98, ME, 2000, XP, Vista, and 7.
    }

}
```

Referring to intentions

Once an intention has been documented, the developer can then construct or update the classes or methods needed to implement that intention.

To link a class to one or more named intentions, the `implementsintention` keyword is used in the class definition as shown in Figure 17.

Figure 17: Linking a class to intention comments

```
class Flashcard implementsintention FlashcardIntention {

    ...

}
```

Multiple intention names can be separated by commas.

It is preferred to have classes link to intentions, rather than directly to goals or requirements; the intention should serve as an intermediary to explain how a goal or requirement is to be implemented. However, if a class needs to link directly to a goal or requirement, the keywords `implementsgoal` and `implementsrequirement` are also provided. These can be combined as shown in Figure 18.

Figure 18: Linking a class to multiple intentions, requirements, or goals

```
class FlashcardSet implements Serializable
        implementsintention FlashcardSetIntention, FlashcardSetFileFormat
        implementsrequirement LoadSetOfFlashcards, ShuffleFlashcards
 {

    ...

}
```

Methods can also use the `implementsintention`, `implementsrequirement`, and `implementsgoal` keywords to signal that the method fulfils or "satisfices" (contributes to satisfying) an intention, requirement, or goal. An example is given in Figure 19.

Figure 19: Linking a method to an intention, requirement, or goal

```
class QuizFrame implementsintention FlashcardTrainerMVCPatternInstance,
        QuizFrameIntention
{

    ...

    public void presentCue(Flashcard flashcard) implementsrequirement
            PresentCueAndAcceptAndCheckAnswer {

        ...

    }

    ...

}
```

The *Java with Intentions* system requires all classes to refer to at least one intention; a compiler error will be generated if a class does not have at least one intention to describe it. (Note that references to requirements and goals cannot be used to bypass this requirement.) Methods can refer to intentions, and this is encouraged, but at present it is not mandatory; however, future *Java with Intentions* implementations might make this configurable.

Inheritance

In object-oriented programming languages, a "subclass" B can extend a "superclass" A, and by so doing, class B inherits all of the properties (in Java, the methods and member variables) of class A. Class B can introduce new properties or override the inherited properties.

Intention comments permit inheritance using the `extends` keyword, as illustrated in Figure 20. All of the fields in the "superclass" intention comment are considered to apply to the "subclass" intention comment, unless that intention comment overrides those fields.

Figure 20: Inheritance of intention comments using the `extends` keyword

```
intention GeneralUserAuthorizationStrategy {

    description {
        Before access will be granted to a function, the system will check
        an authorization table to determine whether the user is permitted to
        access the function.
    }

}

intention AuthorizationStrategy extends GeneralUserAuthorizationStrategy {

    specificDetails {
         Each user will be assigned to one or more of the roles in the ROLES
         table. The PERMISSIONS table will list the functions permitted for
         each role. ...
    }

}
```

Abstract intentions

In object-oriented analysis and design, it is often the case that a number of similar classes have a number of attributes in common. The common attributes can be extracted to a superclass, and the individual classes can then extend the superclass to avoid repeating the common attributes. If the superclass is not designed to be instantiable, it can be declared as *abstract*.

Intention comments can also be declared abstract using the `abstract` keyword. The `abstract` keyword is usually applied to `intention`s, but it can also be applied to `goal`s and `requirement`s as well. This is illustrated in Figure 21.

Figure 21: Declaration of abstract intention comments representing requirements

```
abstract requirement FunctionalRequirement {
    description {
        Functional requirement.
    }
}

abstract requirement NonFunctionalRequirement {
    description {
        Non-functional requirement.
    }
}

requirement ShuffleFlashcards extends FunctionalRequirement {
    description {
        The application shall randomize (shuffle) the flashcards in the
        flashcard set so that the user is not presented with the same sequence
        of cards each time.
    }
}

requirement UseGUI extends NonFunctionalRequirement {
    description {
        The application shall use a graphical user interface.
    }
}
```

Abstract intention comments cannot be referred to directly by classes or methods. Of course, classes or methods can refer to "concrete" (non-abstract) intention comments that have been derived from abstract intention comments.

Abstract intention comments allow empty text or reference fields to be specified which then must be filled out by any non-abstract intention comments that extend it.

Reference fields

An intention comment can refer to other intentions, requirements, or goals. References are handled similarly to variable declarations in classes and methods; each reference field must take a name, and the following "data types" are offered:

- `intentionreference`
- `requirementreference`
- `goalreference`

While it is generally preferred that classes and methods link to intentions, intentions can also link to specific classes, methods, or even objects (using fully-qualified identifiers) using the following data types:

- `classreference`
- `methodreference`
- `objectreference`

These data types are used for referring to a *single* intention, requirement, goal, class, method, or object by name. In many cases, it is suitable to have a reference field refer to *multiple* elements, so in such cases it is permissible to use the array notation "`[]`" after the data type name. This permits a list of intentions, requirements, goals, classes, or objects to be listed, separated by commas, and surrounding with braces.

Fields in abstract intention comments do not need to provide values; fields in non-abstract intention comments must provide values, though `null` and "`{}`" (an empty list) are permitted. Assignment of values uses the "=" operator. Field declarations are terminated with a semicolon.

Examples of the syntax are illustrated in Figure 22.

Figure 22: Examples of single and set reference fields in an intention comment

```
intention OpenFlashcardSetThroughFileMenu {

    description {
        To open a flashcard set and start a new quiz session, the user can use the
        File | Open... option. This will launch an "Open Flashcard Set" dialog which
        allows the user to select a file with the .fcs filename extension.
    }

    requirementreference satisfiesRequirement = LoadSetOfFlashcards;

    intentionreference[] seeAlso = { PullDownMenus, FlashcardSetFileFormat };

    methodreference[] fileHandlingImplementedInMethods = {
        FlashcardSet.readFromFile(File),
        FlashcardSet.writeToFile(String)
    };

}
```

Naming scope

Intention comments can be declared in any .java source code file[12], and these names of intention comments have global scope and visibility across the project. A compiler error will be generated if two intention comments are declared with the same name in the same project.

Classes and methods using the `implementsintention` keyword can refer to intention comment names declared anywhere else in the project; qualification with package names is not necessary.

[12] In the prototype precompiler implementation, *Java with Intentions* source code files have the extension .jwi. The precompiler processes .jwi files by stripping out all instances of *Java with Intentions* language extensions and generates corresponding plain .java files.

References in intention comments to classes, methods, or objects must use fully-qualified identifier names with the full package path (e.g., `com.sampleprojects.flashcardtrainer.QuizFrame`) to refer to any entities outside of the scope of the package in which the intention comment is declared.

B.3.2 Inline intention comments

Within a method, lengthy blocks of code without any descriptive comments will be flagged by the JWI compiler. To associate comment texts with blocks of code, a syntax is provided for "inline intention comments", which consist of start and end tags that can surround blocks of code. The start tag includes the comment text. The use of start and end tags allows inline intention comments to be nested, allowing each step of an algorithm to be broken into smaller sub-steps.

Figure 23 illustrates the syntax for inline intention comments in JWI.

Figure 23: Example of syntax for nested inline intention comments

```
[[ 1 | Shuffle the deck of flashcards (flashcardList) by iterating
       through the list and swapping the card at the current position
       with another randomly-chosen card ]]
for (int i = 0; i < flashcardList.size(); i++) {
     [[ 1.1 | Generate a random number, which will serve as the index
              of the card to be swapped with the current index ]]
     int otherIndex = randomGenerator.nextInt(flashcardList.size());
     [[ /1.1 ]]

     [[ 1.2 | Swap the records at indices i and otherIndex ]]
     Flashcard tempCard = (Flashcard) flashcardList.get(i);
     flashcardList.set(i, flashcardList.get(otherIndex));
     flashcardList.set(otherIndex, tempCard);
     [[ /1.2 ]]
}
[[ /1 ]]
```

"Opening" comment tags take the syntax `[[ commentIdentifier | descriptionText ]]` (where the square brackets and vertical bar are literal characters). The comment identifiers could be virtually any names, but in this example they follow a hierarchical numbering scheme. "Closing" comment tags use a slash in front of the comment identifier, similar to XML.

B.4 Documenting instances of patterns using intention comments

As discussed previously, the object-oriented nature of intention comments lends itself to documenting instances of design patterns via a templating mechanism.

In Figure 24, an abstract intention comment is created to describe the common Model-View-Controller pattern.

Figure 24: An abstract intention defining a general design pattern

```
abstract intention ModelViewControllerPattern {

    description {
        The Model-View-Controller pattern structures the user interface
        code into separate components. This separation of concerns helps
        improve understandability and modifiability.

        The model consists of a representation of the application's data.
        The model notifies listeners (typically, one or more view
        components) when the data changes.

        The view component presents the data to the user in the form of
        UI components. Multiple views based on the same model may exist.

        The controller acts upon input from the user and updates the
        model and/or interacts with the view.
    }

    classreference[] modelClasses;
    classreference[] viewClasses;
    classreference controllerClass;

}
```

A specific instance or application of the Model-View-Controller pattern can then be specified by declaring an intention comment that extends this abstract intention. In the new intention comment, references must be provided for the required fields. Figure 25 shows an intention comment for the specific MVC pattern instance used in the sample Vocabulary Trainer application.

Figure 25: A concrete intention extending the abstract pattern definition to specify a particular instance of the pattern

```
intention FlashcardTrainerMVCPatternInstance extends ModelViewControllerPattern {

    description {
        The flashcard trainer user interface is constructed according to the
        Model-View-Controller pattern.
    }

    modelClasses = { QuizState, FlashcardSet };
    viewClasses = { QuizFrame };
    controllerClass = QuizController;
}
```

The components that take part in this pattern can then link themselves to the intention for the pattern instance; one such example is given in Figure 26.

Figure 26: A component of the pattern instance links itself to the intention comment for the pattern instance

```
class QuizController implementsintention FlashcardTrainerMVCPatternInstance,
        QuizControllerIntention {
     ...
}
```

Now, when new developers join this project and encounter any class that is a part of this pattern instance, they will be able to read the comments and follow the links to locate the other components of the pattern and understand their relationships. A

developer who was not previously aware of this design pattern can follow the links and read the descriptions to gain a better understanding.

B.5 Using graphs of intention comments to represent the design of a system

The primary advantage of the Design Intention Driven Programming approach is that intention comments can be interlinked into graphs that represent the design of a software system at multiple levels of abstraction.

Traditionally, the architecture and design of software systems is written in word-processing documents and augmented with diagrams. The typical hierarchical chapter structure of traditional documents, however, typically does not match the more complex structure of software systems.

With intention comments, the conveniences and rich structuring possibilities of the object-oriented approach are extended to the documentation domain. The design for a system can be broken into small units of textual descriptions, and these units (intention comments) can be interlinked and can take advantage of the inheritance mechanism. Instead of a hierarchical document with no direct linkages to the software code, a "web"-style graph of navigable documentation units with direct links to the software code elements being described can be constructed. A well-written graph of intention comments has the potential to be much more useful to current and future developers.

Seen from a distance, typical graphs might have roughly pyramidal arrangements, with a small number of goals at the top level, followed by a greater number of requirements, and then a large number of intentions, which are linked to potentially an even greater number of code artefacts. While intention graphs could take a true hierarchical form in very small systems, the interlinking between elements at each level and in between the levels, however, means that a more complex graph will emerge in non-trivial systems.

B.5.1 Graphical representation of intention graphs with UML

UML class diagrams, with some minor adjustments, can be repurposed to represent intention graphs. Figure 27 illustrates one possible depiction of the intention graph involving the Model-View-Controller pattern in the Vocabulary Trainer sample application.

Figure 27: UML class diagram representing the intention graph for the Model-View-Controller pattern instance in the Vocabulary Trainer sample application

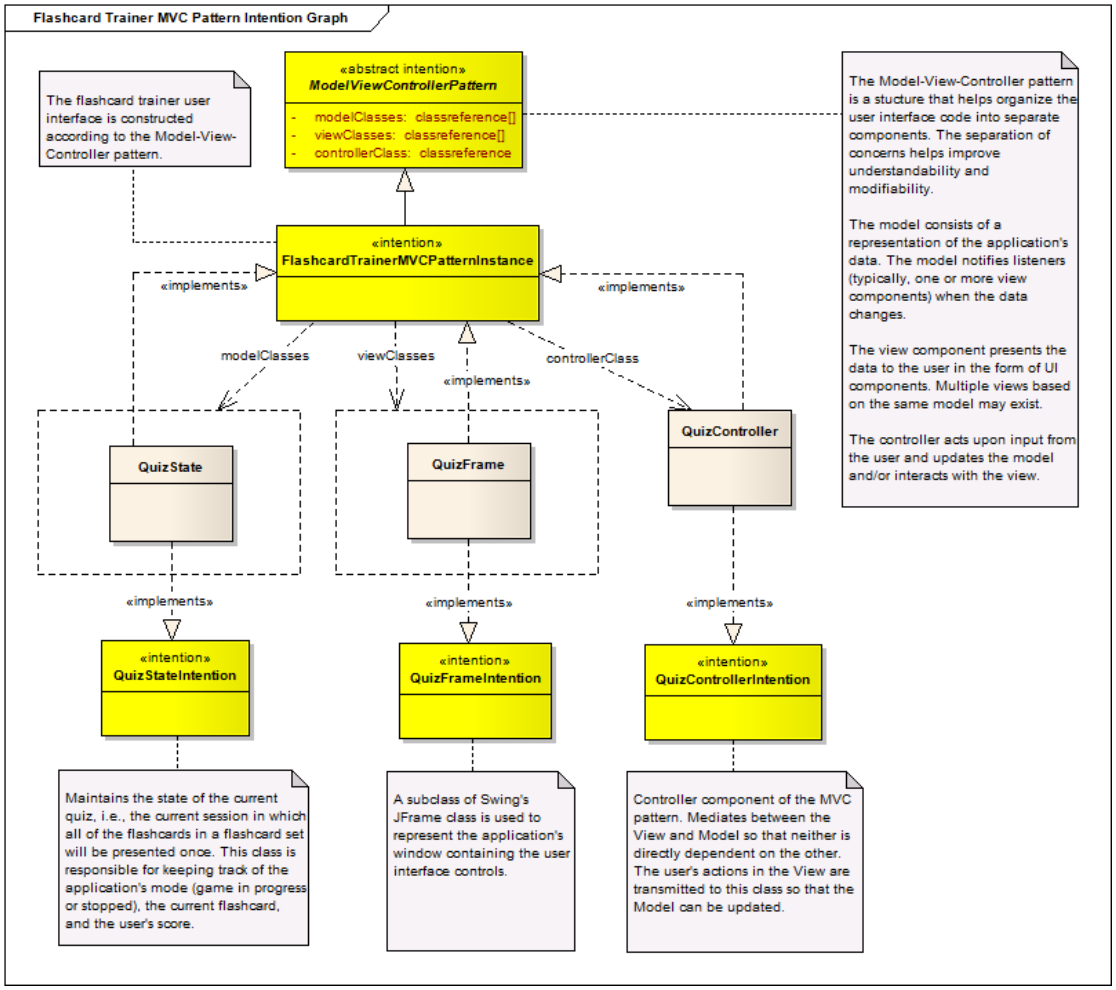
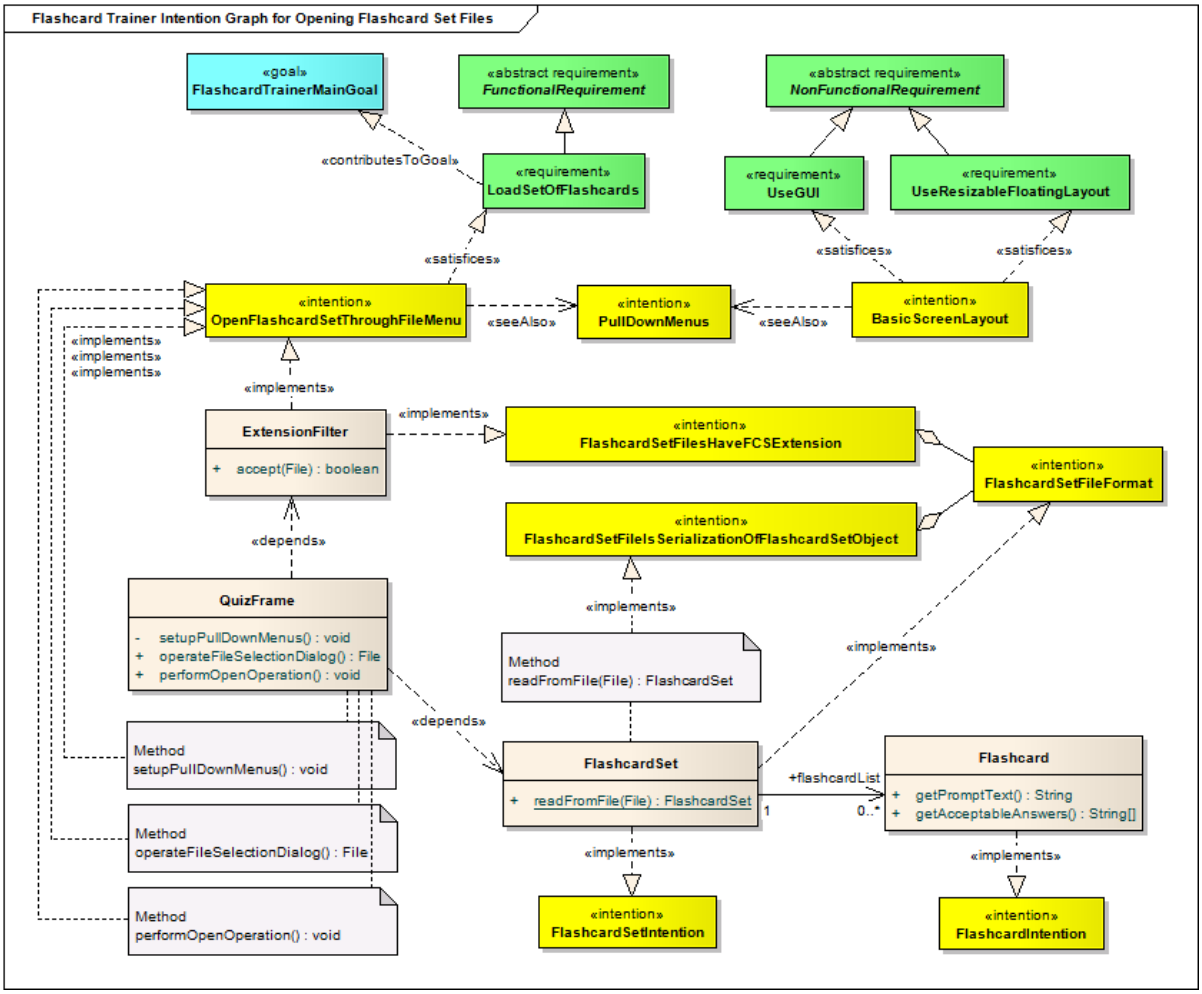


Figure 28 illustrates a more complex intention graph depicting one goal, several requirements, a number of intentions, and classes and methods linked to the intentions. To save space, the actual description texts of the intentions have been omitted, which obviously limits the utility of the diagram. Some liberties have been taken with the notation in this diagram: because UML does not support the depiction of methods as self-standing entities, methods that need to link to intentions have been represented using Note elements.

Figure 28: UML class diagram for the subset of the intention graph for the Vocabulary Trainer application relevant to the loading of flashcard set files



B.5.2 Navigation between intention comments in an IDE

While the entire intention graph for a software system could be represented as a UML diagram, the diagram would become unmanageably large for any non-trivial system. Even omitting the actual description texts of the intentions, the intention graph for the relatively small Vocabulary Trainer application will not fit legibly on a single sheet of paper.

This is one major reason why intention comments were conceived of as a language element rather than a purely graphical modelling element.

In the ideal, fully-realised form of the Design Intention Driven Programming approach, developers would work with an Integrated Development Environment (IDE) such as Eclipse that has been customised with integrated support for the *Java with Intentions* language[13]. The IDE would support easy navigation between intention comments and their references using hypertext-like links.

[13] Please note that the prototype implementation constructed for this dissertation includes only a precompiler; IDE integration has not been implemented due to time constraints.

Currently in Eclipse, for example, a developer can position the cursor on an identifier name and press the F3 key (or press the Control key and click on an identifier name)[14], and the screen will shift focus to display the place in the code where that identifier is defined. The same gestures and behaviours would be supported for *Java with Intentions* language constructs. For example, if a developer were to encounter the method signature "`public void handleEndOfQuiz() implementsintention DisplayScopeInPopUpDialog`", the developer could control-click on the identifier `DisplayScopeInPopUpDialog` to navigate directly to the place in the source code where the intention comment named `DisplayScopeInPopUpDialog` is defined.

B.6 Generating hypertext documentation and the relationship between *Java with Intentions* and *Javadoc*

It is envisioned that a tool similar to the `javadoc` tool would be constructed to generate hypertext documentation sets as a set of HTML pages.

Unlike `javadoc`, which does not output the actual source code, a documentation generator for *Java with Intentions* would have to include the entirety of the project's source code in the output documentation, as the intentions are intimately linked with the source code (and *inline* intentions exist only within the context of source code inside of methods).

Intention comments and Javadoc comments can coexist within the same project. Because there is some degree of overlap between intention comments and Javadoc comments, a project team should consider whether to use *Java with Intentions* exclusively or to use it with Javadoc in a complementary fashion.

Javadoc's generated hypertext documentation is most suitable for documenting APIs that expose classes and methods for public consumption, and so for frameworks and libraries, it is recommended that Javadoc be used so that the traditional Javadoc reference documentation can be published without exposing the source code.

Javadoc is somewhat less effective for documenting the structure and functioning of the "private" implementation behind the publicly exposed façade. Architectural decisions, large-scale software structures (such as abstraction layers), cross-cutting concerns, and design pattern instances can be arguably described better using intention comments.

[14] These gestures assume the standard key bindings on the Windows distribution of Eclipse.

B.7 Responses to common objections and questions

Why not use Java annotations instead?

Some aspects of the intention comments could indeed be represented with annotations. However, the object-oriented nature of intention comments would be awkward to reproduce (annotation definitions but not annotation instances can use the inheritance mechanism), and the integrity of cross-references to other annotations or other program elements cannot be checked by the compiler. Annotations are not mandatory, so enforcement would be impossible. There would also be awkward syntax issues to deal with: multi-line text strings would have to be quoted.

Annotations were not designed for the express purpose of recording design intentions. A construct purposely designed to support the concept is more suitable.

What is the relationship of intention comments to external specification documents?

Traditionally, requirements are managed either in documents or in specialised databases, and architectural designs, functional specifications, and technical specifications are written as documents.

In a project where traditional documents have been already generated, developers might choose to create goals, requirements, and intention comments in the source code to match the existing documents. This would be inconvenient, however, as it would lead to the same information being maintained in two places, and one would have to be designated the master. The JWI system might itself be further extended to be able to refer to requirements and documentation artefacts stored in external repositories. In fact, a wiki-like documentation system could even be envisioned, and intention comments in the source code would be able to refer to the documents or articles by name. The JWI compiler would access the documentation system to verify the references.

A project using Design Intention Driven Programming from the very beginning might base all of the initial goals and requirements and architectural design (in terms of high-level intentions) in the source code. These could then refer to external documents that contain more information, especially in cases where there is a lot of detail or numerous diagrams or tables that will not easily fit in intention comments.

Why not use a formal specification language like Z?

It is exactly in the translation from natural human-language thinking to symbolic representation that encoding errors are most likely to occur and the intention behind the symbolic representation is most likely to be lost – exactly the same problem as translating from natural language to symbolic program code. Formal specification languages are also not commonly used in industry except for safety-critical systems.

Appendix C: The prototype *Java with Intentions* precompiler implementation

C.1 An overview of how JWI programs are processed

Java with Intentions (JWI) is an extension of the Java language as specified by the *Java Language Specification* (Gosling *et al.*, 2005). While the extensions could generally be applied to any version of Java, this specification and the reference implementation are based on Java version 1.5 (i.e., Java SE 5).

The JWI language extensions serve purely for the purpose of documentation. The language extensions have no effect on the behaviour of a program.

Ideally, JWI source code would be stored in text files with the .java extension. However, this would cause conflicts with existing tools that do not expect the JWI language extensions. For that reason, JWI source code files are currently text files with the extension .jwi.

A syntactically-valid .jwi file can be converted to a legal .java source code file by simply stripping out (or commenting out) all instances of the JWI language extensions (i.e., declarations of intention comments, references to intention comments from classes and methods, and inline intention comments within methods).

There are three classes of language processing tools that are conceivable for processing JWI projects:

A. Native compilers that accept .jwi files and output Java .class files;
B. Precompilers that accept .jwi files and output .java source code files (by stripping out instances of JWI language extensions); the .java files can then be passed to the `javac` compiler to create Java .class files;
C. Integrated development environments (IDEs) that continuously parse the source code in the editor and highlight syntax errors, and use either a compiler of class A or a precompiler of class B to compile .jwi files to Java .class files.

The prototype implementation is a precompiler of class B.

Processing tools of all classes must perform the following functions:

- Syntax checking: Validate that each .jwi file matches the language grammar.
- Context analysis: Check that references to and from intention comments are valid, bearing in mind the scope rules specified in Appendix B.
- Stripping of instances of JWI extensions: Remove all instances of JWI language extensions so that the file can be compiled as a plain .java source code file.

C.2 Scope of prototype implementation

Table 42 lists major functional requirements for the precompiler for the *Java with Intentions* language, and indicates which of these requirements have been fulfilled in the prototype implementation. Time constraints limited the functionality completed.

Table 42: Functional requirements defining the scope of the Java with Intentions precompiler reference implementation

No.	Requirement	Planned for implementation?	Implemented?
1	Precompiler must accept from the command line a list of *.jwi files to process	yes	yes
2	For each input .jwi file, the precompiler must output a corresponding, translated .java file	yes	yes
3	Precompiler must output *.java files into the appropriate directory structure dictated by the Java package of each file	yes	yes
4	Precompiler must strip out or comment out all instances of syntactically-valid intention comments when translating from *.jwi to *.java files	yes	yes, partially; working for a limited subset of the syntax
5	*.java files output by the precompiler must conform to Java 1.5 syntax and must be compilable by the javac compiler	yes	yes, but requires further testing
6	Precompiler shall report understandable error messages when a syntax error is encountered in an input file	yes	partially; errors generated by the precompiler's context checking are acceptable; parser errors generated by ANTLR are passed through, and these are less understandable
7	Should a syntax error be detected in an input file, the precompiler shall stop processing that file, but shall continue processing other input files specified on the command line	yes	yes
8	Precompiler shall accept `intention` definition blocks	yes	yes

No.	Requirement	Planned for implementation?	Implemented?
9	Precompiler shall accept `requirement` definition blocks	yes	yes
10	Precompiler shall accept `goal` definition blocks	yes	yes
11	Precompiler shall accept `implementsintention` clauses and omit them from translated output	yes	yes
12	Precompiler shall accept `implementsrequirement` clauses and omit them from translated output	yes	no (in progress)
13	Precompiler shall accept `implementsgoal` clauses and omit them from translated output	yes	no (in progress)
14	Precompiler shall report an error if a Java compilation unit (`class`, `interface`, `enum`) is defined without an `implementsintention` clause	yes	in progress; working for classes but not yet tested with `interface`s and `enum`s
15	Precompiler shall verify that the names of intention comments referenced in `implementsintention`, `implementsrequirement`, `implementsgoal` clauses are defined within the project (where "project" refers to the list of .jwi files supplied on the command line)	yes	partially: yes, for `implementsintention`; no, for `implementsrequirement` and `implementsgoal`
16	Precompiler shall accept inline intention comment syntax and omit the opening and closing comments from translated output	yes	partially; an alternative syntax is temporarily in use due to difficulties with tokenisation involving free text fields
17	Precompiler shall verify that the identifiers in opening and closing inline intention comments match, and shall observe nesting of inline intention comments	yes	no
18	Precompiler shall allow declaration of text fields and reference fields in intention comments marked with the `abstract` keyword	yes	no

No.	Requirement	Planned for implementation?	Implemented?
19	Precompiler shall check that non-abstract intention comments that extend `abstract` intention comments fulfil all of the required text and/or reference fields	yes	no
20	Precompiler shall check that references from reference fields to intention comments names match	yes	no
21	Precompiler shall check that references from reference fields to Java identifiers (e.g., class names) match	no	no
22	Precompiler shall compute information content metrics on intention comments	no	no
23	Precompiler shall compute complexity metrics on code described by intention comments	no	no
24	Precompiler shall allow specification of a threshold for the information content to complexity metric ratio	no	no
25	Precompiler shall ensure that all sections of code adhere to the specified threshold	no	no

C.3 Demonstration of current state of implementation

Please refer to Appendix D for a walkthrough of the current state of the implementation.

Appendix D: Walkthrough of the precompiler implementation and sample application (Vocabulary Trainer)

The prototype precompiler and the sample application are bundled into the file `JWI_Deliverables.zip`, which has been included with the electronic submission of this dissertation, and which can be found on the CD-ROM included with the printed copy of the dissertation.

Please unzip the file before proceeding. Path names in this section are relative to the base directory of the unpacked archive.

D.1 Prerequisites for running the precompiler and sample application

It is assumed that version 1.5 or higher of the Java SE SDK is installed on your machine. Several shell scripts for building and running the precompiler are provided, and executing these will require the use of the "bash" shell, which requires a Unix/Linux environment. (CygWin may be used on Windows machines but this has not been tested.)

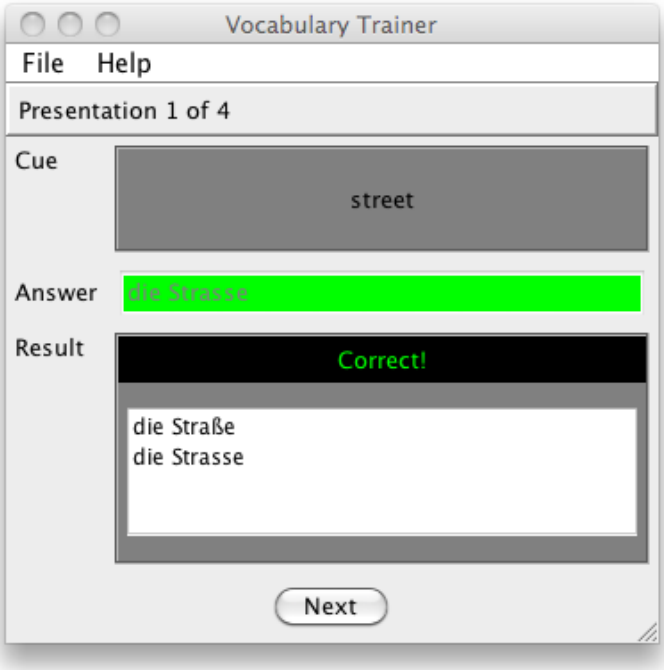
To simply inspect the files, no special software is required as all source code files are plain text files.

D.2 Inspecting the sample application

A sample application has been constructed to demonstrate the use of the *Java with Intentions* language extensions, and to serve as a test case for the precompiler prototype.

The project is a very rudimentary foreign-language vocabulary trainer application that lets the user practise with sets of flashcards. The application uses Swing for the user interface. A screenshot of the application presenting a quiz using a German-English flashcard set is shown in Figure 29.

Figure 29: Screenshot of Vocabulary Trainer application



Once the `JWI_Deliverables.zip` archive has been unpacked, the source tree for the sample project, consisting of *.jwi files, can be found under path `JWI_DemoProject1/src-jwi/`.

(If you wish to actually run the application, a pure Java version that is the same as the JWI version but with all instances of JWI language extensions commented out is provided under path `JWI_DemoProject1/src/`. An Ant build file has not been constructed, but under Eclipse or other IDEs, the necessary files will be compiled automatically if you run the main class, which is `JWI_DemoProject1/src/jwidemos/flashcardtrainer/FlashcardTrainer.java`. Once the application has started, select *File* | *Open*… in the menu and choose the flashcard set file `rsrc/FlashcardSets/German1.fcs`.)

D.3 Inspecting the precompiler's grammar files

The ANTLR grammar files are located under path `JWI_Precompiler/src_in/antlr/`.

The `Java.g` file by Parr (2008), which is a grammar for the Java 1.5 language, was used as a base and extended to support the JWI extensions. For technical reasons, the original `Java.g` file had to be split into separate lexer and parser grammars:

- `JWI_Precompiler/src_in/antlr/JWIPreprocessor_Lexer.g` is the lexer grammar, defining the fundamental tokens (lexemes) such as keywords and operators.
- `JWI_Precompiler/src_in/antlr/JWIPreprocessor_Parser.g` is the parser grammar, defining the syntax of language constructs.

ANTLR grammars are based on production rules using Extended Backus-Naur Form. However, within these production rules, Java code can be embedded which is executed when the parser runs; this enables the parser to build data structures that can be used in the context analysis phase, which takes place after the parsing phase. For example, the production rule for the declaration of an intention comment includes code to add the name of the intention to a symbol table.

Unfortunately, the Java code and other ANTLR-specific options embedded in the production rules can make the grammar files hard to read. There is little separation of concerns possible between the basic grammar and the processing that is triggered when the parser encounters the tokens matching the production rules.

Due to time constraints, not all language features are implemented in the grammar files and the precompiler code. Some temporary compromises with minor alterations of the syntax were made in several cases where implementation problems are still under investigation; most notably, the `description` keyword requires the use of "double braces" ("`{{ }}`") to surround free-form text instead of single braces as shown in this dissertation. (The single braces are still the "ideal" syntax.)

Please refer to the statement of scope in Appendix C, which states which features have been implemented and which have not. Instances of language features not yet supported have been commented-out in the sample Vocabulary Trainer project.

D.4 Inspecting the remainder of the precompiler code

The shell scripts `JWI_Precompiler/make-lexer-grammar.sh` and `JWI_Precompiler/make-parser-grammar.sh`[15] invoke the ANTLR tool to convert the two grammar files described in the previous section into the Java files `JWIPreprocessor_Lexer.java` and `JWIPreprocessor_Parser.java`, which are located under path `JWI_Precompiler/src_in/java/com/kevinmatz/jwi/parser/`.

`JWIPreprocessor_Lexer.java` and `JWIPreprocessor_Parser.java` form the core of the parsing stage of the precompiler, but further Java code is necessary to parse command line parameters, handle context analysis logic, and write the output .java files to disk.

All of the Java code for the precompiler is located under `JWI_Precompiler/src_in/java/`. The main class is `JWI_Precompiler/src_in/java/com/kevinmatz/jwi/JWIPreprocessor.java`.

To compile the entire precompiler project, use the shell script `JWI_Precompiler/make-javac.sh`. This invokes `javac` to build the .class files under path `JWI_Precompiler/executables/`.

[15] Build management with Ant was planned but did not materialise due to time constraints.

D.5 Running the precompiler using the sample application as input

The sample Vocabulary Trainer application, consisting of *.jwi files, is located under `JWI_DemoProject1/src-jwi/`.

To run the precompiler on the sample application, run the shell script `JWI_Precompiler/run-precompiler-on-flashcard-demo.sh`. It simply invokes the preprocessor executable, supplying the *.jwi filenames and an output directory as command-line parameters.

Normally the preprocessor runs silently, generating output on the console only when syntax or contextual errors are detected. For debugging purposes, however, the precompiler currently generates console output as shown in Figure 30.

Figure 30: Console output from JWI precompiler

In this case, no errors have been detected, and all of the input files, i.e., all *.jwi files under `JWI_DemoProject1/src-jwi/`, have been translated into corresponding Java source code files (under the same package structure of subdirectories) under `JWI_Precompiler/tmp/generated_java/`.

To illustrate the generation of an error message, one of the intention references in an input .jwi file was replaced with the name of a fictional intention comment name not defined in the project. Figure 31 shows the error message reported in this case.

Figure 31: Console output showing a contextual analysis error detected by the JWI precompiler

If the precompiler has run successfully without generating any errors, a set of generated .java output files will have been created in a directory specified by a command-line parameter. In the generated output files, all instances of intention comments and reference clauses are commented out, as shown in Figure 32.

Figure 32: Output file QuizState.java generated by commenting-out JWI constructs present in the input file QuisState.jwi

QuizState.java

```
package jwidemos.flashcardtrainer;

import java.util.List;


/* intention QuizStateIntention {

 description {{
        Class QuizState maintains the state of the current quiz, i.e., the current
        session in which all of the flashcards in a flashcard set will be presented
        once. This class is responsible for keeping track of the current flashcard,
        the user's score, and the application's mode (whether a game is in progress
        or is stopped).
    }}

} */


public class QuizState /* implementsintention FlashcardTrainerMVCIntention,
QuizStateIntention */ {
```

```
... remainder of file ...
```

Generated Java source code files can then be compiled with `javac`. Unfortunately, because the precompiler does not yet support all JWI language features, instances of those unsupported features will not be suppressed in the .java output files, so not all generated .java files currently will compile with `javac`.

Appendix E: The questionnaire and summary statistics

Introduction page

Thank you for taking the time to answer this questionnaire.

This research is being conducted by Kevin Matz as a part of his thesis[16] for an MSc in Software Development at the Open University (Milton Keynes, UK).

It is assumed that you are a practicing software developer, and programming is one of your major responsibilities. If this does not describe your situation but you still wish to take part, please contact Kevin at kevin@kevinmatz.com.

This questionnaire has two parts. In part A, you will be asked about your opinions on software documentation and program comments, practices at your current organization, and any difficulties you may have encountered during software maintenance activities.

In part B, you will be asked to read a short article introducing a proposed new approach to in-program documentation. You will then be asked for your opinions and any feedback on this approach.

If a question does not apply to your personal situation, or if you prefer not to answer, please leave that question blank. An empty response to a question will be treated as "not applicable".

Your responses to this questionnaire are anonymous and no personally identifying information will be stored.

Please do not complete the questionnaire more than once.

Thank you -- your participation is very much appreciated!

[Survey version 1.1]

Part A, Page 1 of 6

The following questions relate to the code base of your organization's software product(s) or system(s) that you work on. If you work on multiple projects, either choose a major project, or consider them all together as a whole.

[16] Due to my geographic location (Canada) and the fact that a majority of the survey participants are located here, I have used the North American term "masters thesis" rather than the UK term "dissertation".

[Q01] Approximately how long has the primary system or project you work on been in existence?[17]

Choice	Response count	Percentage
New development	5	13.2%
1-2 years	5	13.2%
3-5 years	6	15.8%
6-10 years	11	29.0%
11-20 years	7	18.4%
20-30 years	2	5.3%
30+ years	2	5.3%

On a scale of 1 to 7, ranging from Strongly Disagree to Strongly Agree, how would you answer the following questions?

		1 Strongly disagree	2 Disagree	3 Slightly disagree	4 Neutral	5 Slightly agree	6 Agree	7 Strongly agree	Mean	Std. dev.
[Q02]	**I consider the size of the project or product to be very large (e.g., large number of staff working on it, large quantity of code).**	2 (5.3%)	4 (10.5%)	3 (7.9%)	6 (15.8%)	8 (21.0%)	10 (26.3%)	5 (13.2%)	4.68	1.74
[Q03]	**The system makes use of a number of different languages or technologies.**	1 (2.6%)	1 (2.6%)	3 (7.9%)	3 (7.9%)	7 (18.4%)	14 (36.8%)	9 (23.7%)	5.42	1.50
[Q04]	**The application domain (e.g., financial, health insurance, medical) of the software I work on is very complex and specialized.**	0	2 (5.3%)	5 (13.2%)	4 (10.5%)	4 (10.5%)	12 (31.6%)	11 (29.0%)	5.37	1.58
[Q05]	**Automated regression test cases are regularly used in our project.**	10 (26.3%)	8 (21.0%)	1 (2.6%)	5 (13.2%)	3 (7.9%)	8 (21.0%)	3 (7.9%)	3.50	2.18
[Q06]	**The system/product uses modern software development technologies and techniques.**	0	5 (13.5%)	5 (13.5%)	8 (21.6%)	3 (8.1%)	11 (29.7%)	5 (13.5%)	4.68	1.67
[Q07]	**The system/product is built using object-oriented technologies.**	4 (10.8%)	10 (27.0%)	3 (8.1%)	2 (5.4%)	5 (13.5%)	7 (18.9%)	6 (16.2%)	4.05	2.15
[Q08]	**In general, I consider the**	2 (5.4%)	6 (16.2%)	6 (16.2%)	8 (21.6%)	4 (10.8%)	10 (27.0%)	1 (2.7%)	4.08	1.67

[17] Mean and standard deviation are not reported for this question as the response categories are not linear.

		1 Strongly disagree	2 Disagree	3 Slightly disagree	4 Neutral	5 Slightly agree	6 Agree	7 Strongly agree	Mean	Std. dev.
	quality of the existing code I work on to be very good.									
[Q09]	**I consider the system/product to have a well-designed architecture.**	3 (7.9%)	5 (13.2%)	8 (21.0%)	6 (15.8%)	9 (23.7%)	5 (13.2%)	2 (5.3%)	3.95	1.66
[Q10]	**During the life of the system/product, the original architectural vision has decayed due to numerous fixes and changes.**	2 (5.4%)	2 (5.4%)	1 (2.7%)	5 (13.5%)	11 (29.7%)	11 (29.7%)	5 (13.5%)	5.00	1.58

The following questions relate to the project management approach used in your organisation or project.

		1 Strongly disagree	2 Disagree	3 Slightly disagree	4 Neutral	5 Slightly agree	6 Agree	7 Strongly agree	Mean	Std. dev.
[Q11]	**The project uses a highly formal, structured approach with strict processes.**	8 (21.0%)	8 (21.0%)	6 (15.8%)	2 (5.3%)	7 (18.4%)	6 (15.8%)	1 (2.6%)	3.37	1.91
[Q12]	**The project uses an agile approach.**	4 (10.5%)	7 (18.4%)	4 (10.5%)	5 (13.2%)	8 (21.0%)	8 (21.0%)	2 (5.3%)	4.00	1.85
[Q13]	**Formal documentation plays a major role in our project (e.g., developers are given formal functional and/or technical specifications).**	14 (36.8%)	6 (15.8%)	1 (2.6%)	5 (13.2%)	7 (18.4%)	3 (7.9%)	2 (5.3%)	3.05	2.04

Part A, Page 2 of 6

Which of the following kinds of documentation do you regularly consume (i.e., read) while you perform your job?

		Available, and I use them	Available, but I do not use them	Not available	Not relevant
[Q14]	**Requirements specifications**	20 (55.6%)	5 (13.9%)	10 (27.8%)	1 (2.8%)
[Q15]	**User story cards**	5 (13.5%)	2 (5.4%)	24 (64.9%)	6 (16.2%)
[Q16]	**Functional specifications**	22 (59.5%)	5 (13.5%)	9 (24.3%)	1 (2.7%)
[Q17]	**Architectural design**	14	10	12	1

		Available, and I use them	Available, but I do not use them	Not available	Not relevant
	documentation	(37.8%)	(27.0%)	(32.4%)	(2.7%)
[Q18]	**Detailed technical design specifications**	13 (36.1%)	3 (8.3%)	19 (52.8%)	1 (2.8%)
[Q19]	**Data dictionaries**	9 (24.3%)	10 (27.0%)	17 (46.0%)	1 (2.7%)
[Q20]	**UML diagrams**	7 (19.4%)	7 (19.4%)	21 (58.3%)	1 (2.8%)
[Q21]	**Test plans and test matrices**	12 (33.3%)	8 (22.2%)	16 (44.4%)	0
[Q22]	**Test cases and test data**	20 (54.0%)	10 (27.0%)	7 (18.9%)	0
[Q23]	**Bug/defect reports**	31 (83.8%)	1 (2.7%)	4 (10.8%)	1 (2.7%)
[Q24]	**Comments in code**	29 (78.4%)	4 (10.8%)	3 (8.1%)	1 (2.7%)
[Q25]	**Informal documentation such as wiki pages**	23 (62.2%)	4 (10.8%)	10 (27.0%)	0

Which of the following kinds of documentation do you regularly write and/or update while you perform your job?

		Yes, I write/update these	No, I don't write/update these	Not relevant
[Q26]	**Requirements specifications**	13 (36.1%)	22 (61.1%)	1 (2.8%)
[Q27]	**User story cards**	7 (19.4%)	23 (63.9%)	6 (16.7%)
[Q28]	**Functional specifications**	15 (41.7%)	20 (55.6%)	1 (2.8%)
[Q29]	**Architectural design documentation**	19 (52.8%)	16 (44.4%)	1 (2.8%)
[Q30]	**Detailed technical design specifications**	16 (44.4%)	19 (52.8%)	1 (2.8%)
[Q31]	**Data dictionaries**	7 (19.4%)	24 (66.7%)	5 (13.9%)
[Q32]	**UML diagrams**	11 (30.6%)	21 (58.3%)	4 (11.1%)
[Q33]	**Test plans and test matrices**	17 (47.2%)	16 (44.4%)	3 (8.3%)
[Q34]	**Test cases and test data**	28 (75.7%)	8 (21.6%)	1 (2.7%)
[Q35]	**Bug/defect reports**	31 (83.8%)	4 (10.8%)	2 (5.4%)
[Q36]	**Comments in code**	33 (91.7%)	2 (5.6%)	1 (2.8%)
[Q37]	**Informal documentation such as wiki pages**	24 (64.9%)	12 (32.4%)	1 (2.7%)

Part A, Page 3 of 6

The following questions ask about comments in the existing code base of the major system or product that you work on.

		1 Strongly disagree	2 Disagree	3 Slightly disagree	4 Neutral	5 Slightly agree	6 Agree	7 Strongly agree	Mean	Std. dev.
[Q38]	**A documentation system such as Javadoc or Doxygen is used.**	11 (30.6%)	4 (11.1%)	3 (8.3%)	4 (11.1%)	4 (11.1%)	4 (11.1%)	6 (16.7%)	3.61	2.31
[Q39]	**Comments appear frequently in the source code.**	0	5 (13.9%)	5 (13.9%)	0	9 (25.0%)	13 (36.1%)	4 (11.1%)	4.89	1.63
[Q40]	**Comments in the source code tend to be accurate (they match the source code they describe).**	1 (2.8%)	2 (5.6%)	5 (13.9%)	5 (13.9%)	11 (30.6%)	6 (16.7%)	6 (16.7%)	4.81	1.58
[Q41]	**Comments tend to be out-of-date (e.g., the code was changed but the comments were not).**	5 (13.9%)	4 (11.1%)	3 (8.3%)	8 (22.2%)	9 (25.0%)	6 (16.7%)	1 (2.8%)	3.94	1.74
[Q42]	**I find the existing comments in the source code very helpful in understanding what the code does and how it does it.**	2 (5.6%)	2 (5.6%)	6 (16.7%)	7 (19.4%)	8 (22.2%)	8 (22.2%)	3 (8.3%)	4.47	1.61
[Q43]	**Comments are written in a consistent way across the source code.**	7 (19.4%)	9 (25.0%)	9 (25.0%)	2 (5.6%)	5 (13.9%)	2 (5.6%)	2 (5.6%)	3.08	1.76
[Q44]	**The general quality of comments is high.**	4 (11.1%)	8 (22.2%)	5 (13.9%)	5 (13.9%)	6 (16.7%)	7 (19.4%)	1 (2.8%)	3.72	1.80

Part A, Page 4 of 6

What are your personal opinions on code commenting?

		1 Strongly disagree	2 Disagree	3 Slightly disagree	4 Neutral	5 Slightly agree	6 Agree	7 Strongly agree	Mean	Std. dev.
[Q45]	**"Program comments are a form of heavy documentation which violate agile principles."**	12 (33.3%)	11 (30.6%)	3 (8.3%)	3 (8.3%)	1 (2.8%)	5 (13.9%)	1 (2.8%)	2.69	1.88
[Q46]	**"If code is written properly, it is self-**	5 (13.9%)	12 (33.3%)	6 (16.7%)	3 (8.3%)	4 (11.1%)	5 (13.9%)	1 (2.8%)	3.22	1.77

		1 Strongly disagree	2 Disagree	3 Slightly disagree	4 Neutral	5 Slightly agree	6 Agree	7 Strongly agree	Mean	Std. dev.
	documenting and doesn't need any comments."									
[Q47]	**"Documenting or commenting code is a waste of time because the documentation and code will drift out of sync as the code is changed."**	13 (36.1%)	12 (33.3%)	4 (11.1%)	2 (5.6%)	2 (5.6%)	2 (5.6%)	1 (2.8%)	2.39	1.64
[Q48]	**"I would like to document my code better, but deadline pressures make that impossible."**	5 (14.3%)	5 (14.3%)	5 (14.3%)	6 (17.1%)	9 (25.7%)	5 (14.3%)	0	3.69	1.68
[Q49]	**"I find that comments get in my way."**	13 (36.1%)	8 (22.2%)	3 (8.3%)	5 (13.9%)	6 (16.7%)	0	1 (2.8%)	2.64	1.69
[Q50]	**"Having better comments and documentation in the existing code would make my job easier."**	1 (2.8%)	1 (2.8%)	2 (5.6%)	5 (13.9%)	7 (19.4%)	11 (30.6%)	9 (25.0%)	5.36	1.51
[Q51]	**"I am very diligent about writing comments when I develop or maintain code."**	3 (8.6%)	2 (5.7%)	0	6 (17.1%)	11 (31.4%)	6 (17.1%)	7 (20.0%)	4.89	1.76
[Q52]	**"I consider myself more diligent than my colleagues/peers in consistently documenting my code."**	3 (8.6%)	1 (2.9%)	1 (2.9%)	13 (37.1%)	7 (20.0%)	6 (17.1%)	4 (11.4%)	4.54	1.62
[Q53]	**"I find comments at the top of classes or files are useful."**	2 (5.7%)	2 (5.7%)	2 (5.7%)	2 (5.7%)	7 (20.0%)	14 (40.0%)	6 (17.1%)	5.17	1.69
[Q54]	**"I find comments at the top of methods or functions useful."**	2 (5.7%)	2 (5.7%)	0	2 (5.7%)	6 (17.1%)	17 (48.6%)	6 (17.1%)	5.37	1.61
[Q55]	**"I find comments within methods or functions useful."**	1 (2.9%)	2 (5.7%)	3 (8.6%)	3 (8.6%)	10 (28.6%)	11 (31.4%)	5 (14.3%)	5.06	1.53

Part A, Page 5 of 6

The following questions ask about your experience with software maintenance and any difficulties that you may have encountered.

		1 Strongly disagree	2 Disagree	3 Slightly disagree	4 Neutral	5 Slightly agree	6 Agree	7 Strongly agree	Mean	Std. dev.
[Q56]	**I work on a system written by developers who have long left the organization.**	4 (11.4%)	1 (2.9%)	1 (2.9%)	7 (20.0%)	4 (11.4%)	6 (17.1%)	12 (34.3%)	5.06	2.01
[Q57]	**I sometimes find it difficult to understand what some parts of the code do or how they work.**	2 (5.6%)	1 (2.8%)	1 (2.8%)	5 (13.9%)	9 (25.0%)	14 (39.0%)	4 (11.1%)	5.11	1.51
[Q58]	**I often find it difficult to gain a "big picture" understanding of a system from reading the code.**	2 (5.6%)	2 (5.6%)	3 (8.3%)	1 (2.8%)	9 (25.0%)	11 (30.6%)	8 (22.2%)	5.17	1.73
[Q59]	**I feel that a lack of domain knowledge hinders my understanding of the system.**	2 (5.6%)	5 (13.9%)	4 (11.1%)	5 (13.9%)	8 (22.2%)	8 (22.2%)	4 (11.1%)	4.44	1.78
[Q60]	**I spend more time reading and debugging code than I feel I should have to.**	2 (5.6%)	5 (13.9%)	3 (8.3%)	4 (11.1%)	10 (27.8%)	9 (25.0%)	3 (8.3%)	4.50	1.73
[Q61]	**The software I work with has a higher bug rate than I am comfortable with.**	5 (13.9%)	1 (2.8%)	6 (16.7%)	7 (19.4%)	8 (22.2%)	7 (19.4%)	2 (5.6%)	4.14	1.76
[Q62]	**Quality issues in our software have led to deadline and/or budget overruns.**	3 (8.3%)	1 (2.8%)	1 (2.8%)	6 (16.7%)	8 (22.2%)	14 (39.9%)	3 (8.3%)	4.92	1.63

[Q63] Of the time you spend programming, what percentage of your time do you estimate you spend on "maintenance" development, i.e., reading and modifying existing code in order to fix defects or make enhancements?

Choice	Response count	Percentage
0%	0	0%
10%	2	7.4%
20%	5	18.5%
30%	5	18.5%
40%	3	11.1%
50%	3	11.1%
60%	2	7.4%

Choice	Response count	Percentage
70%	2	7.4%
80%	1	3.7%
90%	3	11.1%
100%	1	3.7%

Mean: 46.3%[18]

Part A, Page 6 of 6

[Q64] (Optional) Approximately how many years of experience in software development do you have? (free-form numeric response)

Choice	Response count	Percentage
1 years	1	2.9%
2 years	1	2.9%
3 years	2	5.9%
4 years	2	5.9%
5 years	4	11.8%
6 years	1	2.9%
8 years	1	2.9%
9 years	1	2.9%
10 years	3	8.8%
11 years	2	5.9%
12 years	1	2.9%
13 years	1	2.9%
15 years	5	14.7%
18 years	1	2.9%
20 years	1	2.9%
21 years	1	2.9%
22 years	1	2.9%
25 years	2	5.9%
30 years	1	2.9%
34 years	1	2.9%
43 years	1	2.9%

Mean: 13.2 years
Standard deviation: 9.8 years

[18] Assigning response code "1" to represent choice "0%", "2" for "10%", and so on up to "11" for "100%", the mean calculates to a value of 5.63, which corresponds to 46.3%.

[Q65] (Optional) Do you have any further personal opinions on comments and software documentation?

Respondent No.	Response
2	*Proper formal documentation should generally be left at the API level rather than at the individual implementation detail level.*
6	*After thinking about how much time I spend reading/fixing bugs, I just want to say "F*** My Life".*
7	*Commenting on commenting seems redundant.*
8	*should be consistency in terminology everybody is using to place their comments (glossary should go first)*
10	*good comments should add something more than the code is saying: //increment i i++; is utterly pointless!! Comments need to 'talk through' complex algorithms or express ~why~ something is being done ie intent! I'll often write simple comments first to sketch-out the code I'm writing.*
11	*I work for a multinational organisation, where a lot of code is developed in Japan. The comments the Japanese engineers add do not translate (or are not) translated to English when they release code to us, so the comments appear as random characters and are useless. To add to the frustration the functional documentation released by Japan has been translated by a translation package, so some of the translations do not make sense, making comprehension of the document difficult. Also, the documents do not contain enough information describing the intention of the function or how it should be used (i.e. code examples).*
12	*Stale documentation is better than no documentation, as long as you know it could be stale!*
23	*Technical documentation (i.e. for developers or support teams) should be limited to areas of detailed or complicated processing and ignore boiler plate or obvious areas. i.e. don't have developers go into intricate detail over the HTTP -> method binding (which is standard library code anyway) and focus on the intention and design of algorithms and processing functions. User documentation should not be written by developers, nor should any project team member misinterpret technical documentation for user consumption. The two disciplines are far too distinct and aim to achieve distinct ends.*
24	*There were a lot of questions concerning coding and inline comments, but 90% of software engineering is surely in developing the documentation (requirements, analysis, design) before a single line of code is written. After all, one wouldn't build a house without having first checked the land, considered the requirements then design it. So I would suggest casting your net a little wider to consider the really important questions, rather than judging the hypothetical house by its brick work. I hope this is constructive and helps in your thesis. All the best, (name withheld)*
38	*The key idea for me is that Documents, Comments, and Code should each provide a different level of abstraction on the System. During both design and maintenance a developer should start at the documentation level and have accessible tools to drill down into the*

Respondent No.	Response
	comments, then code. As I write I think the key is probably that documentation and code are not cohesive enough in our current tool sets, hence comments are the lazy option and show we are more code centric, as it is the code that delivers business results. Interesting survey and discussion point. Comments are in your face as you code, where as documentation is less accessible.
31	*To get good code it's necessary to have people use appropriate paradigms for appropriate parts of any project. If OO techniques are used to build complex databases or do serious constraint programming or indeed many other things failure will inevitably ensue: developers should use declarative paradigms such as relational algebra, functional programming, constraint programming, and logic programming where appropriate. Comments are important in these declarative paradigms as well as in procedural paradigms, but in my experience programmers educated in the use of declarative paradigms are more inclined to describe design intentions and design justifications in comments (even when writing in appalling languages like C++ and Java) and maintain them that programmers who limit themselves to OO languages (who tend to write only useless comments like "add A to B to get C" - the kind of thing that gives comments a bad name and leads to the "comments are not needed because code is self documenting" error (actually it would not be an error if all comments were like that). Looking at just the question of comments (and the maintenance of comments when code is modified) is not a good approach - it is necessary to get programmers using appropriate tools (and you'll be surprised how easy they find it to write useful comments when using even a flawed attempt at a declarative language like SQL).*
33	*Documentation is extremely important, however management at times do not understand this.*
34	*I feel that comments are extremely important. Many developers think they are a waste of time but I find that this is usually because they are assuming that other developers know what they know. For example, I know developers who won't comment beans because they feel there is no need. However, when a function is called 'isLive' or something - is what live? What does live mean? It makes sense to the developer, but not necessarily to anyone else. Documentation is something that most developers seem to hate. However it does need to be someone's responsibility to create and maintain it for the good of the organization*
36	*Prefer executable documentation i.e. unit/regression/acceptance tests*
37	*In deciding whether or not to apply a comment in code, one has to make assumptions about the level of knowledge of the reader. If a high level of knowledge is assumed, then one might choose not to comment the code or a section of it, if it is felt to be self-explanatory. The addition of comments themselves potentially add to the maintenance overhead of any future changes, if it is expected that those comments will also need to be updated. In small programming teams, it is difficult to make an economic case to have programmers*

Respondent No.	Response
	writing huge amounts of comments and other documentation, thus extending project development times - any "return on investment" with program commenting might be five or ten years away, when a piece of code is eventually revisited. If the quality of comments is not part of any staff appraisal process, there is no incentive to maintain them. Programmers and teams are most commonly judged on how the programs actually work, rather than what's behind the scenes.
38	*I see no value in comments or documentation. They only tell you what the programmer thought they were supposed to be doing which is less irrelevant than either what they have actually done or what they were actually supposed to do.*

Part B, Page 1 of 2

Thank you for completing Part A.

Would you please kindly read the following article before continuing with Part B?

Please copy and paste the following URL into a new tab or window of your browser:

http://www.kevinmatz.com/survey/IntroducingDesignIntentionDrivenProgramming.html [Note: This article is reproduced in Appendix G.]

When you have finished reading the article, please return to this page and continue to the next page of this questionnaire.

Part B, Page 2 of 2

The following questions ask for your personal opinions about the proposed Design Intention Driven Programming and Java with Intentions schemes.

		1 Strongly disagree	2 Disagree	3 Slightly disagree	4 Neutral	5 Slightly agree	6 Agree	7 Strongly agree	Mean	Std. dev.
[Q66]	**"I would be willing to give this approach a try, at least on a trial basis."**	7 (20.0%)	5 (14.3%)	1 (2.9%)	5 (14.3%)	5 (14.3%)	10 (28.6%)	2 (5.7%)	3.97	2.08
[Q67]	**"I see no advantage to this approach over normal comments or existing documentation systems."**	1 (2.9%)	9 (25.7%)	7 (20.0%)	6 (17.1%)	5 (14.3%)	4 (11.4%)	3 (8.6%)	3.83	1.71
[Q68]	**"The practice of recording design intentions before writing code is a**	0	1 (2.9%)	2 (5.7%)	2 (5.7%)	5 (14.3%)	19 (54.3%)	6 (17.1%)	5.63	1.19

		1 Strongly disagree	2 Disagree	3 Slightly disagree	4 Neutral	5 Slightly agree	6 Agree	7 Strongly agree	Mean	Std. dev.
	sensible idea."									
[Q69]	**"Recording design intentions before writing code might be nice in theory, but is impractical for real-world projects."**	2 (5.7%)	5 (14.3%)	11 (31.4%)	5 (14.3%)	8 (22.9%)	3 (8.6%)	1 (2.9%)	3.71	1.49
[Q70]	**"Developers would resent being forced to write documentation. It would only cause frustration and slow down projects."**	2 (5.7%)	4 (11.4%)	5 (14.3%)	7 (20.0%)	10 (28.6%)	5 (14.3%)	2 (5.7%)	4.20	1.59
[Q71]	**"Adding specialized constructs for design intentions to programming languages would help stress the importance of proper documentation."**	3 (8.6%)	3 (8.6%)	2 (5.7%)	5 (14.3%)	9 (25.7%)	9 (25.7%)	4 (11.4%)	4.63	1.78
[Q72]	**"Instances of design patterns should be documented for ease of understanding by later maintainers."**	1 (2.9%)	2 (5.7%)	1 (2.9%)	5 (14.3%)	12 (34.3%)	9 (25.7%)	5 (14.3%)	5.06	1.45
[Q73]	**"This approach could potentially work well in agile projects."**	1 (2.9%)	4 (11.4%)	3 (8.6%)	13 (37.1%)	2 (5.7%)	11 (31.4%)	1 (2.9%)	4.37	1.52
[Q74]	**"This approach could potentially work well in formal, documentation-driven projects."**	1 (2.9%)	1 (2.9%)	0	6 (17.1%)	8 (22.9%)	16 (45.7%)	3 (8.6%)	5.26	1.29
[Q75]	**"Software projects consistently documented in this way would be easier to understand than projects developed using traditional techniques."**	1 (2.9%)	3 (8.6%)	5 (14.3%)	8 (22.9%)	3 (8.6%)	11 (31.4%)	4 (11.4%)	4.66	1.66

[Q76] (Optional) Do you have any remarks on potential problems with the proposed approach?

Respondent No.	Response
1	*In older systems, where a large amount of code has already been written, it would be difficult and time consuming to bring all comments up to date. This would be especially frustrating if intention comments were required to compile code.*
2	*I think Test-driven development is a better approach. You define what your method is supposed to do and then implement runnable tests to make sure that it does what it is supposed to do. This way, the design intentions can be a header block, but when the code and the docs diverge you know that test cases are really what counts for correctness.*
3	*The intention of this approach is good, but may have difficulty to implement in real-life projects.*
6	*Overuse of forced inline comments can sometimes be detrimental to understanding the content as a whole by virtue of making the code longer and more verbose*
9	*Comments shouldn't be compiler objects. They should be free form. That's the point of comments. I like the idea of having a standard methodology for documenting code though, just don't implement it through the compiler. Better implemented through documented design/development standards. Then, if developers still don't follow them, fire them!*
10	*Yet ANOTHER syntax to learn?! And the more it looks like code the more it's going to have code-type problems ie bugs. why not extend javadoc? aka JUnit's @Test*
11	*A lot of function headers and code use cut and paste and then modification to speed up the work. How would this approach affect your proposed approach?*
12	*Yes. I've been doing it myself, for years.*
13	*No advantage over existing documentation (Javadoc etc) or tools (Checkstyle) No advantage to proper process, planning meetings, TDD, pair programming, peer reviews etc. The most likely reason for duplication of documentation is duplication of code. Enforcing principles of any kind is never a good idea, the team should buy into anything being proposed. This ensures it gets done correctly as the team believes in its value. If people don't have time to write comments how do they have time to write intentions? Enforcing them via the compiler would soon get switched off! What makes people more likely to update intentions compared to regular documentation? Would require a separate language and compiler for popular languages such as Java and C#! (however in Java you could probably do something similar with Annotations)*
15	*Design intentions could be stored elsewhere, in technical specifications, for example. Embedding this information within code may cause unnecessary clutter.*
17	*Developers would resist until they could see a clear benefit to themselves. Unfortunately those developing a system are often not the*

Respondent No.	Response
	ones who maintain it.
18	*It doesn't add any clear benefit, and creates more work. If quality has suffered due to time constraints then the time would not be available to adopt this approach. It also clutters the code with the implementation of 'documentation' interfaces. In any good team I believe communication and inline comments are enough. A good design is usually easy to pick up and speaks for itself.*
19	*The main problem I could potentially see is buy-in by all programmers. Many programmers see comments as a waste of time - though I find it very useful when there are well laid out comments. A culture must be created where this is the norm. As well, there may be issues with deciphering what the comments actually say. A programmer might just do a half assed job in writing these comments. Even after passing the compiler test, it may need a code reviewer to look though the comments.*
23	*Abusing any language to add additional documentation constructs is mostly counter-productive. Documentation is one thing, code assets another. Intermingling the two too tightly causes problems with release and other maintenance headaches when edge and corner cases evolve. Requirements are (or should) be a fixed element once a cycle is complete, these can be referred back to at any time when proposed changes in later versions do not accomplish the intended result to repair functionality, if held in code this would result in potential regression issues as it would require an SCM answer to a process problem.*
24	*The proposed approach is basically 'pseudocode' which we used back in the 1980's for procedural design. Yes there should be and indeed has to be, documentation a long time before coding. This is amply served by structured methods, such as UML, OOAD etc. There is no shortcut to doing the leg-work up front - techies tend not to like it, but it is essential, plain fact! I fail to see how one can write an intention in code, without having considered the context of the classes etc. that are being 'typed' - it sounds more like "flying by the seat of one's pants".*
26	*While I understand I would expect that over time the system would degenerate into standard comments with the extra syntax required to write them. If coders are not disciplined enough to write decent comments in the first place why would extra syntax help?*
28	*With no strong automated testing/checking of the descriptive content of intentions I still see an issue as with Javadoc that different developers will describe their intentions in different ways, or worse leave the intention description blank (you have a solution/idea for this but I am sceptical). The other issue I have with this solution is that code and implementation are in the same physical file. You would require tools (i.e. IDE to manage the display of intentions or code). I'm wondering if some kind of reference/linking mechanism would be better?*
31	*It's trying to make a currently overhyped paradigm (Object Orientation) more useful, by addressing one overhyped and not*

Respondent No.	Response
	terribly useful language in that paradigm. Maybe it would succeed if it went for an OO language other than Java (perhaps C# or Smalltalk) but it would still be too restrictive to be useful because there is no imaginable way serious big systems can be written using OO language alone. If it were extended to cover F#, C#, LML, SQL, Prolog, Parlog, and a few other languages it might help with serious system building, but confined to Java it's not going to be useful.
34	*I think that developers would be frustrated and annoyed at being forced to write comments (however, that doesn't mean they shouldn't). It is sometimes difficult to know the intention, especially in agile projects, where planning is minimal*
35	*Unless the compiler can detect whether the comments are correct or not then the approach is vulnerable to developers just adding the bare minimum comments.*
36	*The intentions are just as likely to be incorrect/out of date as comments. There's no way to check/enforce the correctness of any of the intention inheritance/implementation relationships. Increased incidence of value-less comment 'noise'*
37	*The comments about intentions are not next to the actual code that they relate to. Cross-referencing is required. Mandatory requirements to write a certain amount of text based on an arbitrary formula would just get infuriating, and programmers might simply type gobbledegook or (if management objected to that) standard stock phrases that actually mean very little. Alternatively, such requirements might result in the whole approach being abandoned.*
38	*It's difficult enough to get people to write things twice (code + acceptance test). Adding a 3rd (design intention) is likely too much.*

[Q77] (Optional) Do you have any remarks on potential benefits of the proposed approach?

Respondent No.	Response
1	*Definitely would make code easier to read and to understand. Discussion of why a particular approach was chosen would be especially useful when the approach is complicated. Highlighting relationships between pieces of code using comments would also particularly useful, especially in complex systems.*
2	*I think it might be appropriate in very complex/domain specific code which has a very high likelihood of being misunderstood or broken during a correction*
6	*Descriptive comments at the top of functions and structures really ought to be mandatory. Good on you for trying to do it.*
10	*nice idea, and I like the idea of capturing what experienced developers do anyway ie comment to express intent and meaning. A javadoc type system could use these to write a reverse-engineered design spec to check against original spec.*
11	*If the DIDP comments could be automatically extracted to generate*

Respondent No.	Response
	documentation that would be beneficial
12	*Fit it in with test driver design, so you lay out the intention and the test!*
17	*Because the documentation is in the code it adds to more traditional design docs. ie it shows how the software requirements are implemented.*
18	*documentation (although this would be better placed in JavaDoc or system documentation). It might add benefit to more junior coders or for students, although I think they would be better off learning design patterns and API's, algorithms, etc.*[19]
19	*It makes code easier to understand, especially if you are new to the team. Some applications have a steep learning curve, and if the intention of a particular function, or series of functions is written down, it can only help the developer. It is also useful when reading functional/technical specifications to be able to follow along to see what parts of the code are doing what.*
23	*A corporate or team development standard based on the principles you've started on could lead to a shallower learning curve for newer or less familiar team members.*
26	*Writing design intentions is a very good principle and would help. I am not sure more syntax is needed though.*
28	*As discussed early using this method would ensure (or help to ensure) a level of documentation is closely related to the code implementation. The approach you have taken on making a comment an object and allowing them to be syntactically linked to one another is good.*
31	*Obviously if the compilation system refuses to compile stuff that doesn't have design intention (and preferably also design justification and requirement description) comments there is some chance that it will encourage people to write useful comments.*
32	*I think it is a great idea and would create a better quality and maintainable final product. However although maintenance costs of the product would be decreased the initial cost of development would be higher.*
34	*It would enforce documentation, although general code styles could do the same without the frustrating compiler checking*
36	*In a rigorous environment, I can see that the reuse of intentions (e.g. where a pattern catalogue is in use) might be seen as an advantage.*
37	*It makes a statement about the project's values - that developers are expected to frequently document their work.*
38	*Sceptical but could be of some use if coupled with language semantics.*

[19] The beginning part of this remark appears to have been lost.

[Q78] (Optional) Do you have any suggestions or further comments on the proposed approach?

Respondent No.	Response
1	*Could be helpful to include a list of parameters and what they mean/represent/are used for in method-level comments.*
7	*This would be met with initial resistance, but this would logically pass as developers benefit from the comments. Comments should be mandatory at the beginning of programming constructs such as loops, ifs with >n lines, cases, etc. A check in the editor/or compiler that prevents long segments of code with no comments from being generated would be useful.*
10	*would the intent be written first, validated, then code written?*
11	*Can you add references to other documentation such as UML diagrams in the DIDP comments to help explain the code functionality?*
12	*Lay it out by writing a concise comment on that you are going to do. It clears the mind. The do it. Then make the comment say what you have done. It's easy!*
18	*annotations would be better than this approach, for brevity.*
23	*However, there is no way a consensus across all language users will yield a consistent construct or usage to make this a staple of programming in 'x'. As with all methodologies, ideologies and fads (be it agile, waterfall, centralised, decentralised, comment, annotation, or whatever) most software houses will not fully implement any single pattern or method, but will cherry pick from them all until they have something that works for them.*
24	*Consider the whole design/documentation work-flow. If a subsequent software engineer needs to find out what makes a class tick, the external documentation should give him every thing he (or she) needs - with full explanation of intention and all in context. I hope this is helpful in some way - you did ask for comments and I hope they're constructive. By the way - I design software control systems for particle accelerators and speak to international organisations on the subject. Please feel free to contact me if you would like to discuss at all. Best regards, <name withheld>*
26	*You could achieve similar results by analysing the code and ensuring there is a suitable ratio of comments to code. This is a little less OO specific too.*
28	*I have provided suggestions in my previous comments. One thing you might like to add to your thesis is a discussion on international comments? I can read Java in any language even if the comments are in French, German, or Spanish which I can't read. Are comments just a way to help people who can't read java (cynical joke!)*
31	*Make Intention, Requirement, and Justification comments three different kinds of thing. Maybe require each at some level (Requirement at a many files level; Intention at File level, maybe at function level, data structure level [that could be object or schema or ...]; justification - well, clearly there are obvious places for it in the DDL component of SQL, and in old-style ADT languages, and in*

Respondent No.	Response
	constraint solving languages; in OO languages it's probably hard to automate discovery of places where compiler should enforce its presence - too much enforcement would just encourage people to make useless comments, worse than nothing).
32	*This would make business sense if the people developing the code were also going to be tasked with maintaining it. Or if the purchaser of the product was willing to pay a premium for something that was less expensive to maintain*
34	*I think it's a good idea but I don't like the implementation of having to implement interfaces. Perhaps annotations would be better?*
36	*Interesting idea, but IMO flawed.*
37	*Intention comments are described as being object-oriented. I do not work in an object-oriented programming environment, so any approach also needs to be applicable to legacy code. Rather than attempting to force good documentation through compiler-enforced checks, and extra programming constructs, a better approach might be for management to set out clear statements about the commenting approach and standards required, with the standards actually affecting the performance rating of the staff involved.*

Final page

Thank you for participating in this survey! Your help is very greatly appreciated!

Responses to this survey are anonymous. If you wish to inform me that you've completed the survey, please send an e-mail to kevin@kevinmatz.com and mention that the magic word is "CACTUS".

(This is the same magic word for all participants – it cannot be used to personally identify you.)

Thanks again for your help!

Sincerely,
Kevin Matz

Appendix F: Raw survey response data

Table 43 and Table 44 present the raw data from numeric questions on the survey questionnaire. Please see Appendix E for the responses to the open-ended free-text questions (Q65, Q76, Q77, Q78).

Question numbers marked with asterisks are not Likert scale questions.

Table 43: Numeric response data for participants 1 through 20

	Participants																			
Question	**1**	**2**	**3**	**4**	**5**	**6[20]**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Q01*	7	1	6	4	5	1	4	2	3	5	2	3	1	5	6	7	2	3	3	1
Q02	4	4	6	4	6	2	7	5	6	7	5	3	1	5	7	6	1	6	6	3
Q03	6	5	7	1	3	3	7	6	3	7	6	5	6	5	6	6	6	6	4	5
Q04	3	3	7	3	6	3	4	3	2	7	6	4	6	6	7	7	7	6	6	6
Q05	4	5	2	2	2	4	1	4	2	7	1	3	1	1	2	1	1	2	2	6
Q06	7	6	5	.	4	6	5	3	2	7	4	3	2	2	4	2	6	6	4	6
Q07	7	6	2	5	2	7	3	.	6	5	1	2	1	4	2	2	6	2	3	6
Q08	4	6	3	5	4	6	2	5	1	4	1	3	6	2	2	2	5	3	4	3
Q09	5	5	4	5	2	5	3	5	1	4	2	2	5	1	1	2	4	3	5	6
Q10	4	5	6	5	7	1	7	5	6	5	6	6	2	3	7	6	5	1	4	4
Q11	6	5	5	1	3	1	3	2	1	6	1	2	1	3	2	2	5	5	6	2
Q12	6	6	6	2	3	5	6	2	1	4	6	5	7	2	2	4	4	5	3	6
Q13	4	5	6	1	1	4	2	2	1	7	1	1	1	5	1	2	1	1	5	2
Q14*	1	1	1	1	3	.	1	1	2	2	3	3	3	2	1	1	1	3	1	1
Q15*	4	1	3	1	3	.	3	1	4	3	3	3	3	3	3	3	4	3	3	3
Q16*	2	1	1	1	3	.	1	1	3	2	1	3	3	1	1	1	1	1	1	1
Q17*	1	1	2	1	3	.	1	1	2	2	3	1	3	3	2	3	3	3	1	2
Q18*	1	3	1	3	3	.	1	.	2	2	3	1	3	3	3	3	3	3	3	3
Q19*	1	2	2	1	3	.	1	3	3	3	3	3	1	1	1	2	3	3	2	3
Q20*	1	1	3	3	3	.	3		2	3	3	3	1	2	3	3	3	3	3	3
Q21*	1	1	3	2	3	.	1	1	3	2	3	3	3	1	3	3	3	3	1	3
Q22*	1	2	2	2	3	.	1	1	1	2	3	3	3	1	1	1	1	3	1	1
Q23*	1	1	1	3	1	.	1	1	1	1	1	1	1	1	1	1	3	1	1	1
Q24*	1	1	1	1	3	.	1	1	1	1	1	1	1	3	1	1	1	1	2	1
Q25*	1	1	1	1	3	.	1	1	1	1	3	1	3	3	2	1	3	3	3	1
Q26*	1	1	1	2	1	.	2	2	2	2	2	3	2	2	2	2	1	2	2	2
Q27*	2	1	3	1	2	.	2	1	2	2	2	3	2	2	2	2	3	2	2	2
Q28*	1	2	1	2	1	.	2	2	2	2	2	3	2	2	2	1	1	2	2	2
Q29*	1	1	1	1	2	.	1	2	2	2	1	2	2	2	2	2	3	2	2	1
Q30*	1	1	1	2	2	.	1	2	2	2	1	2	2	2	2	2	3	2	2	2
Q31*	1	1	2	2	2	.	2	2	2	2	2	3	3	2	2	2	3	2	2	2
Q32*	1	1	2	2	1	.	2	1	2	2	1	3	1	2	2	2	3	2	2	2
Q33*	1	1	2	2	2	.	1	1	2	2	2	3	2	1	2	2	3	2	1	2
Q34*	1	1	1	2	1	.	1	1	1	2	2	3	2	1	1	1	2	1	1	1
Q35*	1	1	1	2	1	.	1	1	1	1	1	1	1	1	1	1	3	1	1	1

[20] While respondent 6 did not complete the survey, the responses to the questions answered appear legitimate and the decision was made to retain these responses.

	Participants																			
Question	**1**	**2**	**3**	**4**	**5**	**6[20]**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Q36*	1	1	1	1	1	.	1	1	1	1	1	1	1	2	1	1	1	1	2	1
Q37*	1	1	1	1	2	.	1	1	1	1	2	1	2	2	2	1	3	2	2	1
Q38	4	7	1	6	2	.	1	6	6	5	5	4	1	4	2	1	1	5	4	2
Q39	6	6	5	6	5	.	2	6	6	5	3	6	6	3	2	5	6	5	5	2
Q40	6	6	5	5	5	.	3	5	4	4	1	5	7	2	4	5	6	5	5	5
Q41	6	3	3	4	4	.	5	5	4	5	6	4	1	6	5	5	4	7	4	4
Q42	6	5	5	5	6	.	4	4	3	4	1	3	4	4	2	5	7	7	5	3
Q43	6	5	2	2	4	.	2	2	2	4	1	1	6	3	1	3	5	2	5	1
Q44	6	5	2	2	4	.	2	5	2	3	1	3	6	3	1	5	5	2	6	4
Q45	4	1	2	1	1	.	3	3	6	1	1	1	2	6	2	2	2	6	2	6
Q46	4	2	6	5	4	.	5	3	5	2	1	2	2	3	2	1	4	6	3	6
Q47	4	2	2	1	2	.	5	2	2	2	2	1	1	3	2	2	3	6	3	5
Q48	4	5	6	1	2	.	2	5	2	4	4	4	1	3	6	5	5		5	5
Q49	4	4	1	2	1	.	2	4	2	2	3	2	1	3	1	5	1	1	5	5
Q50	4	7	7	6	7	.	4	6	6	5	5	6	4	6	7	7	5	5	6	3
Q51	4	5	5	7	6	.	5	5	7	5	4	7	6	4	6	5	5	7	5	1
Q52	4	4	4	5	6	.	4	4	7	6	4	7	.	4	6	5	5	1	5	2
Q53	4	6	6	6	6	.	6	6	7	5	6	7	2	6	6	7	4	5	5	3
Q54	4	6	7	6	6	.	6	6	7	5	6	7	2	6	6	7	5	6	6	5
Q55	4	6	6	6	6	.	3	6	3	5	5	7	6	6	7	5	5	6	6	2
Q56	5	4	7	5	4	.	7	6	6	7	4	6	1	.	7	7	1	7	4	4
Q57	4	5	6	3	6	.	6	6	5	7	5	6	1	6	6	6	5	7	6	4
Q58	3	5	7	3	6	.	6	5	5	7	2	7	6	5	6	2	5	7	7	5
Q59	4	4	6	2	6	.	5	7	2	7	1	3	1	3	6	5	5	7	6	6
Q60	5	7	5	2	6	.	3	6	6	7	2	4	1	4	6	6	5	7	6	4
Q61	3	4	4	5	6	.	5	5	7	4	1	3	1	5	6	4	5	6	6	3
Q62	4	5	6	5	6	.	5	6	7	4	5	4	1	6	7	6	5	6	6	6
Q63*	.	6	4	5	7	.	8	.	4	.	5	4	3	.	10	10	2	.	8	5
Q64*	.	15	10	6	15	.	11	4	5	25	21	25	15	5	15	10	2	5	4	3
Q66	4	.	7	1	5	.	2	4	2	7	1	2	6	6	6	6	6	4	6	3
Q67	4	.	2	7	5	.	6	2	7	2	3	6	4	3	2	3	5	4	4	5
Q68	4	.	6	6	6	.	6	5	2	7	6	6	6	4	6	6	6	7	6	5
Q69	4	.	3	3	2	.	3	5	4	2	5	2	2	5	3	3	3	7	5	5
Q70	4	.	5	5	4	.	6	3	4	2	4	5	3	3	4	5	3	7	5	6
Q71	4	.	6	2	6	.	5	5	2	7	3	1	4	5	6	5	5	7	6	3
Q72	4	.	6	2	6	.	3	4	2	5	5	6	5	5	6	5	5	7	6	5
Q73	4	.	6	2	6	.	2	6	2	4	4	4	2	6	6	4	6	7	6	3
Q74	4	.	6	2	7	.	6	6	6	6	6	5	5	6	6	6	5	7	6	5
Q75	4	.	6	2	6	.	4	6	2	6	6	3	4	6	7	6	5	7	6	4

Table 44: Numeric response data for participants 21 through 38

	Participants																	
Question	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38[21]
Q01*	4	4	5	5	4	4	4	4	5	1	4	4	3	5	2	4	2	3
Q02	6	2	2	5	5	6	2	5	3	4	7	5	4	4	6	7	6	5
Q03	5	7	6	6	6	6	2	5	7	4	7	7	5	7	6	7	6	4
Q04	7	7	6	5	6	2	6	6	5	4	7	5	4	7	5	7	6	7
Q05	4	6	1	6	1	6	1	2	6	4	7	1	5	5	6	6	6	7
Q06	3	6	4	5	2	6	3	4	6	4	7	3	6	4	6	7	6	7
Q07	2	6	1	5	2	5	1	2	2	4	7	5	7	3	6	7	6	7
Q08	5	2	3	2	4	.	6	6	6	4	3	4	4	6	7	6	6	6
Q09	3	3	3	3	3	3	2	5	7	4	6	4	6	4	7	6	6	5
Q10	6	5	6	7	5	6	6	.	5	4	5	4	5	7	6	2	6	5
Q11	3	3	4	2	1	2	1	6	2	4	5	1	6	3	6	7	5	5
Q12	5	7	3	5	3	6	1	2	2	4	1	1	4	2	6	5	5	5
Q13	6	1	3	1	1	2	1	5	5	4	5	1	7	4	6	4	2	5
Q14*	1	4	3	1	3	1	3	.	1	2	1	3	1	1	1	1	3	2
Q15*	3	1	3	1	3	3	3	4	3	2	4	3	2	4	3	3	3	3
Q16*	1	4	3	3	3	1	1	2	1	2	1	3	1	1	1	1	3	2
Q17*	2	4	3	1	3	1	3	2	1	2	1	3	1	3	1	1	2	2
Q18*	1	4	3	3	3	3	1	1	1	2	1	3	1	3	1	1	3	1
Q19*	1	2	1	3	3	3	4	2	1	2	3	3	2	3	2	3	3	2
Q20*	2	2	3	3	3	1	3	3	4	2	1	3	2	3	1	1	3	2
Q21*	3	2	3	2	3	2	1	.	1	2	2	3	1	1	2	1	3	1
Q22*	1	2	3	2	1	2	1	1	1	2	1	3	2	1	2	1	1	1
Q23*	1	1	3	1	1	1	1	1	1	2	1	3	4	1	1	1	1	1
Q24*	1	2	3	1	1	1	1	1	1	2	1	1	4	1	2	1	1	1
Q25*	1	1	3	1	1	1	3	2	1	2	1	1	1	1	2	1	3	1
Q26*	2	.	1	1	2	2	2	2	1	1	2	1	1	1	2	1	2	2
Q27*	2	1	2	1	2	2	2	.	1	1	2	2	2	3	3	2	2	3
Q28*	1	.	1	2	2	2	1	2	1	1	1	1	1	1	2	1	2	2
Q29*	2	.	1	1	2	1	2	2	1	1	1	1	1	1	1	1	1	2
Q30*	2	.	1	2	2	2	1	2	1	1	1	1	1	1	1	1	1	2
Q31*	2	.	1	2	1	2	2	2	1	1	2	2	2	3	3	2	1	2
Q32*	2	.	2	2	2	1	2	2	3	1	1	2	1	3	2	1	2	2
Q33*	1	.	1	2	2	2	1	3	1	1	1	2	1	2	1	1	1	1
Q34*	1	1	1	2	1	2	1	1	1	1	1	1	1	1	2	1	1	1
Q35*	2	1	1	1	1	1	1	1	1	1	2	2	3	1	1	1	1	1
Q36*	1	.	1	1	1	1	1	1	1	1	1	1	3	1	1	1	1	1
Q37*	1	1	2	1	1	1	2	2	1	1	1	2	1	1	2	1	1	1
Q38	2	1	1	1	7	6	1	1	1	3	7	5	3	3	7	7	7	.
Q39	3	5	2	6	6	7	2	5	6	3	7	5	3	6	7	6	7	.
Q40	4	3	6	4	6	6	2	5	7	3	7	3	5	3	7	7	7	.
Q41	5	5	2	4	1	2	5	2	1	3	1	6	6	5	6	2	1	.
Q42	3	1	5	2	6	6	6	6	5	3	6	3	4	4	7	6	5	.
Q43	2	1	3	1	3	3	3	2	2	3	5	1	3	5	7	3	7	.
Q44	2	1	3	2	4	6	2	4	6	3	5	1	6	4	6	5	7	.
Q45	2	7	4	6	2	1	1	5	2	1	2	1	3	2	4	1	1	.

[21] While respondent 38 did not complete the survey, the responses to the questions answered appear legitimate and the decision was made to retain these responses.

	Participants																	
Question	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**[21]
Q46	2	7	5	6	3	3	1	2	2	2	2	6	1	2	2	1	3	.
Q47	2	7	4	6	1	1	1	2	1	3	1	1	1	2	1	1	1	.
Q48	5	1	6	2	6	5	3	4	2	4	1	5	3	3	6	5	1	.
Q49	1	7	5	5	4	1	1	2	1	5	1	3	2	2	4	1	1	.
Q50	6	2	5	3	5	6	7	7	4	6	6	6	7	5	7	1	4	.
Q51	5	1	2	2	4	6	7	5	6	7	6	4	4	5	.	1	7	.
Q52	5	1	4	1	7	6	6	4	4	6	4	5	4	5	3	4	7	.
Q53	6	2	6	5	5	7	6	.	5	5	6	7	3	6	7	1	1	.
Q54	6	2	6	5	5	7	6	.	6	4	6	6	5	6	7	1	1	.
Q55	5	2	6	4	5	7	6	.	7	3	5	5	5	4	5	1	7	.
Q56	5	7	2	7	7	6	6	7	4	1	7	6	4	5	3	7	1	.
Q57	5	7	6	7	6	6	6	4	5	2	6	5	5	4	4	5	1	.
Q58	6	6	5	7	5	6	7	4	1	3	6	7	6	6	5	6	1	.
Q59	5	2	5	3	6	6	6	5	2	4	5	2	4	3	4	5	7	.
Q60	5	6	2	5	5	5	6	2	5	5	3	5	6	4	3	2	1	.
Q61	5	2	4	6	7	3	5	1	4	6	3	5	4	6	3	1	1	.
Q62	6	6	4	6	6	5	5	4	6	7	3	2	4	6	5	1	1	.
Q63*	7	9	.	3	11	3	3	10	4	.	3	6	6	4	.	.	2	.
Q64*	1	5	22	34	12	11	10	18	43	9	13	8	.	20	3	30	15	.
Q66	6	5	2	1	5	6	6	6	1	1	5	2	4	4	5	1	1	.
Q67	3	5	4	6	6	2	2	3	2	2	7	3	3	5	4	2	1	.
Q68	6	6	6	6	6	6	6	5	7	3	7	6	3	5	5	7	7	.
Q69	3	2	5	6	5	4	3	5	1	4	6	3	4	3	6	1	3	.
Q70	5	6	6	4	6	5	3	2	1	5	7	5	1	4	2	2	5	.
Q71	5	6	6	2	5	6	5	4	5	6	6	7	4	4	7	1	1	.
Q72	6	6	5	6	5	5	5	4	7	7	7	4	4	6	5	7	1	.
Q73	5	6	4	4	4	4	4	4	4	6	6	5	3	6	3	4	1	.
Q74	7	6	4	5	6	6	6	6	4	5	6	4	5	5	4	4	1	.
Q75	5	3	4	3	6	6	6	7	3	4	7	3	4	5	4	2	1	.

Appendix G: The article included with the survey

(The following article was hosted at http://www.kevinmatz.com/survey/IntroducingDesignIntentionDrivenProgramming.html *and a link was made available to survey participants.)*

Introducing *Design Intention Driven Programming* and *Java with Intentions*

(Version 1.1)

In team-based software development, and especially in the maintenance of legacy software systems, developers spend much of their time on the activity of *program comprehension*.

In order to make a change or an enhancement to an existing system, a developer must first understand the code to be modified. It can be very time-consuming to read and analyze complex code artefacts to understand what they do and how they work, and to determine how to modify them. A developer reading someone else's code must try to piece together the *intentions* of the original author – that is, what the author had in mind when writing the code.

If the code has been written carefully, by using good naming of identifiers (variables, methods, and classes), by employing a clear logical structure, and by using commonly-known design patterns, later developers can often figure out what the code does and how it works. Due to time pressure, inexperience, or other factors, however, many systems were not constructed using "best practices", and even in "good" systems, the code quality and structure of the system have often deteriorated over time due to countless changes and quick fixes made by many developers. And often code is extremely complex to understand, simply because it is doing some very complex things.

In the absence of suitable documentation, a developer making changes to existing code that he or she is not familiar with must typically do time-consuming analyses, formulating and testing hypotheses about what is going on. Under deadline pressure, a developer may have to make assumptions and carry out changes based on an incomplete understanding of the code being modified, which often leads to further errors and defects that must be corrected again at a later date.

Obviously, developers can give clues and insights to future developers by writing comments that explain what the code does, how it works, and why it was designed that way. Comments could even contain hints about how to make future expected changes.

In many legacy systems, however, comments are non-existent or are of poor quality. And external documentation such as functional specifications, technical design documentation, or data dictionaries, that might provide additional clues, can become lost or out of date.

When we build new systems that will be maintained over long periods of time, is there any approach we can take to help reduce the burdens of program comprehension and missing documentation?

Introducing *Design Intention Driven Programming*

Design Intention Driven Programming (DIDP) is an approach that encourages developers to record their *design intentions* before they write a piece of code. Developers record their design intentions for a component of a system simply by writing a brief description of what they plan for that component to do, and how it will do it. The description may also include *rationale* – a justification of why one particular solution was chosen over alternative solutions.

In the DIDP approach, a system is written using a programming language that has been extended with special language constructs called *intention comments* that aid in the recording of design intentions. For example, if you were constructing a system based on Java, you would use a language called *Java with Intentions* that extends the Java language syntax with support for intention comments.

In DIDP, before you write a new class, a new method, or a section of code within a method, you should write an intention comment for it, briefly explaining what you plan to do and how the code will work. When you then write the corresponding code, if you have to diverge from your plan, you should then update the intention comment to match the implementation.

Now you're probably asking, why can't we just use normal comments to do this? Well, of course, you certainly could. Intention comments are very much like existing comments in programming languages – they store free-form textual explanations – but they have several unique features:

- Intention comments encourage documentation re-use and help prevent duplication, because they are object-oriented constructs that support the inheritance mechanism. Intention comments can have fields containing either text or references to classes, objects, or other intention comments. An intention comment can be *abstract*, serving as a template, and other intention comments can extend it and fill in the required fields. This is a particularly suitable way of documenting instances of design patterns, as we will see shortly.

- Because intention comments can be named and can refer to each other, and because intention comments representing requirements and goals can be created, interlinked networks of intention comments can be used to represent a design for the system at different levels of abstraction.

- Intention comments can be made mandatory, to enforce the documentation of the code. The compiler enforces that intention comments are present by generating an error message if a class or method or a long section of code within a method does not have a corresponding intention comment associated with it.

  To prevent a programmer from just entering an empty or short gibberish comment to satisfy the compiler, it is envisioned that the compiler will compute a numeric information content metric for the intention comment, and a numeric complexity metric for the code that the intention comment describes. (The simplest metrics are simple counts of characters or lines, but more complex schemes are possible.) If the compiler determines that the information content of the comment is not enough to match the corresponding code (for example, a comment containing just five characters would be considered insufficient to describe a class containing a thousand lines of code), a compiler error will be generated. The metrics and thresholds could be configured for each project. Obviously, this scheme is imperfect and would be easy to circumvent, and cannot guarantee the quality of the comment text, but it is an attempt to "enforce" commenting. (The DIDP approach is obviously not suitable for all projects and teams, but it is suitable as an aid that could be adopted by project teams that value documentation and wish to ensure a certain standard of commenting.)

As this discussion has been very abstract, let's now briefly see what intention comments actually look like in the *Java with Intentions* language.

Introducing *Java with Intentions*

The Java with Intentions language (JWI) simply takes the existing Java programming language and adds support for intention comments. In JWI, intention comments have two basic forms:

1. Free-standing
2. Inline

Free-standing intention comments

Free-standing intention comments sit inside of Java source code files, but outside of classes, or within classes but outside of methods. A class or method can be linked to a freestanding intention comment by referring to its name. For example:

File `Flashcard.java`

```
package vocabularytrainer;

intention FlashcardIntention {
     description {
         To represent a flashcard for learning foreign-language vocabulary,
         with a cue (on one side of the card) and a list of one or more
         acceptable answers (on the other side of the card).
     }
}

class Flashcard implementsintention FlashcardIntention {
     ...
}
```

This explains that the intention of the `Flashcard` class is to represent a flashcard in a vocabulary-training application.

At this point, intention comments don't appear to have any advantage over writing plain comments, or writing comments with Javadoc. However, if you were to compile `Flashcard.java` using the JWI compiler, and class `Flashcard` did not declare that it was an implementation of any intention, of if the compiler judged that the information content of the comment was insufficient, then the compiler would refuse to compile the code.

To demonstrate the object-oriented features of JWI, let's now use intention comments to document an instance of the Model-View-Controller pattern. First, we will create an abstract intention comment to represent the MVC pattern in general:

File `ModelViewControllerIntention.java`

```
package vocabularytrainer.abstractintentions;

abstract intention ModelViewControllerIntention {
    description {
        To implement the Model-View-Controller pattern, in order to
        structure the user interface code into separate components.
        This separation of concerns helps improve understandability
        and modifiability.

        The model consists of a representation of the application's data.
        The model notifies listeners (typically, one or more view
        components) when the data changes.

        The view component presents the data to the user in the form of
        UI components. Multiple views based on the same model may exist.

        The controller acts upon input from the user and updates the
        model and/or interacts with the view.
    }

    classreference[] modelClasses;
    classreference[] viewClasses;
    classreference controllerClass;
}
```

Then we can describe a specific instance or application of the MVC pattern by declaring an intention comment that extends this abstract intention. In the new intention comment, we fill in the required fields (in this case, references to classes):

File `VocabularyTrainerMVCIntention.java`

```
package vocabularytrainer.intentions;

intention VocabularyTrainerMVCIntention extends ModelViewControllerIntention {
     description {
         To implement the vocabulary trainer user interface according to the
         Model-View-Controller pattern.
     }

     modelClasses = { QuizState, Flashcard, FlashcardList };
     viewClasses = { QuizFrame };
     controllerClass = QuizController;
}
```

The classes that take part in this pattern can then link themselves to the intention for the pattern instance. For example:

```
class QuizController implementsintention VocabularyTrainerMVCIntention {
     ...
}
```

Now, when new developers join this project and encounter any class that is a part of this pattern instance, they will be able to read the comments and follow the links to locate the other components of the pattern and understand their relationships.

Inline intention comments

Within a method, lengthy blocks of code without any descriptive comments will be flagged by the JWI compiler. To associate comment texts with blocks of code, we need a syntax for “inline intention comments” that includes start and end tags that can surround blocks of code. This additionally allows inline intention comments to be nested, allowing each step of an algorithm to be broken into smaller sub-steps. The following example illustrates the syntax in JWI:

```
[[ 1 | Shuffle the deck of flashcards (flashcardList) by iterating
       through the list and swapping the card at the current position
       with another randomly-chosen card ]]
for (int i = 0; i < flashcardList.size(); i++) {
     [[ 1.1 | Generate a random number, which will serve as the index
              of the card to be swapped with the current index ]]
     int otherIndex = randomGenerator.nextInt(flashcardList.size());
     [[ /1.1 ]]

     [[ 1.2 | Swap the records at indices i and otherIndex ]]
     Flashcard tempCard = (Flashcard) flashcardList.get(i);
     flashcardList.set(i, flashcardList.get(otherIndex));
     flashcardList.set(otherIndex, tempCard);
     [[ /1.2 ]]
}
[[ /1 ]]
```

"Opening" comment tags take the syntax `[[ commentIdentifier | descriptionText ]]` (where the square brackets and vertical bar are literal characters). The comment identifiers could be virtually any names, but in this example they follow a hierarchical numbering scheme.

“Closing” comment tags use a slash in front of the comment identifier, similar to XML.

Summary

By elevating comments to be “first-class citizens” of programming languages, Design Intention Driven Programming and Java with Intentions attempt to reduce the long-term burden of program comprehension by encouraging (and forcing) programmers to record their design intentions in the code, so that present and future maintainers can spend less time reading and reverse-engineering code.

It is not a perfect scheme, and it is not suitable for all projects and teams. Many developers would be very resistant to any attempt to force them to write comments, so the approach is suitable only for project teams where an agreement has been reached on the value of comments. However, for those who wish to document their systems, it could be a useful tool. ■

Appendix H: Ethical issues

H.1 Ethical issues involving the proposed solution

If the *Java with Intentions* system were to be adopted by a software development organisation, it would change, to some degree, the way that software development work is done. The impact of this could include the following issues:

- Asking a developer to spend his or her time documenting code brings benefits to a future, unknown maintainer, but brings no immediate benefit to the developer working "now". Beck argues that this violates the principle of *mutual benefit* and breeds ill will in project teams (Beck, 2005, p. 14).
- Developers in commercial firms are rarely evaluated on the "quality" of the code and documentation they produce; managers conducting performance reviews seldom inspect and judge code and documentation artefacts (and sometimes have no direct software development experience themselves). Developers who spend more time on documentation may be perceived (or measured) to be less productive than other developers.
- On the other hand, if an organisation's processes involve "documentation police" who review documentation for quality, then those developers who are poor writers, or those whose native language is not the language of the project (English-as-a-Second-Language speakers in Anglophone countries), may be negatively evaluated.
- Time spent on documentation slows down the pace of work in the short term, making software more costly to clients, at least in the short term.
- If expected long-term cost savings and benefits do not emerge, time spent documenting will be seen as costly, wasted effort.

H.2 Ethical issues involving the survey research

- *Due care:* Care must be taken in the design and implementation of the survey to ensure reliability, validity, and neutrality of results and conclusions. Assumptions, threats to validity, and any other weaknesses in the design that could affect the results and conclusions must be stated (Weisberg *et al.*, 1996, p. 352).
- The research should not be intentionally biased to mislead readers or to support or further a personal viewpoint or agenda. Constant vigilance against bias is required; the approach and methods must be re-questioned, limitations of methods must be observed, and work must be reported honestly (M801, 2007, p. 108).
- *Informed consent:* Survey participants must be informed of the research topic, who is sponsoring and/or conducting the research, and the purpose of the survey. Participants should be reminded of the voluntary nature of participation and must be allowed to decline participation (Weisberg *et al.*, 1996, pp. 355-357).
- *Sensitive topics and confidentiality/anonymity:* The planned questionnaire asks the participant to answer questions about practices at his or her current

organisation, and to express opinions and make judgements about those practices and software quality. It could be damaging to a participant if his or her employer became aware of negative views that the participant had expressed in the questionnaire response. For this reason, responses will be anonymous; no personally identifying information will be asked for, nor retained. Note that this policy carries the risk that participants could re-take the survey multiple times, altering the survey results, and other than being on the lookout for suspicious patterns of activity, I will have no way of detecting this.

- I cannot imagine any other way that the survey could cause harm to participants, other than cost them the time needed to complete it.

Appendix I: Hypothesis testing procedures

I.1 Construction of indices

To support the hypothesis tests of Chapters 3 and 8, indices were constructed to form measures that aggregate scores from *batteries* of related questions, using simple addition (Weisberg *et al.,* 1996, p. 210).

For each index, a value is computed for each survey respondent based on his or her responses to the questions included in the index. Table 45 lists the indices used in the analyses.

The notation "inv(*x*)" indicates that the inverse value of a question response is used. For Likert scale questions on the scale of 1 to 7, the inverse of 1 is 7, the inverse of 2 is 6, and so on.

Table 45: Indices used in hypothesis tests

No.	Name of index	Formula to calculate index	Range of index
IND01	General support for documentation and commenting	inv(Q45) + inv(Q46) + inv(Q47) + Q48 + inv(Q49) + Q50 + Q51 + Q52	8..56
IND02	General support for *Java with Intentions*	Q66 + inv(Q67) + Q68 + inv(Q69) + inv(Q70) + Q71 + Q72 + Q73 + Q74 + Q75	10..80
IND03	Likelihood of some degree of frustration or job dissatisfaction due to perceived deficiencies in the organisation's software documentation practices	inv(Q06) + inv(Q08) + inv(Q09) + inv(Q10) + inv(Q40) + Q41 + inv(Q42) inv(Q43) + inv(Q44) + Q52 + Q56 + Q57 + Q58 + Q59 + Q60 + Q61 + Q62	17..119

I.2 Hypothesis testing procedure

The *SAS Learning Edition* software package was used as an aid in performing the calculations involved in the statistical analysis.

The statistical procedures given by Schlotzhauer (2009) are used for testing the hypotheses. As a demonstration, let us use as an example the following hypothesis:

> *Those respondents who report writing comments regularly will tend to express the opinion that the proposed solution could potentially help improve software comprehension.*

We use a significance level (*alpha*) of 0.10, as is discussed in section 3.2.3.

A statistically significant degree of association[22] between the following two questions would provide evidence in favour of that hypothesis:

- Question 51: "I am very diligent about writing comments when I develop or maintain code."
- Question 75: "Software projects consistently documented in this way would be easier to understand than projects developed using traditional techniques."

First, testing for independence, the Pearson chi-square test gives a p-value of p = 0.1854. As 0.1854 is greater than 0.10, we find insufficient evidence at the 10% significance level to reject the null hypothesis of independence between the two variables (*ibid.,* p. 478). In other words, neither variable is dependent on the other. Normally, testing the association between the two variables is not warranted in such cases. However, the chi-square test is not always valid for cases involving very few data points, in which case Fisher's Exact Test can be used instead (*ibid.,* p. 480). Fisher's Exact Test gives a p-value of $p = 1.163 \times 10^{-10}$. This value is less than the significance level of 0.10, so the null hypothesis of independence can be rejected; the variables show dependence at the 10% significance level.

Before we can test the measure of association between the two variables, we must first formulate a null hypothesis as an inverse of the original hypothesis:

> Null hypothesis (H_0): *Those respondents who report writing comments regularly will tend* ***not*** *to express the opinion that the proposed solution can help improve software comprehension.*

Then we must ask whether the measure of association, if it were to be calculated, is significantly different than zero. Kendall's Tau-b test gives a p-value of p = 0.5253, and the Spearman correlation coefficient gives p = 0.6796. As these p-values exceed 0.10, there is no evidence that the association between the two variables is significantly different than zero at the 10% significance level, so we do not bother calculating the measures of association. (Had the p-value for Kendall's Tau-b been below 0.10, the association according to Kendall's Tau-b would have been 0.1104; had the p-value for Spearman's correlation coefficient been below 0.10, the association according to Spearman's correlation coefficient would have been 0.0874.)

We thus *fail* to *reject* the null hypothesis. Essentially, this is equivalent to rejecting the original hypothesis.

[22] *Association*, not *correlation*, is the correct term in this case as the Likert-scale responses are considered ordinal rank measures; measures of correlation apply only to continuous variables.