RIVAR -- Rich Interfaces for Verifiable Aspect Reuse

Collection of empirical data on assumptions made by aspect programmers about the context in which their aspects will be woven.

In the table below, enter information for each advise on a separate line. Use additional lines for different assumptions. Enter assumptions in English text giving as much detail as needed to completely describe the assumption. Coding and classification will be performed in a separate step.

Project:	*HealthWatcher*
Version:	*10 (ExceptionHandling)*
General comments:	*HW was not designed with reuse in mind, so the aspects are often very specific to the concrete usage context. Sometimes, pointcut and binding will be combined within the same aspect, such as where marker interfaces are used, e.g. In HWLocalSynchroniszation.aj. This version of this document contains updates based on my discussion with Phil Greenwood about the data originally collected mainly through manual inspection and interpretation of the code.*

Adivce data:

File	Lines	Source of Assumption (e.g., comment, interview, mailing list, interpretation of code, etc.)	Assumption Description	Comment
hw.a.concurrency.HWLocalSynchronzation.aj	18--22	Interpretation of the code	The advise assumes all methods (including non-public) in all classes implementing the SynchronizedClasses tagger need to be synchronized.	Comment Phil: This uses a slight underspecification, assuming it is not going to do any harm.
hw.a.concurrency.HWLocalSynchronzation.aj	18--22	Interpretation of the code	The advise assumes that synchronisation has to happen on the target object of the call itself rather than on a separate monitor. This implies that other threads may also synchronise on the same monitor introducing additional mutexes or potentially even deadlocks.	Check if this has ever been discussed. Phil: Not sure if it has. However, it is likely. In any case, it is an assumption made by the system.
hw.a.concurrency.HWManagedSynchronization.aj	25--30, 32--37	Interpretation of the code	ConcurrencyManager does not perform any analysis of why a certain key is in the keys database. As a consequence, there is an assumption in this advice combination that there is never an attempt to lock the same key twice within the same cflow (as in lock(k), lock(k), unlock(k), unlock(k)). Doing so would lead to deadlock on the second lock(k).	This is not a problem for the aspect in its current context as it only applies to three 'insert' operations and we can, thus, make sure through inspection of these three operations that this will never be invoked in a nested fashion. However, as soon as the implementation of these functions is touched, we would need to establish this again. Comment Phil: This was not an explicit assumption, but should have been. The developers could have used cflowbelow constructs to avoid this causing any trouble.
hw.a.concurrency.HWLocalSynchronzation.aj, hw.a.concurrency.HWManagedSynchronization.aj		Interview Phil Greenwood	The system assumes either one of these two aspects to be deployed. HWLocalSynchronisation is related to the in-memory implementation of data storage whereas HWManagedSynchronisation is related to the database-based implementation of data storage.	
hw.a.concurrency.HWTimeStamp.aj	41--43, 46--49, 52--93	Interpretation of the code	These pieces of advice assume that insert, update, and search are the only places where a complaint record is modified. Interestingly, they do not take into consideration removal of a complaint record. ComplaintRepositoryRDB.remove is currently implemented as an empty method, but if it were to do anything, this would probably cause a problem.	The HW Application does not allow removing complaints once registered. Thus, there is an assumption that removal will never happen. This is probably an unimplemented requirement, but not having support for it in the aspect is part of the experimental (rather than productive) setup of the system.
hw.a.concurrency.HWTimeStamp.aj	159--163	Comment	This advice is needed because of distribution. When the client side makes an update, the server side complaint is updated, but the client side is not. Thus, we need also to update the client side object (hence the cflow) and increment it. Note that this was not needed before the Observer pattern, because we would update the object just once per request. Now we update it more than once and this synchronization is needed.	This really shows that the advice above was making additional assumptions about its usage context, namely that each complaint record's timestamp would only be incremented once per request regardless of how many individual updates to the record this request entailed. This issue was caught manually and a fix was implemented for a special situation. The new assumption is now that each complaint record's timestamp is only modified once per request, except where the record is involved in an observer pattern (where the timestamp will be incremented once per update of the record).
hw.a.distribution.HWClientDistribution.aj	21	Interpretation of the code	Assumes that all communication between client and server always happens through an IFacade instance. This is an architectural rule that is not made explicit in the code.	
hw.a.distribution.HWServerDistribution.aj	21	Interpretation of the code	Assumes that all data that is ever exchanged between server and client is implemented in classes in hw.model.*.	Comment Phil: This has been ensured by an initial refactoring of the code-base that was used to develop HW.
hw.a.distribution.HWServerDistribution.aj	21	Comment; Interpretation of the code	Also, seems to assume that these only need to be made serialisable on the server side. However, this could also be an error in my interpretation: From the comment: "makes model classes serializable (actually this is also needed in the client)". Hence, it seems that this aspect is actually also meant to be woven into the client code, even though this is not strictly necessary. The assumptions about which part of the code this is to be woven into are not made explicit.	This assumption assumes that this aspect will only be woven for the server code. Looking at build.xml (and the comment above) I'm not sure this is actually the case. Need to check with developers. If it isn't the case it is somewhat unclean in the separation of concerns as the server aspect also contains client concerns. Comment Phil: This aspect is deployed both on the client and the server. This is a bit unclean. It implies an assumption about deployment of aspects.
hw.a.distribution.RMIClientDistribution.aj	20	Comment; Interpretation of the code	From the comment, this seems to assume that RMIClientDistribution is the only specialisation of HWClientDistribution (otherwise it should be within(RMIClientDistribution+) rather than within(HWClientDistribution+).	Comments Phil: - RMIException is RMI specific, so would probably not occur in other subaspects anyway - Even though this has been structured in super- and sub-aspects, there always was an assumption that only RMI would actually be implemented - Even if more than one sub-aspect were to be implemented, the assumption always was that only one of them would be deployed at the same time.
hw.a.distribution.RMIClientDistribution.aj	30--51	Interpretation of the code	This assumes a remote server to be bound to "//" + healthwatcher.Constants.SERVER_NAME + "/" + healthwatcher.Constants.SYSTEM_NAME	This implies an assumption that a corresponding aspect (RMIServerDistribution) will be woven into the system on the server side that appropriately binds the name to a server instance before this lookup is evaluated. So, in a way this is a constraint about the collaboration of two aspects (or rather, two separate pieces of advice) in a system.
hw.a.distribution.RMIServerDistribution.aj	20--34	Interpretation of the code	Assumes that serverStart matches exactly HealthWatcherFacade.getInstance() (line 23 should really be 'proceed()').	This may really be a bit over-exacting for this particular example, but this assumption does have the potential of breaking the encapsulation provided by the explicit pointcut. So, as a consequence, we should be able to document what we assume about pointcuts inherited from super-aspects or expected to be defined in sub-aspects.

RIVAR -- Rich Interfaces for Verifiable Aspect Reuse

Collection of empirical data on assumptions made by aspect programmers about the context in which their aspects will be woven.

In the table below, enter information for each advise on a separate line. Use additional lines for different assumptions. Enter assumptions in English text giving as much detail as needed to completely describe the assumption. Coding and classification will be performed in a separate step.

Project:	*HealthWatcher*
Version:	*10 (ExceptionHandling)*
General comments:	*HW was not designed with reuse in mind, so the aspects are often very specific to the concrete usage context. Sometimes, pointcut and binding will be combined within the same aspect, such as where marker interfaces are used, e.g. In HWLocalSynchroniszation.aj. This version of this document contains updates based on my discussion with Phil Greenwood about the data originally collected mainly through manual inspection and interpretation of the code.*

Adivce data:

File	**Lines**	**Source of Assumption (e.g., comment, interview, mailing list, interpretation of code, etc.)**	**Assumption Description**	**Comment**
hw.a.exceptionHandling.ExceptionHandlingPrecedence.aj	44	Interpretation of the code	Assumes that there are no precedence conflicts other than with ServletCommanding.	I guess, this probably doesn't need any additional code, just checking that indeed there are none.
hw.a.exceptionHandling.HWDistributionExceptionHandler.aj	23	Interpretation of the code		Really just seems to be here for laziness sake to avoid having to catch IOExceptions in the code below. **Comment Phil: Need to ask Nelio**
hw.a.exceptionHandling.HWDistributionExceptionHandler.aj	25--48	Interpretation of the code	Assumes that all HW servlets are subclasses of HWServlet; that is, no user interaction happens through any other channel.	Comment Phil: This is a very dangerous assumption! Actually, one of the first changes introduced into HW was the command pattern. The purpose of this was to decouple HW from its UI, so that it could be used with something other than servlets. No other UI has ever actually been implemented, but if it had been, this assumption here would have broken immediately.
hw.a.exceptionHandling.HWPersistenceExceptionHandler.aj	40--58	Interpretation of the code	Assumes that all HW servlets are subclasses of HWServlet; that is, no user interaction happens through any other channel.	
hw.a.exceptionHandling.HWPersistenceExceptionHandler.aj	62--73	Interpretation of the code	Assumes that Statement.executeQuery does not internally modify the SQL code. Also assumes that there is at most one call to executeQuery within AddressRepositoryRDB.search.	
hw.a.exceptionHandling.HWPersistenceExceptionHandler.aj	75--77	Interpretation of the code	Assumes that all exceptions from AddressRepositoryRDB.search are wrapped into PersistenceMechanismExceptions.	This advice seems to violate an implicit assumption of the two pieces of advice above that the message of PersistenceMechanismExceptions will actually be maintained. Instead, they are thrown away.
hw.a.exceptionHandling.HWPersistenceExceptionHandler.aj	81--102	Interpretation of the code	Assumes a) that ComplaintRepositoryRDB.update invokes internal_update, b) that internal_update invokes at most once Statement.executeUpdate, c) that executeUpdate does not modify the SQL query passed to it, and d) that maintaining this query in an exception will actually have an impact on error handling.	The final assumption is violated by the first advice in the series, as it simply replaces any such exception with a standard RepositoryException. **Comment Phil: Interestingly, Phil seemed to recall that internal_update was explicitly introduced as scaffolding for this to enable picking up some parameter values. Further analysis of the code didn't support this, though. It remains unclear why internal_update exists at all.**
hw.a.exceptionHandling.HWPersistenceExceptionHandler.aj	106--124	Interpretation of the code	Assumes a) that ComplaintRepositoryRDB.insert invokes Statement.executeQuery at most once, b) that Statement.executeUpdate does not modify the SQL query passed to it, and) that maintaining this query in an exception will actually have an impact on error handling.	The final assumption is violated by the first advice in the series, as it simply replaces any such exception with a standard RepositoryException.
hw.a.exceptionHandling.HWPersistenceExceptionHandler.aj	133--148, 156--172	Interpretation of the code		There may be additional assumptions about exception flow between this and other advice in the same aspect, but I cannot immediately understand this. **Comment Phil: Nelio may be able to help here.**
hw.a.exceptionHandling.HWPersistenceExceptionHandler.aj	1--173	Interpretation of the code	Assumes that these explicitly listed pointcuts cover all places where persistence-related exceptions are thrown in the system.	
hw.a.exceptionHandling.HWTransactionExceptionHandler.aj	24--41	Interpretation of the code	Assumes that all HW servlets are subclasses of HWServlet; that is, no user interaction happens through any other channel.	
hw.a.exceptionHandling.HWUpdateObserverExceptionHandler.aj	60--72	Comment	Note that this aspect needs a lot of knowledge about how the observer pattern affects the application and how the application should deal with its exceptions.	**Comments Phil:** **- Pointcut only addresses updates. This is so, because the observer pattern is only applied to update calls, too.** **- Assumes requests are handled in HTML (returning error messages wrapped in an HTTP die()). This has been created against the command pattern, but still makes strong assumptions about this being implemented based on servlets. See above for problems with this.**
hw.a.logging.HWLogging.aj	49--52	Interpretation of the code	Assumes that in any run of the system, either a new HealthWatcherFacade instance is created or a servlet class is initialised. These are cross-assumptions on the existence and weaving of other aspects (e.g., HWServerDistribution).	Really, this seems to be a stupid aspect that would much better be implemented by modifying LogMechanism and setting the default value for logFile to Constants.LOG_PATH. Not sure why this wasn't done in this manner.
hw.a.patterns.RepositoryFactory.aj	26--35	Interpretation of the code	Assumes that all code will use the getRepositoryFactory() method of AbstractRepositoryFactory to access specific repository factories. This seems to imply that all relevant code is aware of the presence of this aspect, as the method isn't actually in the interface of AbstractRepositoryFactory.	The key thing seems to be the implication this has for code using the abstract factories, namely that such code is aware of the presence of this aspect and the opportunity of (and need for) calling getRepositoryFactory. At the very least, there is an assumption about layering here that should be documented in the code. **Comment Phil: This seems to be a general problem with ITDs. Maybe the solution here is really to provide better tooling, that can check for the presence of such methods etc. dynamically during development.**
hw.a.patterns.ServletCommanding.aj	105--140	Interpretation of the code	Assumes HWServlet is only initialised once (as this is a singleton aspect and commandTable is a member of the aspect).	Note that the servlet specification doesn't actually state that this is the case. Servlet engines are free to maintain pools of servlets or even instantiate a new servlet for every connection. This implies the servlet engine would invoke init() multiple times. Of course, this is not a big problem, all that happens is that unneeded objects are instantiated, but it is still a potential waste of resources.

RIVAR -- Rich Interfaces for Verifiable Aspect Reuse

Collection of empirical data on assumptions made by aspect programmers about the context in which their aspects will be woven.

In the table below, enter information for each advise on a separate line. Use additional lines for different assumptions. Enter assumptions in English text giving as much detail as needed to completely describe the assumption. Coding and classification will be performed in a separate step.

Project:	*HealthWatcher*
Version:	*10 (ExceptionHandling)*
General comments:	*HW was not designed with reuse in mind, so the aspects are often very specific to the concrete usage context. Sometimes, pointcut and binding will be combined within the same aspect, such as where marker interfaces are used, e.g. In HWLocalSynchroniszation.aj.* *This version of this document contains updates based on my discussion with Phil Greenwood about the data originally collected mainly through manual inspection and interpretation of the code.*

Adivce data:

File	Lines	Source of Assumption (e.g., comment, interview, mailing list, interpretation of code, etc.)	Assumption Description	Comment
hw.a.patterns.ServletCommanding.aj	182	Interpretation of the code	Assumes the presence of the (apparently unrelated) pointcut on lines 159--162 and advise in the super aspect CommandProtocol dealing with this pointcut and executing a particular command. Furthermore, assumes that the advise in CommandProtocol will be executed after commandTrigger, because the information about which command to invoke will only be available after the call to setCommand.	This seems a very roundabout way of doing things. It would have been easier not to refer back to the abstract aspect for CommandProtocol and to implement command invokation on Line 182 directly rather than using this indirect call. Comment Phil: This is an interesting assumption. However, all the assumptions are essentially true by construction. Still, it may help to have explicit documentation of these assumptions.
hw.a.patterns.ServletCommanding.aj	172--183	Interpretation of the code	This code is a critical section (threads potentially share and co-modify the command object). It is not appropriately guarded here, so it makes strong assumptions about the weaving context to be appropriately synchronised.	In fact, if I understand the servlet specification correctly, there is no such guarantee, so this could actually lead to a race condition when two clients send the same command to the server at the same time. With a bit of bad luck in code interleaving, in such a situation the response to one of the clients could be based (completely or partially) on the request data from the other client. If the command servlets themselves are programmed badly, this may even lead to data corruption.
hw.a.patterns.ServletCommanding.aj	172--183	Interview Phil Greenwood	Also, there is an assumption about the existence of the 'operation' parameter in the URL and that it is correct and refers to an existing command.	
hw.a.patterns.UpdateStateObserver.aj	24--28	Interpretation of the code	Together with the way the subjectChange pointcut is phrased (namely that it mentions Subject+.set* instead of referring to the classes directly) this makes an assumption that no other code in the system uses the ObserverProtocol aspect and makes other classes implement Subject.	This may be a lesser problem, as the advise on Lines 30--32 probably ensures that the subject--observer bindings are still set up reasonably, but there may be circumstances where this leads to problems. In any case, the assumption is not made explicit. Really, I would probably classify this as a programming error or bad smell at least, as the proper thing to do would probably be to create a protected sub-interface of Subject and use this in the declare parents code. **Comment Phil: There is an assumption here, but it is very much implicit.**
hw.a.patterns.complaintState.AnimalComplaintStateAspect.aj	33--47	Interpretation of the code	This relies on the fact that setStatus in ComplaintState is empty and that the state in fact is a constant based on the particular sub-class of Complaint that is being used. Furthermore, this code relies on the fact that even though AnimalComplaintState has no way of returning its status, AnimalComplaintState.setStatus is called obediently and AnimalComplaint.setSituacao is never called directly even though it is public. This is an example of using an empty method as scaffolding to enable advise to be triggered.	This code had me quite confused for a while. It took some time and digging around the code to fully understand what was going on. A good argument for why such assumptions should be made explicit. Below is my initial comment for this (now obsolete, but left in to record my train of thought to some degree): To be honest, I'm confused about this aspect as a whole: 1. It doesn't seem to do anything useful in terms of modularisation. The code in the aspect would be better placed inside AnimalComplaint itself. It would be more localised and easier understood this way. 2. It would appear to me that even if the advise on lines 33--47 is woven into the system, it contains dead code only because its condition will never be true (it is only ever referenced from AnimalComplaint.setStatus, which first sets the situacao...). **Comment Phil: This is not so much an assumption as a really bad implementation. This should really be using around advice and not bother about picking up state from the joinpoint object.**
hw.a.patterns.complaintState.ComplaintStateAspect.aj	49--67	Interpretation of the code	This relies on the fact that setStatus in ComplaintState is empty and that the state in fact is a constant based on the particular sub-class of Complaint that is being used. This is an example of using an empty method as scaffolding to enable advise to be triggered.	Related to above and below
hw.a.patterns.complaintState.FoodComplaintStateAspect.aj	35--49	Interpretation of the code	see above	see above
hw.a.patterns.complaintState.SpecialComplaintStateAspect.aj	31--43	Interpretation of the code	see above	see above
hw.a.persistence.HWDataCollection.aj		Interpretation of the code	Makes an assumption that repositories are a core concept in the application regardless of persistence. So, all that is needed is to provide a different repository implementation and this will be used to store data explicitly by the individual business object classes.	Not sure why this would use an aspect at all. It seems to me that the key thing is the abstract factory pattern, which would have been more easily added in without aspects. **Comment Phil: This could have come about through the particular order in which changes were introduced in different releases. In attempting to minimise changes to existing code, developers would have introduced a new aspect to encapsulate an increment for a particular release.**
hw.a.persistence.HWPersistence.aj	61--62	Interpretation of the code	This seems to make the assumption that HealthWatcherFacade.new is invoked only once in an application or if not it is at least invoked in separate critical sections appropriately mutexed.	If this is not true, there is a danger of creating a connection in the persistence mechanism that is never actually used due to a race condition. **Comment Phil: This seems to be an explicit assumption, as HWFacade 47--52 is synchronised and HWFacade is a singleton.**
hw.a.persistence.HWTransactionManagement.aj				No relevant assumptions as far as I can see
lib.patterns.CommandProtocol.aj				No relevant assumptions as far as I can see
lib.patterns.ObserverProtocol.aj	152--157	Interpretation of the code	Assumes that calls to updateObserver do not lead to calls to add/removeObserver	If they do, a ConcurrentModificationException will be thrown, so the error will be spotted at runtime.