Chapter 7
Software Inspections, Code Reviews, and Safety Arguments

In the chapter on software testing, we have seen that there are numerous strategies by which to assess the quality of a software system by executing it. However, the effectiveness of testing is subject to a range of limitations; there is rarely a complete and reliable oracle, and there is no accepted means by which to generate adequate test sets (or even by which to measure adequacy). Furthermore, many aspects of software quality (such as the maintainability of the source code) cannot be established by testing.

Software inspections and reviews are concerned with the (usually manual) review of software artefacts. Whereas the goal with testing is relatively narrow (to establish correct behaviour), the goals with inspections can be broader; reviewers can consider whether the architecture is sensible, the code is maintainable, a code change is necessary, whether there are better potential solutions, etc. Inspections and code reviews are widely used within organisations, and in open source projects. The requirement for peer-review is an integral part of most development and quality assurance processes and tools.

In this chapter we will briefly cover the origins of software inspections and reviews. In their original form, software inspections were perceived to be too "heavy-weight" for routine software development. This led to the development of new, lightweight "Modern Code Review" techniques, which we will cover in Section 7.2. The more heavyweight traditional inspections do however remain a key part of the development of safety-critical systems. We will thus look at inspections – specifically the safety-specific aspects of software inspections – in Section 7.4.

It is worth noting that the placement of sections on "conventional" software inspections alongside sections on safety assessments of software is somewhat unorthodox; in a traditional text book they would probably be in separate chapters. They are put together here because, conceptually, they are fundamentally related; they involve the manual scrutiny of software artefacts, with the aim of detecting problems and ensuring quality.

N. Walkinshaw, *Software Quality Assurance*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-64822-4_7

7.1 Formal Inspections

Software inspections are well-established. Michael Fagan originally developed the notion in 1976 [48]. He proposed what are now known as "formal" inspections, which were geared towards structured organisations. The idea was that inspections would take place in structured meetings, where different stakeholders would prepare for the meeting by carefully reading through the document in question (e.g. the source code), and would then discuss this during the meeting.

This then led to a very substantial amount of research throughout the 80s and 90s, which focussed on the specifics of reviews and inspection meetings, with a view to maximising their effectiveness. This culminated in various structured reading techniques, such as Active Design Reviews [108] and Perspective Based Reading [16], which were underpinned by extensive empirical studies indicating their efficacy.

Nevertheless, inspections never became particularly widespread in their formal incarnation. They were largely intended for structured work-flows such as the Waterfall model (see section 3.2.1), and failed to adapt to emergent agile processes. The need for synchronicity (for inspectors to meet and discuss a piece of code at the same time) was at odds with the tendency towards distributed software development teams, where different groups of individuals would work on different pieces of code at different times. There also emerged several studies that showed that they did not (in their meeting-centric form) tend to improve defect detection efforts, despite their high costs [75].

This eventually led to the development of more light-weight inspection techniques that could more easily be integrated into routine software development practices. For example, Pair Programming (see section 7.3.2.2), which became an integral part of Extreme Programming, is an apt example of a peer-review technique that does not revolve around costly meetings.

7.2 Modern Code Reviews - Reviewing Code During Development

In recent years, the term 'Modern Code Review'[12] (MCR) has emerged to refer to this new family of inspections. MCR does not rely on face-to-face meetings. On the contrary, it commonly presumes that developers are distributed, and are operating asynchronously. To enable this, MCR operates on tools and processes that are based upon the use of a version repository (as introduced in Chapter 3).

The specifics of MCR vary extensively from one development context to another, and often depend upon the nature of the development process (e.g. whether or not it is agile), the makeup of the development teams, and the work-flow required by the version-control system or the associated code review tools that are used. This section will set out some of the most common MCR features and practices.

7.2.1 Tool-Driven Code Review

One major factor that distinguishes MCR from traditional inspections is the fact that MCR focusses on what are usually small, manageable updates to an underlying system (such as the patches that form a commit to a version repository), as opposed to entire files or modules. To enable reviews, reviewing tools such as Gerrit[1] can easily be integrated with version repositories to act as 'gate keepers' to commits. Aside from Gerrit (which was developed by Google for the development of its Android operating system), other notable examples of reviewing tools include Microsoft's CodeFlow tool [12], and Facebook's Phabricator tool[2].

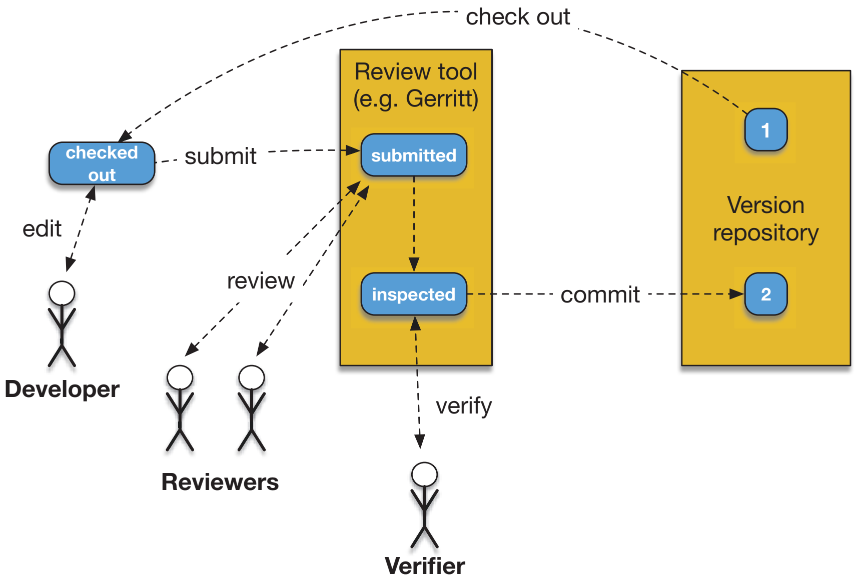

Fig. 7.1 Example of a typical MCR-flow, using a code reviewing tool.

A typical work-flow is illustrated in Figure 7.1. A developer will check out a version from the repository and make their change. This change is then submitted for review (this is often automatically initiated when the developer tries to push their change to the central version repository). Before the commit can be finalised, the patch submitted by the reviewer is subject to review by a pool of reviewers. If the patch is accepted by the reviewers, it is then passed to someone who 'verifies' it - e.g. by testing it, and finally commits the reviewed, verified code fragment to the version repository.

[1] `https://gerrit.googlesource.com/gerrit`

[2] `http://phabricator.org/`

Although MCR is commonly associated with code reviewing tools, these are not essential. Most versioning systems (see Section 4.2.2) provide plenty of mechanisms that can be employed for code review without the use of explicit tools. For example, one convention is to use branching and merging; developers use dedicated branches to make their changes, but these changes are only merged to the 'trunk' by a dedicated user, and only after a review. Reviewers can use plenty of tools (such as repository logs) to inspect the various changes that were made to the code under review. The challenge, in the absence of appropriate code review tools, is to ensure that developers ultimately bother to go through the various reviewing steps, which can easily be neglected if they are not enforced.

7.2.2 Pull-Based Development

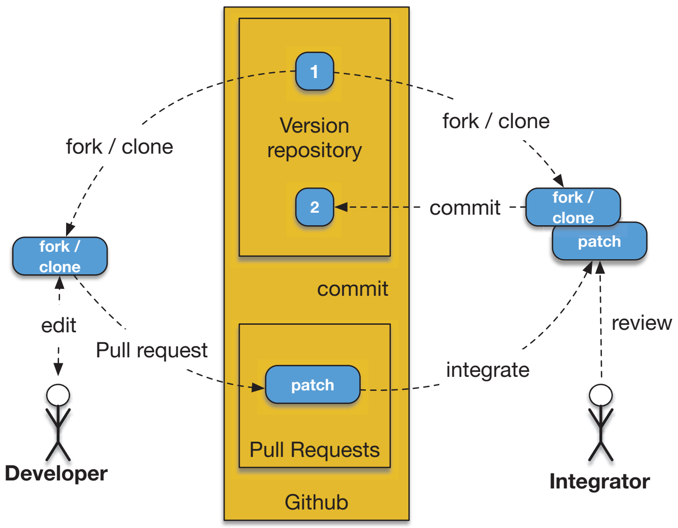

Fig. 7.2 Pull-Based Development

One 'tool-less' development approach that enforces MCR is known as 'Pull-Based Development' [62]. The approach revolves around a mechanism known as a 'Pull Request'. The flow is illustrated in Figure 7.2. A developer (who does not have the privileges to push their changes directly to the central repository) takes a clone of the code from the repository. They make their changes, which are typically small (of the order of a few dozen lines of code [62]). They then submit their proposed changes to the repository as a 'pull request'. A developer with appropriate privileges (often referred to as an 'integrator') then examines their proposed changes, integrates them with their own clone of the source code, and makes the commit. This integration step can often be automated (if there have not been too many intervening changes).

Pull requests are becoming increasingly popular in open source systems because they enable *anybody* to at least attempt to make contributions to any open source system (that supports pull requests). Reviews and discussions of pull requests do not necessarily have to be restricted to a small band of privileged developers, but can be opened up to the wider community of developers. This can, at least in principle, open up software development to greater scrutiny, and a higher level of shared decision making.

It is instructive to skim some of the open-source projects on GitHub to witness how extensively MCR is embedded within large software development projects. We can take, for example, a relatively extensive change proposed as a pull request to the Erlang OTP libraries: `https://github.com/erlang/otp/pull/960`. The pull request includes changes to 52 files within the libraries, and is accompanied by a discussion of 121 (largely substantial and constructive) comments, eventually culminating in the acceptance of the proposed feature.

7.2.3 The Impact of MCR on Software Development and Quality

MCR has, as a practice, emerged out of necessity. Different organisations developed their own internal reviewing approaches. As a result, there is no single canonical definition of the "rules of MCR"; the term covers a relatively broad range of practices. This makes it difficult to analyse or discuss how it is used. Nevertheless, there have been some recent efforts to study the trends underlying MCR, and these are discussed here.

In their 2013 study, Rigby *et al.*[112] analysed the review practices for thirteen projects, including large open-source projects such as Android, Chromium OS, Apache, and Linux. They found the following:

- **Reviews tend to occur before changes have been committed (as per Figure 7.1), and occur frequently.** At AMD the median time for the completion of a review was 17.5 hours. At Microsoft the median review time for the Bing, SQL, and Office projects was 14.7, 19.8, and 18.9 hours respectively. Office had a median of 4384 reviews per month.
- **Change sizes (and thus reviewer loads) are small.** The median change size observed for most of the active open source and industrial projects was below 100 lines of code. For Android and AMD the average was 44 lines.
- **The number of reviewers that typically take part in a review is 2.** This is in line with findings from the era of formal code reviews that 2 reviewers are optimal for defect detection meetings [135].
- **Review is about more than just detecting defects.** Rigby and Bird noticed that the goal of reviews was rarely to record defects – in fact, current reviewing platforms rarely offer such a feature. Instead, the nature of discussions tends to be

'perfective'; reviewers and code authors share the goal of refining and improving a submitted piece of code to a point where it can be merged back into the main code base.

In a similar study, focussed specifically on the use of MCR at Microsoft (with the CodeFlow tool), Bacchelli and Bird[12] interviewed developers who used MCR and analysed their respective changes. Their interview responses emphasised the benefits aside from finding defects that can be brought about by code reviews. These include:

- **Code improvement.** One of the managers at Microsoft noted that the *"discipline of explaining your code to your peers drives a higher standard of coding. I think this process is even more important that the result."*. This sentiment echoes one of the key drivers behind pair programming.
- **Alternative solutions.** 17% of developers put this as their first motivation - the rationale being that different team members could have better ideas as to how to implement a particular solution.
- **Knowledge transfer.** Code reviews can be a key driver to disseminating knowledge about the system within a team, and especially to new members. Being privy to a code review encourages participants to familiarise themselves with aspects of the system that might be outside of their area of expertise.
- **Team awareness.** Code reviews serve as a useful basis for notifying the rest of the team about what is happening in different parts of the system.

Bacchelli and Bird followed up their interviews with a review of the various code changes that arose from the reviews. They found that the majority of code changes were concerned with code improvement. The second largest proportion of changes was concerned with defect finding.

Their findings were supported by a subsequent study by Beller *et al.*[18]. Their studies of two large open source systems also illustrated that 75% of code commits that resulted from code reviews were related to maintenance and improvement, whereas only 25% are related to functionality. These statistics nicely illustrate the complementary nature of software testing and inspection. Whereas testing is primarily concerned with functional software correctness, inspections can support the assessment of a much broader range of concerns.

7.3 Code Reviewing Techniques

Code quality is one of the key factors in whether a software system as a whole is *maintainable* or not. Poor code quality hinders the ability of developers to understand software. This in turn raises the likelihood of accidentally introducing faults, and can lead to further degradation as the source code evolves.

These 'patterns' of poor code quality are often referred to as 'code smells'. The term originated from Martin Fowler [54], to describe the sorts of things that developers should routinely be keeping an eye out for when trying to improve their code.

These can range from highly localised problems (e.g. function or variable names that are non-descriptive, or the existence of a data class in an object-oriented system), to problems that span the whole system, such as the existence of duplicate code.

In this section the focus primarily on the task of detecting such smells. The task of fixing them once they have been detected is somewhat beyond the scope of this book. However, good starting points (at least when dealing with object-oriented software) would be Fowler's refactoring book (which was geared towards addressing code smells) and, for some larger architectural problems, Demeyer *et al.*'s book on software reengineering [41].

There are two families of approaches to inspecting code quality: (1) automated code analysis techniques that can pick out problematic patterns within the source code, and (2) reviews by the developer themselves to pick out problems. In this section we provide a brief overview of the key approaches, and cover what they look for.

7.3.1 Tool-Driven Code Review

Automated code reviews are ultimately driven by source code analysis. These tools go back to the Lint tool for C [76], which first emerged in 1977. Since then, code checkers have become heavily used, and are often routinely built in to IDEs such as Eclipse and IntelliJ. For example, if a variable is declared but never used, or used without being initialised this is commonly flagged up without the developer even having to explicitly invoke a tool.

The nature of the code problems that are (or can be) identified varies from one tool to another, and depends to an extent on the nature of the underlying language. For example, Strongly typed languages such as Java provide more information through which to discern potential problems than dynamically typed languages. Automated tools are especially good at identifying problems that are relatively localised (e.g. are localised to a single method or class). For tasks that encompass larger parts of a system.

Within Java, which is more relatively amenable to static analysis, perhaps the most popular tool (which is not already built in to an IDE) is Findbugs[3]. This tool analyses the byte-code of the compiled program and checks for over 400 problems, from unwritten object fields to non-terminating loops.

One problem that besets tools such as Findbugs is their propensity to overload the developer with warnings. Not every "bug" that a tool checks for is a genuine bug; they can often be false alarms. Nevertheless, when faced with large numbers of fault reports, the developer invariably has to spend a large amount of their time reading through them to separate the genuine problems from the false ones.

[3] `http://findbugs.sourceforge.net/`

Exercise: *Open Eclipse or IntelliJ with the Findbugs plugin, and run it on a software system (preferably your own!).*

7.3.2 Developer-driven Code Reviews

Some properties of code quality are very difficult, if not impossible, to automatically assess. Questions of good design, for example, are often subjective. Some aspects of good design are measurable by algorithms (as we will see in Chapter 8), but others (such as whether the files and modules are intuitively aligned with the problem domain) require a degree of insight that requires a degree of human reasoning.

Developer-driven code review is usually a two-step process [124]. The developer first has to first *understand* the program so that they can form an opinion of its implementation and design. Only then can they appraise the program in properly.

When it comes to the appraisal itself, the specific goals depend on several factors. There is the paradigm of the language that was used to write the system (Object-oriented systems are understood according to a different mental model from functional or imperative programs, for example). There are also the priorities – for example, developers that are focussed on security might look for different potential problems than developers who are focussed on issues such as maintainability.

7.3.2.1 Understanding the Code

How can a developer (or a group of developers) aggregate the knowledge knowledge about a code-base, and use it to form a coherent mental model? It is rarely tractable to try to read through all of the source code to systematically form a complete understanding of code behaviour (imagine being confronted with a truly large-scale system, comprising thousands of files, and millions of lines of code).

The question of code comprehension is a challenging one. Figuring out how to best understand a code-base ultimately requires a sound understanding of human psychology. Although a large body of research has hypothesised how humans understand code, confirming these hypotheses is difficult. Experiments rely on large numbers of human participants, and can be difficult to design. They have to convincingly address all manner of confounding factors (such as the prior experience of participants or the paradigm of the programming language), and also have to somehow measure the knowledge gained during a comprehension exercise, which can be difficult in its own right.

In practice, software comprehension amounts to a mixture of (a) taking stock of what you already know about a program, and (b) exploring the program to find out more about what you do not already know (but need to know) [129]. Elements of a program that you already know about can be indicated by 'beacons' – method names that are sufficiently descriptive to render their functionality obvious, or classes that contain sufficiently detailed comments, or obey design patterns [57] (recall Section 4.2.1) with which you're already familiar.

This is why good programming practice is so important. Sensible names and succinct comments remove the need for other developers to invest precious time into comprehension, and enable them to spend it more productively. This can also be fostered by carrying out light-weight changes to the source code of a program throughout its development [41][4]; if a developer learns something about a program whilst trying to understand it, this understanding can be embedded into the source code as a comment to assist other developers further down the line.

7.3.2.2 Pair Programming

Pair programming is an alternative approach to code review that takes place *during* coding (as opposed to afterwards). The idea is that software development is carried out by pairs of developers, so that each developer is in effect continuously scrutinising the others' work. Pair programming tends to be promoted with the following arguments:

- Code quality is higher, because it is continuously being reviewed.
- Good practice is shared between the programmers.
- There is greater shared knowledge of the code base, which leads to better solutions.

Most of the claims that advocate the use of pair programming have been the subject of empirical studies. In 2009, Hannay *et al.* helpfully collated them in a meta-analysis [68] of 18 relevant articles. This analysis indicated that: (1) pair-programming is faster than solo-programming when the complexity of the task is low, and (2) pair-programming produces results that are of a higher quality than solo programming when the tasks are complex. On the down-side, the higher-quality for complex task comes at a significant cost, requiring much more effort, whilst the quicker completion time for lower-complexity tasks comes at a cost of significantly lower quality

The ability to pair program makes some obvious assumptions about the development context, which often might not apply. For example, it assumes that the development team is collocated, and that the team is sufficiently large. In recent years, the enhanced role of collaborative software development environments and distributed version repositories has perhaps diminished the prevalence of pair programming[5].

[4] See the 'Tie Code and Questions' pattern.

[5] I am not aware of any data on the prevalence of pair-programming, so this is based on intuition, not evidence.

7.4 Safety Arguments and Inspections of Safety Requirements

For the majority of routine software development projects, MCR alone, coupled with some testing, can suffice to provide a sufficient confidence in software quality. For certain systems, however, more evidence is required, which cannot be obtained during routine software development. In the domain of safety-critical systems, for example, software has to be certified before it can be deployed, and certification can often only be achieved by examining a single, fixed version of the system (not one that is continuously in flux).

A profusion of software safety certification standards exist, which are often tailored for particular domains. For civilian aircraft software, there is DO178-B/C[1], which we have already encountered. There is ISO26262 [4] for automotive software, IEC 60880 [3] for software in the nuclear domain, etc. These various standards all share the commonality that they place an onus on the organisation that is responsible for developing the software to collect evidence that demonstrates that the various safety requirements have been met.

Establishing that a software product meets a particular set of safety standards is challenging for the following reasons [100]:

1. The document describing the standards can run into hundreds of pages of natural language text that is subject to interpretation.
2. The evidence that is required to establish a particular safety requirement can be difficult to collect and can, depending on the requirement, at best corroborate but not prove that a requirement has been fulfilled.

The amount and diversity of evidence that can be required can be truly bewildering. In their taxonomy of evidence types, having studied over 200 papers on safety certification, Nair *et al.* present 49 basic evidence types. Although their complete taxonomy is too large to present here, we can look at the high-level categories in their taxonomy, which are shown in Figure 7.3. The figure is instructive because it shows just how far-reaching a safety certification must be.

There are several approaches to collecting and presenting safety evidence. The top two approaches (at least according to their prevalence in research publications [100]) are checklists and what Nair *et al.* refer to as 'qualitative argumentation'. We examine both of these techniques in more detail below.

7.4.1 Checklists

Checklists set out requirements against which a software system should be inspected as a 'to-do' list of guided questions that need to be answered. Ultimately, the rationale of a checklist is to make the inspection repeatable – to ensure that, regardless of the expertise of the individual inspector, they would be prompted to highlight any potential faults.

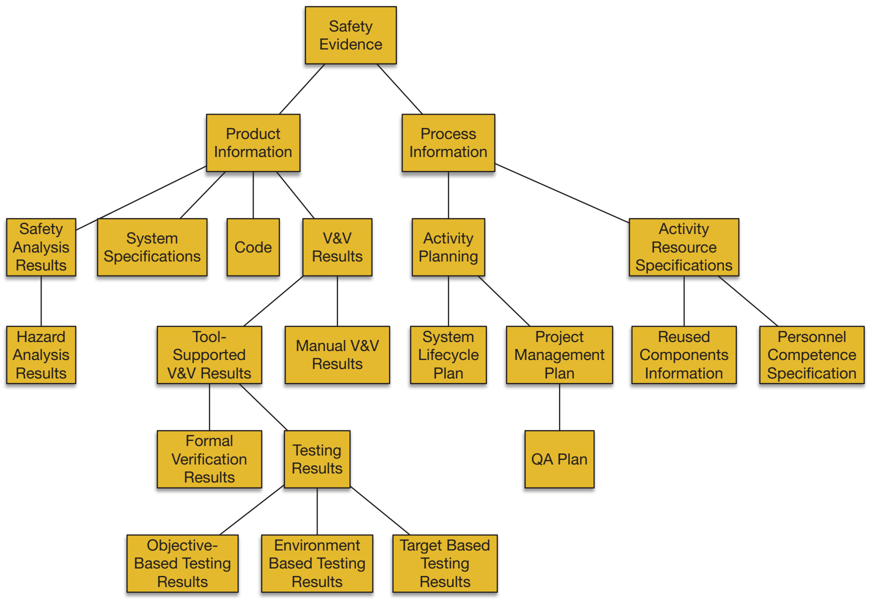
Safety Evidence
Product Information
Process Information
Safety Analysis Results
System Specifications
Code
V&V Results
Activity Planning
Activity Resource Specifications
Hazard Analysis Results
Tool-Supported V&V Results
Manual V&V Results
System Lifecycle Plan
Project Management Plan
Reused Components Information
Personnel Competence Specification
Formal Verification Results
Testing Results
QA Plan
Objective-Based Testing Results
Environment Based Testing Results
Target Based Testing Results

Fig. 7.3 Categories from Nair *et al.*'s safety evidence taxonomy [100].

Exercise: *Checklists are intended to support repeatability. Think back to Chapter 3 - Process-Based Quality Assurance. Which aspects of process-based quality assurance does this remind you of.*

Constructing an effective checklist is an art. If the list is too focussed and granular, the inspector can become too focussed on following the checklist objectives, but can miss out on specific faults that are not in the list. If it is too abstract, the inspector is left to their own devices and the performance of the checklist will vary more from one inspector to the other. The key is to (1) cover as many areas that are of importance from a quality perspective, (2) to ensure that the inspector does a thorough job – goes to the effort to fully understand the item under inspection, and (3) to not require a prohibitive amount of time and effort.

Two potential approaches by which to strike this balance are as follows[6]:

1. Write the questions from the perspectives of different stakeholders[16]. Ideally, task different individuals to adopt the perspectives of, for example, the designer, the tester, the coder, etc., and have them write a set of questions. The rationale is that these will force the inspector to consider the broadest possible range of concerns.

[6] These are techniques that are inspired by two 'formal inspection' approaches that appeared in the 90s – we will not go into their associated methodological details in this book, but their essential lessons can be readily applied to simple checklists.

2. Write the questions in such a way that the inspector is forced to engage with the document under inspection (i.e. by completing some task)[108]. E.g. instead of asking a relatively subjective question that is easy to answer in a hurry *"Are the test sets sufficient?"*, one could write *"Is the branch coverage achieved by all tests greater than 80%?"* – thus forcing the inspector to execute the tests and to monitor their coverage.

7.4.2 Safety Argumentation and the Goal Structure Notation

Although checklists can offer valuable guidance and are easy to use, they also have their weaknesses. If some aspect of safety is not properly captured by way of the questions in a checklist, it is unlikely to be inspected. They only illustrate *what* has been inspected (what has been ticked-off), but not *why*. There is no guidance as to how specific items might help to substantiate a higher-level safety property.

Argumentation-based approaches add an additional dimension to the traditional checklists. The underlying rationale with argumentation-based approaches is that, instead of checking a box, the inspector has to build an explicit argument that explicitly links the evidence to the conclusion that a given safety requirement has been satisfied. For example, instead of merely ticking a box to indicate that test coverage was satisfactory, a safety argument might indicate how the coverage data had been collected, and what coverage criteria had been used, etc.

Safety case argumentation can come in various forms [100]. They can be simply consist of structured natural language arguments, or be graphical in nature. Goal Structure Notation (GSN)[82] is a popular example of such a notation that sets out an argument in a hierarchical form – high level safety goals are broken down into sub-goals and arguments.

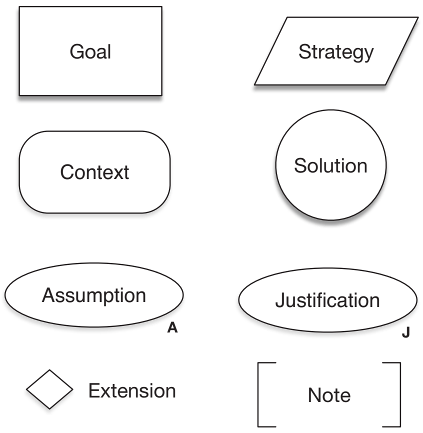

Fig. 7.4 Key GSN Symbols

The main GSN symbols are shown in Figure 7.4. Their purposes are briefly summarised below:

- **Goal:** A goal represents the safety property that we seek to build an argument around.
- **Strategy:** A description delineating *how* a goal is established.
- **Context:** Supporting information required to understand what is meant by a goal.
- **Solution:** A piece of evidence that can contribute towards establishing a goal.
- **Assumption:** Conditions under which a piece of evidence can validly be held to establish a goal.
- **Justification:** Reasons justifying the use of a piece of evidence.
- **Extension:** An indicator that further argumentation and support is required to support a particular (sub-)goal.
- **Note:** Additional textual annotation.

When put together, arguments can be constructed, forming hierarchical structures[7], where high-level safety goals can be broken down into sub-goals, and linked to the sources of evidence that would be required to establish them.

A nice, comprehensive example of a safety case is the safety case that was published in 2004 for the Panavia Tornado combat aircraft [128] (which can be downloaded in its entirety from the Gov.UK website). This was constructed, in part, to comply with legal obligations stating, for example, that it was necessary to show that the risk to its stakeholders was ALARP (As Low as Reasonably Practicable).

A small example GSN fragment is shown in Figure 7.5. This shows the high-level goal at the top, which is broken down into sub-goals further down. Where relevant, goals are linked to contexts, to explain what they mean, of justifications, to contribute to the argument of why they exist. At the lowest level, goals are tied to solutions – techniques that are to be used ot fulfil the goals.

7.5 Key Points

- **Software inspections were formally proposed by Fagan in 1976.** These 'Formal Inspections' remain widespread in certain sectors, but tend to be considered 'heavyweight', involving a lot of time and effort from the development team.
- **In recent years, the notion of a 'Modern Code Review' has become widespread.** These are characterised by being more lightweight, tool-based, and suited to distributed, asynchronous development. Reviews are often conducted on a per-commit basis to a version repository. These can be facilitated by tools such as Gerritt. An alternative is to adopt pull-based development, where proposed changes to the code base are sent to the development team in the form of a pull-request.

[7] Specifically, these are directed acyclic graphs.

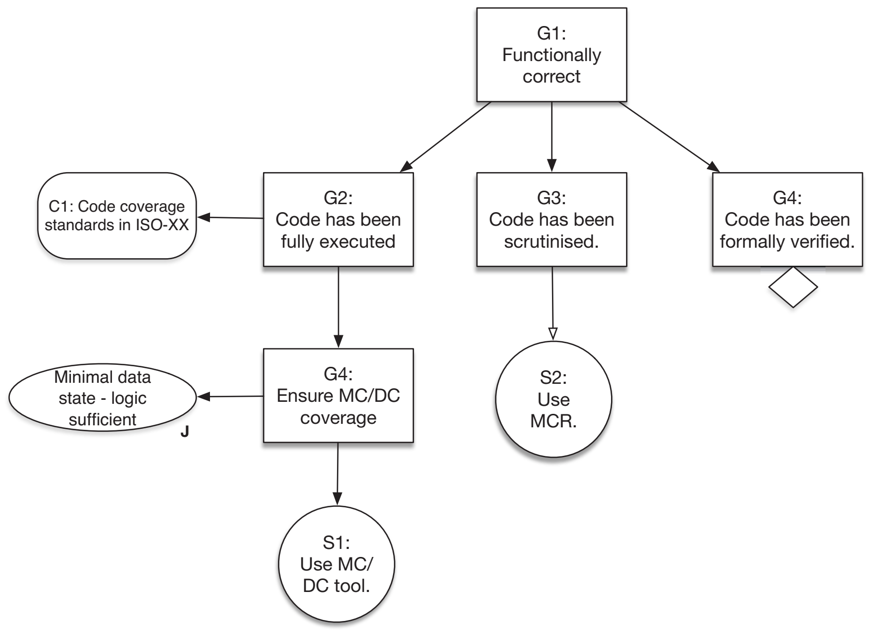
G1: Functionally correct
C1: Code coverage standards in ISO-XX
G2: Code has been fully executed
G3: Code has been scrutinised.
G4: Code has been formally verified.
Minimal data state - logic sufficient
J
G4: Ensure MC/DC coverage
S2: Use MCR.
S1: Use MC/ DC tool.

Fig. 7.5 GSN example.

- **A series of empirical studies have emerged to show several positive impacts of code reviews.** A series of empirical studies have emerged to show several positive impacts of code reviews. Results indicate that the code quality increases, along with team awareness of the code. They do not merely serve the narrow purpose of detecting bugs, but also spread knowledge about the system within the team, and lead to code improvement.
- **Code reviews necessarily start with a phase of 'program comprehension' – understanding the structure and functionality of the source code.** The activity is facilitated when there are areas of the source code ('beacons') that can be used as a basis for orientation.
- **Pair-programming is a means by which to review code *during* coding, as opposed to afterwards.** It was popularised with the emergence of agile software development. Instead of development being undertaken by individuals, pair programming sees pairs of developers sitting together to write code. Whilst one developer does the typing, the other can assist and can help to guard against errors.
- **Safety inspections constitute a special type of software review.** This tends to be carried out by independent teams of inspectors, who seek to inspect a system with respect to particular safety properties. These safety properties are often specified in the form of check-lists or graphically in the form of Goal-Structure Notation.