Software architecture reconstruction

Krikhaar, R.

Link to publication

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 5

Redefined Architecture

In the previous chapter we have discussed the described level of software architecture reconstruction. The next level of the SAR method concerns the improvement of existing software architectures: the redefined level.

5.1 Introduction

A described architecture consists of the explicit description of the software architecture of an existing system. This is for example useful for helping developers comprehend that system. The improvement of an existing software architecture is logically the next subject in our discussion. Improving an existing software architecture may be of great help in simplifying a system’s maintenance and its extensions. The realization of improvements results in a redefined architecture which will be discussed in this chapter [KPS+99]. The process of improving a software architecture is called (software) re-architecting.

Changing a software architecture may affect many parts of the software. Before a change can be introduced, an architect must know exactly which parts will be affected but also the cost of implementing the change. Impact analysis is a technique for calculating the consequences of a change in advance, without realizing it in the actual source code. An architect can quietly consider all the pros and cons of a change before he or she decides to implement it.

Figure 5.1 shows the various activities involved in architecture improvement. A software model containing the ArchiSpecs as discussed in e.g.
Chapter 4 is derived from the source code, documentation and information obtained from system experts. This software model is subjected to impact analysis: an architect has an idea (e.g. moving a function to another module by means of changing the partof relation) that can be simulated (e.g. by means of recalculating certain ArchiSpects). This results in a modified software model. This new model must be evaluated by the architect, which may influence the original idea, resulting in an adapted or new idea. After some iterations, the refined idea may be accepted or discarded. Accepted ideas must be transformed into a prescription of modification for the implementation. This prescription can be applied to the source code and documentation, resulting in a new system. Note that when we extract information from the modified system, we obtain the same software model as we had constructed after simulating the accepted idea.

The advantages of this approach are clear. An architect can apply his or her ideas to a software model and figure out the consequences without involving many people and without affecting the actual system. Note that we have described a purely architecture-driven analysis, which does not take into account business, organisation or process-related issues. For example, the implementation of an improvement may be beneficial from a software-engineering point of view, but it may have disastrous consequences for the product’s time-to-market. Besides the impact analysis described above, considerations of the latter kind must also be taken into account in a com-
In this chapter we will discuss some ArchiSpects that are helpful in performing impact analysis aimed at improving the software architecture of an existing system; see Figure 5.2. The following ArchiSpects will be discussed in an arbitrary order:

- *Component Coupling*, the quantified dependencies between the software parts of a system;
- *Cohesion and Coupling*, metrics that describes connectivity between various software parts;
- *Aspect Coupling* (using the *Aspect Assignment* InfoPack), quantified dependencies between various parts, taking into account a certain aspect.
5.2 ArchiSpect: Component Coupling

5.2.1 Context

The Component Coupling ArchiSpect belongs to the module view. The results of the Import (Section 4.6) and PartOf (Section 4.7) InfoPacks are required. The Component Dependency (Section 4.9) and Using and Used Interface (Section 4.10) ArchiSpects are closely related.

5.2.2 Description

The Component Coupling ArchiSpect quantifies dependencies as depicted in the resulting diagram of the Component Dependency ArchiSpect, e.g., see Figure 4.11. Quantification is useful for example when we want to remove a dependency between X and Y. The size of a relation can be of help in estimating the amount of effort that will be involved in removing that dependency. Assume that components consist of files that import each other. Then, the number of import statements is a measure (or weight) of the intensity of dependency between the components. But, a relation can be quantified in different ways. We define the following weights:

- size-oriented weight, meaning that the number of relations at the lower level is reflected in the weight of the lifted relation at the higher level;
- fan-in-oriented weight, meaning that the number of entities (e.g., files) used by a component is reflected in the weight;
- fan-out-oriented weight, meaning that the relation is quantified by the number of a component’s using entities.

To illustrate this, consider the use relation depicted in Figure 5.3. Component CompHigher uses component CompLower because High1 uses Low1, amongst other relations. According to the above definition, we obtain the following weights:

- size-oriented: 4 (the number of dashed arrows)
- fan-in-oriented: 3 (the number of files a dashed arrow points to)
- fan-out-oriented: 2 (the number of files at which one or more dashed arrows start)

A combination of the three weights helps to re-architect a system. Assume we want to remove a dependency between two components. The size-oriented weight indicates how many import statements must be removed to
achieve this. On the other hand, the fan-out-oriented weight indicates how many files of the including component will be affected. So, it defines the number of files that have to be changed to remove the dependency between the components.

If we want to replace a component, then the component’s interface with other components must be known. The fan-in-oriented weight defines the number of files that will be used by the including component.

5.2.3 Example

The reconstruction of Component Dependency of two systems has been discussed in Section 4.9. In this section we present the Component Coupling of these systems. The resulting diagram of this ArchiSpect, created by Teddy-PS, contains the same arrows as the diagram of Component Dependency, but the arrows are of different thicknesses. The thickness of an arrow\(^1\) is a measure of the weight of the relation. For the corresponding relation, a tuple with a large (small) weight is represented by a thick (thin) arrow.

The cells in the table representation of Component Coupling contain the three different weights of the various tuples, i.e. the fan-in-oriented, size-oriented and fan-out-oriented weights.

\(^1\)We have used a logarithmic function to map the weight onto an arrow width of a few points.
Med

The (size-oriented) Component Coupling of Med is presented in Figure 5.4. The corresponding diagram of Component Dependency is depicted in Figure 4.11.

A table representation of Med is given in Table 5.1 (with its related Table 4.7). Each affected cell contains three figures, i.e. the fan-in-oriented, size-oriented and fan-out-oriented weights. Note that the table representation explicitly shows the identity relation (if applicable) in contrast with the diagram representation. Furthermore, all the variants of quantification
are presented in a single table.

Comm

The Component Dependency of the Comm system is depicted in Figure 4.12. Figure 5.5 shows the (size-oriented) Component Coupling of this system.

5.2.4 Method

The steps to be executed are similar to those of the Component Dependency method (described in Section 4.9). The required input consists of²:

²For clarity, we will still use the Files and Comps decomposition levels in the description, but any other pair of levels could be used.
<table>
<thead>
<tr>
<th>fan-in size fan-out</th>
<th>Div</th>
<th>Top</th>
<th>Est</th>
<th>Sens</th>
<th>Pres</th>
<th>Util</th>
<th>Netw</th>
<th>UTX</th>
<th>Bot</th>
<th>Bas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Div 9 9 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>66</td>
<td>11</td>
<td>2</td>
<td>41</td>
<td>18</td>
<td>59</td>
<td>119</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>151</td>
<td>11</td>
<td>2</td>
<td>52</td>
<td>25</td>
<td>204</td>
<td>719</td>
<td>60</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>9</td>
<td>2</td>
<td>16</td>
<td>16</td>
<td>52</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Top 1301 11 9 39 4 23 117 186 8220 41 75 6 55 1623 9324 1170 34 43 6 29 522 1191</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Est 32 117 4 4 11 124 11 968 144 333 4 52 968 27 98 2 23 105</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sens 20 151 611</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pres 75 611</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Util 41 147</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>236 193</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Netw 208 1028</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UTX 23 55 27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bot 2 4 4 227 860 1626 193</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bas 303 3725 338</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Component Coupling (fan-in, size, fan-out) of Med
• \textit{imports}_{Files,Files}, a multi-relation that can be constructed by mapping the \textit{imports} relation: [\textit{imports}\_{Files,Files}];

• \textit{partof}_{Files,Comps}, a part-of relation which may be the result of a composition of a chain of \textit{partof} relations, for example (\textit{Med}):

\[
\text{partof}_{Files,Comps} = \text{partof}_{Packs,Comps} \circ \text{partof}_{Files,Packs}
\]

The three variations of weight are calculated as follows (note that we introduce two new lift operator notations $\uparrow$ and $\downarrow$):

**size-oriented:**
\[
\text{imports}_{Comps,Comps} = \text{imports}_{Files,Files} \uparrow \text{partof}_{Files,Comps}
\]

**fan-in-oriented:**
\[
\text{imports}_{FI,Comps,Comps} = \text{imports}_{Files,Files} \downarrow \text{partof}_{Files,Comps} = \left[\text{partof}_{Files,Comps}\right] \circ \left[\left[\text{imports}_{Files,Files} \circ \left[\text{partof}_{Files,Comps}\right]\right]\right]
\]

**fan-out-oriented:**
\[
\text{imports}_{FO,Comps,Comps} = \text{imports}_{Files,Files} \downarrow \text{partof}_{Files,Comps} = \left[\left[\text{partof}_{Files,Comps} \circ \text{imports}_{Files,Files}\right] \circ \text{partof}_{Files,Comps}\right]
\]

**explanation**

In the first definition, the \textit{imports}_{Files,Files} multi-relation is lifted to component level. The lift operator for multi-relations takes into account the number of dependencies at file level in constructing the dependency at component level (as defined in Section 3.5).

To explain fan-in-oriented and fan-out-oriented lifting, we must consider an alternative formula for the lifting of relations: $U \uparrow P \equiv P \circ U \circ P^{-1}$. This alternative formula also exists for multi-relations: $U \uparrow P \equiv [P] \circ U \circ [P^{-1}]$. Note that we must first map the \textit{partof} relation (which is always a binary relation) onto a multi-relation. Furthermore, $[P^{-1}] \equiv [P]^{-1}$.

Figure 5.6 shows the various steps of fan-in-oriented lifting (this figure is an alternative to the view presented in Figure 5.3). In the last part
of the given formula (i.e. the second composition), part of the lift operation is performed ("lifting in its domain"). This intermediate result describes a relation from $\text{Comps}$ to $\text{Files}$ (represented in Figure 5.6 as $\text{imports} \ _{\text{Comps} \rightarrow \text{Files}}$). We are interested in the files imported by a component; we are not interested in the number of times these files are imported. So the intermediate result is normalised, i.e. setting its weight is set to 1 by mapping it first to a normal relation and then back to a multi-relation. The first part of the fan-in-oriented formula performs the rest of the lift operation ("lifting in its range"). The number of file dependencies is thus taken into account in obtaining a fan-in-oriented weight.

The fan-out-oriented lift is defined in a similar manner. We start with the first part of the formula: the intermediate result contains the normalised multi-relation $\text{imports} \ _{\text{Files} \rightarrow \text{Comps}}$. The last step is to lift the domain of this intermediate result to the component level, to obtain the relation $\text{imports} \ _{\text{Comps} \rightarrow \text{Comps}}$ with a fan-out-oriented weight.

5.2.5 Discussion

We have discussed an ArchiSpect which is useful in the context of re-architecting. This ArchiSpect should be part of the software model (as depicted in Figure 5.1).
Undesired component dependencies are intuitively considered less harmful when they have a small weight and more harmful when they have a large weight. An architect may use this information to detect weak spots and to analyse these spots. A large size-oriented weight between two components and a small fan-in-oriented weight may indicate that a file is located in the wrong component. To check whether this is true for the system at hand we will modify the partof relation (idea) and will recalculate the Component Coupling ArchiSpec (simulate).

There exists a relation between the various weights described above. Let’s call the three kinds of weights (size-oriented, fan-in-oriented and fan-out-oriented) $s$, $f_i$ and $f_o$, respectively. We will explain that the following inequation holds: $\max(f_i, f_o) \leq s \leq f_i \times f_o$. Consider all the tuples in the $\text{imports}$ file relation that are responsible for the dependency between two components. The number of tuples in this restricted $\text{imports}$ relation (let’s call it $\text{imp}$) corresponds to the size-oriented weight $s$. The size of the range of the $\text{imp}$ relation describes the fan-in-oriented weight $f_i$. The size of the domain of $\text{imp}$ describes the fan-out-oriented weight $f_o$. The largest possible $\text{imp}$ relation consists of the cartesian product of the domain and the range, which has a size of $f_i \times f_o$. The smallest possible $\text{imp}$ relation contains at least the set of tuples that span the domain (which is sized $f_i$) and the range (which is sized $f_o$). Therefore, the $\text{imp}$ relation has a minimum size being the maximum of $f_i$ and $f_o$.

5.3 ArchiSpec: Cohesion and Coupling

5.3.1 Context

The Cohesion and Coupling ArchiSpec belongs to the module view. It requires the results of the Import (Section 4.6) and PartOf (Section 4.7) InfoPacks. It is related to the Component Coupling ArchiSpec (Section 5.2).

5.3.2 Description

The complexity of a system highly affects the system’s comprehensibility, maintainability and testability. Cohesion and coupling play important roles in expressing complexity. Topics relating to module cohesion and module coupling were discussed by Yourdon and Constantine in the seventies [YC79, SMC74]. These measures have also been used to develop tools
that automatically cluster parts of software, e.g., [MMR+98].

A system’s cohesion describes the connectivity of entities within the part comprising them. It is defined as the ratio of the number of actual dependencies between these entities and the number of all possible dependencies (this agrees with the definition of intra-connectivity given in [MMR+98]). Coupling describes the connectivity between two different entities in terms of their comprising parts (in [MMR+98] this is called inter-connectivity). It is defined as the ratio between the number of actual dependencies of these entities and the number of all possible dependencies. A general rule of thumb (and not more than that) for achieving, amongst other things, a high degree of comprehensibility, is that a system should minimize coupling in favour of maximising cohesion.

Consider the system depicted on the left side of Figure 5.7. The components CompLeft, CompRight and CompLow contain some files that import each other (dotted arrows). The components CompLeft and CompRight import each other (solid arrow) by means of file Y. By moving this file Y from CompRight to CompLeft, we obtain the situation depicted on the right side of this figure. As we can see in the diagram, the degree of coupling between the components has decreased and the degree of cohesion between the files of CompLeft has increased. Without having any knowledge of the system’s semantics we may conclude that the structure has been improved and that the new system is easier to understand. We must note that a good software architecture cannot be created simply by optimizing the cohesion and coupling quality measures; many other aspects also play a role (e.g., decomposing a system into parts that semantically belong together).

5.3.3 Example

Comm

The Cohesion and Coupling of Comm at subsystem level are presented in Table 5.2 (− means that there is no cohesion/coupling). In this example we have chosen files as the constituents of a subsystem (we can take other entity levels, too, e.g., modules). We conclude that for all subsystems the degree of cohesion is low. Files are small units with respect to subsystems and therefore we may expect a low degree of cohesion. We may even state that a high degree of cohesion would be suspect in the case of this system. For the same reason the coupling figures are also low. This discussion shows that a proper understanding of the system is required to be able to draw
In this section we will discuss the calculation of Cohesion and Coupling, given the imports and partof relations. For clarity in the discussion we will use only two decomposition levels, namely Files and Components.

The Dominating Ratio (DR) between two components $X$ and $Y$ relates the actual file imports to all possible file imports between these two components. For example, in Figure 5.8, component $X$ imports file $y_1$ twice,
indicated by the solid arrows. The dashed and solid arrows (6 in total) indicate all the possible imports between the files of component X and those of component Y. Therefore, in this example, the dominating ratio is $DR_{X,Y} = \frac{2}{6} = 0.333$.

We define the Dominating Ratio of component X with respect to component Y, denoted as $DR_{X,Y}$, as follows:

\[
\begin{align*}
Files &= \text{dom}(\text{partof} \ Files, \text{Comps}) \\
\text{imports}_{\text{Comps}, \text{Comps}} &= [\text{imports} \ Files, Files] \uparrow \text{partof} \ Files, \text{Comps} \\
\text{impAll}_{\text{Comps}, \text{Comps}} &= [Files \times Files] \uparrow \text{partof} \ Files, \text{Comps} \\
DR_{X,Y} &= \frac{||\text{imports}_{\text{Comps}, \text{Comps}} \mid \text{dom} \{X\} \mid \text{ran} \{Y\}||}{||\text{impAll}_{\text{Comps}, \text{Comps}} \mid \text{dom} \{X\} \mid \text{ran} \{Y\}||}
\end{align*}
\]

**explanation**

The size-oriented weight of the multi-relation $\text{imports}_{\text{Comps}, \text{Comps}}$ refers to the number of file import statements in the code. The multi-relation $\text{impAll}$ describes all the possible imports between components. The $\text{imports}$ multi-relation is restricted in its domain with X and it is restricted in its range with Y, resulting in a multi-relation $(X,Y,w)$. The weight $w$ refers to the number of file imports between components X and Y. The size of this singleton multi-relation is equal to $w$. Analogously, the number of possible imports is calculated by starting with the multi-relation $\text{impAll}$. We obtain the dominating ratio $DR_{X,Y}$ by dividing both sizes of doubly restricted relations.
The cohesion of a component indicates the degree of connectivity between its comprising files. One way of interpreting connectivity is to look at the imports relation. Cohesion is defined as follows:

\[
\text{Cohesion}_X = \frac{||\text{imports}_{\text{Comps,Comps}}|_{\text{car}} \{X\}||}{||\text{impAll}_{\text{Comps,Comps}}|_{\text{car}} \{X\}||}
\]

\[
= \frac{||\text{imports}_{\text{Comps,Comps}}|_{\text{dom}} \{X\} |_{\text{ran}} \{X\}||}{||\text{impAll}_{\text{Comps,Comps}}|_{\text{dom}} \{X\} |_{\text{ran}} \{X\}||}
\]

\[
= DR_{X,X}
\]

**Explanation**

The numerator defines the number of imports between files within component \(X\) (solid arrows in Figure 5.9). In fact, the restriction results in a singleton relation \(\langle X, X, w \rangle\), where \(w\) indicates the number of actually imported files. The denominator contains the number of possible imports inside \(X\), as we have seen above (solid and dashed arrows in Figure 5.9). This corresponds to the dominating ratio of \(X\) with respect to \(X\).

The cohesion of component \(X\), as illustrated in Figure 5.9, is \(\frac{4}{6} = 0.667\).

**Coupling**

Coupling is a measure of the degree of connectivity between two components. Given the imports relation as a connectivity artefact we define coupling as:
\[
\begin{align*}
imp_{X,Y} &= \text{imports}_{\text{Comps}, \text{Comps}} \quad \text{dom} \left\{ X \right\} \quad \text{ran} \left\{ Y \right\} \\
imp_{Y,X} &= \text{imports}_{\text{Comps}, \text{Comps}} \quad \text{dom} \left\{ Y \right\} \quad \text{ran} \left\{ X \right\} \\
impAll_{X,Y} &= \text{impAll}_{\text{Comps}, \text{Comps}} \quad \text{dom} \left\{ X \right\} \quad \text{ran} \left\{ Y \right\} \\
impAll_{Y,X} &= \text{impAll}_{\text{Comps}, \text{Comps}} \quad \text{dom} \left\{ Y \right\} \quad \text{ran} \left\{ X \right\}
\end{align*}
\]

\[
Coupling_{X,Y} = \frac{\| \imp_{X,Y} \cup \imp_{Y,X} \|}{\| \impAll_{X,Y} \cup \impAll_{Y,X} \|}
= \frac{\| \imp_{X,Y} \| + \| \imp_{Y,X} \|}{\| \impAll_{X,Y} \| + \| \impAll_{Y,X} \|}
= \frac{\| \imp_{X,Y} \| + \| \imp_{Y,X} \|}{2 \times \| \impAll_{X,Y} \|}
= \frac{1}{2} \left( \frac{\| \imp_{X,Y} \|}{\| \impAll_{X,Y} \|} + \frac{\| \imp_{Y,X} \|}{\| \impAll_{Y,X} \|} \right)
= \frac{\text{DR}_{X,Y} + \text{DR}_{Y,X}}{2}
\]

**Explanation**

The numerator of \( Coupling_{X,Y} \) counts the number of file import statements of component \( X \) importing files from component \( Y \) and vice versa (solid arrows in Figure 5.10). As in the cohesion definition, the denominator contains the number of all possible imports (solid and dashed arrows in Figure 5.10). The ratio is a measure of the degree of coupling, which can be rewritten in terms of dominating ratios. Because \( X \neq Y \), and therefore \( \imp_{X,Y} \cap \imp_{Y,X} = 0 \), we can rewrite the numerator by adding the sizes of the two multi-relations. Analogously, the denominator can be written as the sum of two multi-relations. Furthermore, these latter two multi-relations are both of the same size (due to the construction of \( \impAll \) it holds that: \( \impAll \equiv \impAll^{-1} \)).

We note that the degree of coupling between \( X \) and \( Y \) is equal to the degree of coupling between \( Y \) and \( X \) (due to the associative + operator).

The degree of coupling between components \( X \) and \( Y \), illustrated in Figure 5.10, is \( \frac{1}{2} \times \left( \frac{2}{6} + \frac{3}{6} \right) = 0.417 \).
5.3.5 Discussion

Given a set of entities and relations between them (e.g., the imports relation between Files), one can cluster these entities by applying the heuristic “maximise cohesion and minimise coupling”. Note that it makes no sense to increase cohesion simply by artificially creating extra relations between entities within a cluster. Therefore we should consider the above heuristic more carefully. By creating various clusters one in fact divides the set of existing tuples of the relation, e.g., imports, into two parts: a set of tuples that do not cross a cluster’s border and a set of tuples that do cross a cluster’s border. So it is better to define the heuristic as: “maximise the number of tuples in the cohesion part and minimise it in the coupling part”.

Clustering of software parts at different levels in the decomposition hierarchy is a task which can indeed not be performed automatically [Wig97, MMR+98]. Cohesion and coupling metrics can help an architect to make the right decisions about clustering (a tool for software re-clustering, based upon these metrics, has been implemented by Brook [Bro99]). Note that clustering can never be driven by optimising the value of two metrics. Many factors play a role in the clustering process, but only a few can be expressed in metrics.
5.4 InfoPack: Aspect Assignment

5.4.1 Context

The Aspect Assignment InfoPack belongs to the code view. We may need the results of the PartOf InfoPack (Section 4.7). The results are used by Aspect Coupling (Section 5.5).

5.4.2 Description

The notion of aspects has already been discussed in Section 1.5.3. Aspects are a design concept, but they should also be reflected in the implementation in some way. This can be realized in various ways. For example, a file addresses only a single aspect of the system: the aspect to which the file belongs can be encoded in the file name.

The main result of this InfoPack consists of the addressesFiles,Asps relation and the Aspects set.

5.4.3 Example

The Tele system explicitly engineers the notion of aspects during system development in all its phases (i.e. in the forward-architecting process). A file, having an aspect-related name, addresses exactly one aspect.

We will illustrate this with an example of a system that did not initially consider aspects.

Med

During our re-architecting activities we introduced aspects into the Med system. Although it was not possible to apply it precisely in its full meaning, it helped us to construct a new view on the system. We were able to identify the following aspects:

- Clinical: all the software that is sent to a hospital along with the medical system;
- Test: the software needed to test the system during its development;
- Development: the software that comprises all the dedicated tools that are required to develop the system (e.g. for code generation).
We refine the first aspect into:

- *Operational*: software activated by an operator in the hospital;
- *Research*: software prepared for academic hospitals for clinical research purposes;
- *Diagnostic*: software relating to all the service operations performed by a service mechanic;
- *Installation*: software required only to install the system at a (new) site.

So the *Aspects* set and the *Clinical Aspects* subset are defined as follows:

\[
\text{Aspects} = \{ \text{Operational, Research, Diagnostic, Installation, Test, Development} \}
\]
\[
\text{Clinical Aspects} = \{ \text{Operational, Research, Diagnostic, Installation} \}
\]

### 5.4.4 Method

The first step is to determine the various aspects of the system. This task is easy when aspects have been used already during architecture design. If not, we will have to discuss the notion of aspects with architects and designers. The second step consists of identifying these aspects in the software. Although the notion of aspects may not have been explicitly identified so far, it may be possible to determine aspects in code. For example, an aspect like *Logging* may express itself in the naming of functions (*WriteLogMessage*) and/or the naming of files (*LogUtilities*). Given such naming conventions, one can assign an aspect to most of the functions (or files). Because of the heuristic nature of extraction, the results of this extraction should be carefully checked. The functions (or files) that cannot be assigned to an aspect in this way must be assigned by hand.

With the *Med* system, we used the names of the packages to assign aspects. So we were able to extract the \( \text{addresses}_{\text{Packs,Asps}} \) relation by analysing the package name. We can lower this relation to the *Files* decomposition level:

\[
\text{addresses}_{\text{Files,Asps}} = \text{addresses}_{\text{Packs,Asps}} \downarrow \text{partof}_{\text{Files,Packs}}
\]
5.4.5 Discussion

This InfoPack may be hard to construct in the case of systems in which aspects are not handled explicitly. To reconstruct Aspect Assignment, one should take the decomposition level that fits such an assignment best. The partof relation can be used to bring the assignment to any requested level. For example, if it is possible to reconstruct aspect assignment at Functions level \((\text{addresses}\_\text{Funcs},\text{Asps})\), one can bring it to a Files level (using composition). In that case, the resulting \(\text{addresses}\_\text{Files},\text{Asps}\) relation is not necessarily functional.

If aspects appear only at statement level, it is practically impossible to obtain a useful \(\text{addresses}\) relation. Sometimes the notion of aspects must be relaxed somewhat to realize aspect assignment. Although this particular result does not comply precisely with the definition of aspects, it may be helpful in re-architecting a system.

5.5 ArchiSpect: Aspect Coupling

5.5.1 Context

The Aspect Coupling ArchiSpect belongs to the module view. The results of the Import (Section 4.6), PartOf (Section 4.7) and Aspect Assignment (Section 5.4) InfoPacks are part of input. This ArchiSpect is related to the Component Coupling ArchiSpect (Section 5.2).

5.5.2 Description

Consider a programmer who is working on message logging. He is not interested in all the code, but only in the parts concerning statements about logging. If we can offer the programmer a reduced logging view on the system, it will be easier for him to perform his logging task.

Aspects structure a system in addition to e.g. functional structuring. Both structuring means are more or less orthogonal, which helps to create two completely different views on the system. Aspect structuring plays the most important role during certain development activities, e.g. when dealing with message logging while functional structuring is important, e.g. when adding a new feature to a system (e.g. Follow Me into a telecommunication switching system).
Design decisions relating to aspects should also be reflected in source code. For example, an aspect can be reflected as a set of files, which means that a single file belongs to exactly one aspect.

A structuring mechanism is effective only when properly applied. This will be apparent from e.g., low degree of connectivity between the parts that result from structuring. A rule of thumb for aspect connectivity is: an aspect $A$ may only use functionality that belongs to aspect $A$. Other dependencies between aspects could be defined by an architect, but, for clarity, they should be minimal.

5.5.3 Example

Med

The Aspect Coupling of the Test aspect is presented in Figure 5.11 (created by the Teddy-PS tool). This diagram is of the same type shown for Component Coupling. In fact, it is a subset of the diagram depicted in Figure 5.4 on page 104. The dependency between aspects is represented in Figure 5.12.

5.5.4 Method

We will start with the imports multi-relation at the proper decomposition level, which corresponds to the decomposition level of the domain of the addresses relation. For each aspect we construct a diagram or table similar to the diagram shown for Component Coupling. Assume we want to construct the diagram for an Asp aspect, then we have to restrict the imports multi-relation for this aspect:

$$ FilesAsp = addresses_{Files, Asps, Asp} $$

$$ importsAsp_{Files, Files} = imports_{Files, Files} | dom FilesAsp $$

**explanation**

The $FilesAsp$ set contains all the files assigned to the $Asp$ aspect. We reduce the imports relation by looking only at the files that belong to

---

$^{3}$An arrow can occur in the aspect diagram only if it occurs in the component diagram, with the same or greater weight.
Figure 5.11: Test Aspect Coupling for Med

Figure 5.12: Dependencies between Aspects of Med
this set. This means that we have to restrict the \textit{imports} relation in its domain using \textit{FilesAsp}.

The dependencies between aspects yield an alternative abstract view on the system. Given the \textit{imports} relation, we can derive the dependency between the aspects as follows:

\[
\text{depends}_{\text{Asps,Asps}} = [\text{addresses}_{\text{Files,Asps}}] \circ \text{imports}_{\text{Files,Files}} \circ [\text{addresses}^{-1}_{\text{Files,Asps}}]
\]

\textbf{explanation}

If the \textit{addresses} relation defines a partition, we can lift the \textit{imports} relation to the level of \textit{Aspects}. But we may not assume this, so we must bring both the domain and the range of the \textit{imports} relation to the aspect level through composition.

The last step of the method consists of presenting this information. We use the same presentation techniques as used for Component Dependency, e.g., by means of the Teddy-PS tool.

\subsection*{5.5.5 Discussion}

The proper application of aspects results in a clear division of a system into slices (each belonging to a single aspect) that make virtually no use of each other. The \textit{Aspect Cohesion} and \textit{Aspect Coupling} metrics can be defined in a similar manner to the description in Section 5.3. The containment relation is defined by the \textit{addresses}_{\text{Files,Asps}} relation (although it is not necessarily a partition). Furthermore, the \textit{imports}_{\text{Files,Files}} relation represents the dependencies between files. We define the dominating ratio (\textit{DR}_{A,B}) between aspect \textit{A} and \textit{B} as follows:

\[
\text{imports}_{\text{Asps,Asps}} = [\text{addresses}_{\text{Files,Asps}}] \circ \text{imports}_{\text{Files,Files}} \circ [\text{addresses}^{-1}_{\text{Files,Asps}}]
\]

\[
\text{impAll}_{\text{Asps,Asps}} = [\text{addresses}_{\text{Files,Asps}}] \circ [\text{Files} \times \text{Files}] \circ [\text{addresses}^{-1}_{\text{Files,Asps}}]
\]
\[
DR_{A,B} = \frac{\lvert \text{addresses}^{-1}_{\text{Files,Asps}} \rvert_{\text{dom } \{A\}, \text{ran } \{B\}}}{\lvert \text{imports}_{\text{Asps,Asps}} \rvert_{\text{dom } \{A\}, \text{ran } \{B\}}}
\]

So, as with cohesion and coupling, we define aspect cohesion and aspect coupling as follows:

\[
\begin{align*}
\text{Cohesion}_A &= DR_{A,A} \\
\text{Coupling}_{A,B} &= \frac{DR_{A,B} + DR_{B,A}}{2}
\end{align*}
\]

## 5.6 Concluding Remarks

Improving a software architecture often involves a lot of questions and deducing more precise questions from the answers or defining improvements. Relation Partition Algebra offers a flexible way of asking these questions in a formal manner; the answers are obtained by executing the RPA formulas. Impact analysis (see Figure 5.1 on page 100), or what-if analysis, consists of an iterative process of defining an idea, simulating it on a software model and evaluating the results. For example, an idea could be to move a function from one file to another. The simulation of this idea consists of changing the appropriate sets, relations and multi-relations of the software model (including re-calculation of the derived relations to retain a consistent model). We have also presented a number of quality metrics (cohesion, coupling, aspect cohesion, aspect coupling) that can support the evaluation of a software model. But the intuition of architects also plays an important role [Cor89]. An ArchiSp ect like Component Coupling helps an architect shape his or her intuition.

We could use our experience to develop a dedicated tool that supports the impact analysis process (Computer-Aided Impact Analysis). An idea can be transformed into actions that must be executed during simulation (i.e., a script for modifying the software model). The tool could be designed to keep the software model consistent. To support evaluation, such a tool should be able to present various metrics and diagrams and tables as results of various ArchiSp ects.
There exist clustering algorithms that e.g. try to cluster functions into cohesive groups that are minimally coupled. In large systems, functionalities are grouped at various levels, which makes these algorithms hard to apply. In addition, factors that are hard to measure play a role with respect to deciding whether to cluster functionalities, e.g. the semantics of functions.

Accepted ideas must be implemented in the actual software. An idea, e.g. move function $f$ from file $x$ to file $y$, must be translated into a prescription that can be applied to the source code. Dedicated transformation tools (often based on compiler technology to a great extent) can automatically apply simple prescriptions to the source code. The rest must be specified in terms of change requests and must be performed manually by developers.