Stevenson, Jamie and Wood, Murray (2017) Recognising object-oriented software design quality : a practitioner-based questionnaire survey. Software Quality Journal. ISSN 1573-1367 , http://dx.doi.org/10.1007/s11219-017-9364-8

This version is available at http://strathprints.strath.ac.uk/60262/

CrossMark

Recognising object-oriented software design quality: a practitioner-based questionnaire survey

Jamie Stevenson[1] · Murray Wood[1]

Abstract Design quality is vital if software is to be maintainable. What practices do developers actually use to achieve design quality in their day-to-day work and which of these do they find most useful? To discover the extent to which practitioners concern themselves with object-oriented design quality and the approaches used when determining quality in practice, a questionnaire survey of 102 software practitioners, approximately half from the UK and the remainder from elsewhere around the world was used. Individual and peer experience are major contributors to design quality. Classic design guidelines, well-known lower level practices, tools and metrics all can also contribute positively to design quality. There is a potential relationship between testing practices and design quality. Inexperience, time pressures, novel problems, novel technology, and imprecise or changing requirements may have a negative impact on quality. Respondents with most experience are more confident in their design decisions, place more value on reviews by team leads and are more likely to rate design quality as very important. For practitioners, these results identify the techniques and tools that other practitioners find effective. For researchers, the results highlight a need for more work investigating the role of experience in the design process and the contribution experience makes to quality. There is also the potential for more in-depth studies of how practitioners are actually using design guidance, including Clean Code. Lastly, the potential relationship between testing practices and design quality merits further investigation.

Keywords Software · Design · Quality · Object-oriented · Survey · Questionnaire · Industry · Practitioner · Maintenance

✉ Murray Wood
murray.wood@strath.ac.uk

Jamie Stevenson
jamie.stevenson@strath.ac.uk

[1] Department of Computer and Information Sciences, University of Strathclyde, Livingstone Tower, Richmond Street, Glasgow G1 1XH Scotland, UK

Springer

Jamie Stevenson is a third year PhD student in Computer Science at the University of Strathclyde, Glasgow. His research interests are in software engineering, particularly software design. His PhD research includes practitioner attitudes to object-oriented design quality and patterns of interface and inheritance usage in open-source systems.

Murray Wood is a senior lecturer in Computer Science at the University of Strathclyde Glasgow. He holds a PhD in Computer Science, also from the University of Strathclyde. His research interests are in empirical software engineering. The recent focus of his work is software design, specifically studying design practices used in large-scale software systems.

1 Introduction

The key practices and techniques used in the design of software are long established and widely known. Multi-edition software engineering textbooks (Sommerville 2010; Pressman 2014) have included chapters on software design for decades; there is a large collection of textbooks that specifically address software design (Coad and Yourdon 1991; Gamma et al. 1994; Riel 1996; McLaughlin et al. 2006), and there is a huge body of empirical research literature constantly proposing and evaluating new approaches to design. However, there does not appear to be much research exploring the techniques that software developers actually use in their day-to-day practice and which of these techniques they find most useful.

This paper describes a questionnaire survey designed to discover how important software design quality is to practitioners working in industry and the techniques they find most useful for achieving good quality designs. The research questions addressed the extent to which practitioners concern themselves with design quality, how they recognise design quality, what guidelines and techniques they use and the information they use to make design decisions.

The survey is concerned with software design quality—the internal quality of the design of software and the properties that make the software easier to understand and change in the future (maintainability). It focuses on object-oriented technology because much of today's software is developed using that technology (Cass 2015). It addresses design quality at the class and package level—including source code organisation, class identification, interface use, design patterns and the application of design guidelines.

This research was motivated by a lack of prior research investigating what practitioners find useful in their day-to-day development work. It is important for the software engineering research community to know what practitioners are really doing in terms of design, what is useful and what is less so, to help guide the direction of future research. It is also of potential interest to developers working in industry to know the techniques and practices that their colleagues are using and find useful. It is important for educators to base their teaching on techniques and practices that are actually used in industry, and known to be useful in practice.

This research contributes to the problem identified by Baker et al. in their foreword to an IEEE Software special issue on Studying Professional Software Design (Baker et al. 2012): "*Although we have quite a few high-level design methodologies we don't have a sufficient understanding of what effective professional developers do when they design … To produce insights that are relevant to practice, researchers need to relate to practice and be informed by it*". The research is further motivated by evidence-based software engineering (EBSE) (Dyba et al. 2005): "*A common goal for individual researchers and research groups to ensure that their research is directed to the requirements of industry and other stakeholder groups*". Finally, Carver et al. recently explored practitioners' views of empirical software engineering research (Carver et al. 2016), and one topic

practitioners were keen to see more research on was "*Define good qualities of software and help developers to implement software with great quality*".

To address the goal of discovering how practitioners approach design quality, an online questionnaire consisting of 37 questions, in three sections, was developed. The first section explored high level views on design quality: how it is ensured by practitioners, its relative importance compared to functional correctness and the development methodologies used. The second, main section, explored opinion on well-established design guidelines and practices. The third and final section captured demographics. Questions consisted of a mixture of fixed-option, Likert-item options followed by free-text fields intended to gather more detailed insights.

The questionnaire was distributed by email, initially using personal and university contacts in the UK. Thereafter, the authors searched the mainstream software engineering literature for papers related to software design published by authors with an industrial affiliation. Two hundred and seventy-five contacts were identified who were then emailed individually inviting them to participate and requesting that they might distribute the questionnaire to relevant colleagues. As a result, a total of 102 questionnaires were completed by respondents, all with industrial experience of object-oriented design.

The resultant data was analysed using a combination of quantitative analysis of the fixed-option Likert-item questions and qualitative analysis of the free-text data (25 k words of text). This resulted in the identification of 14 key findings. These key findings were then used to answer the original research questions and identify a number of key themes which form the main results from the study.

The contribution of the paper is a series of practitioner-based findings on the importance of software design quality and how it is achieved in industry. While some of the findings are already widely advocated, to-date, they have lacked practitioner confirmation of their real value. The quantitative findings are also enhanced by many more detailed qualitative insights which start to reveal the specific challenges that occur during software design and the practices which are currently used to overcome them.

The main finding is that individual and peer experience are seen as major contributors to design quality and to confidence in design decisions. Classic design guidelines, well-known lower level practices (e.g., Clean Code), reviews, tools and metrics all can contribute positively to design quality. There is a potential relationship between testing practices used in industry and design quality. Factors that can have a negative impact on design quality include inexperience, time pressure, novel problems, novel technology and imprecise or changing requirements. Most of the findings did not seem to be influenced by experience, programming language or development role, though there is a suggestion that respondents with more experience have higher confidence in their design decisions, place more value on reviews by team leads and are more likely to rate design quality as very important.

These findings have implications for researchers, practitioners and educators. For researchers, the results emphasise the importance of experience in the design process. There is scope for much more empirical research investigating exactly what constitutes practitioner experience, how it is used in practice and to what extent experience can be translated into better guidelines, metrics and tools. Researchers might also follow up these findings to investigate whether experience, testability, Clean Code, etc., do actually provide detectable improvements in practice. For practitioners, these results serve to highlight the techniques, guidelines and tools that other practitioners find to be useful. For educators, these results help to identify some of the fundamentals that students should know and understand. The results

also highlight the need for educators to find effective case studies and design projects which may help accelerate their students' progress from novice to expert.

The remainder of the paper is organised as follows. Section 2 presents a summary of related work on software design quality using both the theoretical and empirical software engineering literature. It also includes a brief summary of related survey-based research. Section 3 describes the study's research goal and its research questions. Thereafter, Section 4 describes the design of the survey in detail. Section 5 provides a detailed description of the main results from the study. Section 6 investigates whether respondent experience, programming language used or development role appears to have influenced the main results. Section 7 describes how the results are interpreted to answer the original research questions. Section 8 provides a detailed discussion of the results, interpreting them and relating them to related work. The paper finishes with a discussion of potential limitations and threats to validity, Section 9, followed by the conclusions in Section 10.

2 Related work

2.1 Software design quality

The focus of this research is the design of object-oriented software at the class and package level—including source code organisation, class identification, interface use, design patterns and the application of design guidelines. It corresponds to the 'Product View'—an examination of the inside of a software product (Kitchenham and Pfleeger 1996).

In terms of the current international standard on Software Quality Characteristics (BSI 2011), the focus of this work is 'Maintainability':

- Degree to which a system is composed of discrete components such that a change to one component has minimal impact on other components (Modularity).
- Degree of effectiveness and efficiency with which it is possible to assess the impact on a system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified (Analysability).
- Degree to which a system can be effectively and efficiently modified without introducing defects or degrading existing product quality (Modifiability).
- Degree of effectiveness and efficiency with which test criteria can be established for a system, and tests can be performed to determine whether those criteria have been met (Testability).

Since the arrival of object-oriented technology in the mainstream of software engineering in the 1980s, there has been a vast amount of research investigating definitions of 'good' object-oriented design and the associated identification of techniques and tools to help achieve the desired quality.

One significant approach has been the adoption of traditional, pre-object-oriented design guidelines on coupling and cohesion to the object-oriented domain (Coad and Yourdon 1991; Riel 1996). Coupling is concerned with the strength of connections amongst components in a design, aiming to minimise these, whereas cohesiveness is concerned with the relatedness of the elements within a component, aiming to maximise these. Early work by Briand et al. suggested that long-stand design guidelines by Coad and Yourdon such as coupling and

cohesion can have a beneficial effect on the maintainability of object-oriented designs (Briand et al. 2001).

Closely related to these guidelines is a significant body of work defining metrics that aim to measure properties associated with these design guidelines. The most widely used metrics suite is that proposed by Chidamber and Kemerer (C&K), and variants derived from these, covering coupling, cohesion and inheritance relationships (Chidamber and Kemerer 1994). Chidamber and Kemerer used viewpoints collected from practitioners to drive the development of their metrics. Concerns about the theoretical validity of some of the cohesion and coupling aspects of the original C&K metrics (Hitz and Montazeri 1996; Mayer and Hall 1999; Kitchenham 2010) have led to refinements such as LCOM4 (lack of cohesion) (Hitz and Montazeri 1995) and Low level design Similarity-based Class Cohesion metric (LSCC) (Al Dallal and Briand 2012).

Object-oriented design guidelines and their associated metrics have been subject to a range of empirical evaluation. Most of the investigations explore potential relationships between internal factors, such as coupling and cohesion, and external factors such as fault-proneness, development effort, change ripple effect and expert opinion (Briand and Wüst 2002). A systematic literature review of 99 studies (Jabangwe et al. 2015) explored the potential link between object-oriented measures and external quality attributes (e.g., reliability and maintainability). In terms of maintainability, the review found that "*Overall, coupling, complexity and size measures seem to have a better relationship with maintainability than inheritance measures*". Al Dallal found that most of the cohesion, size and coupling metrics could predict the more maintainable classes in three open-source Java systems (Al Dallal 2013). Ferreira et al. identified threshold values for metrics which could help distinguish well-designed classes from less well-designed classes (Ferreira et al. 2012). Dallal and Morasca suggest that coupling and size have a positive impact on 'reuse-proneness' (Al Dallal and Morasca 2014). Bajeh et al. found a correlation between coupling and cohesion metrics and 'testability' (Bajeh et al. 2015). Metrics have also been widely used for fault prediction (Radjenović et al. 2013) and identifying bug-prone code (Palomba et al. 2016).

Kitchenham performed a preliminary mapping survey of empirical research on software metrics (Kitchenham 2010). The key findings were that there is a large body of research related to metrics but that this is dogged by significant issues: invalid empirical validations (especially related to theoretical limitations of the C&K LCOM metrics and the treatment of code size), contradictory results and too many comparisons in single studies (e.g., 66 different metrics). Other work has raised concerns about the value of such metrics in predicting design quality. Riaz et al.'s systematic review of maintenance prediction and metrics (Riaz et al. 2009) found weaknesses in terms of comparison with expert opinion and external validity, as well as differences in the definition of maintenance used in the studies. Sjøberg et al. found that a range of size, complexity, coupling, cohesion and inheritance metrics were not mutually consistent, and that only size and low cohesion were strongly associated with increased maintenance effort (Sjøberg et al. 2012). Ó Cinnéide et al. studied the use of five cohesion metrics to guide refactoring and concluded that the metrics often captured differing notions of cohesion (Ó Cinnéide et al. 2012, 2016). Veerappa and Harrison repeated Ó Cinnéide et al.'s study focussing on coupling metrics (Veerappa and Harrison 2013) finding that coupling metrics appeared less conflicting than cohesion metrics and that improving coupling did not necessarily improve cohesion.

There has also been a large amount of research on the closely related topics of 'code smells' (Fowler et al. 1999) and 'anti-patterns' (Brown et al. 1998). Marinescu used metrics-based rules to identify a range of 'design flaw smells' (Marinescu 2012)

associated with technical debt (Cunningham 1993). Yamashita and Moonen found that code smells were "*partial indicators*" of design quality; the smells which were most closely associated with maintenance problems were again associated with size, complexity and coupling (Yamashita and Moonen 2013b). Khomh et al. found that classes affected by design problems in the form of 'anti-patterns' are more change and fault prone (Khomh et al. 2012). On the other hand, Sjøberg et al. investigated the relationship between code smells and maintenance effort using professional developers and found that none of the investigated smells were significantly associated with increased effort (Sjøberg et al. 2013). Fontana et al. used a literature review to identify a range of examples of anti-patterns that might "*not be as detrimental as previously conjectured*" (Fontana et al. 2016).

A similar approach to the high level design guidelines of coupling and cohesion are object-oriented principles such as 'SOLID' (Martin 2003). Martin argued that "... *dependency management, and therefore these principles, are at the foundation of the '-ilities' that software developers desire*" (Martin 2005). The key principles associated with SOLID are the following:

Single Responsibility (SRP)—A class should have one and only one reason to change
Open-Closed (OCP)—Extend a class' behaviour, without modifying it
Liskov Substitution (LSP)—Derived classes must be substitutable for their base classes
Interface Segregation (ISP)—Make fine-grained interfaces that are client specific
Dependency Inversion (DIP)—Depend on abstractions, not on concrete classes

As well as the SOLID principles, Martin advocated a wide range of 'Clean Code' principles (Martin 2009) which include a much wider range of advice focussing on both design and code intended to make programs (and their designs) easier to understand and change. This advice covers package and class design and minimising their dependencies, method design with recurring themes of small size (e.g., classes less than 100 lines, methods less than 20 lines and much shorter), simplicity (e.g., methods do one thing) and clarity (e.g., descriptive naming, minimal value-added commenting).

In a similar vein, Larman identified the 'GRASP' principles (Larman 2004) which were intended to capture the knowledge of expert designers to help in the construction of 'better' quality object-oriented designs. GRASP consists of nine principles, some of which are closely related to the concepts covered above.

Over the last two decades, the idea of design patterns has become widely established as a source of object-oriented design advice (Gamma et al. 1994). Design patterns present generalised solutions to common design problems, making use of design techniques such as interfaces, abstract types, inheritance, dependency inversion, polymorphism, and double dispatch to create '*simple and elegant*' (Gamma et al. 1994) design solutions. The original catalogue consisted of 23 patterns whose names and intent are now widely-recognised in the design community. Design patterns embody two fundamental design principles that are recognised in their own right:

- Program to an interface, not an implementation—Do not declare variables to be instances of concrete classes, commit only to interface types
- Favour object composition over class inheritance—Rather than using inheritance to define a subclass, use the 'parent' class to define a field and delegate to it.

In contrast to the significant amount of empirical investigation into guidelines such as coupling and cohesion and their associated metrics, there has been relatively little empirical investigation of the Clean Code, GRASP, design pattern-type principles. These principles and patterns capture the experience of recognised experts in the field, such as Fowler (Fowler et al. 1999), the 'Gang of Four' (Gamma et al. 1994), 'Uncle Bob' Martin (Martin 2009), Brooks (Brooks 2010) and Booch (Booch 2011). They are derived from decades of experience, both actively working in the field and from challenging and debating other experts within the software engineering community. Much of this literature is largely in the form of *advocacy* (Zhang and Budgen 2012) with little in the way of formal evaluation in the published literature.

One aspect that has attracted some limited empirical evaluation is design patterns. Zhang and Budgen's systematic literature review found 11 formal studies and seven experience reports (Zhang and Budgen 2012). From their analysis, they could not, however, derive any firm findings, though they did suggest that design patterns can provide a "*framework for maintenance*" and found "*some qualitative indication that they do not help novices learn about design*".

Conley and Sproull used aspects of Martin's SOLID principles to define 'modularity' (Conley and Sproull 2009). Using static bugs and complexity as proxies for quality, they found a decrease in complexity but also an increase in the number of static bugs detected as their measure of modularity increased.

Finally, there is a range of software tools that are intended to actively support improved design quality. Typically, these tools aim to enforce adherence to guidelines such as those described above or use metric values to flag up potential weaknesses or 'bad smells'. An example of this kind of tool is SonarQube (Campbell and Papapetrou 2013). SonarQube builds on existing widely used code quality tools such as FindBugs (Ayewah et al. 2008) and PMD (Copeland 2005) to highlight design quality issues such as cyclic dependencies between packages and classes (tangle), metrics such as LCOM4 and Response for Class (interactions with other classes), code complexity, code duplication and test coverage.

2.2 Software engineering surveys

Surveys have been used to discover the state of industrial software engineering practices for more than three decades (L. Beck and Perkins 1983). In recent years, they have become particularly popular, taking advantage of online distribution, email and social media for contacting potential participants and using highly-regarded survey guidance (Pfleeger and Kitchenham 2001).

One of the closest pieces of design-related survey research was on design patterns (Zhang and Budgen 2013). Their goal was to discover which of the Gang of Four patterns were considered useful by experienced users. They approached all authors of papers relating to design patterns in the mainstream software engineering literature. They received 206 usable responses (response rate of 19%). More than half, the respondents had over 10 years of experience in object-oriented development. The survey was administered using the SurveyMonkey[1] commercial site. It used a combination of Likert-item questions and open-ended questions but received "*relatively few comments*"—338 in total. Their main finding was that only three design patterns—observer, composite and abstract factory—were highly regarded.

---

[1] https://www.surveymonkey.com/

Simons et al. surveyed 50 professional developers comparing their opinion of design quality to that of metrics based on design size, coupling and inheritance (Simons et al. 2015). They found that there was no correlation between the metrics used and professional opinion when assessing UML class diagrams. One of their main findings was the need to involve humans within the software design quality assessment process; metrics alone were insufficient. Yamashita and Moonen surveyed 85 professional developers exploring their knowledge and use of code smells (Yamashita and Moonen 2013a). Thirty-two percent of respondents did not know of code smells. Only 26 were concerned about code smells and associated them with product evolvability. Relevant to the research reported in this paper, respondents highlighted that they often had to make trade-offs between code quality and delivering a product on time. At Microsoft, Devanbu et al. found that practitioners have strong beliefs largely based on personal experience rather than empirical evidence (Devanbu et al. 2016). They call for empirical research to take practitioner beliefs into account when designing new empirical studies.

Other recent, but less closely related, software engineering surveys include the practices used in the Turkish software industry (Garousi et al. 2015), the use of modelling in the Italian software industry (Torchiano et al. 2013), the use of UML modelling for the design of embedded software in Brazil (Agner et al. 2013), testing practices used in the Canadian software industry (Garousi and Zhi 2013), the data and analysis needs of developers and managers at Microsoft (Buse and Zimmermann 2012) and the use of software design models in practice, in particular UML (Gorschek et al. 2014).

A number of common themes emerge from these surveys. All surveys tend to use an online, web-based delivery mechanism such as SurveyMonkey. Many closely follow the guidance of Pfleeger and Kitchenham on survey methodology (Pfleeger and Kitchenham 2001). Most focus on the use of Likert-item or fixed-response questions with associated (quite straightforward) quantitative analysis. There is some use of free-text questioning and, again, fairly routine qualitative analysis. Sampling is challenging, with a tendency to use personal contacts, public records of relevant organisations and mailing lists/social media. The number of respondents typically ranges from about 50 to 250, in most of the studies respondents typically had 5–10 years of experience. Most of these survey papers tend to identify high level findings or guidelines, focussing on raising issues to be investigated further in more rigorous research studies.

3 Survey goal and research questions

This research was motivated by a lack of previous work that explores the techniques that software developers actually use in their day-to-day practice and which of these techniques they find most useful. The goal of the survey was to discover the extent to which practitioners concern themselves with software design quality and the approaches practitioners use when considering design quality in practice. The design of the questionnaire was guided by the following research questions:

> RQ1: Design Priorities—To what extent do practitioners concern themselves with design quality?
>
> This research question was motivated by the fact that there is a significant amount of research literature and empirical studies that investigate software design quality but there

is not much research that actually investigates the importance of design quality to practitioners' in their day-to-day work.

RQ2: Recognising Quality—How do practitioners recognise software design quality?

This question was motivated by the need to discover how practitioners distinguish 'good' (or better) design quality from 'bad' (or worse). Again, there seems to be little prior research that investigates how practitioners actually recognise quality in their day-to-day work.

RQ3: Guidance—What design guidelines do practitioners follow?

This question was motivated by the need to discover whether well-known guidelines and practices from the literature are actually used, and are useful, in practice. Also, which of these are the most useful?

RQ4: Decision Making—What information do practitioners use to make design decisions?

This question was motivated by the need to discover the design information that practitioners use to make decisions about quality. Is it information from designs, from code, from metrics, from tools …?

The answers to these research questions are important to researchers to help guide future research, to practitioners to see how other practitioners are addressing design quality, and to educators to teach students the principles that underpin industry-relevant techniques.

The Goal Question Metric approach (Van Solingen et al. 2002) was used to derive the research questions from the original research goal. Prior to the distribution of the questionnaire, the research questions were mapped to survey questions, see Table 1.

4 Software design quality questionnaire survey

The design of this survey was based on reputable empirical advice popular in the related studies (Pfleeger and Kitchenham 2001). The questionnaire was based on a mixture of closed questions and open, free-text questions which were intended to gain deeper insights into the participants' responses.

The questionnaire was developed, distributed and analysed using the Qualtrics Survey Software.[2] Qualtrics contains professional-grade survey building tools, including support for questions paths and customisation by scripting and stylesheets. Further analysis of the results was carried out using nVivo.[3]

The questionnaire consisted of a mixture of Likert-item questions, mostly with five items including a neutral response in the middle. Including a neutral option avoids forcing a positive or negative choice (Shull et al. 2008), which seemed appropriate given the exploratory nature of the survey instrument. To increase respondent speed and comprehension, a similar question layout was used throughout the questionnaire. Each question topic finished with a free-text box intended to gather more detailed insights into the participants' responses.

---

[2] http://www.qualtrics.com/
[3] http://www.qsrinternational.com/

Table 1 Original GQM mapping of survey goal to research questions to answerable survey questions

High level goal: *To what extent do practitioners concern themselves with design quality?*

RQ1: Design Priorities—*To what extent do practitioners concern themselves with design quality?*
Questions about time spent improving quality, responses where poor quality is the cause of a problem.
Q1, Q3, Q5, Q17, Q19, Q27, Q37

RQ2: Recognising Quality—*How do practitioners recognise software design quality?*
Questions about recognising quality and responses where following a particular guideline or practice is
    considered positive/reassuring
Q3, Q5, Q6, Q7, Q9, Q11, Q14, Q28

RQ3: Guidance—*What design guidelines do practitioners follow?*
Direct questions about selected guidelines, general question about any other useful guidelines. Responses where a
    tool, guideline or methodology is mentioned as useful.
Q4, Q8, Q10, Q12, Q13, Q15, Q16, Q18, Q20, Q22, Q24, Q26

RQ4: Decision Making—*What information do practitioners use to make design decisions?*
Direct questions about design decision making, confidence, and what makes decisions difficult.
Responses that mention choice, design decisions, doubt or managing uncertainty.
Q1, Q4, Q9, Q20, Q11, Q14, Q15, Q16, Q28

The questionnaire landing page stated the study aim, identified its authors and gave their contact details. It confirmed that the survey had university ethics approval (obtained December 6, 2014). It stated that questionnaire was only concerned with object-oriented design and defined design in the context of the survey: "*The term design is used in this survey to discuss the organisation of packages and classes, class identification and interaction, and interface use.*" This was followed by a section on privacy which explained how responses were to be kept anonymous, how the data was to analysed, how long data would be retained, how to quit the questionnaire, if required and how to contact the survey authors. There was also a consent section which blocked access to the rest of the questionnaire until consent was given.

The first section of the survey explored the participant's high level views of software design quality: how they assess quality, their confidence in design decisions that they make, the relative importance of design quality compared to functional correctness and the development methodologies that they use.

The next section explored how design guidelines for each of the following were used in practice (a questionnaire page dedicated to each in turn): program to an interface, inheritance, coupling, cohesion, size, complexity and design patterns. Again, the questions were a mixture of Likert-item followed by free-text questions to explore each topic further.

Demographic questions were included at the end in order that these questions did not deter potential respondents from completing the questionnaire; as Kitchenham and Pfleeger note, this is suggested by Bourke and Fielder (Kitchenham and Pfleeger 2002b). These questions were used to screen participants, ensuring that they appeared to have sufficient and appropriate experience and also to provide a general summary of participants' backgrounds.

4.1 Pilot study

Prior to formal and widespread distribution, the initial questionnaire was evaluated in a pilot study. The questionnaire was sent to ten individuals asking them to complete it and provide feedback on the content and form of the questionnaire. The ten included six individuals from the target population, two researchers with software development experience and two

academics with experience of survey research. The pilot was distributed on June 23, 2014, a reminder was sent on July 4, 2014 and the pilot study was closed on July 11, 2014.

Four completed and two partially completed questionnaires were received. Only one of the pilot respondents appeared to complete the questionnaire in a single session, taking 36 min. Four of the respondents indicated their number of years working in industry: One had 2–5 years, two had 6–10 years and one had 11–20 years. The respondents used the (then) generous free-text spaces in the questionnaire to provide feedback. As a result of feedback on the questionnaire structure, the following changes were made:

– Some of the optional free-text boxes were removed as these had been perceived as mandatory or excessive despite being marked as optional.
– A question about the Law of Demeter (Lieberherr et al. 1988) was removed as it seemed some respondents were not familiar with this concept; and therefore, it did not meet the aim of a commonly known guideline.
– Some terms that appeared to cause confusion (e.g., framework, pattern) were clarified.
– The options in some questions were simplified.

The final version of the questionnaire consisted of an initial consent question and then 37 questions: 29 topic questions (15 multiple choice, 1 numerical, 13 free-text), seven demographic questions, and the final general free-text feedback question. A summary of the final questionnaire is available in the appendix, and the complete questionnaire is available in the experimental package.[4]

4.2 Population and sampling strategy

The target population for the survey was software developers with industrial experience using object-oriented technology. Obtaining access to a representative sample of this population was a major challenge in performing the study. Industrial software development is a worldwide activity currently involving perhaps as many as 11 million professional developers.[5] As Zhang and Budgen recognise "... *this particular community is defined in terms of their specific skills and knowledge, rather than by location or through membership of any formal organisation, there is no way of knowing the size of the target population…*" (Zhang and Budgen 2013).

The survey therefore used non-probabilistic sampling, convenience and snowballing (Kitchenham and Pfleeger 2002a). The questionnaire was initially distributed by email in July 2014 to 48 of the authors' university contacts who worked in the UK software industry. At the same time, the questionnaire was published to various relevant social network communities, and 34 email invitations were sent to business incubators affiliated with the Scottish Informatics and Computer Science Alliance (SICSA).

Two months after the initial distribution, the authors then targeted a more global population using industrial authors who had recently published design-related research in key software engineering destinations. The authors manually searched a range of software engineering journals and conferences (EMSE, ICSE, ICSM, IEEE Software, TSE, IST, SPE, SQJ, OOPSLA, SPLASH, MOBILESoft) through the period 2009–2014 for software design-related papers with an author who appeared to have an industrial affiliation. Many of the publications printed the author's email address. In cases where the email address was missing,

---

[4] http://dx.doi.org/10.15129/879fb7cf-7f98-49f5-bfa0-183dc5f7d85f
[5] http://www.infoq.com/news/2014/01/IDC-software-developers

a web-search was performed to discover whether the author's email address was publically available in another source e.g., the individual's web page. This resulted in the identification of a further 275 contacts.

Each contact was then sent a short email. This email introduced the authors and the purpose of the survey. It explained why they were being approached e.g., as an author of a technical paper in an identified journal or conference and their affiliation. The email stated that the questionnaire should be completed by programmers, designers or architects based on how they approach object-oriented design quality in practice. The email also stated that the questionnaire took between 15 and 30 min to complete (depending on how much text was written in the open-ended questions). The email also included an anonymised, reusable link to the questionnaire. (The Qualtrics survey system uses a machine's IP address to identify unique participants. This information is not available for inspection and is destroyed on completion of the survey. Duplicate responses are prevented by the use of cookies.) The invitation email also included a request that the email be passed on to other relevant parties e.g. within the recipient's organisation, if appropriate (snowball recruitment). The survey was closed at the end of December 2014, having run for 6 months.

Using the template suggested by Linåker et al. (Linaker et al. 2015), the *Target Audience*, *Unit of Analysis*, *Unit of Observation* and *Search Unit* for this study are all: *Software developers with industrial experience using object-oriented technology*. The *Source of Sampling* are the authors' contacts in the UK software industry and software engineering papers published in the period 2009–2014 by authors who held an industrial affiliation.

4.3 Data analysis

Two hundred and twenty-two responses were received in total. Of these:

- 96 questionnaires were complete.
- 93 questionnaires were blank indicating that candidates had seen the landing page and not continued or had answered 'no' to the consent question.
- 23 questionnaires were partially completed—these respondents had completed a variety of multiple choice questions, omitting some free-text questions or some of the demographic questions.
- 6 completed questionnaires from the pilot study were included excluding the data from questions that were removed between the pilot and the main study. These did not include the two academics as it was known they did not respond; email tracking was used in the pilot so it was possible to tell who responded but not which response corresponded to which email.

The mean completion time for the questionnaire was 31 min.

An examination of the partially completed questionnaires found that they appeared to contribute very little—mostly a few questions were answered and small comments in the free-text sections. A decision was therefore made to exclude all of these from the analysis. This resulted in a total of 102 complete questionnaires being used for the main analysis. All 102 respondents stated that they were currently working in industry or had previously done so.

For each of the closed questions, the data is summarised and presented 'as is'. The questionnaire contained 14 free-text questions. The responses from the free-text areas yielded over 25 k words of feedback. The approach to qualitative analysis of free-text responses was

similar to that adopted by Garousi and Zhi—synthesis, aggregation and clustering of responses to identify key categories (Garousi and Zhi 2013).

There are many approaches available for coding such qualitative data (Saldaña 2015); here, an iterative approach was used to allow the data to inform the coding scheme:

- Holistic coding—collecting all the relevant sources for each basic theme or issue together for further analysis.
- In vivo/verbatim coding—short words or phrases from the material—this may reveal significant subjects, topics or themes that are explicit in the material.
- Descriptive/topic coding—identifying the topic, not necessarily the content of the material.
- Versus coding—any discussion of mutually exclusive activities, policies etc.

Each of these types of coding was performed in its own pass over the free-text material. The coding process was iterative—with topics and relationships becoming more refined as more data was analysed. This inevitably carries the risk of bias—as coding is heavily dependent on the researcher. In this case, there was no pre-determined coding scheme, and the aim was to let the responses 'speak for themselves'.

Coding was used to identify key topics in the responses for each individual free-text question. In the topic tables that follow, the first column identifies the topic code, the second column lists illustrative quotes for that topic and the third column shows a count of how many respondents mentioned the topic for that question. It should be noted that this count of the number of mentions is not viewed as a sample from any population, but rather only an indication within this particular sample about popularity.

To save space, only the most popular topics for each question are included in this paper along with one or two illustrative quotes. Occasionally, where an illustrative quote was judged to be particularly interesting, some less popular topics have also been retained. As a result, some detail is inevitably lost to the reader—the complete set of raw data, and full results tables are available in the online experimental package.

5 Survey results and findings

5.1 The respondents

One hundred and two respondents completed the questionnaire, 56 from the UK, 40 from all over the world outside of the UK, and six who did not provide a location. The self-reported

Fig. 1 Q32—How many years have you been involved in the software industry as a designer/developer? (102 responses)

years of experience for respondents is shown in Fig. 1. Unless shown otherwise, all figures consist of 102 responses. Compared to the experience levels recorded in the surveys discussed in the Related Work section the experience level here is at the high end; 45% of respondents in this study have over 10 years of industry experience.

Figure 2 summarises the roles that the respondents fill in software development (they could fill more than one role). Ninety-three of the respondents have some programming duties. Only two respondents were solely managers; however, these individual's responses in other areas indicate that this is a technical lead, rather than an administrative role.

Figure 3 summarises the programming languages used by respondents (again respondents could choose more than one response). There is a good correspondence with the top programming languages identified by IEEE Spectrum (Cass 2015) which ranked 1: Java, 2: C, 3: C++, 4: Python, 5: C#, 6: R, 7: JavaScript.

Figure 4 shows the application domains that respondents work in.

As the study used non-probabilistic sampling, the findings cannot be generalised to a wider population (Stavru 2014). However, the respondents do appear to represent a diverse group of experienced practitioners, and the common findings should therefore provide a useful indication of widely used approaches to design quality. Overall, it does appear that the respondents are all from the target population.

5.2 General approaches to design quality

Question 1, the first Likert-item question that focussed on design quality, asked respondents about their general approach to ensuring quality in their development process. The results are summarised in Fig. 5.

Responses show a clear bias towards personal experience—this is the strongest response for each of the categories, followed closely by peer review, team review and expert review—all people-mediated inputs. This leads to the first key finding (KF) from the study.

> KF1: People-mediated feedback, including personal experience, is highly important when determining software design quality.

Table 2 shows the free-text responses associated with Q2—further details on ensuring quality in the development process. The topic code for each category of response is shown in

Fig. 2 Q36—What role(s) have you carried out in a software development environment? (102 responses, respondents could choose more than one option)
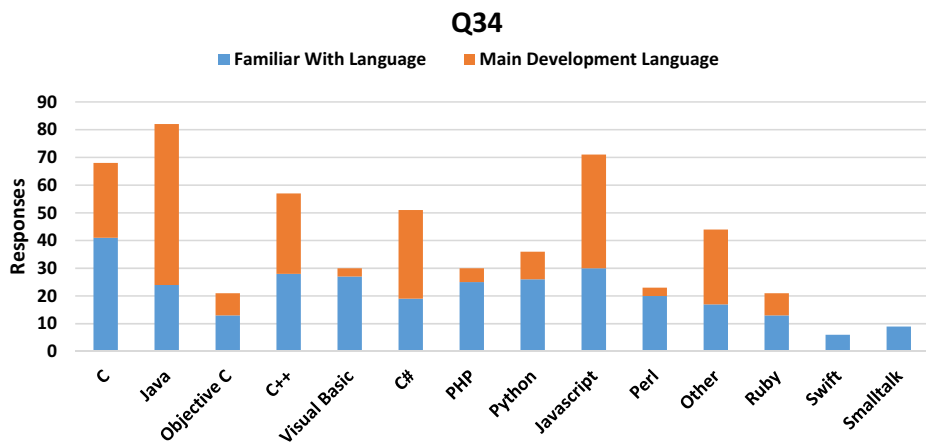
Q34
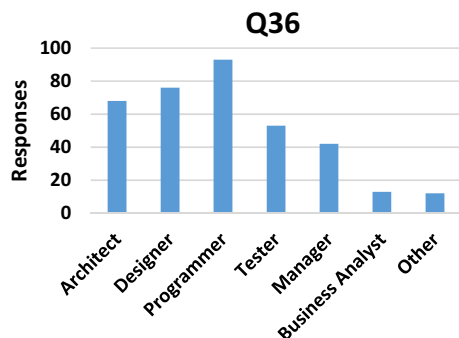
Fig. 3 Q34—What programming language(s) do you use when developing software? (102 responses, respondents could choose more than one)

the first column, illustrative examples are listed in middle column and the total count of respondents who made responses in that category is provided in the third column. The complete set of response data is available in the online experimental package.

The free-text responses in Table 2 might be summarised as '*knowing what to do*'—whether this is from previous experience, a review process, guideline, process, specification, or domain knowledge. The feedback on experience might be separated into two further categories:

- General conscientiousness, following guidelines and design practices (discipline)
- Prior experience with a particular technology or problem domain (specific knowledge), allowing the avoidance of common pitfalls

The next general design question was concerned with confidence in design decisions, see Fig. 6 (Q3).
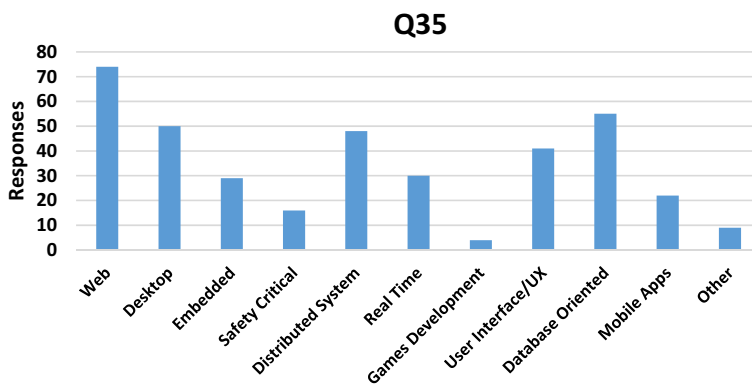
Q35

Fig. 4 Q35—What kind of problems are you most familiar with? (102 responses, respondents could choose more than one)
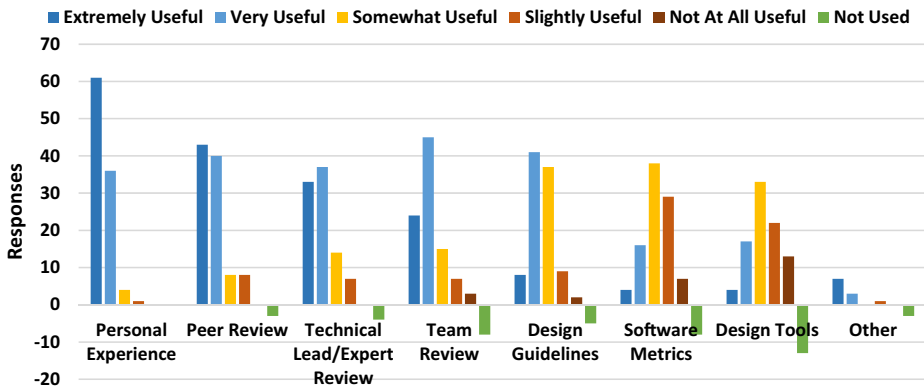
Q1

Fig. 5 Q1—How important is each of the following in ensuring the quality of software design in your development process? Note that the horizontal categories have been ordered from most to least positive (total of first two columns in each category). Counts for 'not used' are shown as green negative values for clarity

The quantitative results suggest generally high confidence; however, analysis of the free-text responses (Table 3) revealed a range of situations where there is less certainty. Many responses highlighted uncertainty in terms of the problem domain, requirements, choices amongst design options, lack of control, and design complexity.

KF2: Respondents were generally confident about design decisions that they make. Issues that reduce confidence include lack of knowledge about the problem domain or

Table 2 Q2 responses—Further details on how design quality is ensured in the development process (example free-text responses)

Topic code	Illustrative quotes	Mentions
Review	"…tight review by 1 or 2 technical experts."	10
Personal Experience	"I think personal/peer experience is the starting point, as you can't consider using what you/your team don't know about."	9
Tools	"…with our build system, Sonar by SonarQube is ran and this runs Checkstyle, EclEmma, ..."	9
Peer experience	"…we are much more reliant on personal experience, team experience and access to domain experts than design guidelines. The latter can only take us so far..."	7
Design space problem	"…there is no perfect choice but there are always different ways to reach your goal."	5
Teamwork	"…a general consideration and acknowledgement that others will have to use your code once you're done is IMO the single most important factor. There are experienced people who will produce lower quality designs than less experienced but conscientious designers…"	4
Processes	"Separating design from implementation is itself an impediment to good design. Good design depends on feedback."	4
Guidelines	"…professional guidelines are very helpful."	4
Feedback	"…feedback that matters is that which comes from end users and those concerned about the long-term evolution of the system."	3
Design for change	"…Re-architecting an existing design is rarely a simple task and so is often not a desired option."	3
Domain familiarity	"Generally our team deals with a certain business domain… we can have a good idea whether or not a design proposal will work…"	3

Fig. 6 Q3—When making design decisions, how confident are you that you have chosen the best option from the alternatives you have considered?
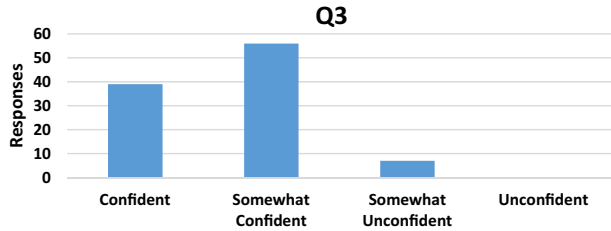
technologies used, having a range of design options, imprecise or changing requirements, lack of control over all aspects of the system, design complexity.

The next question (Q5) asked how designers recognise design quality. Table 4 summarises the main responses. Many topics are associated with well-defined practices such as Clean Code (Martin 2009): clarity of design including naming, small size, simplicity, modularity, documented, full set of tests.

KF3: Respondents identified many well-defined practices that help improve design quality: clarity of design especially naming, small size, simplicity, modularity, documented, full set of tests.

The other group of topics are much less concrete and relate back to the emphasis placed on experience in responses to Q1, intuition and even design hindsight: maintainable, clarity, complete, subjective "*design is an art*", reflects the problem domain.

Table 3 Q4 responses—What type of problem makes it difficult to have confidence in a design decision? (sample free-text responses)

Topic code	Illustrative quotes	Mentions
Knowledge gap	"…lack of familiarity with the problem generally leads to more iterations over the designs as the project moves forward." "...inexperience with any of the technologies involved and the feasibility of what can be done with them."	20
Unclear requirements	"Customers who don't understand their own problem."	18
Choice amongst options	"Many options may be good, and weighing their advantages and disadvantages is an inexact science"	17
Changing requirements	"It is far more time and cost efficient to be able to refactor code when we know how it is really being used, rather than try and concoct a perfect plan first time around."	15
Lack of control	"Confidence levels can drop when the use of external resources … is used as part of an overall project."	14
Unknown future use	"The best you can do is design for the requirements you were given, and with a bit of foresight…"	13
Complexity	"Convoluted designs almost always suggest there's something wrong with design, regardless of system complexity"	6

Table 4 Q5—What are the key features that allow you to recognise good design in your own work and in the work of others? (sample free-text responses)

Topic code	Illustrative quotes	Mentions
Maintainable/Extensible	"…well designed components are easy to maintain over time though it takes longer to recognise this…"	31
Clarity	"A good design should make it as simple as possible to understand the responsibility of each piece of code, and how they interact."	28
Modular/separation	"Partitioning of functionality across interfaces i.e. a target balance of loosely coupled components. I tear my hair out on a regular basis when I see components poisoned with dependencies between logical boundaries (this is something you always see with graduate programmers but never with older hands)."	18
Smallness/simplicity	"…no genius code where a task is compacted into an unmaintainable mess."	18
Tested/testable	"…code that doesn't have a decent set of tests is badly designed…"	14
Complete	"Easy to understand, reuse, alter and maintain without needing to seek external explanation" "…you should be able to read the code as if it were written in English, without any comments…"	13
Documented	"…programmers who think their code is 'self-documenting' or who give other excuses for failing to document are usually incompetent, in my experience."	12
Quality is subjective	"There's a quote from R. Buckminster Fuller 'When I am working on a problem, I never think about beauty, but when I have finished, if the solution is not beautiful, I know it is wrong.'" "…mostly gut feeling, I think design is an art."	10
Reflects problem	"How well the model describes the domain and how little the technical platform has influence on the model"	10

> *KF4: Respondents identified a range of desirable properties for designs that are hard to define and acquire and which seem derived from experience, intuition and hindsight.*

There then followed a pair of questions that explored the relationship between functional correctness and design quality. The first of these (Q6) asked about the importance of functional correctness and design quality in the respondent's work—see Fig. 7. The second (Q7) asked about the split of effort between focusing on correctness and design quality—see Fig. 8.

Functional correctness was defined in the questionnaire as "*Ensuring that the software being created will meet the specified requirements*". Design quality was defined as "*The suitability of the structural organisation of packages and classes, class identification and interaction, and interface use*".

Perhaps unsurprisingly, a high majority of respondents (83) stated that functional correctness is *Very Important* in their work, with almost the rest (15) stating that it is *Important*. The responses for design quality (39 *Very Important* and 53 *Important*) suggest that respondents, while not viewing design quality as important as correctness, still view design quality as an important consideration in their development work.

Figure 8 shows the distribution of effort spent on design quality compared to functional correctness, where each point is an anonymous respondent. Note that
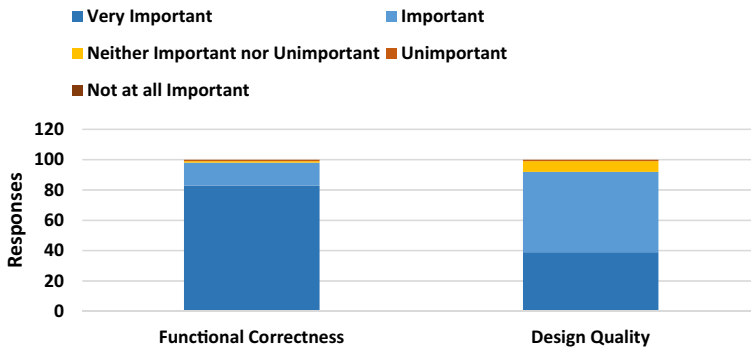
Q6

Fig. 7 Q6—Importance of Functional Correctness versus Design Quality (99 respondents answered both questions, two answered one each, one answered neither question)

respondents were able to choose 'Other Tasks' which is why many of the respondents are not on the diagonal representing a time split totalling 100%. The focus of this question was the split between functional correctness and design quality so no further questions were asked about other tasks.

The chart has been scaled to 100% on each axis, and the data points have also been jittered by $< = 2$. (Jittering is the process whereby a small random amount is added or subtracted from each point to avoid overlapping points in scatter charts (Kitchenham et al. 2002)). The tendency towards the top left in Fig. 8 shows respondents favouring functional correctness over design quality.

Overall, Fig. 8 shows most responses around two-thirds correctness and one-third quality (or about twice the effort spent on correctness compared to design quality), but also highlights a large spread in responses. Almost all of the respondents spend some time ensuring functional correctness, with a few respondents almost exclusively working on this aspect. Some respondents work more on design quality (those to the right of

Fig. 8 Q7—In the course of your software development work, approximately what percentage of your effort is split between the following tasks? (Ensuring Functional Correctness; Ensuring Design Quality; Other Tasks) Note that the data points have been jittered by $< = 2$ on each axis to avoid overlapping points
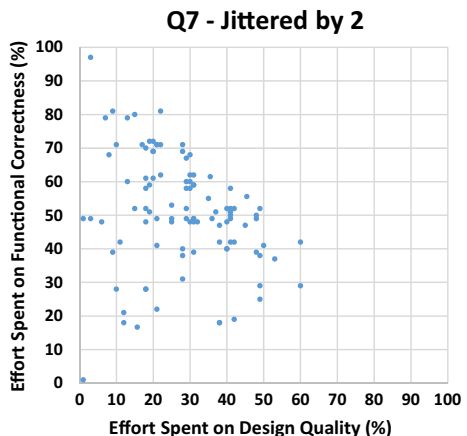
the diagram)—but this aspect has no dedicated full time practitioners (as expected). Finally, one respondent pointed out the importance of both: "*These are not mutually exclusive activities, and unless you're doing both all the time you aren't doing your job.*"

> KF5: Almost all respondents see both functional correctness and design quality as important or very important in their practice. Functional correctness is more important than design quality with respondents typical spending as much as twice the effort on functional correctness as design quality.

The final general design question (Q8) explored the development methodology that most influenced how practitioners do design. The key themes that emerged from this question are generally well known—the importance of an iterative approach and design for change, and the pressures of time and commercial factors. A more general finding was the need to adapt the methodology to each new project rather than always adopting the same set process.

5.3 Design practices—program to an interface

There then followed a series of questions that explored well known design practices. The first of these (Q10) asked about 'program to an interface' (Gamma et al. 1994). A follow-up question (Q11) asked about specific factors that help to decide when to use an interface in a design, rather than a concrete (or abstract) class.

Figure 9 shows that there is positive support for the guideline program to an interface, with 84/102 rating it *Very Important* or *Important* and no respondents rating it *Not at all Important*.

Figure 10 illustrates that there is strong support for many of the program to an interface factors with over 60% of respondents seeing avoiding dependencies, multiple implementations, dependency injection (Fowler 2004) and containing change as *Very Important* or *Important*. On the other hand, there is still a significant group of respondents (at least 25%) who view these criteria neutrally or as unimportant. It is possible that program to an interface is not important in the particular development context that these respondents work, rather than them viewing the practice itself as unimportant.

Table 5 shows the free-text responses for Program to an Interface. From these responses, the key factors that are identified include isolating systems, dependency
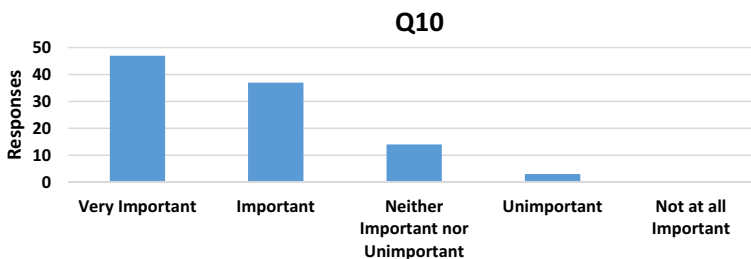

Fig. 9 Q10—How important is Program to an Interface in your design work?
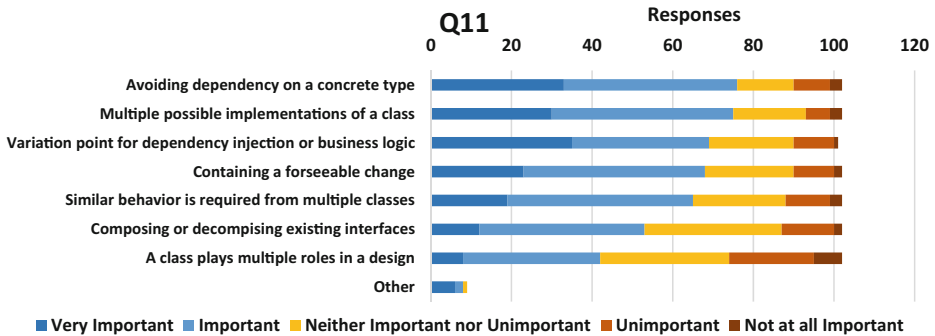
Fig. 10 Q11—How important are the following factors when deciding if an interface should be used? Note that the vertical order is in most-to-least positive (total of the 'important' responses in each category)

injection, 'Single Responsibility' and aiding testing. It is also important to note the warnings regarding potential overuse of interfaces potentially 'cluttering the code base'.

> KF6: Program to an Interface was identified as an important design guideline by a large majority of respondents (83%). The most important motivators for interface use are: depending on abstractions (avoid direct concrete dependency, multiple implementations, dependency injection), isolating (sub-) systems and aiding testability. Respondents warned against overuse of interfaces and stressed the Single Responsibility Principle in interface design.

5.4 Design practices—inheritance

The next section of the questionnaire explored inheritance—its importance (Q13), factors influencing its use (Q14) and the design of hierarchies (Q15).

Table 5 Q12—Feedback for use of interfaces (sample free-text responses)

Factor	Examples	Count
API/isolating systems	"Abstractions for core infrastructure classes such as repositories, data storage, messaging etc." "…layer boundaries in application design."	7
Dependency injection	"Depending on a concrete type becomes acceptable with dynamic or configurable composition."	6
Testability	"As that module inherits that interface and we can run tests against that interface out of the system we can hot-swap a module in a live system without worry of error."	5
Composability/abstraction	"Reducing each class to a single role gives better compartmentalisation and composability…"	5
Overuse of interfaces	"Often the answer is: it depends. Overuse of interfaces can clutter the code base and make changes tedious."	5
One role per class	"Multiple interface inheritance (beyond obvious ones like IDisposable) is a code smell"	4

Figure 11 shows that there was more ambivalence regarding inheritance use than programming to an interface, although overall there were still more respondents rating it *Very Important* or *Important* (55) than *Unimportant* or *Not at all Important* (16).

Figure 12 shows that Code Reuse, Type Substitution and Future Flexibility are all regarded positively (*Very Important* or *Important*) by over half of the respondents (with very few identifying *Other* reasons for inheritance usage). It is notable, however, that some respondents consider these inheritance factors *Not at all Important*.

The free-text responses for Q15 (Table 6) indicate that code-reuse is the most commonly mentioned motivation for using inheritance, followed by reuse of concepts. However, in keeping with the responses to Q14, there are many negative responses regarding inheritance: not used (code smell), prefer composition (Gamma et al. 1994), as well as the view that inheritance hierarchies should be kept shallow (Daly et al. 1996), and that inheritance should follow a type substitution relationship (Liskov and Wing 1994).

Figure 13 shows the factors considered important when designing inheritance hierarchies. There is relatively strong support for consideration of inheritance depth, more modest support for consideration of dependency inversion, and more mixed opinions on inheritance width.

The free-text responses for Q17 (Table 7) provide useful insight into respondents' thoughts on the use of inheritance in object-oriented designs. Again, there are strong views that inheritance should be avoided, and composition preferred, or, when used, it should be limited to inheritance from abstract classes only. In terms of hierarchy design, the most common views suggest that inheritance hierarchies should be 'shallow', model the problem domain, and adhere to 'is-a' relationships. Good tool support was recognised as important in understanding inheritance hierarchies.

> KF7: There was a clear spread of opinion on the role of inheritance in design quality with some practitioners considering it a design flaw. Code reuse, type substitution and future flexibility were all seen as positive reasons for using inheritance. Hierarchies should model the problem domain, adhere to 'is-a' relationships and be kept 'shallow'.

5.5 Design practices—coupling

Respondents were very positive about the role of coupling in considering design quality with 89/102 rating it at least *Important* and none rating it *Not at all Important*, see
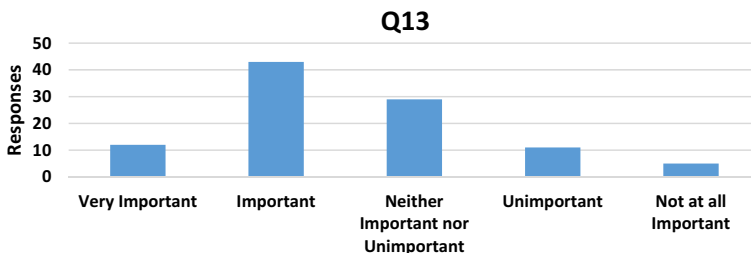

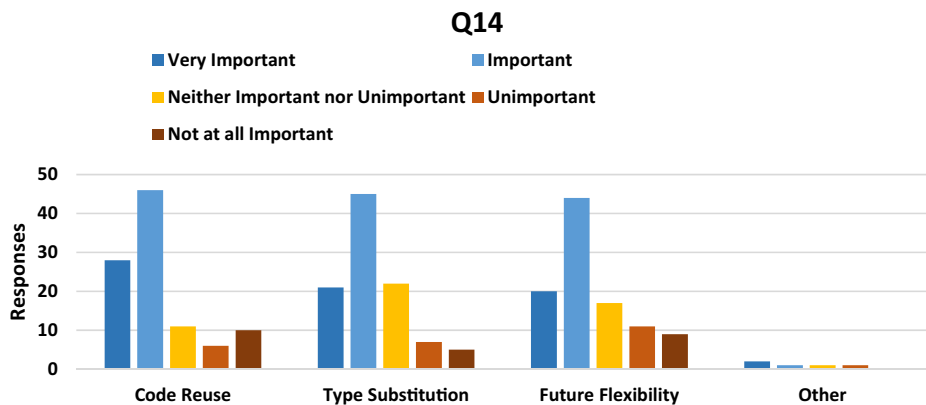**Fig. 11** Q13—How important is inheritance in your design work?

Q14

Fig. 12 Q14—What factors or indicators do you consider to be most important when deciding to use inheritance?

Fig. 14. Coupling received the most positive responses of all the design considerations presented in the questionnaire.

The free-text responses in Table 8 emphasise the popularity of coupling as an indicator of design quality with many practical approaches to its detection. These approaches include the use of tools (SonarQube) and methods such as dependency graphs. Recurring themes of experience and ease of testing were again highlighted, along with guidelines on keeping class interfaces small and simple, being able to understand classes in isolation and avoiding access to, and minimising knowledge of, the state within other classes. One reason why coupling is seen as popular may be the range of practical techniques for detecting it.

> KF8: Respondents viewed coupling as an important consideration in design quality (87% rated at least Important), with the most positive responses out of all the guidelines explored. Many different approaches to minimising coupling were identified including: use of tools, dependency analysis, experience, design guidelines, simple and small class

Table 6 Q15—Deciding on abstract or concrete inheritance

| Factor	Examples	Count
Code reuse	"…very important so as not to use the same code over and over…"	12
Behaviour/concept reuse	"… identify concepts which are core and likely to evolve on the same path and refactor those to use a common base."	9
Not used	"…avoid inheritance where possible … it's always ended up biting me." "Creates code which is hard to read, difficult to maintain."	9
Prefer composition	"Composition of behaviour has proven to be far more flexible and reusable."	7
Flexibility	"Future flexibility … Abstract Inheritance over Object Inheritance."	5
Dependency injection/	"…it's easier and generally considered safer and more testable/scalable, to inject interface dependency and use composition over inheritance."	4
Framework/API	"…abstract inheritance which allows us to share code and API between very similar classes	4
Shallow hierarchy	"…very shallow hierarchies (probably just one (maybe abstract) parent with few direct children), every child class 100% 'is a' [parent class]."	4
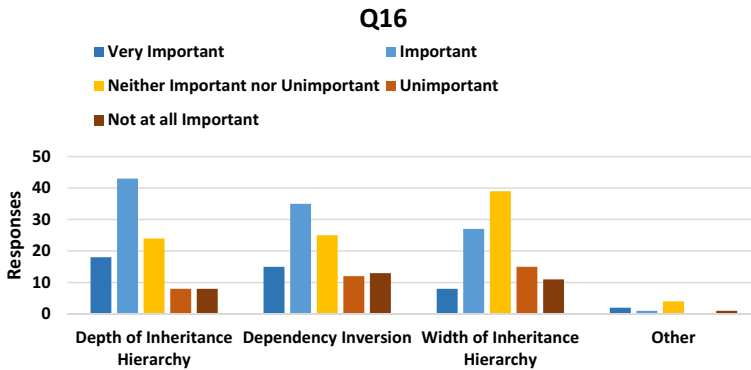
Fig. 13 Q16—When designing inheritance hierarchies, how important are the following features?

interfaces, understanding classes in isolation, avoiding access to state within other classes and ease of testing as an indicator.

5.6 Design practices—cohesion

The summary of responses to Q20, Fig. 15, on the importance of cohesion, again shows a strong positive response (75/102 rating it at least *Important*).

One theme that emerges from the free-text responses on cohesion (Q21), Table 9, is a difficulty in providing clear guidelines for detecting good or bad cohesion—many of the guidelines were statements of other design principles e.g., "Low Coupling" or small size. Indeed, one of the responses acknowledges this "*It's quite hard to explain and measure cohesion…*".

One of strongest positive guidelines was the Single Responsibility Principle (SRP) from SOLID again (Martin 2003). There was some mention of tool use (SonarQube again), metrics and dependency analysis. Once again, ease of testing was highlighted as an indirect indicator of cohesion (difficulty in achieving high coverage).

KF9: Cohesion was identified as an important indicator of design quality (70% at least Important). Respondents suggest that cohesion is harder to assess than other design properties. Guidance on assessing cohesion included Single Responsibility

Table 7 Q17—Dealing with inheritance

| Factor	Examples	Count
Hierarchy structure	"If a design calls for a deep inheritance hierarchy, so be it. Ditto for the width…" "You don't want your ears to pop when traversing down the inheritance hierarchy."	13
Avoidance	"…we very rarely find ourselves using inheritance."	12
Side effects	"…derived types must satisfy the Liskov Substitution Principle … very difficult to achieve, so we try to use composition…"	7
Tools	"…modern IDEs help, but they only help traverse the hierarchy, they don't necessarily aid in understanding the structure."	5

Q18
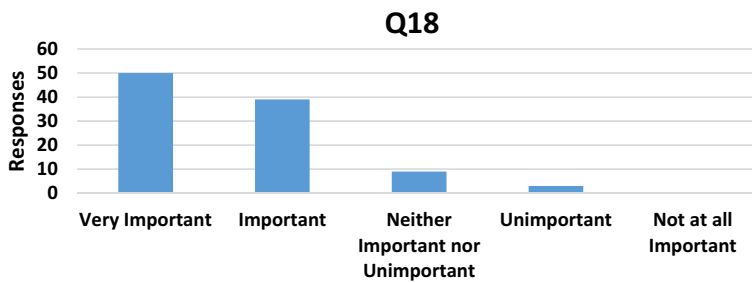
Fig. 14 Q18—How important is the consideration of coupling in your design work?

Principle, use of tools and metrics, experience, and as a by-product of other design guidelines (coupling, size, interface use) and the ease with which classes can be tested.

5.7 Design practices—size

The next question, Q22, explored both class size and method size. Figure 16 summarises responses showing that a clear majority of respondents see both class size and method size as important indicators of design quality. There is a slight indication (about 10% of respondents) that method size is more important. Around 10% of respondents suggest that size is not important to design quality.

In the free-text responses (Q23) many respondents expressed rules of thumb when considering class and method size. For example: "*A method should rarely exceed the size of one page on the screen*". What constitutes a 'screen' will vary amongst working environments. Table 10 summarises the specific size recommendations received.

In contrast to the view that size should generally be 'small', another common view was to avoid excessive decomposition— "*the kind of abstraction that discards essential complexity*". Similar to some views expressed under the coupling and inheritance questions, respondents suggested that some 'substantial' (large, complex, coupled) pieces of code can be central to the design in terms of managing complexity and/or representing the domain. One of the challenges of design appears to be the recognition of such 'essential complexity'.

Table 8 Q19—Dealing with coupling

| Factor	Examples	Count
Tools and methods	"SonarQube, JUnit and the overall TDD approach." "Dependency graphs and a well understood domain model…"	19
Guidelines	"'Keep It Simple', 'Tell don't Ask' and 'Open/Closed' (from SOLID) …" "Essential coupling in the business should show up as essential coupling in the code."	10
Technique	"Explicit API between classes and modules, and prohibiting classes from accessing others' internal state."	10
Smells/problems	"…if you find yourself having to mock too many other objects … your subject under test probably has too many dependencies."	5
No issue	"We don't lose sleep over coupling"	1

Q20
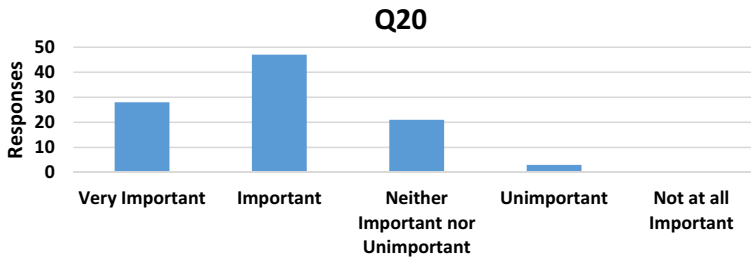
Fig. 15 Q20—How important is cohesion in your design work?

> KF10: Both class and method size were viewed as important design factors (70–80% respondents), though about 10% viewed size as unimportant. Many respondents provided explicit comments on size which tended to constrain classes and methods to be 'small'.

5.8 Design practices—complexity

The next question (Q24) explored the role of complexity in design quality. Like size, there were separate questions on class and method complexity. Figure 17 shows that there was a strong view that complexity is an important consideration in design quality. Seventy to eighty percent of respondents identified both class and method complexity as at least *Important* and less than 5% viewed it as *Unimportant*.

There were many comments made in the free-text section on complexity, see Table 11. One of the main themes that arose was the relationship with size, keeping classes and methods 'small' helps reduce complexity: "*This is the same as size, just measured with different metrics*". Related to this was the recurring theme of striving for simplicity—"*simplify, simplify, simplify*"—and the emphasis once again of SOLID/Clean Code-like principles (Martin 2003, 2009) of separating conditionals into separate methods, one abstraction per method etc. There was some identification of some useful tools (SonarQube) and some metrics: cyclomatic complexity, npath (Nejmeh 1988). Once again, the recurring themes of review, experience and the relationship with the ease of testing were highlighted.

Table 9 Q21—Dealing with cohesion

| Factor	Examples	Count
Guidelines	"…programming to interfaces…should encourage splitting classes…" "Code size is important." "…few dependencies are better…"	9
Tools and methods	"SonarQube uses the LCOM2 metric." "Dependency Structure Matrix" "…low cohesion…often difficult to write tests for to get high coverage."	7
SRP	"One class should do one thing well and nothing else."	7
Middle ground	"…aim for high cohesion in models but certain things like logging libraries, utility classes and other things drag it down a bit…"	3
Not an issue	"Not having cohesion is an indication of a bad design, but I never strive to get it for its own sake."	3
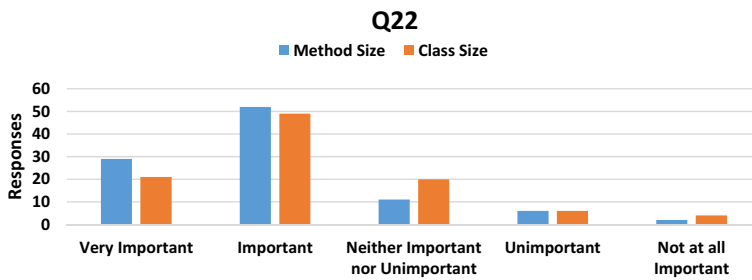
Fig. 16 Q22—Q22. How important is size in your design work? (Method Size; Class Size)

Although the majority of responses suggested that, for the most part, complexity was avoidable and addressable by breaking down classes and methods into ever simpler components, there was also, (again) the recognition that some complexity is 'essential' and is better explicitly modelled as such rather than artificially broken down. Pushing complexity out of a class may increase overall systemic complexity—which is less visible: "*I'd rather have a more complex class than one that removes some of the real complexity*". This raises the fundamental question of what constitutes a 'good' decomposition for a particular problem: "*We do not know what should be the aspects that need separating, and we do not know when it is worth separating them and when it is not*" (Fowler 2003).

> KF11: Both class and method complexity were viewed as important design factors (70–80% respondents), with less than 5% viewing them as unimportant. Practical approaches to managing complexity include the use of tools, metrics and the relationship to small size. More informal approaches include keeping classes and methods 'simple', using experience and reviews, and also the ease of testing. There was recognition that 'essential' complexity in the problem or design domain should be kept together rather than artificially broken up.

5.9 Design practices—design patterns

The second last question on design practices, Q26, explored the role and importance of design patterns, see Fig. 18. Although there is a similar shape to the distribution of responses (~60% *Important*), there was more ambivalence on the role design patterns than the previous design practices. The free-text responses in Table 12 provide more detailed insight into this.

Responses were, in the majority, positive, with comments about their use improving the quality of design and also the 'common language' and common understanding that patterns provide. However, there were also strong opinions expressed about their overuse and the blind application of patterns. Another view is that fundamental design principles such as low coupling and SOLID are more important than design patterns. There was also an issue of how patterns should be used in design, up front— "*...identify up front possible design patterns ...*" - or emerging as appropriate - "*They emerge, they are not the starting point*" —which seemed to be the more common view and more in line with the original proposal for design patterns (Gamma et al. 1994). Nine design patterns were explicitly mentioned by respondents (five of these corresponding to the six identified by Zhang and Budgen as the most favoured patterns in their survey results (Zhang and Budgen 2013)).

Table 10 Q23—Summary of responses on size issues (LOC = Lines of Code)

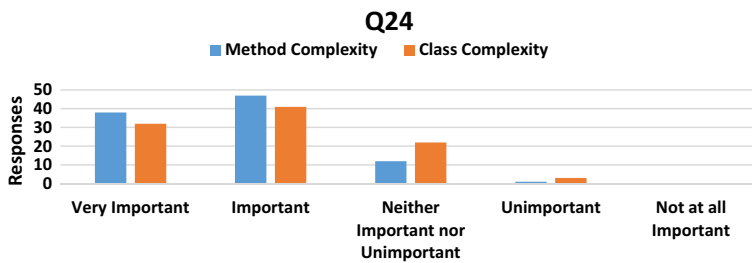| Attribute	Response										
LOC in a Method	1	"a few"	< = 5	<8	5–10	< = 20 non-comment	< = 0	<50	Half page	Page/screen	Short, no fixed limit
Agreement	2	1	1	1	2	1	1	1	1	6	6
Method Parameters	2–3	3–4	< = 5	<7							
Agreement	1	1	1	1							
Class Size	< = 50 LOC	< = 100 LOC	< = 500 LOC	< = 7 non-get/set methods	< = 12 methods	< = 15 methods	<page/screen	<2 pages/-screens			
Agreement	1	1	2	1	1	1	3	1			
Inheritance Depth	<=3										
Agreement	3										

Fig. 17 Q24—How important is Complexity in your design work?

> *KF12: There were mixed views on the role of design patterns as a contributor to design quality, though, overall, most respondents viewed them as at least Important (~60%). They contribute positively to design structure and common understanding. They can also be over-used. There was mixed opinion on whether patterns should be used up-front during design or emerge where necessary and appropriate.*

5.10 Design practices—refactoring

The final section explored the use of refactoring to address each of the previously covered design concerns: complexity, coupling, size etc. Figure 19 summarises the responses, again ordered by popularity. Complexity and size lead the *Most Often* category for refactoring, with 62 respondents saying they *Very Often* or *Quite Often* refactor due to complexity issues. Coupling and cohesion are ranked next. It is striking that the refactoring ranking has a broad similarity to the overall importance respondents gave to the design practices in the previous sections. It may also be that complexity and size degrade more easily, or are easiest to detect, and are also closely related: "*Size and complexity are often related and are much more likely to get out of hand as time goes by*".

The free-text responses on design in general (Q29) are summarised in Table 13. Although, for this question, there was a lack of commonality in the responses, they

Table 11 Q25—Dealing with complexity

| Factor	Examples	Count
Guidelines	"…making sure each class only has a single responsibility… (although at the cost of some increase in the complexity of arranging objects)…"	16
Tools	"Using Sonar is like having a very knowledgeable Java programmer looking over your shoulder…"	13
Definition	"Complexity is everything… the complexity of the solution should match the complexity of the problem."	8
Techniques	"… logical chunks of code are often moved to separate methods…"	5
Experience	"All personal experience, try to keep things as simple as possible."	3
Cause	"… the problem is not sufficiently understood but it can also be an indicator of poor coding and design skills."	2
Problems	"Method complexity is something we consider as it affects our Unit Testing and Code Coverage."	2
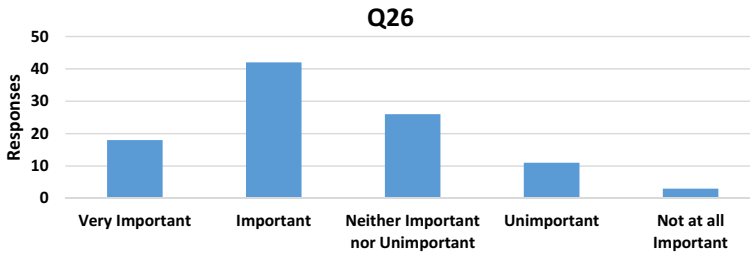
Fig. 18 Q26—How important are Design Patterns in your design work?

do identify a range of techniques for determining when to refactor: when changes are made code should always be improved, peer review, experience, formal and informal metrics, tools and as part of test-driven development. Finally, there was an illuminating comment on why refactoring is perhaps not practised as widely as it should be: "*Refactoring is rare, not because it isn't important but because it is rarely the most important thing you need to do next*".

> KF13: Strong reasons for refactoring were complexity, coupling, size and cohesion, in that order. The relative importance of reasons to refactor broadly matched the overall importance of the corresponding design practice. Developers use experience, peer review, formal and informal metrics, tools and test driven development to determine when to refactor. Ideally, code should be refactored whenever it is changed but other pressures can make this unrealistic.

6 Experience, programming language used and development role

The questionnaire data was also analysed to investigate whether there was any noticeable difference in responses based on experience, main programming language used and software development role. Rather than 'fishing' for statistically significant

Table 12 Q27—Dealing with design patterns

| Factor	Examples	Count
Positive	"General awareness of design patterns is essential for well written code."	15
Usage	"…applied, but very pragmatically. It's not about using DPs for the sake of using them."	13
Common language	"…makes it instantly clear what something does and how it should be used, as we all understand the pattern"	10
Specific pattern	Builder, Factory, Observer, Template, Strategy, MVP, ViewHolder, Singleton, MVC, Composite, Command, Façade	9
Resistance/controversy	"Don't know of any developer I work with that makes use of design patterns."	8
Guidelines	"Simpler rules like inheritance, compositions, SOLID, etc. are better to master than design patterns."	3
Negative	"…younger programmers tend to shoehorn into every single problem… results in a lot of technical debt that other developers have to maintain"	3
Framework	"…framework we use dictates what design patterns we use to interact with the hooks."	2
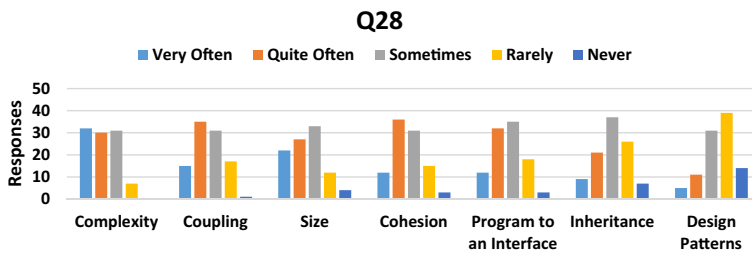
Fig. 19 Q28—How often to you alter or refactor a design due to the following considerations?

results the response data was explored visually using the Qualtrics software. Given the relatively small number of respondents overall, and the even smaller number in each sub-category, the aim was to check if there were any differences amongst responses based on subject background. (Only 97 of the subjects identified their years of experience and so the total numbers shown below are less than the 102 shown elsewhere in the paper.)

For each of the main results reported in this paper (17 in total), a series of line graphs was constructed which enabled visualisation of any major differences in response based on sub-category (experience, language or role). For almost every result, there did not appear to be a major difference amongst sub-categories. Figure 20 shows the typical shape of the graphs, in these cases for Main Programming Language Used versus Inheritance (left) and Experience versus Inheritance (right).

Both graphs show the same general trend as the overall Inheritance result shown in Fig. 11. All the sub-categories which have more than a few respondents in them peak at *Important*, with *Neither Important or Unimportant* the second most popular option. Almost all the other graphs followed a similar breakdown pattern, with each major sub-category following the same trend as the corresponding overall result for the particular question. This suggests a broadly similar view amongst respondents regardless of experience, main programming language used or development role.

There were three cases where some difference in response based on sub-category could be seen. Figure 21 shows a plot of Experience versus Confidence in Design

Table 13 Q29—Final thoughts on design and assessing quality

Factor	Examples	Count
Methods	"By adopting BDD and TDD methods in order to follow the high level Red-Green-Refactor cycle you will find that refactoring happens often…" "Refactoring just means that when you touch code, you should strive to leave it in better shape than when you found it." "The Mikado Method"	9
Indicators/drivers	"Annoyingly, I find a lot of it comes down to experience - sometimes a solution just feels wrong. Once you've refactored it it's easy to show that it's better, but I can't always articulate exactly what's wrong beforehand." "… Bad names allow responsibilities to be assigned incorrectly. Good names make that a little bit harder."	7
Tools/metrics	"Sonar keeps everyone honest on a day-to-day basis." "Using introspection tools and metrics is important."	5

Decision making (see also Fig. 6). It is interesting to compare those with 21 to 40 years of experience (black line) against those with 2 to 20 (orange, brown and yellow) years. (The number of respondents with less than 1 year and over 40 is very small). Thirteen (50%) of the respondents with more than 20 years of experience were *Somewhat Confident* or less in their design decisions while the other 13 (50%) were *Confident*. On the other hand, for those with 20 years or less experience, 45 (63%) were *Somewhat Confident* or less and 26 (37%) were *Confident*. So, perhaps understandably, there is an indication that those with much more experience tend to be more confident in their design decisions.

Figure 22 shows a similar plot to Fig. 21 when comparing Experience against the Usefulness of Technical Lead/Expert Review in ensuring software design quality. The trend towards *Extremely Useful* can be seen in those with 21–40 years of experience (black line), whereas the peak is *Very Useful* for those with less experience. Again this finding is not too surprising—those respondents with the most experience, who might have technical lead responsibilities themselves, were more inclined to see the value of expert reviews in assuring software design quality.

The third plot which showed some variation in response was Experience versus the Importance of Design Quality (see Fig. 23). Again, the variation is based on those with the most experience in the 21–40 years category. While the overall result found respondents suggesting that design quality was *Important* rather than *Very Important* (in contrast to Functional Correctness which is consistently *Very Important* (see Fig. 7), the experienced respondents show more tendency to view design quality also as *Very Important* (*black line*). This would suggest that those respondents with most experience are more inclined to value design quality.

Overall, these findings show that there was a consistency in response for the design topics explored, regardless of years of experience, main programming language used and development role. This consistency across all question topics may help strengthen confidence in the general findings that are being observed. Only three out of the 51 graphs showed any distinctive variation (Figs. 21, 22 and 23). The patterns observed in those three figures suggest that experienced respondents may have more confidence in their design decisions, place more value on the role of technical leads in reviewing design quality and place more importance on the role of design quality in the development process.
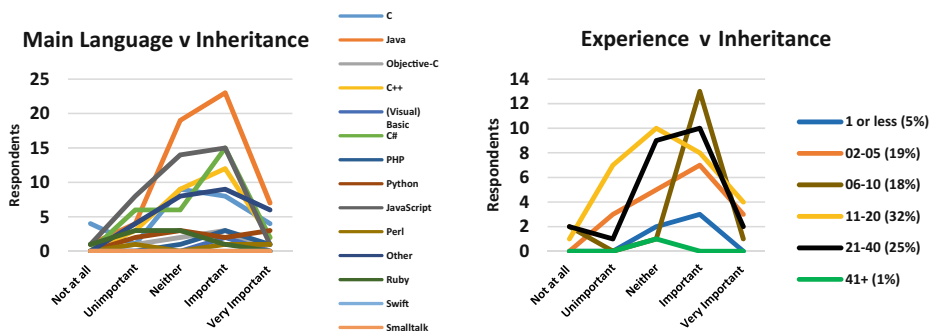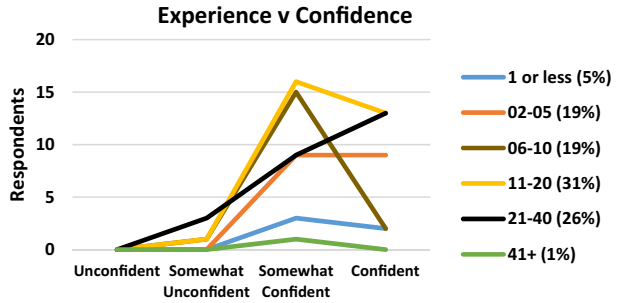

Fig. 20 Response Trends for Importance of Inheritance versus Programming Language (*left*) and Experience (*right*). Percentage of respondents in each category is shown in *brackets*. (See also Fig. 11)

Experience v Confidence

KF14: Findings did not appear to be influenced by experience, programming language or development role, though there was a suggestion that those with most experience are more confident in their design decisions, place more value on reviews by team leads and are more likely to rate design quality as very important

7 Answers to research questions

This section attempts to answer each of the research questions in turn. Some answers to research questions follow directly from the survey questions. For the rest, the Key Findings (KFs) are the main source of answers to the research questions. Table 1 in Section 3 lists the original research questions and the survey questions that were designed with the intention of providing responses that could help answer these research questions.

The first research question (RQ1) was: *To what extent do practitioners concern themselves with design quality?* This question is mainly addressed by answers to the survey questions that explicitly explored functional correctness and design quality (Q6 and Q7). It was clear that practitioner priority is almost always functional correctness over design quality (KF5). However, most respondents said that design quality was still important or very important in their work. As a rough estimate, participants spend around twice as much effort on functional correctness than design quality. Commercial and other time pressures impact on the time spent on design quality.

The second research question (RQ2) was: *How do practitioners recognise software design quality?* This was mainly addressed by answers to survey questions that

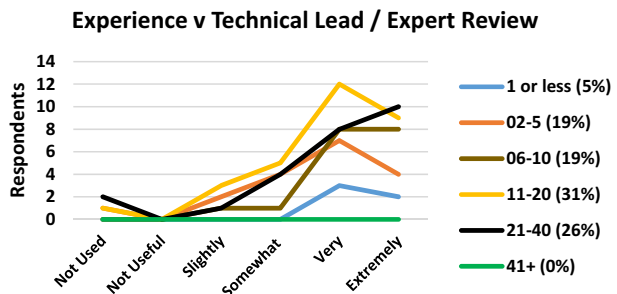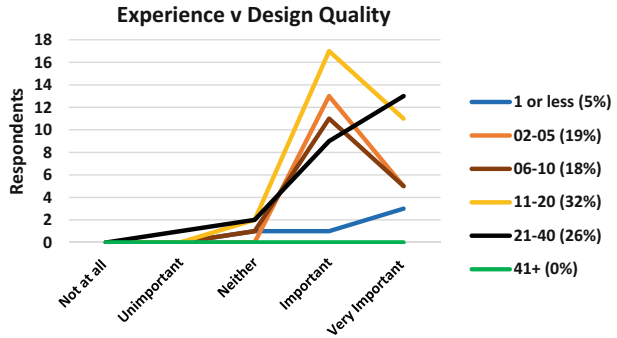**Experience v Technical Lead / Expert Review**

Fig. 23 Experience versus the Importance of Design Quality. Percentage of respondents in each category is shown in *brackets*. (See also Fig. 7)

Experience v Design Quality

Legend:
- 1 or less (5%)
- 02-05 (19%)
- 06-10 (18%)
- 11-20 (32%)
- 21-40 (26%)
- 41+ (0%)

explored possible approaches to quality (Q1, Q5), the follow-up free text questions (Q2) and questions which explored specific design practices (most of the questions Q10 to Q25). The key findings identified a range of mechanisms that practitioners use to determine software design quality: personal experience (KF1), peer and manager review (KF1), design practices: naming practices, 'simplicity'—keep things simple, modularity, documentation, full set of tests (KF3), coupling (KF8), program to an interface (KF6), class and method size (KF10), cohesion (KF9) and class and method complexity (KF11). Respondents also identified the role that tools (e.g., SonarQube), some metrics and standard design guidelines can play in determining design quality.

Overall, it is concluded that designers use a combination of experience, peer review, tools, design guidelines and some metrics to recognise design quality. There appears to be a core of important design guidelines based on small size, minimal complexity, low coupling, good naming, high cohesion, ease of testing, etc. It should be acknowledged, however, that a high level questionnaire such as the one used here is only identifying indicators used by practitioners, indicators which *may* suggest a design problem. The survey has not revealed exactly how they are linked to design quality or the extent to which there is any explicit relationship. It seems that experience stops the blind application of these indicators, determining when to break the guidelines, dealing with new problem domains and technologies, helping to address changing and incomplete specifications, and recognising essential complexity when it arises.

The third research question (RQ3) was: *What design guidelines do practitioners follow?* This was mainly answered by survey questions which explicitly concerned guidelines (Q10 through to Q27). There was clear, strong support for many of the long-standing design guidelines amongst the respondents. Eighty-seven percent of respondents said that coupling was an important or very important design guideline (KF8). Eighty-three percent stated that programming to an interface was at least important (KF6). Seventy percent said that cohesion was important. Over 70% said that both class and method size were at least important (KF10). Over 70% also said that both class and method complexity were at least important (KF11). There were more mixed views on inheritance (KF7) and design patterns (KF12), though both were recognised as potentially important when they are used appropriately.

The fourth and final research question (RQ4) was: *What information do practitioners use to make design decisions?* To answer this research question, responses to a

range of survey questions were used, often from the free-text responses. As highlighted under RQ2 above, it seems clear that respondents believe that experience is a key factor in decision making—designers recognising situations similar to those they have encountered before and reusing design solutions (KF1). As such, unfamiliar problems can cause difficulty in making good design choices. Beyond experience, it is clear that practitioners do make use of the main, established design guidelines (see RQ3) and other well-established design practices such as Clean Code and SOLID. There was evidence of the use of tools (e.g., SonarQube) and some metrics (e.g., LCOM for complexity) in supporting decision making. An interesting finding was the regular mention of the close link between testing and design, higher quality designs tend to be easier to test, and difficulty in writing test cases and achieving test coverage can indicate poor design quality.

8 Discussion

The main contribution from this research is that it emphasises the importance of personal and peer experience to practitioners, the experience that comes from seeing similar problems and knowing what works in those situations. Practitioner experience has long been recognised as important in software development, going back to the days of chief programmer teams (Mills 1971) and the emphasis placed on retaining experience in Lister and DeMarco's Peopleware (Lister and DeMarco 1987). These findings show that the practitioners surveyed here see experience as the most important contribution to design quality. Figure 5, based on Question 7, highlights the relative importance of experience and peer review compared to guidelines tools and metrics (which were still seen as useful). In discussing Lister and DeMarco's Peopleware, Boehm and Turner say: "*The agilists have it right in valuing individuals and interactions over processes and tools ... Good people and teams trump other factors*" (Boehm and Turner 2003). In their recent work on design smells and technical debt, Suryanarayana et al. argue that developers lacking in experience are often responsible for poor quality designs (Suryanarayana et al. 2014). Simons et al. also highlighted the need involve humans within the software design quality assessment process; metrics alone were insufficient (Simons et al. 2015). Wu et al. have also recently found a strong link between developer quality and software quality (Wu et al. 2014).

How can inexperienced designers and novices gain such experience? One answer is learning 'on the job' especially from experienced peers (Begel and Simon 2008). Can education and training do more? A solution might be to ensure that students are exposed to many examples of design, especially high quality design. A useful contribution would therefore be the identification of a range of widely-recognised, high-quality software design case studies to be used in education and training. JHotdraw is often presented as one such example (Ferreira et al. 2012). Ducasse also argues that "*... patterns (and smells) are really effective ways to capture such [design] experience*" (Suryanarayana et al. 2014). Design patterns may therefore have a role to play, being small examples of high quality software design—though previous work (Zhang and Budgen 2012) and the findings from the current survey suggest mixed views on the value of patterns as a learning mechanism.

Respondents said that functional correctness was more important than design quality, but that design quality was still important. There was an indication that practitioners spend about twice as much effort on functional correctness as design quality. Commercial pressures reduce the time afforded to design quality and for refactoring designs. In Yamashita and Moonen's survey of practitioners exploring their knowledge and use of code smells (Yamashita and Moonen 2013a), respondents highlighted that they often had to make trade-offs between code quality and delivering a product on time. Again, Demarco and Lister recognised the commercial pressure of time to market (Lister and DeMarco 1987). The implications of this finding suggest the need for approaches to design quality that fit efficiently and effectively into the development process—this is one reason why experience is so important, but also it also suggests the need for efficient and effective tools and metrics.

It is clear that in this study, respondents viewed many of the long-standing design guidelines as helpful to identify and improve design quality. A large majority of respondents said that coupling was an 'important' or 'very important' indicator of design quality, similarly for programming to an interface, cohesion, class and method size and complexity. The same key design factors were identified as important motivators for refactoring designs. The importance of inheritance and design patterns were more contentious. That respondents see coupling, complexity and size measures as important, and inheritance less so, is in keeping with the findings of Jabangwe et al.'s major systematic literature review (Jabangwe et al. 2015). These findings are also consistent with Al Dallal's finding that most of the cohesion, size and coupling metrics could predict class maintainability (Al Dallal 2013) and also Yamashita and Moonen's finding that the code smells most closely associated with maintenance problems were associated with size, complexity and coupling (Yamashita and Moonen 2013b).

Tools were identified as useful, particularly for checking coupling, cohesion and complexity (with SonarQube receiving many mentions). The results suggest that there is a role for the right tools and that some metrics are useful, at least as indicators of minimum quality or to flag-up areas requiring careful attention. There is therefore a need to identify what tools and metrics are particularly useful, what aspects of these are helpful, and to what extent they can be improved. There is an ongoing research effort to convert general guidance on design guidelines into theoretically sound and empirically demonstrated metrics (Kitchenham 2010). It would appear, though, that despite theoretically weaknesses and inconsistences, metrics are used in practice—perhaps only as indicators, with additional expert judgement being used as the final arbiter.

As an example of the importance of expert judgement being used to temper the blind application of metrics, there was a recurring theme of careful management of essential complexity from the problem domain. Respondents suggested that blocks of essential complexity should be kept together, and understood as a whole rather, than blindly following design rules that would encourage breaking them down into less understandable, less cohesive, smaller units.

An unanticipated but very important contribution of this study was the repeated emphasis that respondents put on Clean Code and the related SOLID principles. Respondents consistently identified a core set of these practices as positively contributing to software design quality. These included 'good' naming practices, 'small' size

of methods and classes, simplicity (or reduced complexity), modularity and documentation, single responsibility, interface segregation and Liskov substitution. This finding raises an important future research question: How much design quality can be achieved just by carefully following a set of such principles (perhaps along with the previous higher-level guidelines and some tool and metrics support)? This relates to Becks view that "... *a certain amount of flexibility just comes from doing a clean job, doing a workman-like job, acting like an engineer*" (Beck 2014).

Another unanticipated and important contribution was the recurring theme of a potential relationship between ease of testing and design quality. The claim is that well-designed software systems are easier to test and, importantly, vice-versa, that using practices such as Test Driven Development (Beck 2003) can encourage better quality designs. In particular, respondents noted that reduced coupling, increased cohesion and reduced complexity all made testing easier. This appears to be consistent with Bajeh's recent work suggesting a correlation between coupling and cohesion metrics and 'testability'(Bajeh et al. 2015). Again, some of this finding may be related to experience, knowing how to construct a design to make it easier to build test cases and achieve coverage. There would appear to be further valuable research to be done exploring the relationship between testing practices and software design quality. Some of this experience may reside in processes and organisations rather than with individuals.

There was also some identification of practices that can have a negative impact on design quality: overuse of interfaces, inheritance not modelling the problem domain, inheritance not adhering to 'is-a' relationships (Liskov substitution), 'non-shallow' hierarchies and overuse of design patterns, particularly by novices. There were also negative comments about "*striving for cohesion for its own sake*" and warnings of excessive decomposition that "*discards essential complexity*". Again, this may be where experience is used to make the final decision rather than the unquestioning application of design guidelines. It is interesting that many of these issues are related to the application of interfaces and particularly inheritance. There is the potential for future research that examines in detail the potential relationship between inheritance practices and design quality.

Finally, these research findings did not appear to be influenced by respondent experience, programming language used or development role, though there was a suggestion that those with most experience are more confident in their design decisions, place more value on reviews by team leads and are more likely to rate design quality as very important.

There are implications from these findings for practitioners, researchers and educators. For practitioners, the findings highlight the importance of experience and peer review. As has been long recognised in the literature, it is vital that organisations recognise, reward and retain their experienced developers. However, it is also important for industry to recognise the need to support less experienced staff and to support their development. Practitioners may also find some of the specific findings reported here on practices and tools useful in their own working environment.

These finding identify a variety of topics for further research. There seems to be little research to-date explicitly exploring the potential of Clean Code and SOLID practices to impact on design quality. Similarly, there is potential to investigate further the possible relationship between ease of testing, testing practices and design quality. This research emphasises the importance of experience to practitioners, there is the potential for research explicitly investigating the contribution experience can make to design quality and related qualities. There is also more research to be done on techniques that support novices in becoming experienced. The findings suggest that design guidelines associated with coupling,

cohesion, size, complexity and programming to an interface are considered useful by practitioners. There is scope for further work investigating exactly how these are being used in practice, whether as final arbiters of quality or just indicators flagging up areas of design and code that require more detailed consideration. If they are only being used as high-level indicators then their measurement-theoretic properties might not be so important. Lastly, there is the possibility of replicating this survey with other practitioners, including refinements to the topics that are discussed in Section 9 Limitations, Improvements and Validity.

There are also implications for the educators of students. Given that experience is so important in design, are there mechanisms that can accelerate students acquisition of appropriate experience? One possibility is through the study of high quality software designs and architectures, perhaps supported by experienced practitioners in doing so. Another is to ensure that students understand the fundamental principles underpinning design guidelines such as coupling, cohesion, complexity, size, and Clean Code practices. At the moment, many students are taught the basics of key techniques such as inheritance, programming to an interface and design patterns —these findings suggest that experienced practitioners know more than this—they appear to know when and how to best use these techniques.

9 Limitations, improvements and validity

The final question (Q37) asked whether the respondents had any comments on the questionnaire itself. The actual feedback here was quite sparse, with only a few issues raised.

- One respondent questioned the focus on object-oriented technology.
- One respondent pointed out that 'agile' and 'iterative' were not mutually exclusive development methodologies (Q8). (The questionnaire text did define what was intended by these two terms.)
- One respondent commented on the lack of questions on testing. As the results of this survey have shown, design quality and testing are closely related. It may have been better to explore this relationship further.
- The questionnaire presented design quality and functional correctness as separate topics. One respondent commented that these should not be viewed separately.

Overall, respondents had very few issues with the wording of the questionnaire, the quality and variety of responses suggest that most had a clear understanding of what was being asked.

If the questionnaire was to be used again, there are a few ways it might be improved. Some of those would be to address the concerns mentioned above—being clearer regarding 'agile' and 'iterative' and perhaps introducing a few questions on the role of testing. Removing the focus on object-oriented technology would completely change the purpose of the questionnaire. The issue of separate consideration of design quality and functional correctness may require some change of wording (though this issue was only raised by one respondent).

With hindsight, the main omission was some exploration of thoughts from respondents on where researchers could mostly usefully contribute to future improvements in software design quality.

In this survey, it is recognised that there are numerous, important threats to validity. In terms of construct validity, perhaps the biggest concern is the choice of design topics explicitly covered by the survey such as program to an interface, coupling and cohesion. It may be that other design topics should also have been explored; however, the length of the questionnaire

and the associated time taken to complete it were major concerns. Most questions had 'Other' options and associated free-text fields, and there was little indication that important topics had been omitted.

The pilot study was intended to help identify any major concerns with the design of the questionnaire. As a result, the original questionnaire was reduced in length, some potentially confusing terminology was clarified and questions exploring the 'Law of Demeter' were removed. An estimation of the time to complete the questionnaire was also included in the introduction.

The most significant threat to internal validity is the methodology used to derive the key findings and associated conclusions. The key findings were derived using a mixture of quantitative and qualitative analyses, in attempt to 'abstract out' the main messages. In a similar manner, these key findings were then merged together to form the main conclusions and answer the initial research questions. It is clear that this procedure is open to researcher bias and others may have identified different findings. To address this, all the original questionnaire data has been made available publicly. In the analyses there are traceable paths from the questionnaire data to the key findings (KF1–14) and then to the corresponding high level conclusions and answers to the original research questions.

The selection of participants for this study is also a significant threat to the internal validity. Initially, participants were identified using industrial contacts of the authors and their university and, thereafter, by searching for industrial authors in the academic literature. This is clearly a limited selection procedure. Furthermore, the respondents who completed the questionnaire have been self-selective and may have particular biases and opinions which they were keen to express, quite possibly views that are not representative of the mainstream. Also, there may be a tendency for subjects to give known, 'best practice' responses. Lastly, there was very limited control of the questionnaire once distributed, it is possible that the questionnaire could be completed carelessly or even dishonestly. It is difficult to address such concerns, though, as stated earlier, given the general consensus and quality of free-text responses, it is believed that these issues have not had a major impact on the findings.

The ordering of questions is another potential threat, again, particularly, the ordering of specific design topics covered. There is the potential for 'rating fatigue' (Zhang and Budgen 2013) where respondents lose interest as they progress. There seems little indication of this, with generally positive responses in initial questions, then more mixed views on inheritance, then being more positive again on coupling and finishing with more mixed responses on design patterns and refactoring.

There are recognised concerns regarding external validity and the generalisability of the findings. The target population was the millions of industrial software developers working around the globe. A key question is whether the actual population sampled is a representative subset of that target population (Kitchenham and Pfleeger 2002c). The respondents in this survey appear to have more industrial experience compared to related work (45% have over 10 years); work in a wide range of locations around the world are mostly active programmers (93/102) and have experience of multiple software development activities. The textual feedback provided in the questionnaires showed many respondents responding thoughtfully and knowledgeably which increases confidence in the relevance of the results.

Finally, the number of respondents is relatively small (102), certainly in comparison to the potential size of the target population. However, this size is in keeping with that of recent, comparable studies.

10 Conclusions

Based on quantitative and qualitative analysis, drawing together key findings and trying to answer the original research questions, the following conclusions emerge:

1. Design quality is seen as important to industrial practitioners, but not as important as functional correctness.
2. Commercial and other time pressures limit the time spent on design quality and associated activities such as refactoring.
3. Practitioner experience is the key contributor to design quality, both when working as an individual and as part of the peer review process.
4. Novel problems and changing or imprecise requirements make design more difficult.
5. It is important to recognise essential complexity in a problem and be mindful of how it is distributed in design.
6. Many of the traditional design guidelines are viewed as important to design quality: coupling, cohesion, complexity, size and programming to an interface. Inheritance and design patterns are also important but controversial and susceptible to misuse.
7. There are many lower level practices identified as contributing to good quality design, e.g., SRP, LSP, OCP, small size, simplicity and good naming practices.
8. Tool support such as SonarQube is often used to support software design quality; there was also some evidence to support the use of metrics.
9. There is a close relationship between testing practices and design quality. Good testing methodology can help improve design; testable designs are thought to be better, and poor designs can be harder to test.
10. The findings did not appear to be influenced by experience, programming language or development role, though there was a suggestion that those with most experience are more confident in their design decisions, place more value on reviews by team leads and are more likely to rate design quality as very important.

The contribution of this research is that the practitioners who were surveyed saw experience and peer review as the most important contribution to design quality. These practitioners do generally find design guidance (coupling, cohesion, complexity and size), metrics and tools useful but not as important as experience. The practitioners see design quality as important but not as important as functional correctness. More surprising, perhaps, was the importance of Clean Code/SOLID principles to practice—as important as traditional guidelines and metrics— and also the potential relationship between 'testability' (ease of testing) and design quality.

These findings have implications for researchers, practitioners and educators. For researchers, more work could be done investigating the role of experience in the design process—to what extent can the nebulous notion of experience be translated into better guidelines, metrics and tools? The survey has only identified the high-level practices that those working in industry find to be useful indicators of quality. Much more work needs to be done with industry to discover exactly how these indicators are used in practice and the extent to which an explicit link can be demonstrated between them and design quality. Other practitioners may find useful advice and lessons within these results to help improve their own design practices. Finally, the results help to identify some of the fundamental techniques that educators should emphasise to their students, ideally using high-quality examples of industrial-strength design case studies to do so.

Appendix - Survey questions

1	How important is each of the following in ensuring the quality of software design in your development process? *Personal Experience; Design Guidelines; Software Metrics; Design Tools; Peer Review; Team Review; Technical Lead / Expert Review; Other*	Likert
2	Please use this space to provide any details that would help to support or clarify your answers above.	Free Text
3	When making design decisions, how confident are you that you have chosen the best option from the alternatives you have considered?	Likert
4	What types of problem make it difficult to have confidence in a design decision?	Free Text
5	What are the key features that allow you to recognise good design in your own work and in the work of others?	Free Text
6	How important are the following elements in your work? *Functional Correctness; Ensuring Design Quality; Other Tasks*	Likert
7	In the course of your software development work, approximately what percentage of your effort is split between the following tasks? *Ensuring Functional Correctness; Ensuring Design Quality; Other Tasks*	Number (%)
8	What software development methodology most influences your approach to design? *Process based (*e.g. *Waterfall); Iterative (*e.g. *RUP); Agile based (*e.g. *Scrum); Other*	Multi-Choice + Other
9	If you have any other comments about how development methodology affects design quality in your work, please note them here.	Free Text
10	How important is Program to an Interface in your design work?	Likert
11	How important are the following factors when deciding if an Interface should be used? *Avoiding dependency on a concrete type; Multiple possible implementations of a class; A class plays multiple roles in a design; Similar behaviour is required from multiple classes; Composing or decomposing existing interfaces; Containing a foreseeable change (flexibility/hotspot); Variation point for dependency injection or business logic; Other*	Likert
12	Metrics or tools which you find useful for managing Interfaces.	Free Text
13	How important is Inheritance in your design work?	Likert
14	Which factors or indicators do you consider to be most important when deciding to use Inheritance? *Code Reuse; Type Substitution; Future Flexibility; Other*	Likert
15	Which factors do you consider to be most important factors when deciding between Object Inheritance (concrete super class) or Abstract Inheritance (abstract super class)?	Free Text
16	When designing inheritance hierarchies, how important are the following features? *Depth of Inheritance Hierarchy; Width of Inheritance Hierarchy; Dependency Inversion (keeping concrete classes at the fringes of the inheritance hierarchy); Other*	Likert
17	Please use this space to provide any additional details about how you deal with Inheritance, such as any Guidelines, Metrics, or Tools that make this task more manageable.	Free Text
18	How important is the consideration of Coupling in your design work?	Likert
19	Are there any Guidelines, Metrics, or Tools that you find especially useful when thinking about Coupling?	Free Text
20	How important is Cohesion in your design work?	Likert
21	Are there any Guidelines, Metrics or Tools that you find especially useful when thinking about Cohesion?	Free Text
22	How important is Size in your design work? *Method Size; Class Size*	Likert

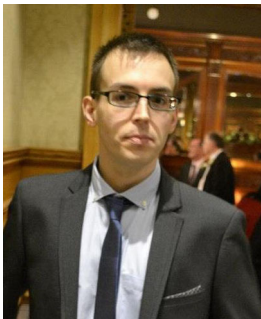| 23	Are there any Guidelines, Metrics or Tools that you find especially useful when thinking about Size?	Free Text
24	How important is Complexity in your design work? *Method Complexity; Class Complexity*	Likert
25	Are there any Guidelines, Metrics or Tools that you find especially useful when thinking about Complexity?	Free Text
26	How important are Design Patterns in your design work?	Likert
27	Please provide any other details on how you deal with Design Patterns in your designs in this space.	Free Text
28	How often do you alter or refactor a design due to the following design considerations? *Program to an Interface; Inheritance; Coupling; Cohesion; Size; Complexity; Design Patterns*	Likert
29	Are there any Guidelines, Metrics or Tools that have not already been discussed that you find particularly helpful when designing software or assessing design quality?	Free Text
30	In which country is your software house or production environment located?	Multi-Choice
31	What is the nature of the software development you have taken part in? (Please select all that apply)	Multi-Choice
32	How many years have you been involved in the software industry as a designer/developer?	Multi-Choice (ranges)
33	What formal training have you completed in relation to your role as a designer? *College; University; Masters; PhD; Apprenticeship; Industry Accreditation; Other*	Multi-Choice + Other
34	What programming language(s) do you use when developing software?	Multi-Choice + Other
35	What types of design problems are you most familiar with?	Multi-Choice + Other
36	What role(s) have you carried out in a software development environment?	Multi-Choice + Other
37	If you have any final thoughts or comments on Design Quality or on the survey in general, please note them here.	Free Text

References

Agner, L. T. W., Soares, I. W., Stadzisz, P. C., & Simão, J. M. (2013). A Brazilian survey on UML and model-driven practices for embedded software development. *Journal of Systems and Software, 86*(4), 997–1005.

Al Dallal, J. (2013). Object-oriented class maintainability prediction using internal quality attributes. *Information and Software Technology, 55*(11), 2028–2048.

Al Dallal, J., & Briand, L. (2012). A precise method-method interaction-based cohesion metric for object-oriented classes. *ACM Transactions on Software Engineering and Methodology (TOSEM), 21*(2), 8.

Al Dallal, J., & Morasca, S. (2014). Predicting object-oriented class reuse-proneness using internal quality attributes. *Empirical Software Engineering, 19*(4), 775–821.

Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix, J., & Pugh, W. (2008). Using static analysis to find bugs. *IEEE Software, 25*(5), 22–29.

Bajeh, A. O., Basri, S., & Jung, L. T. (2015). An Empirical Validation of Coupling and Cohesion Metrics as Testability Indicators. In *Information Science and Applications* (pp. 915–922). Springer.

Baker, A., van der Hoek, A., Ossher, H., & Petre, M. (2012). Guest Editors' introduction: studying professional software design. *Software, IEEE, 29*(1), 28–33.

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Beck, K. (2014). Software Design: Why, When & How. JavaZone 2014 Conference.

Beck, L., & Perkins, T. (1983). A survey of software engineering practice: tools, methods, and results. *IEEE Transactions on Software Engineering*, (5), 541–561.

Begel, A., & Simon, B. (2008). Novice software developers, all over again. In *Proceedings of the Fourth international Workshop on Computing Education Research, 2008* (pp. 3–14). ACM.

Boehm, B., & Turner, R. (2003). *Balancing Agility and Discipline: A Guide for the Perplexed, Portable Documents*. Addison-Wesley Professional.

Booch, G. (2011). Draw me a picture. *IEEE Software, 28*(1), 6.

Briand, L., & Wüst, J. (2002). Empirical studies of quality models in object-oriented systems. *Advances in Computers, 56*, 97–166.

Briand, L., Bunse, C., & Daly, J. (2001). A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Transactions on Software Engineering, 27*(6), 513–530.

Brooks Jr, F. (2010). *The design of design: Essays from a computer scientist*. Pearson Education.

Brown, W., Malveau, R., McCormick, H., & Mowbray, T. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

BSI (2011). Systems and software engineering. Systems and software quality requirements and evaluation (SQuaRE). System and software quality models.

Buse, R., & Zimmermann, T. (2012). Information needs for software development analytics. In *Proceedings of the 34th international conference on software engineering* (pp. 987996). IEEE Press.

Campbell, G., & Papapetrou, P. P. (2013). *SonarQube in Action*. Manning Publications Co.

Carver, J., Dieste, O., Kraft, N. A., Lo, D., & Zimmermann, T. (2016). How Practitioners Perceive the Relevance of ESEM Research. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 56). ACM.

Cass, S. (2015). The 2015 top ten programming languages. *IEEE Spectrum, July, 20*.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493.

Coad, P., & Yourdon, E. (1991). *Object-oriented design* (Vol. Vol. 92). NJ: Yourdon Press Englewood Cliffs.

Conley, C. A., & Sproull, L. (2009). Easier said than done: an empirical investigation of software design and quality in open source software development. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on.* (pp. 1–10). IEEE.

Copeland, T. (2005). *PMD Applied: An Easy-to-Use Guide for Developers*. Centennial Books.

Cunningham, W. (1993). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger, 4*(2), 29–30.

Daly, J., Brooks, A., Miller, J., Roper, M., & Wood, M. (1996). Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering, 1*(2), 109–132.

Devanbu, P., Zimmermann, T., & Bird, C. (2016). Belief & evidence in empirical software engineering. In *Proceedings of the 38th International Conference on Software Engineering* (pp. 108–119). ACM.

Dyba, T., Kitchenham, B. A., & Jorgensen, M. (2005). Evidence-based software engineering for practitioners. *IEEE Software, 22*(1), 58–65.

Ferreira, K. A., Bigonha, M. A., Bigonha, R. S., Mendes, L. F., & Almeida, H. C. (2012). Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software, 85*(2), 244–257.

Fontana, F. A., Dietrich, J., Walter, B., Yamashita, A., & Zanoni, M. (2016). Antipattern and Code Smell False Positives: Preliminary Conceptualization and Classification. In *2016 I.E. 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Vol. 1, pp. 609–613). IEEE.

Fowler, M. (2003). Who needs an architect? *IEEE Software, 20*(5), 11–13.

Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html. Accessed Jul 2016.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). Refactoring: Improving the design of existing programs. Addison-Wesley Reading.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Garousi, V., & Zhi, J. (2013). A survey of software testing practices in Canada. *Journal of Systems and Software, 86*(5), 1354–1376.

Garousi, V., Coşkunçay, A., Betin-Can, A., & Demirörs, O. (2015). A survey of software engineering practices in Turkey. *Journal of Systems and Software, 108*, 148–177.

Gorschek, T., Tempero, E., & Angelis, L. (2014). On the use of software design models in software development practice: an empirical investigation. *Journal of Systems and Software, 95*, 176–193.

Hitz, M., & Montazeri, B. (1995). *Measuring coupling and cohesion in object-oriented systems.* Paper presented at the Proceedings of the International Symposium on Applied Corporate Computing Mexico.

Hitz, M., & Montazeri, B. (1996). Chidamber and Kemerer's metrics suite: a measurement theory perspective. *IEEE Transactions on Software Engineering, 22*(4), 267–271.

Jabangwe, R., Börstler, J., Šmite, D., & Wohlin, C. (2015). Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. *Empirical Software Engineering, 20*(3), 640–693.

Khomh, F., Di Penta, M., Guéhéneuc, Y.-G., & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering, 17*(3), 243–275.

Kitchenham, B. (2010). What's up with software metrics?–a preliminary mapping study. *Journal of Systems and Software, 83*(1), 37–51.

Kitchenham, B., & Pfleeger, S. (1996). Software quality: the elusive target. *IEEE Software, 13*(1), 12.

Kitchenham, B., & Pfleeger, S. (2002a). Principles of survey research part 2: designing a survey. SIGSOFT Softw. *Eng. Notes, 27*(1), 18–20.

Kitchenham, B., & Pfleeger, S. (2002b). Principles of survey research: part 3: constructing a survey instrument. *ACM SIGSOFT Software Engineering Notes, 27*(2), 20–24.

Kitchenham, B., & Pfleeger, S. (2002c). Principles of survey research: part 5: populations and samples. *ACM SIGSOFT Software Engineering Notes, 27*(5), 17–20.

Kitchenham, B., Pfleeger, S., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K., et al. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering, 28*(8), 721–734.

Larman, C. (2004). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. In (3 ed.). Prentice Hall.

Lieberherr, K., Holland, I., & Riel, A. (1988). Object-oriented programming: an objective sense of style. *ACM SIGPLAN Notices, 23*(11), 323–334.

Linaker, J., Sulaman, S. M., Maiani de Mello, R., & Host, M. (2015). Guidelines for Conducting Surveys in Software Engineering.

Liskov, B. H., & Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS), 16*(6), 1811–1841.

Lister, T., & DeMarco, T. (1987). *Peopleware: Productive projects and teams*. New York: Dorset House.

Marinescu, R. (2012). Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development, 56*(5), 9: 1–9: 13.

Martin, R. C. (2003). *Agile software development: principles, patterns, and practices*. Prentice Hall PTR.

Martin, R. C. (2005). Principles of OOD. http://www.butunclebob.com/. Accessed Jul 2016.

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Mayer, T., & Hall, T. (1999). A critical analysis of current OO design metrics. *Software Quality Journal, 8*(2), 97–110.

McLaughlin, B., Pollice, G., & West, D. (2006). *Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D*. " O'Reilly Media, Inc.".

Mills, H. D. (1971). Chief programmer teams: Principles and procedures. *IBM Federal Systems Division Report FSC*, 71–5108.

Nejmeh, B. A. (1988). NPATH: a measure of execution path complexity and its applications. *Communications of the ACM, 31*(2), 188–200.

Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S., & Hemati Moghadam, I. (2012). Experimental assessment of software metrics using automated refactoring. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement* (pp. 49–58). ACM.

Ó Cinnéide, M., Moghadam, I. H., Harman, M., Counsell, S., & Tratt, L. (2016). An experimental search-based approach to cohesion metric evaluation. *Empirical Software Engineering*, 1–38.

Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., & Oliveto, R. (2016). Smells like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*. Raleigh, USA: IEEE.

Pfleeger, S. L., & Kitchenham, B. A. (2001). Principles of survey research: part 1: turning lemons into lemonade. *ACM SIGSOFT Software Engineering Notes, 26*(6), 16–18.

Pressman, R. S. (2014). *Software engineering: a practitioner's approach* (8 ed.).

Radjenović, D., Heričko, M., Torkar, R., & Živkovič, A. (2013). Software fault prediction metrics: a systematic literature review. *Information and Software Technology, 55*(8), 1397–1418.

Riaz, M., Mendes, E., & Tempero, E. (2009). A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement* (pp. 367–377). IEEE Computer Society.

Riel, A. J. (1996). *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc.

Saldaña, J. (2015). *The coding manual for qualitative researchers*. Sage.

Shull, F., Singer, J., & Sjøberg, D. I. (2008). *Guide to advanced empirical software engineering* (Vol. 93). Springer.

Simons, C., Singer, J., & White, D. R. (2015). Search-based refactoring: Metrics are not enough. In *International Symposium on Search Based Software Engineering* (pp. 47–61). Springer.

Sjøberg, D. I., Anda, B., & Mockus, A. (2012). Questioning software maintenance metrics: a comparative case study. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement* (pp. 107–110). ACM.

Sjøberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., & Dybå, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering, 39*(8), 1144–1156.

Sommerville, I. (2010). *Software Engineering. International computer science series* (9ed., ed). Addison Wesley.

Stavru, S. (2014). A critical examination of recent industrial surveys on agile method usage. *Journal of Systems and Software, 94*, 87–97.

Suryanarayana, G., Samarthyam, G., & Sharma, T. (2014). *Refactoring for software design smells: Managing technical debt*. Morgan Kaufmann.

Torchiano, M., Tomassetti, F., Ricca, F., Tiso, A., & Reggio, G. (2013). Relevance, benefits, and problems of software modelling and model driven techniques—a survey in the Italian industry. *Journal of Systems and Software, 86*(8), 2110–2126.

Van Solingen, R., Basili, V., Caldiera, G., & Rombach, H. D. (2002). Goal question metric (gqm) approach. *Encyclopedia of software engineering*.

Veerappa, V., & Harrison, R. (2013). An empirical validation of coupling metrics using automated refactoring. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 271–274). IEEE.

Wu, Y., Yang, Y., Zhao, Y., Lu, H., Zhou, Y., & Xu, B. (2014). The Influence of Developer Quality on Software Fault-Proneness Prediction. In *Software Security and Reliability (SERE), 2014 Eighth International Conference on.* (pp. 11–19). IEEE.

Yamashita, A., & Moonen, L. (2013a). Do developers care about code smells? An exploratory survey. In *WCRE* (Vol. 13, pp. 242–251).

Yamashita, A., & Moonen, L. (2013b). To what extent can maintenance problems be predicted by code smell detection?–an empirical study. *Information and Software Technology, 55*(12), 2223–2242.

Zhang, C., & Budgen, D. (2012). What do we know about the effectiveness of software design patterns? *IEEE Transactions on Software Engineering, 38*(5), 1213–1231.

Zhang, C., & Budgen, D. (2013). A survey of experienced user perceptions about software design patterns. *Information and Software Technology, 55*(5), 822–835.

Jamie Stevenson is a third year PhD student in Computer Science at the University of Strathclyde, Glasgow. His research interests are in software engineering, particularly software design. His PhD research includes practitioner attitudes to object-oriented design quality and patterns of interface and inheritance usage in open-source systems.

Murray Wood is a senior lecturer in Computer Science at the University of Strathclyde Glasgow. He holds a PhD in Computer Science, also from the University of Strathclyde. His research interests are in empirical software engineering. The recent focus of his work is software design, specifically studying design practices used in large-scale software systems.