CHAPTER 6

DESIGN

Once the problem has been clarified to a sufficient level of detail, a move is made from the analysis to the design phase, which aims to specify the solution. There is not a strong boundary between these two phases, and while iterating on the analysis or design, one can move between the two (see Figure 2.1). Since from this point on, the proposed methodology focuses on the JADE platform (and hence, the constructs provided by it). Carrying out the design phase allows to reach a level of detail that is sufficient enough to have a relatively straightforward transition to the implementation, with the possibility of a significant amount of code being generated.

Similar to the analysis, the design phase has been carried out by following a number of logical steps, with a certain degree of overlap. The steps in the design phase are discussed in detail in Sections 6.1-6.10.

6.1 STEP 1: AGENT SPLITTING/MERGING/RENAMEING

This step involves observing the artifacts produced in the design phase and determining whether the agent types produced in the agent diagram should be split or merged. This step is considered important, since it has a direct effect on overall system efficiency and complexity. Based on this, the following rules have been applied in this step:
• Data duplication should be avoided. If there are two or more agents that share a large majority of the information required to carry out their tasks, these agents can possibly be merged into a single one.

• Duplication of code to access resources should be avoided. If there are two or more agents that need to access the same resource, these agents can possibly be merged.

• Avoid splitting agents unless there are good reasons for doing this (see below). Dealing with too many agents increases the overall system complexity and decreases system efficiency since unnecessary communication between agents will possibly take place.

• Each agent is situated on a single machine. A major factor which leads the splitting of an agent is deployment issues (based on the deployment diagram – see Section 5.6). If two pieces of functionality must be provided on different machines, these pieces of functionality must be provided by different agents.

• Avoid having agents that are too big and complex. This makes them difficult to design and to maintain.

• In some cases where the wrapper approach is used (see Section 5.2), the agent is assumed to cover and take the size of the java code it is wrapping. Thus, in such a case, it may be difficult to merge or split such an agent. Furthermore, this is another good reason to adopt the transducer approach discussed in Section 5.2.
Looking at the third and fifth rules, there may seem to be a contrast. However, what is being advocated here is a rational balance between an excessive number of simple agents and a small number of large complex agents.

In the case of the multi-agent system, there are a relatively small number of agent types identified in the analysis phase. Hence, splitting or merging of agent types is not considered to be a major issue.

6.2 STEP 2: INTERACTIONS SPECIFICATION

In this step, for each agent type, all responsibilities that are related to an acquaintance relation with another agent (based on the responsibility table produced in analysis) are taken into account and an interaction table is produced for each agent type. Each row in the table represents an interaction and will include:

- A descriptive name for the interaction.
- The responsibility (identified in the responsibility table produced in the analysis phase) that originates this interaction. This links design artifacts to analysis artifacts and can be used later to check consistency.
- A suitable interaction protocol (IP) chosen to implement the interaction. The standard FIPA interaction protocols should be considered as a candidate first (Milgrom 2001).

If none of these protocols are deemed suitable, an ad-hoc interaction protocol should be defined, as described in Step 3 of the design phase.
- The role played by the considered agent in the interaction protocol. This can be \(I\) for initiator or \(R\) for responder.
- The agent type and name (if relevant) of the complementary role.
- The trigger condition, i.e. when this interaction takes place. This condition is expressed in an informal but descriptive way.

Table 6.1 shows how the interaction table might look for the multi-agent system. Similar tables can be filled for the other agent types identified in the analysis phase.

**Table 6.1 Interaction table for multi-agent system after Step 2 of design**

<table>
<thead>
<tr>
<th>Interaction</th>
<th>Resp.</th>
<th>IP</th>
<th>Role</th>
<th>With</th>
<th>When</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invite user to enter the details</td>
<td>2</td>
<td>Contract Net</td>
<td>I</td>
<td>Generic Agent</td>
<td>User Specifies SSN and Signal Type</td>
</tr>
<tr>
<td>Transfer the data file</td>
<td>4</td>
<td>Contract Net</td>
<td>I / R</td>
<td>Specific Agent – EEG / ECG / EMG</td>
<td>An request is received for processing</td>
</tr>
<tr>
<td>Interact with Expert Systems</td>
<td>8</td>
<td>FIPA Request</td>
<td>I</td>
<td>Specific Agent – EEG / ECG / EMG</td>
<td>Acquiring the Processing Report</td>
</tr>
<tr>
<td>Insert into DB</td>
<td>9</td>
<td>FIPA Request</td>
<td>R</td>
<td>DB Agent</td>
<td>Updating Database report</td>
</tr>
<tr>
<td>Retrieve the Details</td>
<td>5</td>
<td>FIPA Response</td>
<td>I</td>
<td>Generic Agent</td>
<td>Report details displayed on the user machine</td>
</tr>
</tbody>
</table>
Similar tables can be filled for the other agent types identified in the analysis phase. It should be noted that since the focus is now on JADE, it is necessary to consider the features provided by the JADE platform. Hence, it is possible to start mapping the yellow pages agent with the ready-made JADE directory facilitator. More information on this is provided in Step 4 of the design phase (Section 6.4).

6.3 STEP 3: AD-HOC INTERACTION PROTOCOL DEFINITION

Whenever possible, existing interaction protocols defined by FIPA should be adopted www.fipa.org/repository/ips.php3. However, it is often the case that an interaction requires an ad-hoc interaction protocol to be defined (i.e. when none of the FIPA defined interaction protocols are deemed adequate). In such cases, the interaction protocol should be defined by means of a proper formalism. Two options are suggested for this:

- The interaction protocol formalism defined in AUML www.auml.org.
- Other (possibly user-defined) interaction protocols such as the FSM7-based formalism (see Further Work, Section 9), Petri-nets Cost et al (1999), or enhanced Dooley graphs Parunak (1996). Some good reviews on the modeling of agent conversation are provided in Nowostawski et al (2001) and Paurobally (2003).

The reason for including alternative interaction protocols in addition to AUML is that although AUML is suitable for simple
conversations, it is not practical for expressing complex interaction sequences.

In the case where an ad-hoc interaction protocol is required, a schema should be provided with its definition. This schema should be compliant with the AUML formalism, or, in the future, with the proposed FSM-based formalism.

6.4 **STEP 4: MESSAGE TEMPLATES**

All the interaction protocol roles identified in the previous step are implemented as JADE behaviours (see Section 6.7). In this step, suitable MessageTemplate objects are specified to be used in these behaviours to receive incoming messages, and these templates are added to the rows of the interaction table. The following rules have been applied in this step:

- Use MessageTemplates based on the conversation ID in behaviours implementing initiator roles.
- Merge in a single behaviour responder roles dealing with the same combination of initiation message performative, ontology, and language.
- Use MessageTemplates based on the above combinations of performative, ontology, and language, in all always-active behaviours implementing responder roles.
- Analyze conflicts and modify MessageTemplate used in responder behaviours.
- If conflicts cannot be solved working on performative, ontology, and languages, consider applying the dynamic template pattern described in Section 6.3.1.
It should be noted that at this stage some assumptions are made about the ontology and language used in the system when specifying the templates. This will, therefore, necessitate refinements in later steps. Based on the above rules, Table 4 is updated Table 6.2 which is partially shown below.

**Table 6.2 Interaction table for multi-agent system after Step 4 of design**

<table>
<thead>
<tr>
<th>Interaction</th>
<th>Resp.</th>
<th>IP</th>
<th>Role</th>
<th>With</th>
<th>When</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invite user to enter the details</td>
<td>2</td>
<td>Contract Net</td>
<td>I</td>
<td>Generic Agent</td>
<td>User Specifies SSN and Signal Type</td>
<td>Conv-id</td>
</tr>
<tr>
<td>Transfer the data file</td>
<td>4</td>
<td>Contract Net</td>
<td>I / R</td>
<td>Specific Agent – EEG / ECG / EMG</td>
<td>An request is received for processing</td>
<td>Perf=CFP</td>
</tr>
</tbody>
</table>

**6.4.1 The Dynamic Template Pattern**

The dynamic template pattern shown in Figure 6.1, designed for multi-agent system is based on the usage of a jade.lang.acl.ConversationList object inside the agent. All initiator behaviours register to the ConversationList in their onStart() method and deregister in their onEnd() method. The ConversationList therefore can keep track of all interactions initiated by the agent and is able to provide a MessageTemplate that matches all messages *not* belonging to any of these conversations. Responder behaviours with conflicting templates can then use the template provided by the ConversationList (properly refined with combinations of performative, ontology, and language) to avoid conflicts with all initiators.
6.5 **STEP 5: DESCRIPTION TO BE REGISTERED/SEARCHED (YELLOW PAGES)**

In this step, the naming conventions and the services registered/searched by agents in the yellow pages catalogue maintained by the JADE directory facilitator are formalized (when relevant). A *class diagram*
form is proposed to describe service registrations/searches as shown in Figure 6.2 with reference to the multi-agent system.

6.6 STEP 6: AGENT-RESOURCE INTERACTIONS

It is often the case that one or more agents in the system must interact with external resources such as databases, files storing information, or legacy software. In some cases some hardware appliances must be controlled or monitored, but this always happens through some dedicated software that actually hides the hardware behind it. Agents interacting with external resources have been identified in Step 2 of the analysis phase (Section 5.2), and are expressed in the agent diagram by an acquaintance relation with a resource element. Such resources can be classified into two main categories:

- **Passive resources**: resources that change their status only as a consequence of some stimulus issued by the agent controlling the resource itself.
- **Active resources**: resources that may change their status independently from the controlling agent.

Agent interactions with passive and active resources as defined above are discussed in Sections 6.6.1 and 6.6.2, respectively.

6.6.1 Passive Resources

Examples of passive resources are a database fully controlled by the interacting agent and a data file in the local file system. Moreover, a JADE agent is, in effect, a piece of Java code and standard Java techniques is
being used to handle these cases. For example, in the case of a database, JDBC has been used, in the case of a data file java.io has been used.

6.6.2 Active Resources

Examples of active resources are a database where a human operator (or an external program) need to insert or modify data, a log file continuously filled (updated) by an external program, an appliance that can raise alarms and software controlling a sensor detecting changes in the local environment. Active resources provide a listener-based interface so that the controlling agent can immediately detect changes inside the resource. In other cases, the resource provides an interface with methods that block until a change is detected, e.g. a network socket where some data is expected to be received. Finally, in certain cases the only way to detect relevant changes in an active resource is to periodically poll the resource itself.

Though several approaches are possible to deal with active resources, a single approach is used, which attempts to homogenize all the possible combinations of cases described above. This approach is based on the following rules:

- If no listener-based interface is available, use a dedicated Java thread, or pool of threads, to emulate it, i.e. to detect relevant changes inside the resource and act as listener notifier.
- Provide the notifier with a listener implemented so that each call from the notifier results in adding a proper Behaviour to the agent according to the Listener adding behaviours pattern depicted in Figure 6.3.
- Use a jade.util.Event object and its waitUntilProcessed() and notifyProcessed() methods to synchronize the listener and the added Behaviour when a result (produced by the behaviour) must be passed back to the notifier as the return value of a method of the listener interface.

The proposed approach is quite flexible and avoids synchronization problems between the notifier threads and the agent thread since all relevant operations are carried out by the agent thread within the added behaviours. Moreover, using different behaviours to serve the events generated by the notifier transparently deals with the case where the notifier holds several threads that may notify events concurrently.

**Figure 6.3 The listener adding behaviour pattern**

### 6.6.2.1 Querying a Relational Database

A particular case that deserves some more consideration is that of a relational database. Since querying a database is typically performed using a very flexible language such as SQL, just applying the transducer approach discussed in Step 2 of analysis phase (Section 5.2) to a database, may not be the right choice, since only the controlling agent (i.e. the transducer) can
actually exploit the power of SQL. Another agent wishing to retrieve information from the database should send a query expressed in, for example, SL (see Section 4.10 for more information on Content Languages including SL) to the transducer, that, on its turn, should translate it into SQL, get the result from the database, and send them back to the initiator. However, expressing SQL queries in SL is definitely not a trivial task and typically one ends up with mapping all possible queries that other agents may wish to perform on the database to dedicated actions, thus making the domain ontology much more complex than it should be.

The approach proposed is to adopt the transducer approach and avoid embedding JDBC code inside all agents that may need to query the database, as depicted in Figure 6.4, and is based on the Iterated version of FIPA-Request-like protocols and on the JDBC ontology that is currently under development. The former is just an extension of the normal FIPA-Request protocol, but allows one to obtain the results divided in chunks, rather than all together. The latter is an ontology that provides a single concept basically mapping the JDBC ResultSet.

![Figure 6.4 Performing SQL queries on a “transduced” database](image-url)
The proposed solution preserves the full power of SQL/JDBC without the need for embedding JDBC code (and the related complexity) inside all agents that need to perform queries on the database. Furthermore, this shows the effectiveness of the transducer approach over other approaches.

### 6.7 STEP 7: AGENT-USER INTERACTIONS

In many cases, an agent needs to interact with a user. Agents interacting with users have been identified in Step 2 of the analysis phase (Section 5.2) and are expressed in the agent diagram by an acquaintance relation with an actor element. There are several ways a human user can interact with a piece of software such as a JADE agent. Here, the focus is on the graphical user interface (GUI), which is by far the most commonly used type of user interface. Two cases are distinguished:

- A local GUI, typically implemented using Swing, the Abstract Windowing Toolkit (AWT) or some other graphical toolkit.
- A web GUI implemented using Java Server Pages (JSP) technology.

As it sufficient to use the local GUI and it is discussed in more detail in the following section.

#### 6.7.1 Local GUI

Here, the main issue is that the agent and the GUI must typically work on the same data (the agent to perform its domain specific tasks, and the GUI to present the data to the user), but must organize this data in different
ways. Using a toolkit such as Swing, which implements the Model-View-Controller architecture, it is possible to overcome the problems by applying the following rules:

- Store data inside the agent in structures that are suitable to accomplish agent tasks.
- Make these structures fully synchronized.
- Make these structures implement the proper swing model interfaces and link them to the related graphic classes inside the GUI module.
- Define proper renderer classes when necessary.

Keeping the rules in mind, two local GUIs are developed namely, GenericGUI for getting the patient’s info and FileGui for getting the Data file name.

6.8  **STEP 8: INTERNAL AGENT BEHAVIOURS**

The actual job an agent has to do is typically carried out within the agent’s “behaviour(s)”. Hence, in this step, the agent responsibilities are looked up (via the responsibility table) identified in the analysis phase and mapped them to agent behaviours.

First of all, the following rule has been applied:

For a responsibility related to an interaction in the interaction table described in Step 2 of the design phase (Section 6.2), obtain the JADE class implementing the interaction protocol and role selected for that interaction and provide a suitable extension.
In the Multi-agent system, for instance, the “serve requests to initiate invitations from the system user” responsibility corresponds to an interaction in the interaction table where the GenericAgent plays the initiator role in a Contract-Net protocol. Therefore the behaviour implementing that responsibility should be a proper subclass of the jade.proto.ContractNetInitiator class.

Other responsibilities must be implemented using completely application-specific behaviours and therefore it is quite difficult to provide a formal guidance for this process. The suggestion, in any case, is not to extend the ade.core.Behaviour class directly, but to start from the JADE classes that implement the skeletons for commonly required types of task. These classes include:

- **OneShotBehaviour**: implementing an atomic task that runs once and terminates immediately.
- **CyclicBehaviour**: implementing a task that is always active, and performs the same operations each time it is scheduled.
- **TickerBehaviour**: implementing a task that periodically executes the same operations.
- **WakerBehaviour**: implementing an atomic task that runs once after a certain amount of time, and then terminates.

When dealing with complex responsibilities, it is suggested to attempt splitting them into a number of simpler tasks combined together and adopt one of the composite behaviour classes provide by JADE. These composite behaviour classes include:
• SequentialBehaviour: implementing a composite task that schedules its sub-tasks sequentially.

• FSMBehaviour: implementing a composite task that schedules its sub-tasks according to a Finite State Machine.

Composite behaviour can be nested and therefore there can be, for instance, a subtask of a SequentialBehaviour that is on its turn a FSMBehaviour and so on. In particular, all complex responsibilities that can be modeled as Finite State Machines can be effectively implemented as FSMBehaviour instances.

To demonstrate an example of a complex responsibility, take for instance the “let the Generic agent enters the SSN, Signal Type and Data file name” responsibility (see Table 4.1). This responsibility can be modeled as a Finite State Machine, and is shown by the State Transition Diagram in Figure 6.5 (note that this is a hypothetical example to demonstrate the point and doesn’t necessarily represent efficient operation).

The State Transition Diagram in Figure 6.5 demonstrates that for the responsibilities defined in the analysis phase, there may be many “sub-responsibilities” (unanticipated responsibilities arising from the main responsibilities in the responsibility table) when mapping to agent behaviours (leading to an update of the responsibility table). Moreover, during the process of defining agent behaviours, an interaction may arise that was unanticipated in the earlier stages of the design phase. For example, the “update agents list” state (sub-responsibility) shown in Figure 6.5, requires an interaction with the other specific agents in order to update the agents list. This will lead to the update of the interaction table and subsequent definition of a behaviour that is a proper subclass of the jade.proto.ContractNetInitiator class (in this case the Generic agent will play the Initiator role in a Query protocol).
6.9 STEP 9: DEFINING AN ONTOLOGY

When agents in the system interact, they exchange information that refers to entities, abstract or concrete, that exist in the environment agents reside in. These entities may be primitive, such as a String or a number, or may have complex structures defined by templates specified in terms of a name and a set of slots whose values must be of a given type. These complex entity templates are referred to as Concepts. In the multi-agent system, there are concepts such as those shown in Figure 6.6.
Moreover, entities are typically related by means of relations that can be either true or false. Similar to complex entities, relations also have structures defined by templates and again, these templates are specified in terms of a name and a set of slots whose values must be of a given type. These relation templates are referred to as *Predicates* and, considering in the multi-agent system, predicates such as those shown in Figure 6.7 are allocated.

Finally, a particular kind of complex entity is represented by descriptors of actions that agents can perform. The templates of these action descriptors are referred to as *AgentActions*. Figure 6.8 shows an agent action (to be performed by the generic agent) relevant to the multi-agent system.
Figure 6.8 A possible agent action performed by generic agent

Actions, when executed, may produce an effect and/or generate a result to be sent back to the requester. An ontology is a set of concepts, predicates and agent actions referring to a given domain. Some more details on expressing ontologies are provided in the following section.

6.9.1 Formalisms for Expressing Ontologies

Different formalisms can be adopted for expressing an ontology. In the proposed methodology, a graphical formalism is advocated, based on UML class diagrams.

The following points should be noted:

- Each ontological template is expressed as a class.
- The stereotype is used to differentiate between concepts, predicates and agent actions.
- A slot of an ontological template whose type is primitive is expressed as an attribute of the corresponding class.
- A slot of an ontological template whose type is itself a concept in the ontology is expressed as a role of an association linking the ontological element that owns the slot with the concept representing the type of the slot.
• Effects and results produced by the execution of an action are documented as comments attached to the agent action.

• The inheritance relation is used as usual to indicate that an ontological template is a specialization of another ontological template.

6.9.2 Heuristic rules

Defining an ontology is typically not an easy task since the same domain can typically be described by means of several different sets of concepts, predicates, and agent actions, i.e. by means of several different ontologies. Generally, there are a number of choices that must be made. The most important ones are indicated in the following three sections (Sections 6.9.2.1-6.9.2.3). It should be noted that the details that motivate the outcome of each choice become clearer as the development process progresses. For this reason, the ontology is typically refined while iterating through the steps in the design phase.

6.9.2.1 Ontology Boundaries

An ontology is essentially a model of the application domain addressed by the system. Moreover, it is not always trivial to decide which types of entities and relations to model inside the ontology. On the one hand, it is desirable to keep the ontology as simple as possible, while on the other hand, it must be ensured that the ontology is complete enough to allow agents to perform their jobs. The guideline provided to drive this choice is the following:
Include in the ontology only concepts and predicates that agents need to talk about, i.e. whose instances must be encoded inside the content of ACL10 messages exchanged by two or more agents in the system.

6.9.2.2 Predicates versus Concept Slots

Each time entities of a complex type X are related to entities of type Y, this can be expressed either by adding a slot of type Y in concept X or by adding a predicate that relates X and Y. For example, the fact that a cinema has a given address, can be expressed either by a slot of type Address in the Cinema concept (as in Figure 6.7) or by a predicate IsIn that relates a cinema and an address (as in Figure 6.8).

To guide this choice, the following heuristic rule is provided:

When, given an entity of type X, the related entity of type Y is fixed and will never change, use a slot. Conversely, when the latter can change during the lifetime of the system that is being developed, use a predicate.

In the example above, applying the rule would lead to the first choice, since it can be confidently assumed that the address of a given cinema will never change.

6.9.2.3 Information Retrieval

It is often the case that an agent in the system must retrieve some information from another agent. Using the ACL language, as JADE agents do, this may be achieved through either a QUERY_REF message including a proper Identifying Referential Expression (e.g. (iota ?x (p ?x))) as content, or
a REQUEST message specifying an action whose result is the information that must be retrieved. This choice is clearly related to the ontology. As a heuristic rule, the following is provided:

- If the agent providing the information to be retrieved includes a knowledge base that is able to handle Identifying Referential Expression directly, define predicates in the ontology. Otherwise, define actions.

### 6.9.3 Tools for Defining an Ontology

JADE provides a sophisticated mechanism, described in Caire et al (2004), to handle ontological elements as instances of Java classes that are basically beans with proper get and set methods for all the slots in the template and to automatically convert them back and forth strings to be used as message contents. When moving from the design to the implementation, the creation of these ontological Java classes is very straightforward, but, especially when dealing with large ontologies with a lot of templates, it may be quite time consuming. Thanks to a proper plug-in, (called beangenerator) implemented by C.J. Van Apart from the Department of Social Science Informatics at the University of Amsterdam http://acklin.nl/beangenerator/, it is possible to define the ontology using Protégé and then let the beangenerator automatically create the ontology definition class and the predicates, agent actions, and concepts classes. Use of this approach is particularly convenient when:

- There are several templates in the ontology.
- The ontological classes do not require any other method than the get and set methods corresponding to the ontological template slots.
Furthermore, if other methods or fields are added annually after the automatic generation, and if the ontology must subsequently be modified and therefore re-generated, all manual modifications are not preserved by the bean generator.

6.10 STEP 10: CONTENT LANGUAGE SELECTION

JADE provides codecs for two content languages: the SL language and the LEAP language (through the jade.content package).

The SL language is a human-readable string-encoded content language, while LEAP is a non-human readable byte-encoded content language. Based on this, some heuristics on choosing the appropriate content language are as follows:

- SL is suitable for agent based applications that are (or can become) open (i.e. where agents from different developers, running on different platforms must communicate).
- The LEAPCodec class is lighter than the SLCodec class. Thus, when there are strong memory limitations the LEAP language is preferable. For example, for applications which rely on portable devices such as mobile phones, the LEAP language is much more preferable. Conversely, for applications where high capacity computers are involved, the SL language is more suitable.
- When it is required that a content language be used that should be readable by humans, SL should be chosen; otherwise, if there are no such requirements, it is advisable to use LEAP.
In the multi-agent system, since agents from different developers, running on different platforms communicate with each other, the SL language is undoubtedly the best option.

6.11 DESIGN SUMMARY

Once the analysis has been carried out, a move is made to the design phase, which aims to specify the solution. The solution focuses on the JADE platform. It is possible to move back and forth between the analysis and design whenever necessary. The steps in the design have been summarized below:

- **Step 1: Agent Splitting/Merging/Renaming.** By considering system performance and complexity in relation to the agent deployment diagram produced in analysis, it is determined whether agents should be split, merged or left as is.

- **Step 2: Interaction Specification.** All responsibilities in the responsibility table related to an acquaintance relation with another agent are considered, and the interaction table produced for each agent type.

- **Step 3: Ad-Hoc Interaction Protocol Definition.** In the case that an existing interaction protocol can not be used for an interaction, an ad-hoc interaction protocol is defined using a suitable formalism.

- **Step 4: Message Templates.** The interaction table is updated to specify suitable MessageTemplate objects in behaviours to receive incoming messages.

- **Step 5: Description to be Registered/Searched (Yellow**
Pages). The naming conventions and the services registered/searched by agents in the yellow pages catalogue maintained by the JADE directory facilitator are formalized. A class diagram form is used as a representation.

- **Step 6: Agent-Resource Interactions.** Based on the agent diagram produced in analysis, passive and active resources in the system are identified, and it is determined how agents will interact with these resources.

- **Step 7: Agent-User Interactions.** Based on the agent diagram produced in analysis, agent-user interactions are identified and detailed.

- **Step 8: Internal Agent Behaviours.** Based on the responsibility table produced in analysis, the agent responsibilities are mapped to agent behaviours. Different types of responsibilities (including interactions) require different types of agent behaviours have been specified.

- **Step 9: Defining an Ontology.** An appropriate ontology for the domain is specified, by making a number of considerations.

- **Step 10: Content Language Selection.** By following some rules, a suitable content language is selected.

- **Iterate Steps 1-10.** Move back and forth between analysis and design whenever necessary.

The artifacts produced in each step of the design and their relationships to those produced in the analysis are summarized in Figure 6.9.
Figure 6.9 Summary of the design phase