Aspect Assumptions

A Retrospective Study of AspectJ Developers’ Assumptions About Aspect Usage

Steffen Zschaler
Department of Informatics
King’s College London
London, UK
szschaler@acm.org

Awais Rashid
School of Computing and Communications
Lancaster University
Lancaster, UK
marash@comp.lancs.ac.uk

ABSTRACT
Aspect developers constantly make a range of assumptions about the context in which their aspects will be deployed: ranging from assumptions about other aspects deployed to assumptions about semantic properties of the base and the joinpoints at which an aspect is woven. Although it has been acknowledged that such assumptions need to be made explicit to validate aspects in the face of evolution (both of aspects and the base) and reuse as well as to mitigate the fragile-pointcut problem, so far no study exists that identifies the types of assumptions aspect developers make. In this paper, we present a retrospective study of three medium-sized open-source AspectJ projects and assumptions identified in these. This leads to an initial classification of assumptions that can form the basis for further research into how best to support each type of assumption.

Categories and Subject Descriptors
D.1.m [Programming Techniques]: Miscellaneous—Aspect-Oriented Programming; D.2.2 [Software Engineering]: Design Tools and Techniques—Modules and Interfaces; D.2.4 [Software Engineering]: Software/Program Verification—Programming by Contract

General Terms
Documentation, Languages, Verification

Keywords
AspectJ, developer assumptions

1. INTRODUCTION
With any modularisation, individual modules make assumptions about the functionality and realisation of other modules. A module can provide its intended functionality only if its assumptions are satisfied by the remainder of the system [1]. Aspect-oriented programming (AOP) [24] provides new means of modularisation to software development, in particular with respect to the separation of scattered or tangled concerns. Because AOP relies on cross-cutting the original modularity of a system and because aspects often aim to be oblivious to other parts of a system [11], assumptions aspects make about the rest of the system are particularly critical. Developers modifying other parts of the system may not even be aware of any aspect assumptions their changes may violate. In the context of AspectJ-like approaches to AOP [23]—which are the focus of this paper—this has often been referred to as the fragile-pointcut problem [26].

It is important that we improve our understanding of the assumptions aspect developers make about the system an aspect is deployed into. A catalogue of such assumption types would be useful in a range of situations:

- **Code improvement.** Some assumptions indicate bugs or coding mistakes. Such assumptions can form the basis for extracting aspect-oriented anti-patterns [5, 29, 34], which could be used to provide improvement hints to developers.

- **Assumption elicitation.** Some assumptions in aspect-oriented code may be implicit; that is, developers may not have been aware of making these assumptions. A catalogue of assumption types can help elicit such assumptions in two ways:
  1. It can be used in code inspections. Developers can ask for each category and aspect: “Am I making an assumption similar to this in my own code here?”
  2. We can attempt to extract patterns from the examples of aspect assumptions we have already identified. These patterns can then be used to semi-automatically identify assumptions in other aspect code.

- **Assumption verification.** Finally, where assumptions are made (whether implicitly or explicitly) it would be useful to enable developers to make them explicit in their code in a formalised manner. This enables (semi-)automatic verification of these assumptions as aspects are reused in new contexts or as the base evolves [13]. A catalogue of assumptions should enable us to provide more dedicated notation and verification strategies, thereby making fully automatic verification a feasible goal for at least some of the assumption categories without requiring developers to be fluent in formal specification techniques. This can be useful in a number of situations:
  1. **Evolution of base system.** Current syntactic pointcut expressions break easily when the base system evolves. If explicit aspect assumptions were available, they could be checked for an evolved base to establish which pointcuts have been broken and may need
inspection and updating by the aspect developer. If assumptions are expressed in a formal manner, such checks can be performed automatically, enabling direct feedback to developers as part of the compilation and weaving process.

2. Aspect reuse. When aspects are reused in a new context, their pointcuts may need to be adjusted. Without explicitly expressed assumptions, it is almost impossible to do this safely. When assumptions have been made explicit, they could be checked in the new usage context. Given the updated pointcut expressions, explicit assumptions could be used to verify that the aspect has been reused correctly.

3. Library aspects. Library aspects are designed to be reusable. When using AspectJ to develop library aspects, a typical pattern is to use abstract aspects with abstract or empty pointcuts to be refined by sub-aspects for particular reuse contexts. As the abstract aspect cannot explicitly express the assumptions it makes about how a pointcut can be refined, sub-aspects can easily break the library aspect’s contract.

Making an aspect’s assumptions explicit effectively introduces a more formal interface between the aspect and the rest of the system. Although it has been generally acknowledged that such interfaces are needed [2, 6, 17, 19], and some work on providing verification of aspect-oriented systems based on such interfaces and specifications of aspect assumptions has been published [2, 13, 17, 21, 27], so far no study exists that identifies the types of assumptions aspect developers actually make. In this paper, we present an initial, retrospective study of three medium-sized open-source AspectJ projects, in which we have manually identified developer assumptions.

From this, we have derived a categorisation of types of assumptions made by aspect developers. Many of these categories can be expressed using parameterised formal statements, which may form the basis for providing dedicated language support for expressing and verifying these assumptions in AspectJ code. Already, the categories identified, as well as the act of working through projects to elicit the assumptions have proven useful: As a result of our discussing them with developers from the projects, these developers report improved understanding of their systems, bugs identified and fixed, as well as pointing out assumptions that have frequently caused problems in the past. We believe that the assumptions and categories identified can form the basis for interesting future work in finding bugs, eliciting previously implicit assumptions, and supporting evolution of base and aspects.

The key contribution of this paper, thus, lies in the collection of fundamental real-world knowledge on the assumptions aspect developers make. This can serve to underpin, complement, direct, and further develop the work on novel language constructs for expressing interfaces and contracts in aspect-oriented programmes.

The remainder of this paper is structured as follows. We introduce the setup of our study in Sect. 2. Section 3 presents the categories of assumption types followed by a discussion of some threads to the validity of our findings in Sect. 4. Section 5 presents related work and Sect. 6 concludes the paper.

2. STUDY DESIGN

In order to understand the assumptions made by AspectJ developers about the context in which their aspects are used, we manually analysed the source code of three medium-sized open-source AspectJ-based projects. Here, we outline the design of our study beginning with a definition of key terms, followed by a discussion of the three projects selected and our reasons for selecting them. Finally, we discuss the individual steps of our study in more detail.

2.1 Notation

From a development perspective, a system can be characterised as a set \( S \) of interacting and integrated components. In asymmetric AO systems, there is a (typically non-empty) set \( B \subseteq S \) that corresponds to the base system; that is, it contains no aspect code. For symmetric aspect systems, \( B = \emptyset \). Let there be \( n \) aspects deployed within \( S \). Then there are further subsets \( A_i \subseteq S \); \( i = 1..n \) so that \((\bigcup_{i=1}^{n} A_i) \cup B = S \) and all \( A_i \) and \( B \) are pairwise disjoint; that is, the \( A_i \) and \( B \) form a partition of \( S \). Based on these definitions, we can then define the weaving context of an aspect \( A_i \) in system \( S \) as \( \text{wctx}_S(A_i) = S \setminus A_i \). Note that by this definition the weaving context of an aspect is different from the base of an asymmetric aspect-oriented system. In particular, in addition to the base the weaving context also contains any other aspects deployed in the system. Notice that \( A \nsubseteq \text{wctx}_S(A) \) even though an aspect may advise itself. However, for the purposes of this study we are only interested in assumptions about things outside the control of the developer of \( A \). We assume that the contents of \( S \) is under its developer’s control. Finally, it is useful to define a predicate \( \rightarrow \) over aspects of a system. \( A_1 \rightarrow A_2 \) indicates that \( A_1 \) advises \( A_2 \).

Similarly to Goldman and Katz [13], we say that a system \( S \) satisfies a certain property expressed as a formula \( \phi_S \), denoted \( S \models \phi_S \). Further, we say an aspect \( A \) guarantees a certain property \( \phi_A \) so that \( \forall S = A \cup \text{wctx}_S(A) \models \phi_A \). As with any other unit of modularity, aspects do not guarantee properties unconditionally. Rather, aspects make assumptions (expressed as a formula \( \psi_A \)) about their weaving contexts, and make their guarantees conditional on these assumptions being fulfilled:

\[
\forall S = A \cup \text{wctx}_S(A) \models \psi_A \implies S \models \phi_A
\]

Aspect assumptions are not explicitly available in current aspect-oriented code. Instead, in our study we needed to retrospectively reconstruct the assumptions. To this end, the notion of a developer assumption was useful for our study: Let \( R = \text{wctx}_S(A) \). For any aspect, we look for potential changes \( R \rightarrow R' \), such that \((R \cup A \models \phi_A) \land (R' \cup A \nmodels \phi_A)\). The differences between \( R \) and \( R' \) can be used to derive developer assumptions. In what follows, we will also use the words “assumptions of an aspect” to refer to developer assumptions.

We have intentionally kept this definition broad, as in this study we are interested in identifying the range of assumptions made. A too narrow definition carries a danger of eliminating interesting assumptions prematurely. In collecting assumptions, we were equally interested in those that were explicit (i.e., assumptions developers were aware of, as indicated, for example, in comments or through discussions) and those that were implicit (i.e., assumptions developers were not aware of, but that were still present in the code).

2.2 Project Selection

We have analysed the following three projects:

1. HealthWatcher [15, 36] is an AspectJ application refactored and evolved from an existing object-oriented solution for the management of public health-and-safety complaints in Brazil. The aspects have been developed specifically for this application, although some effort has been invested in generalisation, resulting in a small aspect framework that is specialised for HealthWatcher. HealthWatcher is an academic
Inclusion assumptions

2.3 Study Execution

Note that we are not claiming that we have identified all the assumptions in the study presented in this paper. Studying additional projects, this would not invalidate the findings. Therefore, while we may find further types of assumptions when providing evidence that they would never be made. We provide empirical evidence for assumptions made, not to eliminate assumptions AspectJ developers make. The goal of our study was to provide evidence that they would never be made.

3. We had reasonable connections to developers on these projects, which allowed us to validate our findings with these developers, removing wrongly identified assumptions and adding assumptions we had missed in our own analysis.

We have chosen these specific systems for a number of reasons:

1. They represent a range of different characteristics of aspect-oriented systems written in AspectJ: academic vs. industrial systems, aspects intended for reuse vs. aspects developed specifically for one system, aspect systems developed from scratch vs. aspect systems refactored from object-oriented solutions, and the use of static vs. dynamic crosscutting.

2. All three projects are open-source projects, which enabled analysis of the source code.

3. We had reasonable connections to developers on these projects, which allowed us to validate our findings with these developers, removing wrongly identified assumptions and adding assumptions we had missed in our own analysis.

Note that we are not claiming that we have identified all types of assumptions AspectJ developers make. The goal of our study was to provide empirical evidence for assumptions made, not to eliminate assumptions by providing evidence that they would never be made. Therefore, while we may find further types of assumptions when studying additional projects, this would not invalidate the findings of the study presented in this paper.

2.3 Study Execution

Our study consisted of two phases:

1. Data Collection: In this phase, we manually analysed the source code of the selected projects including any comments as well as other documentation and publications available (e.g., [4, 10, 36]) and extracted developer assumptions. As a first validation step, we then discussed these assumptions with key developers from the projects, which led to some clarifications as well as adding a small number of assumptions to the collection. In total, this analysis led to a collection of 183 assumptions for all three projects combined.

2. Clustering: In a second phase, we manually clustered these assumptions into a classification hierarchy, which is the key result of our work (see Sect. 3). This classification hierarchy was then again discussed with aspect developers to validate the generalisations made.

3. ASPECT ASSUMPTIONS

In this section, we present the types of aspect assumptions we have found in this study. A complete list of all assumptions from the three projects and how they have been classified is available on-line. Taking inspiration from the work on design patterns [12], we use a consistent template for the presentation of each assumption category. For intermediary categories this includes the name of the category, a description, and a list of sub-categories. For leaf categories, the template includes the name of the category, a description, an example, and a discussion of the assumption types.

Figure 1 shows the three top levels of our classification of aspect assumptions. The classification splits the field into two major categories: 1) Assumptions about aspect–aspect coordination, and 2) Assumptions about aspect–base coordination. Deviating slightly from common usage, the term base, here, should be understood as a relative rather than absolute term. In particular, it may also include other aspects that have been deployed and may be advised by the aspect currently under study.

3.1 Aspect–Aspect Coordination

Assumptions in this category are assumptions an aspect A makes on the behaviour of other aspects in the system, whether or not they are advised by A. More precisely, \( \psi_A \Rightarrow P(A_1, \ldots, A_n) \) with \( A \cap A_i = \emptyset \) where \( P \) is some predicate over aspects \( A_1, \ldots, A_n \).

We distinguish three broad categories, which we discuss in more detail in the following: 1) Inter-aspect assumptions, 2) Inter-advice assumptions, and 3) Inter-process assumptions.

3.1.1 Inter-Aspect Assumptions

Inter-aspect assumptions are assumptions one aspect makes on other aspects as a whole, as opposed to assumptions on specific pieces of advice. While these assumptions may relate to a lower-level assumption about particular behaviour provided by other aspects, in the context of a specific aspect-oriented system, it may be useful to think about them in terms of the available aspects rather than their required properties, which can sometimes be quite complex or difficult to capture.

We have identified five categories of inter-aspect assumptions, which we will discuss in the following.

Assumptions on Aspect Deployment.

Individual aspects sometimes make assumptions about the deployment (or lack of deployment) of other aspects in the same application instance. We have found three categories of such assumptions.

In their description below, we will use \( d(A) \) to formally denote that aspect A is deployed; that is \( S \models d(A) \) iff \( A \subseteq S \).

**Name. Inclusion assumptions**

**Description.** Some aspects require other aspects to be deployed to function correctly; that is \( \psi_{A_1} \Rightarrow d(A_2) \). Generally, this is the case when an aspect \( A_1 \) relies on another aspect \( A_2 \) to either modify the


development, widely accepted as a testbed for research on AOSD in the academic community. The majority of aspects provided in HealthWatcher use dynamic crosscutting; that is, before, after, or around advice. Data was collected for Version 10 of HealthWatcher [14].

2. **MobileMedia** [8, 10] is an AspectJ application refactored and evolved from an initial object-oriented solution for the management of pictures in a mobile-phone context. It has been evolved to provide support for other types of media, using primarily static crosscutting to encapsulate the logic related to different types of media or target devices. The aspects have been developed specifically for this application without generalisation effort. MobileMedia is an academic development originally intended to provide a testbed for research in software product line engineering; it has been used in a number of different previous studies comparing aspect-oriented and object-oriented implementations of the same system. Data was collected for Version 7 of MobileMedia [9].

3. **Glassbox** [4, 32] is a well-known, open-source, industrial AspectJ application for monitoring application servers and applications deployed in such servers. Data was collected for the project of Glassbox version 2.0 [33].

We have chosen these specific systems for a number of reasons:

1. They represent a range of different characteristics of aspect-oriented systems written in AspectJ: academic vs industrial systems, aspects intended for reuse vs aspects developed specifically for one system, aspect systems developed from scratch vs aspect systems refactored from object-oriented solutions, and the use of static vs dynamic crosscutting.

2. All three projects are open-source projects, which enabled analysis of the source code.

3. We had reasonable connections to developers on these projects, which allowed us to validate our findings with these developers, removing wrongly identified assumptions and adding assumptions we had missed in our own analysis.

Note that we are not claiming that we have identified all types of assumptions AspectJ developers make. The goal of our study was to provide empirical evidence for assumptions made, not to eliminate assumptions by providing evidence that they would never be made. Therefore, while we may find further types of assumptions when studying additional projects, this would not invalidate the findings of the study presented in this paper.

2.3 Study Execution

Our study consisted of two phases:

1. **Data Collection:** In this phase, we manually analysed the source code of the selected projects including any comments as well as other documentation and publications available (e.g., [4, 10, 36]) and extracted developer assumptions. As a first validation step, we then discussed these assumptions with key developers from the projects, which led to some clarifications as well as adding a small number of assumptions to the collection. In total, this analysis led to a collection of 183 assumptions for all three projects combined.

2. **Clustering:** In a second phase, we manually clustered these assumptions into a classification hierarchy, which is the key result of our work (see Sect. 3). This classification hierarchy was then again discussed with aspect developers to validate the generalisations made.

3. **ASPECT ASSUMPTIONS**

In this section, we present the types of aspect assumptions we have found in this study. A complete list of all assumptions from the three projects and how they have been classified is available on-line. Taking inspiration from the work on design patterns [12], we use a consistent template for the presentation of each assumption category. For intermediary categories this includes the name of the category, a description, and a list of sub-categories. For leaf categories, the template includes the name of the category, a description, an example, and a discussion of the assumption types.

Figure 1 shows the three top levels of our classification of aspect assumptions. The classification splits the field into two major categories: 1) Assumptions about aspect–aspect coordination, and 2) Assumptions about aspect–base coordination. Deviating slightly from common usage, the term base, here, should be understood as a relative rather than absolute term. In particular, it may also include other aspects that have been deployed and may be advised by the aspect currently under study.

3.1 **Aspect–Aspect Coordination**

Assumptions in this category are assumptions an aspect \( A \) makes on the behaviour of other aspects in the system, whether or not they are advised by \( A \). More precisely, \( \psi_A \Rightarrow P(A_1, \ldots, A_n) \) with \( A \cap A_i = \emptyset \) where \( P \) is some predicate over aspects \( A_1, \ldots, A_n \).

We distinguish three broad categories, which we discuss in more detail in the following: 1) Inter-aspect assumptions, 2) Inter-advice assumptions, and 3) Inter-process assumptions.

3.1.1 **Inter-Aspect Assumptions**

Inter-aspect assumptions are assumptions one aspect makes on other aspects as a whole, as opposed to assumptions on specific pieces of advice. While these assumptions may relate to a lower-level assumption about particular behaviour provided by other aspects, in the context of a specific aspect-oriented system, it may be useful to think about them in terms of the available aspects rather than their required properties, which can sometimes be quite complex or difficult to capture.

We have identified five categories of inter-aspect assumptions, which we will discuss in the following.

**Assumptions on Aspect Deployment.**

Individual aspects sometimes make assumptions about the deployment (or lack of deployment) of other aspects in the same application instance. We have found three categories of such assumptions. In their description below, we will use \( d(A) \) to formally denote that aspect \( A \) is deployed; that is \( S \models d(A) \) iff \( A \subseteq S \).

**Name. Inclusion assumptions**

**Description.** Some aspects require other aspects to be deployed to function correctly; that is \( \psi_{A_1} \Rightarrow d(A_2) \). Generally, this is the case when an aspect \( A_1 \) relies on another aspect \( A_2 \) to either modify the
Inter-Process

Super-Aspect Structure

Wormhole

Advice Execution Sequence

Aspect Assumptions

Inter-Aspect

Deployment

ITDs

Precedence

Inter-Advice

HealthWatcher defines two aspects

Inter-Advice

Super-Aspect

Aspects may also be mutually exclusive; that is,

Aspect deployment is specified separately from the ac-

MobileMedia uses aspects to configure different prod-

Examples. MobileMedia uses aspects to configure different prod-

HWUpdateObserverExceptionHandler,

documented as

the system it is woven. For example, HealthWatcher has an aspect

the generic-pattern aspect being deployed, but also about where in

where exactly the generic pattern is woven. An aspect provid-

be woven in a potentially large number of places. Where an

pattern affects the application and how the application should deal

Some aspects are very specific and must be woven in one par-

Behaviour \(A_1\) introduces or to introduce base behaviour that \(A_1\)
can then modify. If \(A_1\) requires \(A_2\) to be deployed because it uses
inter-type definitions (ITDs) defined in \(A_2\), this is classified with
the other ITD assumptions below.

A special case is when an aspect \(A_1\) has been introduced to re-
solve a feature interaction between two other aspects \(A_2\) and \(A_3\)
(or more). In this case, there will be an inclusion assumption
\(\forall A_1 \Rightarrow d(A_2) \land d(A_3)\).

Examples. MobileMedia uses aspects to configure different prod-

cucts of a product line. For example, deploying 

MusicAspect implies features for both photo and music manage-

have been implemented in separate aspects (PhotoSelector and Music-

Selector in this example), there is an inclusion requirement that

these implementation aspects also be deployed.

HealthWatcher shows an example of an aspect introduced be-

cause of a feature interaction: Aspect HWTimeStamp had to be

adjusted to provide special behaviour for the case when 

UpdateStateObserver (implementing the observer protocol) is dep-

loyed. In particular, HWTimeStamp maintains a modification
counter for objects stored in the database. Before the deployment of

the observer protocol it was sufficient for this aspect to update

the modification counter for every request coming into the system.

Deployment of the observer pattern invalidated this assumption as

a request may now lead to a chain of modifications triggered by

observers. The modification counter may need to be updated more

than once for each request coming into the system. Here, the in-

clusion assumption is on the deployment of HWTimeStamp and

UpdateStateObserver. However, the former is trivially true

because the advice has been directly included into HWTimeStamp.

Some aspects are very specific and must be woven in one par-

icular way; other aspects capture more generic patterns and can

be woven in a potentially large number of places. Where an

aspect provides special behaviour depending on an aspect cap-

turing generic patterns, whether to provide the special behaviour

(and possibly also what behaviour to provide) may depend on

where exactly the generic pattern is woven. An aspect provid-

ing such special behaviour not only makes an assumption about

the generic-pattern aspect being deployed, but also about where in

the system it is woven. For example, HealthWatcher has an aspect

HWUpdateObserverExceptionHandler, documented as

"[...] this aspect needs a lot of knowledge about how the observer

pattern affects the application and how the application should deal

with its exceptions" [14]. Further analysis of the code shows that,

among other things, this aspect assumes that the Observer-Pattern

aspect is only woven into the system for data updates (and not, for

example, for data creation).

Beyond the general case, we have identified a specific variation,

where one aspect defines a marker interface that is then used by

another aspect in a declare parents clause. In this case, the

second aspect assumes deployment of the first, so that the semantics

of the marker interface are operationalised.

For example, Glassbox defines interface MonitoredType.

This is defined as a top-level interface, but is documented to “start

[the] agent when loading any such type. Should be added to any

controlled code whose loading would trigger monitoring [...]”

[33]; that is, it is used as a marker interface. The starting-the-agent

functionality is implemented in aspect AutoInitialization.

Any aspect declaring MonitoredType as the parent of some
class (for example, BerkeleyDbMonitor, CommonsHttp-

Monitor, or ServletRequestMonitor) implicitly requires

AutoInitialization to be deployed.

Discussion. Aspect deployment is specified separately from the ac-
tual aspect definitions, either on the ajc command line or in pa-

rameter files for ajc. When inclusion dependencies are not made
explicit in an aspect definition, the presence of a required aspect

cannot be checked by ajc at weaving time, potentially leading to

timeout errors or unexpected system behaviour.

The inclusion assumptions we have found, seem to occur mostly

where aspects are used to encapsulate individual features to be

added to or removed from the system as required. In this sense,

they seem closely related to the ordering constraints between so-
called derivatives as described in [25].

Name. Mutual exclusion assumptions

Description. Aspects may also be mutually exclusive; that is,

\(\forall A_1 \Rightarrow \neg d(A_2)\). In some cases, at least one aspect from a mu-

tually exclusive set of aspects has to be deployed to provide the

required system functionality.

Examples. HealthWatcher defines two aspects HWLocalSync-

chronization and HWManagedSynchronization providing

alternative implementations for the synchronisation of access
to data classes of HealthWatcher. HWLocalSynchronization

is intended to be used in a debugging version of HealthWatcher,

which uses an in-memory representation of data. In contrast, HW-

ManagedSynchronization is used in the live system, which

uses a relational database.

A special case is the mutual exclusion between sub-aspects
of a common abstract library aspect. HealthWatcher shows an example of this kind of assumption in the RMIClient-Distribution aspect. This aspect contains pointcuts using withincode(HWClientDistribution+) (referring to the super-aspect of RMIClientDistribution). These pointcuts would also—erroneously—match joinpoints in other sub-aspects of HWClientDistribution should any be deployed. In the case of HealthWatcher, this is currently not the case, and there is an explicit assumption that only RMI will be supported. Should this change in the future, this assumption will break.

Discussion. As mentioned above, broken aspect-deployment assumptions cannot be caught by the compiler unless made explicit in the aspect definition. Mutual exclusion assumptions are likely to indicate a conflict between two aspects. If both were deployed within the same system, errors are very likely to occur at runtime.

Name. Mandatory aspect assumptions
Description. An aspect-oriented system may contain one or more aspects that are required by all other aspects in the system and are mandatory for the system to work. This is not strictly an assumption of a single aspect, but rather of the entire system. We choose to include it in our categorisation because of its potential for substantial impact on the intended system functionality.

Examples. HealthWatcher expects a HWFacade to be constructed or a HWServlet to be started. This requires at least one of the following aspects to be deployed: RMIServerDistribution, HWServletDistribution, or ServletCommanding.

Discussion. If a mandatory aspect is not deployed, the system will present unexpected, incomplete, or erroneous behaviour. ajc cannot currently capture and verify such assumptions, so that these errors would only manifest at runtime. Mandatory aspect assumptions cannot be expressed by inclusion assumptions from all other aspects, as this could still be satisfied by not deploying any aspects at all, including the mandatory ones. Mandatory-aspect assumptions are assumptions on the system as a whole rather than assumptions of a single aspect and, therefore, need to be treated separately.

Assumptions on ITDs.
In addition to before, after, or around advice, aspects can use inter-type declarations (ITDs) to extend the interface or internal structure of existing types. There are two categories of assumptions in relation to ITDs:

Name. Assumptions on ITD provision
Description. An aspect may make use of features that need to be introduced through an ITD from another aspect. Let def(itd) indicate that an element with signature itd is defined in a system: S |\= def(itd) iff itd /\ S. Then we might formalise this type of assumption as \( \psi_A \Rightarrow def(itd) \) where itd denotes the specific ITD aspect A relies on.

Examples. In MobileMedia, aspect SMSOrCapturePhoto uses method getImage in class PhotoViewScreen. This method is introduced by aspect SMSAspect.

Discussion. Such assumptions are easily checked by ajc as the ITDs will need to be available at compile time.

Name. Assumptions on ITD use
Description. Aspects typically use ITDs for one of two reasons: 1) to associate additional data with objects advised so that data can be used internally to the aspect, 2) to provide an interface to additional features introduced by the aspect. In the second case, the ITDs are often not used from the declaring aspect, so, unless developers intentionally produce dead code, there is an implicit assumption that other code will be aware of these ITDs and will use them appropriately. In some cases, this assumption is even more fundamental as the ITDs must be used to configure essential parameters of the aspect before it can work properly. For an ITD \( itd \in A \) let inv(itd) indicate that a statement invoking itd exists somewhere in a system. Then we might formalise this assumption as \( \psi_A \Rightarrow inv(itd) \).

Examples. In MobileMedia, aspect MusicSelector provides ITDs for setting the music controller and music album data and expects them to be called from another aspect (e.g., PhotoAndMusicAndVideo or PhotoAndMusicAspect) before any of the advice in MusicSelector is invoked.

Discussion. This is particularly critical for ITDs that are used for configuration and must be called. If they are not invoked, unexpected behaviour or errors may occur at runtime. The compiler might be enhanced to track unused ITDs and present warnings about them. However, it cannot declare them as an error, because in some reuse situations unused ITDs may be perfectly normal. If the assumptions could be made explicit—particularly where an aspect requires its ITDs to be called to configure the aspect—the compiler could check the assumptions and issue errors for broken explicit assumptions and warnings for broken implicit assumptions.

Assumptions on Super-Aspect Structure.
Sub-aspects regularly make assumptions about details of their super-aspect(s). This can be critical when super-aspects can be changed outside the control of the developers of sub-aspects—for example, if the super-aspects are library aspects in a third-party library. To support the discussion in this and the following section, we introduce relation \( \succ \) over aspects, where \( A \succ B \) indicates that A is a (direct or indirect) super-aspect of B.

Name. Assumptions on the semantics of (abstract) pointcuts
Description. This category groups assumptions an aspect \( A_1 \) makes about how its super-aspect \( A_2 \succ A_1 \) advises pointcuts it introduces. Abstract aspects can introduce pointcuts to allow sub-aspects to specify points that should be advised in a particular way. When a sub-aspect decides to refine such a pointcut, it implicitly makes an assumption about the advice provided for this pointcut by the super-aspect. Formally we might represent this as \( \psi_{A_1} \Rightarrow P(A_2,p_1,...,p_n) \). Here \( P \) represents a predicate expressing how \( A_2 \) affects the system behaviour at pointcuts \( p_1,...,p_n \).

Examples. In HealthWatcher, aspect ServletCommanding subclasses CommandProtocol, an abstract implementation of the command design pattern [12, 18]. CommandProtocol provides an abstract pointcut commandTrigger that must be refined to define what events should lead to a command being created. This pointcut exposes a CommandInvoker parameter, which is used by CommandProtocol to obtain the correct command from a central registry of commands. ServletCommanding refines this pointcut to match calls to CommandProtocol’s method setCommand, which is used to store commands in this central registry. It also provides advice for post and get methods of all servlets implemented for HealthWatcher and for each such request registers a new command against an anonymous command invoker. Because of the refinement of commandTrigger, this has the side effect of invoking the command. Apart from assuming that commandTrigger identifies joinpoints where commands should be triggered, this also assumes that commandTrigger will be advised by after advice, so that the command has been successfully registered before it is issued by CommandProtocol.

Another example shows an aspect making an assumption about
more than one pointcut. In Glassbox a large range of aspects implement monitoring of specific elements of enterprise applications by inheriting (directly or indirectly) from AbstractMonitor. This aspect introduces a pair of abstract pointcuts monitorBegin and monitorEnd to allow sub-aspects to define the scope of monitoring. All sub-aspects assume that defining these pointcuts will mean that the aspect measures the time and resources taken from a joinpoint in monitorBegin to a joinpoint in monitorEnd.

Discussion. Abstract aspects and pointcuts are a key instrument in building reusable aspects in AspectJ. Typically, library users do not have control over how the library code evolves. Consequently, to make library reuse robust, it is important to be able to make assumptions on the library code explicit. Evolution of library aspects may result in unexpected behaviour or runtime errors.

Name. Assumptions on concretisation of pointcuts

Description. When a hierarchy of aspects inherits from an abstract aspect, pointcuts will be concretised (defined) at different points in the hierarchy, and possibly repeatedly redefined (e.g., aspect A₁ defines pointcut p and aspect A₂ ≺ A₁ also defines p). In such a case, the definition lowest in the hierarchy will be used. An aspect A₁ defining a pointcut p, but wishing to maintain the behaviour introduced by its super-aspect makes an assumption that this pointcut is not already defined by its super-aspect(s):

\[ \forall Aₙ \supseteq A₁ : \neg \exists f(p, A₁) \]

Note that these assumptions are frequently broken in Glassbox, because sub-aspects of AbstractMonitor often ignore monitorBegin, but still use monitorEnd. The underlying assumption about Resource nesting is still maintained, because these aspects provide their own (partially redundant) advice to create new Resource objects. From our discussion with Ron Bodkin, the lead developer of Glassbox, we know that the inability to verify these assumptions of AbstractMonitor has frequently been a cause of problems in the past.

Note that these assumptions are frequently broken in Glassbox, because sub-aspects of AbstractMonitor often ignore monitorBegin, but still use monitorEnd. The underlying assumption about Resource nesting is still maintained, because these aspects provide their own (partially redundant) advice to create new Resource objects. From our discussion with Ron Bodkin, the lead developer of Glassbox, we know that the inability to verify these assumptions of AbstractMonitor has frequently been a cause of problems in the past.

Discussion. As AspectJ does not provide a way of checking if a pointcut has already been concretised (apart from the quite radical final keyword) or of reusing a pointcut concretisation from a super-aspect when redefining the pointcut in the sub-aspect, it is important that assumptions about what pointcuts have been concretised further up in the hierarchy can be made explicit.

Assumptions on Sub-Aspect Structure.

Name. Assumptions on sub-aspect structure (pointcut contracts)

Description. When an aspect declares an abstract pointcut it makes certain assumptions about the joinpoints that will be matched by any concretisation of the pointcut in a sub-aspect. In many ways these are linked to the converse assumptions mentioned in the previous section. However, these assumptions work more as a contract that an abstract aspect would like to impose on its specialisations. For an abstract aspect A we might formalise this as

\[ \forall A₁ \supseteq A : \exists P(A₁, p₁,..., pₙ) \]

Where P is a predicate on the system behaviour as affected by aspect A₁ at pointcuts p₁...pₙ.

Examples. The types of pointcut parameters can already be specified in AspectJ, their semantics in terms of base control- or dataflow are also very important. For example, CopyMultiMediaAspect in MobileMedia defines an abstract pointcut handleCommandAction that has two parameters CopyTargets controller and Command c. It assumes that this will be defined to match calls to specialisations of CopyTargets’ abstract handleCommand method, with controller exposing the instance of the specific subclass of CopyTargets and c exposing the command parameter passed into this method call. It further assumes that the handleCommand implementation so captured will return true to indicate that is has handled a given command itself.

Sometimes, multiple abstract pointcuts may need to be concretised in a coordinated fashion, for example to ensure match-up conditions whereby every control flow that matches pointcut p₁ eventually reaches a joinpoint that matches pointcut p₂. A specific variant of this is where such matches need to be nested appropriately. For example, Glassbox’s AbstractMonitor defines a pair of abstract pointcuts monitorBegin and monitorEnd. It advises monitorBegin to create a new Resource object (recording the time it was created) and maintains a stack of these. When monitorEnd is reached, the top Resource object is taken off the stack, the current time is recorded and this information is stored in the monitoring database. There are two assumptions here:

1. Any Resource object created will eventually be taken off the stack again. This can be achieved by ensuring that any control flow that reaches monitorBegin eventually reaches monitorEnd, too.

2. The correct Resource object is used to measure end time. This can be translated into an assumption that monitorBegin and monitorEnd ‘events’ are correctly nested within one control flow. In particular, every monitorEnd joinpoint should find at the top of the stack a Resource object that has been put there by a corresponding monitorBegin joinpoint (i.e., one that has been defined within the same sub-aspect of AbstractMonitor).

Note that these assumptions are frequently broken in Glassbox, because sub-aspects of AbstractMonitor often ignore monitorBegin, but still use monitorEnd. The underlying assumption about Resource nesting is still maintained, because these aspects provide their own (partially redundant) advice to create new Resource objects. From our discussion with Ron Bodkin, the lead developer of Glassbox, we know that the inability to verify these assumptions of AbstractMonitor has frequently been a cause of problems in the past.

Discussion. Currently, AspectJ does not provide support for making such contracts explicit. The work by Griswold et al. [17] addresses this problem to a certain extent by encoding some contracts into their crosscut programming interfaces (XPIs).

Precedence Assumptions.

Name. Precedence assumptions

Description. Some advice may make assumptions about the relative precedence between aspects. This may mean expecting precedence to be explicitly declared in a particular way. Additionally, it may mean that the standard AspectJ precedence (in particular between sub- and super-aspect) should not be modified by an explicit declare precedence clause.
**Examples.** The Glassbox aspect JxtaSocketMonitor assumes to have precedence over its super-aspect AbstractMonitor. This is the default semantics of AspectJ. However, an assumption remains that this is not changed by any declare precedence clauses anywhere in the code.

**Discussion.** Precedence may be influenced by declare precedence clauses directly referencing a set of aspects, but also quite indirectly. For example, AbstractMonitor defines an interface LowerPrecedence. Any aspect implementing this interface will automatically receive lower precedence than AbstractMonitor.

However, AspectJ’s declare precedence clause can also be used to make precedence assumptions explicit. Then, if they are violated, ajc will produce an error. This has been done for the Glassbox example as a result of the discussion of our findings with Glassbox’ lead developer.

### 3.1.2 Inter-Advice Assumptions

These assumptions refer to the interactions between individual pieces of advice rather than aspects as a whole.

**Name.** “Wormhole” assumptions

**Description.** A common pattern for advice interaction is the so-called “wormhole” [28]. This involves collecting context information that is available at different points throughout a control flow and making it available at a different point in this (or another) control flow. The original form of the pattern involves exposing all context information as parameters of a single pointcut. However, this is not always convenient (especially when not all points involved are within one cflow). A common alternative is to temporarily store some context data in aspect-local variables to be used at a later point. In both scenarios, there is a requirement that the values picked up by one advice / at one joinpoint remain valid (and unchanged) until another piece of advice is invoked. This can imply a synchronisation assumption for the aspect.

**Examples.** MobileMedia implements a mechanism for maintaining a collection of favourite media that can be accessed directly from a list of favourites. This feature is implemented in aspect FavouritesAspect using a set of heterogeneous advice on different parts of the overall system. All of these advice need to maintain state about whether the favourites view is currently activated. They do this using an aspect member favorite. Hence, the aspect requires that this variable is maintained consistently in accordance with the application state. This requirement is satisfied in the current MobileMedia implementation, because there is only one application thread. It may break when the aspect is reused in a multi-threaded context.

**Discussion.** Probably, wormhole assumptions should be classified as a bug that should be fixed by using appropriate synchronisation mechanisms and possibly thread-local variables (or, of course the original form of the pattern, which may avoid some of the assumptions because AspectJ would implement it internally using a thread-local variable). Hence, this assumption category may be most useful for assumption elicitation and as an anti-pattern. There may also be situations in which the above corrections would not be applicable. In these cases it will still be useful to make the assumption explicit so that it can be checked when the context changes.

**Name.** Assumptions on sequential execution of advice

**Description.** Advice may modify the state of base objects. Other advice may rely on this state. In effect, such advice relies on other advice having been executed before it for a particular application instance. Let \( e_s(a, t) \) indicate that advice \( a \) has started executing in thread \( t \). Conversely, let \( e_f(a, t) \) indicate that advice \( a \) has finished executing in thread \( t \). Then, we might formalise the assumption that advice \( a_1 \in A \) assumes previous execution of advice \( a_2 \) as \( \forall t_1, G \neg e_s(a_1, t_1) \cup e_f(a_2, t_2) \).\]

**Examples.** For example, MobileMedia has an aspect PhotoAndMusicAndVideo, which is deployed when all three named features are selected. This provides two pieces of advice: one performing some setup code, the other reacting to selections of menu items installed by the first advice. Clearly, there is an assumption here that the first advice has run before the second one is invoked.

**Discussion.** If broken, this type of assumption can cause unexpected behaviour or errors at runtime. For example, in the MobileMedia case, if the setup code were not to be executed, the new feature would not be accessible to the user, because all of its user-interface elements would not have been created.

If \( a_1 \) and \( a_2 \) are located in different aspects, this assumption could also be rendered as an inclusion assumption about aspect deployment. In this respect, it is a more semantic representation of the cause for such an inclusion assumption. This category is also closely related to the synchronisation assumptions and the assumptions on coding patterns discussed later. The key difference is that here we discuss an assumption on other aspects regardless of whether they are being advised by the aspect making the assumption. Also, this assumption is independent of threading issues, which is different from the synchronisation assumptions.

### 3.1.3 Inter-processual Aspect Interaction Assumptions

**Name.** Inter-processual aspect interaction assumptions

**Description.** Some aspects may affect communication between application processes. This may lead to assumptions about the interaction of aspects deployed in different application processes.

**Examples.** These assumptions may relate, for example, to data persistence and to aspects deployed in serial invocations of the same application (or versions thereof). Alternatively, they may relate to distribution and to aspects deployed in (potentially different) applications running simultaneously, but possibly on different nodes. While in principle, all types of assumptions discussed above might also exist in an inter-processual variant, we have specifically found examples of the following types: 1) Inter-processual deployment assumptions, and 2) Inter-processual precedence assumptions.

For example, HealthWatcher defines RMIClientDistribution to be deployed in the client version of the software. This attempts to link to the server version at a hard-coded address. Another aspect (RMIServerDistribution) is deployed in the server version of the software and makes sure that the server is actually registered at that address. RMIClientDistribution assumes RMIServerDistribution to be deployed in the server version of the software.

As another example, MobileMedia provides a number of aspects contributing data fields to the serialisation (i.e., persistence) feature of the software. Because these data are serialised in an order depending on the precedence between these aspects, there is an inter-processual precedence assumption that subsequent invocations of

---

3The association to a specific thread is not actually relevant here. We introduce these forms of the predicates, because we can reuse them in Sect. 3.2.1.

4Using the usual operators of LTL [35]:

- \( G \phi \) – globally \( \phi \); that is \( \phi \) holds for all states from now
- \( E \phi \) – eventually \( \phi \); that is \( \phi \) will hold in some future state
- \( X \phi \) – next \( \phi \); that is \( \phi \) holds in the next state
- \( q_1 \to q_2 \) – \( q_1 \) until \( q_2 \); that is \( q_1 \) will hold in all states until \( q_2 \) holds and \( q_2 \) will eventually hold.
the software (which might correspond to different versions) will contain appropriate precedence constraints.

Discussion. Obviously, these assumptions cannot be checked by ajc at all. Furthermore, because they are concerned with the interaction of different processes, possibly including a time difference (e.g., for persistence-related assumptions), their runtime effects may not become immediately obvious making them difficult to spot and correct.

3.2 Aspect–Base Coordination

These are assumptions on the relation between aspect and base. Keep in mind that 'base' in this context may mean anything that an aspect advises, including other aspects. Aspect–base coordination assumptions take the form

\[ \psi_A \Rightarrow P(B) \wedge \forall A_i. A \rightarrow A_i \Rightarrow P_i(A_i) \]

where the \( P_i \) are predicates further detailing the assumption made.

In the following, we discuss three broad sub-categories.

3.2.1 Synchronisation Assumptions

If aspects are deployed in a multi-threaded environment, they may make assumptions about synchronisation.

Name. Weaving context ensures synchronisation

Description. Whenever an aspect does not provide synchronisation of its own for any behaviour sensitive to concurrency, it effectively makes an assumption that the weaving context will ensure appropriate synchronisation. Such assumptions typically concern specific advice, or groups of advice, as this is where behaviour is encapsulated within an aspect. In the three projects studied, we have specifically found examples of the following types of assumptions:

1. An aspect \( A \) may assume that advice \( a \in A \) is called only once per run of the system; that is

\[ \psi_A \Rightarrow \forall t_1. e_s(a, t_1) \Rightarrow \forall t_2. \exists X. \neg e_s(a, t_2) \]

This is an assumption that, for example, may occur with advice initialising aspect-internal data structures.

2. An aspect \( A \) may assume that advice \( a \in A \) is invoked only by one thread at a time; that is

\[ \psi_A \Rightarrow \forall t_1. e_s(a, t_1) \Rightarrow \forall t_2. \neg e_s(a, t_2) \]

This is the classical assumption of critical sections. The aspect assumes that the critical section represented by a particular advice will be guarded appropriately at every weaving context.

3. An aspect \( A \) may assume that some advice \( a_1 \in A \) can only be invoked again after some other advice \( a_2 \) has been called in the same thread:

\[ \psi_A \Rightarrow \forall t. e_s(a_1, t) \Rightarrow \exists X. \neg e_s(a_2, t) \]

Examples. Examples of the first type of assumption by initialisation code occur in HealthWatcher—for example, in aspects HWPersistence and ServletCommanding.

HealthWatcher's ServletCommanding aspect also is an example where the second type of assumption was made incorrectly. This aspect provides unsynchronised advice responsible for receiving requests from web browsers, extracting incoming and outgoing data streams, and obtaining and triggering a command to handle response to the request. Because command objects are reused between requests, the aspect makes an implicit assumption that the advice will never be executed in parallel by multiple requests with the same command type. However, the servlet specification makes no guarantees about threading [30], so this assumption is not actually valid in the current weaving context of the aspect, which could potentially even lead to data corruption.

HealthWatcher's HWManagedSynchronization aspect provides its own notion of locking, which is not reentrant. The aspect makes an assumption of the third type, namely that lock(\( k \)) is never invoked more than once for the same \( k \) unless a corresponding unlock(\( k \)) has been invoked. This assumption is valid for the current pointcut definition and code, but this may change when the base evolves.

Discussion. The most critical of the three types of assumptions we have found is certainly the second one, where an aspect assumes a critical section to be synchronised already appropriately in its weaving context. If such an assumption breaks, it may lead to unwanted behaviour due to race conditions that can be very difficult to identify. It is tempting to classify such an assumption as a bug that should be fixed by introducing explicit synchronisation in the advice. However, this may introduce its own complications, in particular the danger of inadvertently introducing deadlocks. Breaking the first type of assumption is likely to cause inefficiencies. It can also cause runtime errors.

Synchronisation assumptions cannot currently be checked by ajc.

Name. Assumptions about who could share monitors

Description. When an aspect implements synchronisation, it needs to decide what monitors to use. At a minimum, there is a choice between using an object private to the aspect and an object that can be accessed from outside the aspect. This decision is important, because for publicly accessible monitors it implies an assumption about what can happen to the monitor outside the aspect. In particular, the aspect may need to assume that the monitor object is not used in synchronized clauses outside the aspect, as this may lead to deadlocks with the synchronisation code inside the aspect.

Examples. HealthWatcher's HWLocalSynchronization aspect provides synchronisation for a number of method calls in the system. It effectively wraps each call in a synchronized(target) clause. Because this uses the target object itself as a synchronisation monitor, this monitor is easily accessible to other code in the system, which could also attempt to synchronise on it. Aspect HWLocalSynchronization, thus, assumes that either no other code in the system contains such synchronized clauses or that they will not cause deadlocks in interaction with HWLocalSynchronization where they exist.

Discussion. When aspect and base evolve independently and aspect monitors are accessible outside the aspect that uses them, deadlocks may occur as an unexpected side effect of evolution. At the same time, aspects may need to make their internal monitors available so that they can synchronize their behaviour with other aspects.

Synchronisation assumptions cannot currently be checked by ajc.

3.2.2 Architectural Assumptions on the Base

Pointcut specifications can be fragile because they use syntactic properties of the base. At a deeper level, they can also become fragile, because they rely on certain architectural properties of the base, which may change as the base evolves.

Name. Well encapsulated communication

Description. Aspects may assume communication between important architectural layers or components to be well encapsulated in the code. Let calls(\( C, D \)) indicate that some elements of \( C \) di-
HealthWatcher has evolved from a strictly servlet-based architecture to an architecture where all requests are encapsulated as command objects, but may originate from different user interfaces. This evolution has broken the assumptions of many aspects about the layered architecture of their weaving context. Some aspects have not been updated correctly, so that they still reference the servlet interfaces rather than the new command classes.

**Examples.** HealthWatcher has evolved from a strictly servlet-based architecture to an architecture where all requests are encapsulated as command objects, but may originate from different user interfaces. This evolution has broken the assumptions of many aspects about the layered architecture of their weaving context. Some aspects have not been updated correctly, so that they still reference the servlet interfaces rather than the new command classes.

**Discussion.** Aspects that make assumptions about the architecture of their weaving context, break easily when this architecture evolves. Because, with AspectJ, the base is intended to be oblivious of the aspects woven into it, base evolution may easily happen without the developers realising the need to co-evolve aspects. Such situations could only be identified if the aspect assumptions have been made explicit and are verified during weaving.

### Name. Well encapsulated data

**Description.** Aspects may also need to advise all data types in an application (e.g., to add the Serializable marker interface). In such scenarios they may make assumptions about how data classes can be differentiated from classes that also provide behaviour. Let $data(e)$ indicate that $e$ represents a data class for $e \in S$. Then, an assumption about well-encapsulated data can be formalised as $\forall_A \Rightarrow \forall e \in S. P(e) \Rightarrow data(e)$. Here $P(e)$ is a predicate that captures how aspect $A$ identifies data classes—for example, through a naming convention.

**Examples.** In HealthWatcher, the system had initially been refactored, grouping all data classes in their own package, so that aspects could assume a naming convention based on package structures. As with the more behavioural assumptions on base architecture above, assumptions about encapsulation of data are easily broken when the base evolves.

### 3.2.3 Assumptions on Particular Coding Patterns

Aspects regularly make assumptions about the internal structure of the base code they advise.

**Name.** Assumptions on semantic patterns in the base

**Description.** Aspects may make assumptions on details of the control-flow or on data values in the base—for example

- **Method $m1$** is invoked exactly once within the control flow of method $m2$;
- **Method $m1$** is only invoked in the control flow of method $m2$;
- **Method $m1$** invokes method $m2$ before invoking method $m3$;
- Input provided will always follow a certain pattern or contain certain distinct elements;
- Particular return values on control flow (e.g., in connection with chain-of-responsibility-like design patterns [12]) have a particular meaning; or
- **Method $m1$** will not modify parameter $p$ before invoking method $m2$.

**Examples.** `HWPersistenceExceptionHandler` in HealthWatcher makes very distinctive assumptions about the control flow in the base and the order in which methods are invoked in order to pick up data to be used in constructing meaningful exception detail. As another example, `UtilAspectEH` in MobileMedia assumes that method `MediaUtil.readMediaAs.ByteArray` is only used in methods reading in images and uses this assumption in constructing error messages. In fact, the code is also invoked for music data, which might lead to slightly misleading error messages should an error occur.

**Discussion.** Aspects relying on such patterns in the base will break when the base is refactored or otherwise evolves so that the patterns are no longer manifested. This is a fundamental issue in aspect-oriented programming, as such patterns are often needed to precisely indicate where an aspect should be woven into the base.

### Name. Assumptions on advice structures in aspects advised

**Description.** Aspects may advise the execution of advice in other aspects. Unfortunately, there is no opportunity in AspectJ to specify the type of advice that should be advised (or in fact select specific advice from one aspect). Since the correct behaviour to weave in may depend on the type of advice, aspects may need to make assumptions about what types of advice will be present.

**Examples.** Glassbox provides mechanisms for containing errors occurring in the monitoring infrastructure rather than in the actual system being monitored [4]. For technical reasons, this can only be done safely for before or after advice, but not for around advice. The `ErrorContainment` aspect currently can only assume that no monitoring code will contain around advice.

**Discussion.** This should probably rather be classified as a current limitation of AspectJ. It should be possible to extend AspectJ so that pointcuts can that distinguish between different types of advice.

#### 3.2.4 Assumptions On the Code Called from Advice

Advice may make assumptions on code it invokes. This is very much similar to assumptions in non-aspect-oriented code. However, because the base code may be oblivious of the aspect code, new versions of the base code may inadvertently invalidate such assumptions. This leads to an even stronger need to make these assumptions explicit.

### 4. Threats to Validity

We have presented a categorisation of assumption types found in three AspectJ projects. If we want to generalise from these findings, there are three main threats to validity to be taken into account:

1. We have studied a limited number of projects only;
2. We have identified and clustered assumptions manually; and
3. All of these projects used AspectJ.

We will discuss these in more detail in the following.

#### 4.1 Limited Number of Projects

Because we have only studied three projects, there may still be a number of types of assumptions developers make but that we have not seen in our sample. As we have already said in Sect. 2, our study has not been designed to eliminate assumptions by providing evidence that they would never be made. Instead, the goal of our study is to find evidence of assumptions that are being made, so that we can then move on to provide support for developers making such assumptions. If more types of assumptions are identified later on, we can then try to provide additional support for these assumptions if that is desirable. Even so, the study is already based on a
collection of 183 concrete assumptions from all three projects combined. Furthermore, as discussed in Sect. 2.2, we have specifically selected the three projects studied to cover a wider range of different uses of AspectJ. We believe that this gives us substantial coverage and helps to avoid missing major categories of aspect assumptions.

4.2 Manual Identification and Clustering of Assumptions

As we have performed a manual analysis of the AspectJ code and the documentation for the three projects studied, there is a danger that we may have misinterpreted code or documentation or may have overlooked essential elements. In this case, our assumption examples might not correctly reflect the assumptions actually made by the aspect developers. To address this concern, we have asked developers from the three projects to validate our collection of assumptions for their respective project. This resulted in some small adjustments as well as in some added assumptions.

For HealthWatcher, we have performed a two-hour interview session with one of the key developers, discussing each individual assumption identified as well as any additional assumptions highlighted by the developer. For example, the mutual exclusion assumption between \texttt{HWMManagedSynchronization} and \texttt{HWMManagedSynchronization} was identified through these discussions. For Glassbox, a similar interview was performed with the lead developer via email. This resulted in one added assumption and four separate updates to the Glassbox code removing assumptions identified and classified by the lead developer as design or implementation problems. Interestingly, the assumption about the correct nesting between \texttt{monitorBegin} and \texttt{monitorEnd} pointcuts in \texttt{AbstractMonitor} was cited as a major cause of problems in the past, specifically because it cannot currently be made explicit and consequently cannot be tested or verified easily as the system evolves. Unfortunately, no feedback was received from the MobileMedia developers contacted.

As we have manually clustered the assumptions into categories of assumption types, there is a danger that these categories may inaccurately reflect the assumptions of AspectJ developers. We have used a bottom-up approach in constructing our categories: In an initial step, we have clustered all assumptions from a project that were actually repeated instances of a pattern of assumptions. Based on this initial clustering step, we have then derived a set of basic assumption categories by removing project-specific information and clustering assumptions that were effectively similar. Starting from these basic categories, we then manually clustered categories into a hierarchy of categories. After we had done this for one project, we added in information from the other projects in a similar manner, restructuring and refining the current hierarchy as and where necessary. Most of the categories eventually presented in this paper correspond to a cut-off point such that every category is supported by at least two examples from more than one project. Categories ‘Assumptions on advice structure’, ‘Well-Encapsulated Data’, ‘Monitor Sharing’, and ‘Mandatory Aspect’ are each supported by one example only, but have been included because they represent interesting complements to the other categories in their group. Additionally, assumptions on synchronisation, ITD provision, and inter-advice assumptions are each supported by examples from only one of the projects. Finally, the categories we identified have again been validated in discussion with aspect developers. This discussion largely confirmed the categories identified. An interesting point was raised by Ron Bodkin from Glassbox, who indicated that some of the assumptions in Glassbox had actually “crept in through incremental evolution of aspects”\textsuperscript{6}. This suggests the change of aspect developer assumptions over the lifetime of an aspect-oriented system as an interesting field for further study.

4.3 AspectJ Specificity

Our study focused on AspectJ-based projects only. Aspect orientation encompasses more than AspectJ. Although we believe that some of the assumption types found are equally applicable to other types of aspect orientation, further research is required to understand the assumptions made by developers using different modularisation techniques.

While this means that the assumption types we have found cannot be generalised directly to other modularisation techniques, it does not invalidate the usefulness of the categorisation in itself. We believe, an AspectJ-specific categorisation is useful in its own right because AspectJ is probably the most popular aspect-oriented language at the moment.

5. RELATED WORK

Greenwood et al. [16] present an empirical study of the pointcut-frailty problem, in which they identify specific sources of fragility in aspect-oriented code using the PointcutRejuvenator tool [22]. In our study, we look beyond the technical notion of pointcut frailty and aim to extract the original assumptions that a particular pointcut was designed to capture. This should enable us to check if these assumptions are actually represented adequately by a particular pointcut. Khatchadourian et al. [22] discuss an approach using the joinpoints actually referenced by a pointcut to infer something about the intention behind the pointcut and to suggest potential improvements in pointcut expression in order to make the pointcuts more robust. The categories of assumptions we have identified could potentially be used to improve their approach by providing more focused feedback to aspect developers, in particular about assumptions that are only made implicitly.

Katz [21] classifies different types of aspects by how much they modify the base behaviour. Clifton and Leavens [6] had previously defined two initial classes in a similar spirit. Katz then goes on to show that these different categories correspond to different proof complexity for the verification of behavioural properties of the overall system composed of aspect and base. In our study, we are less interested in the specific functionality provided by an aspect and more in what assumptions about its weaving context an aspect relies on. Only when this is known and these assumptions can be made explicit, can we begin to verify properties of aspects and aspect-oriented systems.

Nonetheless, there is already work studying how properties of aspects and aspect-oriented systems could be verified: A number of papers focus on showing that properties of a base module are maintained even if an aspect is woven [2,21,27]. Goldman and Katz [13] present work that allows to prove that an aspect guarantees certain properties for any application that it is woven with, as long as this application fulfils certain assumptions. All of these works assume that aspect developers make certain types of assumptions and that aspect behaviour can best be captured using particular formalisms. Our work is the first study attempting to identify what assumptions aspect developers actually make. Some of the assumptions we have found—for example in the Coding-Pattern categories—can probably best be treated with some of the techniques described in [2, 21, 27]. Others—for example the assumptions on aspect deployment—may be more easily handled using different, more static formalisms. For many of the assumption types, we have iden-

\textsuperscript{6}Personal communication

\textsuperscript{5}A mindmap with the complete hierarchy and links to the concrete assumptions found in the projects studied can be found on-line.
6. CONCLUSIONS

This paper presented the results of a first retrospective study into assumptions made by developers of aspects in AspectJ. This work has resulted in a categorisation of types of assumptions. As discussed in Sect. 1, the assumption categories we have identified can be used in different ways, depending on what we believe to be the source of the assumption:

- **Code improvement.** This includes assumptions such as the example we found for the mutual exclusion between sub-aspects of one super-aspect. These should probably be reclassified as bugs or coding mistakes.

- **Assumption elicitation.** For example, the synchronisation assumptions found in HealthWatcher seem to be largely implicit, as indicated by discussions with one of the lead developers. These categories could help make developers aware of the assumptions they might be making so that they can inspect them and possibly adjust their code appropriately.

- **Assumption verification.** For many of the categories, we have provided parameterised formalisations of the aspect assumptions. Even though these still need more work and refinement, it is clear that having such patterns can be beneficial in providing language and tool support for making assumptions explicit and verifying these assumptions: In particular, we can provide a dedicated language concept for each type of assumption that allows developers to provide concrete values for the parameters in the corresponding formal representation. The predicates that we have defined in this paper in a somewhat ad hoc fashion can be realised through code analysis or additional speculative aspects [21] and the parameterised formula can then be instantiated to a complete formula to be used, for example, in verification based work such as [13, 27].

Thus, the classification can form the basis for future work in developing language and tool support for making these assumptions explicit in the source code and using them for verification purposes or for detecting bad smells and suggesting code improvements to aspect developers.

Acknowledgments

This work was supported by the European Union through FP7 Marie-Curie Fellowship RIVAR. The authors would like to thank Phil Greenwood for discussions on the HealthWatcher project and Ron Bodkin for discussions on GlassBox as well as Shmuel Katz for discussions on an earlier version of this paper.

7. REFERENCES


