Pattern-driven Partitioning in Designing Distributed Object Applications

Widayashanti P. Sardjono Dr. Anthony J. H. Simons

Department of Computer Science, University of Sheffield
211 Portobello Street, Sheffield S1 4DP, United Kingdom
{W.Sardjono|A.Simons}@dcs.shef.ac.uk

Abstract

One of the fundamental challenges in designing distributed object applications (DOAs) is application partitioning. Partitioning is a technique through which all the necessary partitions or components of the application, particularly distributable ones, are discovered. This research proposes a technique to help designers in partitioning DOAs such that a justifiable application structure is achieved. The technique combines the decomposition approach from the software architecture field with the techniques to discover subsystems found in some object-oriented methods. The technique is represented as Partitioning Pattern Language and Partitioning Process Patterns. The technique is developed within the context of an existing object-oriented method, called Discovery [Sim], which serves as the framework for object-oriented modelling.

1 Introduction

Previous work on application partitioning proposed different and independent approaches. One approach is the architecture-based approach, which assumes an a priori imposition of a particular structure, in the form of an architectural style. e.g. [WF98]. This top-down decomposition approach is a specialisation of architectural design methods, e.g. [Bos00]. The approach emphasises forming fixed high-level application architectures, leaving out the detailed design of its components. Unfortunately, establishing the application structure at such an early stage of development prevents the developers from reworking the design later. This is necessary to discover the optimum partitioning according to the internal needs of the system.

Another approach is the technique to discover subsystems found in typical object-oriented methods. This bottom up approach shows that the formation of the fixed structure of subsystems can be delayed until all the main classes and their collaborations have been discovered. This allows the designer to make informed partitioning decisions based on the distribution requirements [Eel00, LR00]. These approaches disregard the fact that some presumptive structures may exist as a consequence of addressing the concerns raised early by the application stakeholders. In this case, applying patterns and styles would yield a sound application design more effectively than pure object-oriented methods.

This research addresses the challenge of bridging the gap between top-down decomposition and bottom-up subsytem discovery and taking the benefits of both approaches. The proposed partitioning process patterns would enable designers to utilise architectural and design patterns or styles and yet allow them to optimise the resulting designs to obtain better distributed object application architecture based on the internal forces of the system requirements. The partitioning pattern language would allow designers to apply patterns or styles at the right stage. Similar works on hybrid approach [CHHKC01, Sto99] make initial system hierarchy based on functional partitioning and use object-orientation in refining the design. However, unlike this research, both approaches have not taken patterns and architectural issues into account and do not allow further reconfiguration of the initial structure based on the feedback from later design steps.

The research is carried out by investigating different practices in designing DOAs from the software architecture and object-oriented design fields. Similarities between these practices and Discovery on the levels of abstraction, deliverables and process are explored in order to classify each approach and fit it into a common framework. This framework consists of the pattern language for the design artefacts and the process patterns for applying the design process. The pattern languages are applied to typical examples of distributed applications to show the fitness of the languages for resolving partitioning problems.

This paper presents the process patterns related to the top-down partitioning. The overview of the bottom up partitioning process patterns and the partitioning pattern language are given in the appendices for further reading.

2 Motivational Example: FoodMovers - A Food Distributor System

An e-commerce case study is a classic example, which represents thousands of internet applications today. Treatments for variants of this case are given independently by Gomaa [Gom00] and Wijegunaratne and Fernandez [WF98]. The FoodMovers case study [Tra03] exemplifies supply chain management applications, which is a kind of e-commerce application. An adaptation of the FoodMovers example is given below.

Problem Description FoodMovers is a distributor of food products for small, like corner stores, and medium-size grocery retailers. It buys products from food manufacturers (suppliers), and sells them to its retailer customers (stores). Between buying and selling, FoodMovers stores food items in their warehouses. FoodMovers has several warehouses, from which the actual food products are distributed. All but one of these warehouses are located in different sites (cities) than the main office. Each of these warehouse is responsible in maintaining its own inventory level. The main FoodMovers office handles the orders to suppliers, updates food information from the suppliers, and handles the orders from the customers.

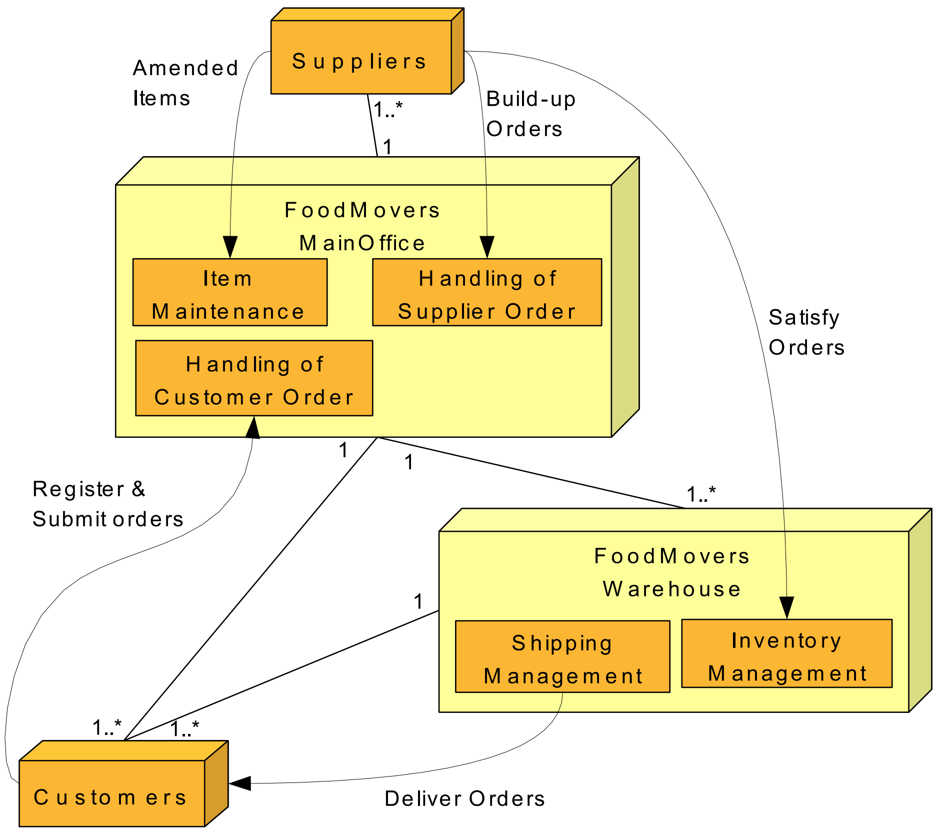

Figure 1: An illustration of the FoodMovers example

Clearly, FoodMovers must face the challenge of dealing with all sizes of customers, from huge multinational corporations to small stores, as quickly and efficiently as possible. This is because the profit margins are slim and the some of the products are perishable.

For simplicity purposes, the invoicing and payment processes are not included in this case study. The business processes included in this case study, as depicted in figure 1, are:

- Item Maintenance

  The purpose of the Item Maintenance business process is to record, update, or delete the UPC (Universal Product Code) and all related information regarding a food item. Such information are supplied by the manufacturer. Each manufacturer is responsible for maintaining its block of UPC numbers. Therefore, it is the responsibility of the manufacturer to inform FoodMovers about any new item it produces, or any item that has been discontinued or changed. The manufacturer can supply this information directly to the FoodMovers system because it provides interface to be used by the manufacturer to enter such information. FoodMovers assigns its own price for each item to be referred to by its customers.

- Handling of Supplier Orders

  FoodMovers has to maintain its level of inventory to ensure that there is enough of each product in the warehouses so it will be available when the stores make their orders. The inventory level needs to be optimal. FoodMovers can have a contract with its supplier that allows the supplier initiate a supplier order when the FoodMovers' stock gets low. FoodMovers provide an interface that exposes the inventory level information to its trusted supplier partner bound by the contract. Using the interfaces provided by FoodMovers, the supplier can build up order based on the queried inventory level information.

- Inventory Management

  When a warehouse received food items from a supplier, each of these items are recorded as arrived items. Further manual checks to the items include making sure that the number ordered are satisfied, and no defect to these items. If there is any discrepancy, an exception is raised and the supplier is informed. When all checks are clear, the status of these items are changed from pending supplier order to the live inventory. At this point, the item is considered to be in active inventory, awaiting shipment to stores.

- Handling of Customer Orders

  FoodMovers registers all of its customers and provides a facility for them to securely enter their orders. If the credit check shows that the customer is worthy, then FoodMovers checks each item of the customer's order against the inventory level. If the item is in stock, then it is included in the shipping instruction, ready to be delivered as requested. Obviously, the handling of customer orders need to take into account the location of the customers, so that the order can be satisfied from the nearest warehouse.

- Shipping Management

  Every morning, each warehouse checks the shipping instruction. Items in the shipping instruction are collected from the shelves and allocated to delivery vehicles. For each of these items, the warehouse marked the item as shipped and decrease the inventory level. For each shipping to the customer's site, the warehouse generates a shipping manifest to be used as a based for invoicing.

3 Partitioning Process Patterns

Our literature study shows that there are two approaches to carry out the partitioning process. The first is the top-down system decomposition, which is employed at the architectural level. The second one is the bottom-up partitioning, which considers the detailed distribution aspects. Both approaches justify the design of the application partitions, but a smooth progression from one approach to another is needed so that a clear transformation of the system design could be obtained.

This research proposes a new arrangement of partitioning process. The new way combines strategies from both approaches and is informed by the existing patterns. It is the purpose of the research to allow integration of this partitioning process into an established object-oriented method. The selected object-oriented method to be used as a workbench is Discovery [Sim], which has already embraced the relevant issues for the purpose of this research.

The partitioning process is in fact a generic approach which provides solutions to recurring problems of partitioning. Therefore, the process is expressed as patterns. Figure 2 shows how these patterns intermingle with some of the existing Discovery activities and artifacts. Section 3.1below gives an overview of those activities and artifacts.

3.1 Discovery: The Object Modelling Framework

As the first step in eliciting the system requirements, the original Discovery method suggests the *non-directive interviewing* techniques in exploring the clients' needs, which are recorded in the interview ***notes*** and the task ***sketches***. The interview notes and task sketches are used in the *task analysis* activity, which identifies the ***task model*** comprising both the ***task flows*** and the ***task structures***. A task in the task model represents a coarse-grained business activity, which has a goal or a purpose of its own. It may decompose into subtasks, which at the lowest level represents a complete interaction with the systems (also known as use cases in UML terms). The task flows denote the ordering of the tasks involved in fulfilling

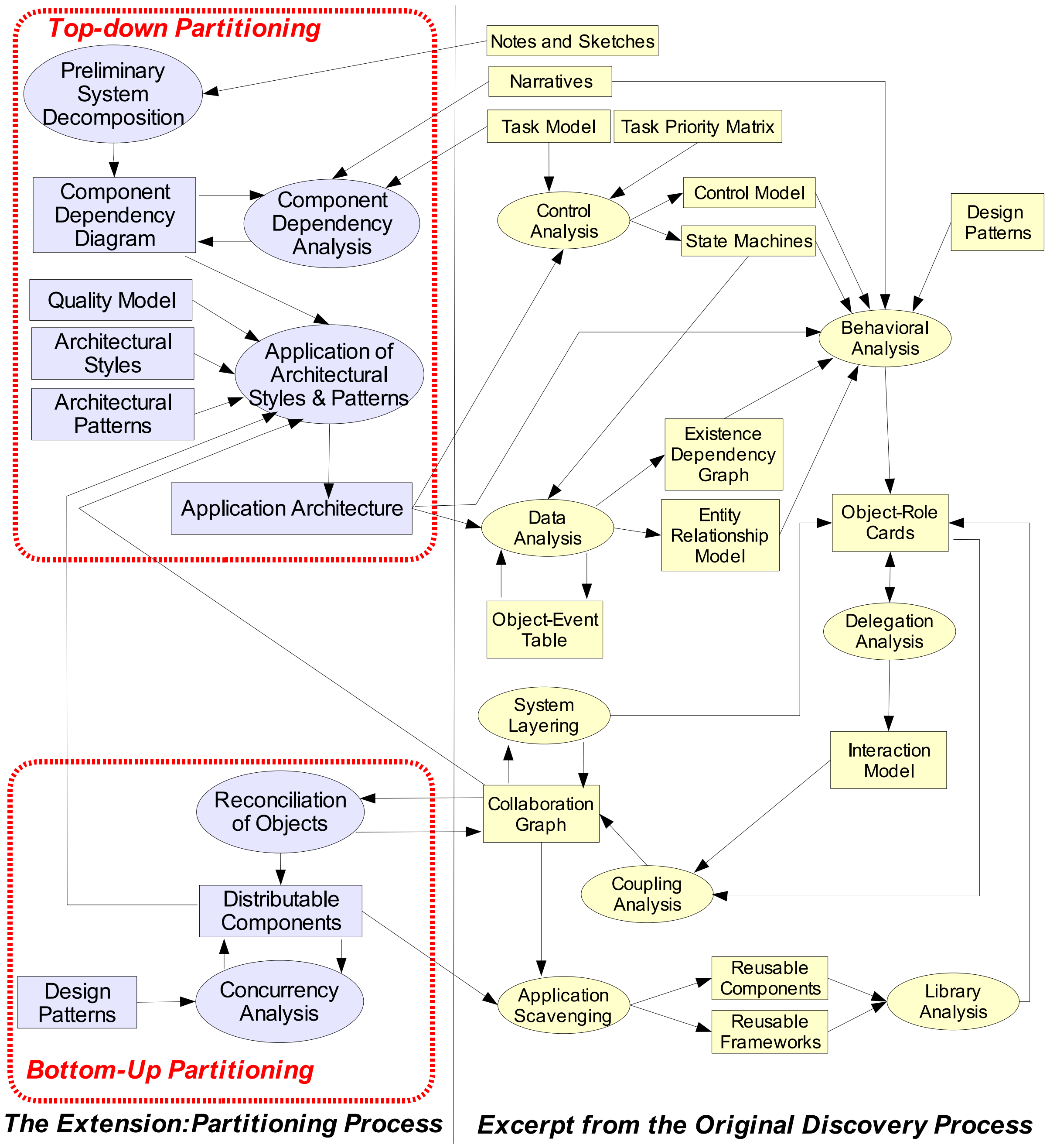

Figure 2: Integration of the Process Patterns and the Partitioning Pattern Language into Discovery

certain system functionalities. The task structures represent the structural relationship between tasks. The task model is then used as the inputs in the *task specification* activity to formalise the tasks by producing ***narratives***. A narrative is a structured form for encoding the textual information pertaining to a task. It records the purpose, actors, salient objects involved, preconditions, descriptions, postconditions, and exceptions of the task.

The complete set of task models and narratives constitutes the specification of the system, on which the construction of the initial design model is based (see Figure 2). In the *control analysis* activity, the task flows of the task models previously defined are inverted into ***state machines*** such that all of the tasks in the tasks flows become the transitions in the state machines, whereas the transitions in the tasks flows become the states in the state machines. The purpose of the state machines is to capture the control and contingency structures embedded in the system. By specifying the state machines, events generated in the systems are elicited. In the *data analysis* activity, entity-relationship modelling can be performed to model objects when the data structure is a strong requirement of the system. Such object model is expressed by the ***entity-relationship model***. Alternatively, the events elicited before are fed into the *data analysis* activity, in which the ***object/event table*** and the ***existence dependency graph*** are constructed. The purpose of this event-driven design [SSH99] is to capture the participation of these objects in an event of the system and to specify the existence (or lifetime) dependency between these objects.

Further refinement of the system specification is carried out in the *behavioural analysis* activity to elicit the responsible objects, who carry out identified actions in the narratives. The data dependency graph contributes stabilised data-centric objects as an initial set of object candidates. The state machines provide a set of control-centric object candidates that encode business process. In this activity, an object is seen, not only as a data carrier as in the data analysis activity, but also as an agent with a purpose. Thus, the object candidates are further refined such that their existence is judged by their responsibilities in achieving the system functionality. These responsibilities of objects are recorded in the ***object role cards***.

The *delegation analysis* activity breaks down over-burdened objects by forcing the delegation of functionality to ensure its balance. Too many responsibilities owned by an object indicates that an object is over-burdened and needs to be decomposed. Eventually, when all the data and the system (application) logic have been captured and managed by the identified objects, the ***interaction model*** is built. The interaction model shows the dynamic viewpoint of the system, which is a complement to the object role cards that show the static viewpoint of the system.

In the *coupling analysis* activity, the ***collaboration graph*** is produced based on the object role cards and the interaction models that have been built in the previous phase. The colaboration graph shows the total client-server connections created between two classes. The collaboration graph is then used by the *system layering* activity to rearrange the system configuration by introducing a subsystem hierarchy in order to achieve minimum coupling. The class reorganisation in the system layering activity may produce new abstractions whose object roles cards should also be written or updated. In turn, the new or updated object roles cards are fed back to the delegation analysis activity to be further reworked. This iteration of the delegation analysis activity may update the object roles cards and the interaction model. This may indicate the need of another iteration of the coupling analysis and the system layering activities.

The reusability of the system solution is explicitly enforced in this phase through both the *application scavenging* and the *library review* activities. The aim of the library review activity is to maximise reuse of existing designs, whereas the application scavenging activity aims to harvest new ***reusable components*** and ***reusable frameworks*** from the current design.

3.2 Principles of Top-Down Partitioning

The adopted top-down approach in partitioning considers these principles simultaneously:

- *aiming for high cohesion among the system elements within a single partition*

  By the first principle, the semantic relevance between one element and another is intuitively identified, hence the initial grouping is crystalised. This grouping is driven by the concerns raised early by system stakeholders and should cluster certain types of system functionality by theme, which allows the developer to focus their attention on one specific aspect of the system at a time.

- *aiming for minimal software dependencies between the system partitions*

  The second principle provides the solid ground for obtaining minimal coupling between application components at the later stages, when details of the implementation have crystalised.

- *establishing the application architecture*

The third principle allows designers to think about the priority of requirements and reusable artifacts, which leads to establishing the initial application architecture.

The top-down approach is represented by three patterns: Preliminary System Decomposition, Software Dependency Analysis, Application of Architectural Styles and Patterns.

3.3 Top Down Partitioning Patterns

3.3.1 Process Pattern 1: Preliminary System Decomposition

Problem How do you start to partition a system?

Context The system requirements have been relatively explored. In the context of Discovery this is shown by having the relatively stable task model and narratives. In the context of other object-oriented methods, such requirements are usually represented as use cases. In this case, the use cases and their scenarios are relatively established. However, for any non-trivial system there will be a large body of requirements that must be addressed by the design process. This implies that there needs to be some form of simplification or decomposition of the system to make such a design process manageable. Furthermore, the system decomposition should be carried out in such a way that the concerns of the system stakeholders and the constraints from the system environment are addressed.

Forces

- Customers or the system owners might impose their own preferred structure, regardless of the actual needs of the system.
- Some systems might have some constraints regarding physical settings, technical heterogeneity, or social related issues.
- Based on their previous experience, developers of the system might apply abstraction to simplify or encapsulate the complexity of the implementation.

Solution Determine whether, and how, all of the above forces affect the system. Create initial groupings/partitions by considering in turn each of the relevant design forces. The suggested alternative partitions emerging from this process may be total, or partial. They may overlap, or may be coincident. Merge coincident partitions, but keep all alternative partitions for resolution later (see Process Pattern 2: Software Dependency Analysis).

It is a good practice to document the discovered partitions in such a way that the intention and the scope of each partition is recorded. Such notes will be useful when they are analysed to obtain a more sensible and formal structure of the system components.

Implementation The client or the system owner may impose preferential groupings of the application. However, this groupings should be further analysed to ensure that other constraints are satisfied. This may result in either the adoption of the preferred groupings by the client or further negotiation with the client regarding the groupings.

> *Example*
>
> In the FoodMovers example above, the client might want to have the Price Assignment subsystem to be delivered in different packages from the Item Maintenance subsystem. Technically, there is no reason why these two functions should be separated into two packages (partitions) because they are so closely related. However, the client might want to have them separate because of the delivery schedule, budget, the company's internal task assignments, or any other non technical reason.

The typical constraints which may also drive the initial grouping decision include the physical settings of the system, the technical heterogeneity involved in the system and the relevant social issues:

- Physical settings

  The configuration of the physical sites in which the application would reside usually implies several parts of the application. The distinction between parts exists because there are more than one type of sites and each type of sites plays different role than the others. Therefore the initial groupings are determined based on these types of sites.

In case where there is only one type of sites, further analysis is needed to determine whether these sites play the same role or they have two or more different roles. In the former case, preliminary system decomposition cannot be achieved from the physical settings viewpoint and should take alternative viewpoints. In the latter case, although the type of sites are the same, since they play different roles from each other then the initial groupings are determined based on these roles.

> *Example*
>
> From the FoodMovers example above, it is clear that the system consists of four different types of physical site. Two types of site are within the FoodMovers' authority. They are the *main office* and the *warehouses*. All these warehouses play the same roles and have the same set of responsibilities, even though they reside in different cities than the main office.
>
> The other two types of site are collaborative institutions, over which FoodMovers has no direct authority. They are the *suppliers* (or manufacturers) and the *customers* (or stores). Just like the warehouses, in the context of the system, there are several instances of suppliers and customers, and they play the same role. Although each of them is an autonomous institution, they would either access a FoodMovers subsystem to place orders, or use FoodMovers external interfaces to communicate to the FoodMovers system. Therefore, the FoodMovers should provide access or external interfaces for them to use as a part of its application. Based on this early knowledge, the preliminary partitions of this system would be: FOODMOVERS, WAREHOUSE, SUPPLIER, AND CUSTOMER.

- Technical heterogeneity

Technical heterogeneity may come in different aspects: hardware, software, and data. The hardware heterogeneity exists when the computing hardware are of different manufacturers. In most cases, this implies different computing architectures exists, hence different implementation strategy would be adopted. Therefore, the initial groupings are determined based on types of hardware to be used, each of which has different implication on the application implementation.

> *Example*
>
> FoodMovers wants to ensure the correct record of inventory. This includes checking out the goods that are collected from the shelf in the warehouse using the hand-held computing technology. The applications to be run in the hand-held computers must be programmed using a specific API which is different from the main application programming language used by FoodMovers. This leads to a GOODSCOLLECTORS partition of the FoodMovers.

The software heterogeneity may exist at different levels of applications. Different operating system or platform implies yet another different implementation strategy to be adopted. As with the hardware heterogeneity, the initial groupings are determined based on the types of operating system or platform deployed. Different strategy to handle software heterogeneity includes the encapsulation of existing or legacy software to be connected to the distributed application, which are deployed within the system.

> *Example*
>
> If the FoodMovers application is to be made complete, there should also be the accounting subsystem. Since there are a lot of accounting applications available in the market, FoodMovers may decide to use one of those off-the-shelf packages. To enable the FoodMovers application to communicate with this proprietary accounting software, FoodMovers needs to create a partition designated to translate data from the FoodMovers internal format to the format of this software. (Since the accounting subsystems of FoodMovers would not be exercised, no particular partition is defined for this).

The heterogeneity in data structures also implies the creation of different parts of the application to handle them. The variation of these data structures stems from the use of different hardware architectures or legacy software.

> *Example*
>
> Some of the Foodmovers' suppliers are long-established manufacturers who are used to trade electronically using the legacy EDI (Electronic Data Interchange) systems. Since the implementation of EDI takes a lot of investments, often these suppliers want to keep using it. This needs a special translation of

EDI messages from the structure defined according to EDI standard languages, for example: ANSI X12, HL7 (Health Level 7), or EDIFACT, to commands that the FoodMovers application understands. Therefore, FoodMovers needs a partition to provide EDI access and translation to be used by these suppliers, which is named, for example, SUPPLIEREDIACCESS.

- Social issues

  Social related issues such as laws, regulations, or organisational boundaries, can also influence the system decomposition.

  Typically, when a system should abide newly introduced laws or regulations, one or more dedicated subsystems are identified to handle necessary functions, which ensure that the laws or regulations are followed accordingly.

  Example
  As FoodMovers is a general food distributor, it may distribute food that has particular temperature requirements or other regulated conditions during transport. These requirements and constraints should be taken into account when allocating the food item into the delivery vehicles. One customer order may be satisfied by more than one batch of shipping, depending on the transport requirements of the items being ordered. Therefore, assigning delivery vehicles may be a separate partition: VEHICLEALLOCATION. This partition should be designed to ensure that each food item ordered by customer is assigned to the appropriate vehicle so that all orders can be delivered as efficient as possible according to the law.

  The organisational structure or roles, in which the client and the user of the application reside, may also impose initial groupings. The authority of the users and the owner of the systems dictates the their access rights, which in turn determine the partitioning.

  Example
  There are at least two kinds of staff roles played in the warehouses, in the FoodMovers example. First role is played by those who collects items from the shelves and load them into the delivery vehicles. The other role is played by those who authorise the shipping itself by checking the delivery vehicles, despatch them and actually decrease the inventory level. This leads to two preliminary partitions: one to handle the collection of items and their allocation to particular delivery vehicle according to the shipping order (GOODSCOLLECTORS), one to handle the actual updates to the inventory level (SHIPPINGAUTHORIZATION).

- Functionality decomposition

  Grouping based on the systems functionality is always useful. In particular cases where constraints concerning physical settings, technical heterogeneity and social issues are not evident at such an early stage, the decisions on initial groupings can only be made according to the functionality of the problem. The groupings are determined based on the closeness of the basic concepts found in the problem. These concepts are inherent within the functions of the applications. Therefore, by classifying together several functions which are similar in nature, the initial groupings of the application is achieved.

  Example
  Early analysis on the FoodMovers case study reveals that the FoodMovers' application may be decomposed into five partitions according to the business process. They are: ITEMMGT, SUPPLIERORDER, INVENTORYMGT, CUSTOMERORDER, and SHIPPINGMGT. This grouping makes sense as each of them clearly addresses specific aspect of the system.

Resulting Context The groups discovered so far become the initial partitions of the application, which have taken into account the easily perceivable system constraints. At this point, only the partitions of the systems that are discovered, while the relationship between them have yet to be established.

For the FoodMovers application example, the compiled results are:

- MainOffice (based on the analysis of the physical settings)
- Warehouses (based on the analysis of the physical settings)

• Suppliers	(based on the analysis of the physical settings)
• Customers	(based on the analysis of the physical settings)
• SupplierOrder	(based on the analysis of the system functionalities)
• CustomerOrder	(based on the analysis of the system functionalities)
• InventoryMgt	(based on the analysis of the system functionalities)
• ItemMgt	(based on the analysis of the system functionalities)
• ShippingMgt	(based on the analysis of the system functionalities)
• SupplierEDIAccess	(based on the analysis of the heterogeneity)
• SupplierWebServices	(based on the analysis of the heterogeneity)
• CustomerWebAccess	(based on the analysis of the heterogeneity)
• VehicleAllocation	(driven by social issues)
• GoodsCollectors	(driven by the analysis of heterogeneity and social issues)
• ShippingAuthorization	(driven by social issues)

Rationale This is a pattern that represent an activity to intuitively capture the obvious grouping of elements of the problem. It is natural to acknowledge the first intuition about conceptual grouping. Even when clients are indifferent to the structure of the application, decomposing the system is a natural divide-and-conquer way of handling vast amount of information of the system. This initial decomposition is far from ideal, but at least, designers would have an initial model to work on, which has already taken notable constraints into account. Some of these constraints suggest a strong groupings which cannot be altered anyway. For example, it is very unlikely that the physical settings on which the application is deployed would alter later. Other constraints, such as organisational boundaries, still leave some room for modification. This makes the resultant design of this pattern quite solid as a basis for further design process, and yet flexible enough for further refinement.

Related Patterns This pattern is similar to the *Distributed Requirements Pattern*, proposed by Silva et. al [SHM+96] to the extent that they both are techniques to explore the requirements for partitioning. Silva et. al. suggest that the requirements should be expressed in terms of problem space and no solution should be suggested at this stage. However, their pattern seems to allow exploration of the problem more intensely than this pattern. Not only explicit partitioning requirements, for example the physical settings requirements, that are captured, but also implicit partitioning requirements such as replications of Account in a banking system as a result of different grade of availibility of an account. In contrast, this pattern does not yet address implicit requirements because such an activity is deemed to be a different step of the partitioning process which would be reinforced later when more information are revealed.

Next The Software Dependency Analysis pattern.

3.3.2 Process Pattern 2: Software Dependency Analysis

Problem How do you arrange system partitions, such that they enable the end result design to have minimal coupling between partitions?

Context The observable partitions of the system has been identified. These partitions are originated from the intuitive partitioning and shaped by both the implicit as well as the imposed constraints. However, the relationships and the dependencies between these partitions and the mapping between these partitions with the systems tasks (Task Model of Discovery or use cases in other object-orientation methods) have yet to be established.

Forces

- The information acquired at this stage is not detailed enough to define a full specification partitions and the connections between them.
- As the partitions are discovered by looking at the problem from different viewpoints, there may be partitions which essentially map to the same sets of system tasks. This means that reorganisation of the partitions are needed. However, it is not advisable to delete the tasks yet because, as implied from the first force, there may be implicit requirements for these partitions to exist which are not recognised yet.
- Indicated by the Task Flow, several partitions may be involved in completing one course of action within the system. This implies that there are dependencies between partitions.
- The number of partitions should be kept managable.

Solution The establishment of the relationships and the dependencies between the already discovered partitions involves analysing the specification of the system. In the context of Discovery, this means analysing the Task Model (Task Structures and Task Flows) as well as the Narratives. In the context of other object-oriented methods, this involves analysing the use cases and their scenarios. These specifications are revisited to allocate the tasks into the partitions, to establish the dependencies between partitions, to analyse the fitness of these tasks in their allocated partitions and to rearrange these tasks and partitions based on the inter-partition dependencies.

Implementation The process of identifying dependencies among the partitions and the restructurisation of the partitions is as follows:

1. Analyse the fitness of the tasks from the Task Structure and allocate them into the most appropriate partitions. The top level tasks should be allocated first, taking with it its subtasks.

    - Most of the tasks would fit into the partitions created based on the functional decomposition, as the tasks are identified from the same viewpoints.
    - The rest of the tasks would fit into partitions created based on physical settings.

    This initial allocation of tasks to partitions are driven by the fact that the partitions created based on these viewpoints are *fully partitioned* (the partitions cover the whole system) whereas the other forces only create *partial partitions* (the partitions only cover specific parts of the system). An example of this step can be seen in figure 3, indicating the allocation of the FoodMovers tasks into partitions.

2. Arrange partitions by identifying their relationships, including association and generalisation. This is to allow to cluster together closely related partitions. At this stage the subtasks in each partition are revisited to assess their fitness in the associated partitions. Further reference to the Task Flows would highlight the unsuitability of subtasks in a partition, allowing us to reallocate tasks into more appropriate partitions.

    In the FoodMovers example, the result of this step can be seen in figure 4. By arranging partitions, it is clear that the Inventory Reservation, Inventory Enquiry, and Shipping Instruction tasks are less related to their original parititons. Therefore they are reallocated to different partitions, which have tasks that manipulate similar things.

3. Analyse the dependencies between partitions based on the Narratives. The Narratives provide more detail scenarios of the system tasks. The dependency between one tasks to another is reflected by Preconditions of the narratives as well as Subtasks. The dependency between tasks means that the dependency between partitions also exists. Therefore, from the analysis of Narratives, we can draw the dependency between partitions. An example of this step is shown in figure 5.

4. As we can see in figure 5, the analysis of the narratives results in such an undesirebly complex dependency diagram. Such a complex diagram is difficult to understand. Therefore, in order to achieve minimal dependencies we analyse this diagram further. There are two things to be looked at for each dependency:

    (a) Whether the partition, on which another partition depend, is required to perform tasks in a same session as the depending task.

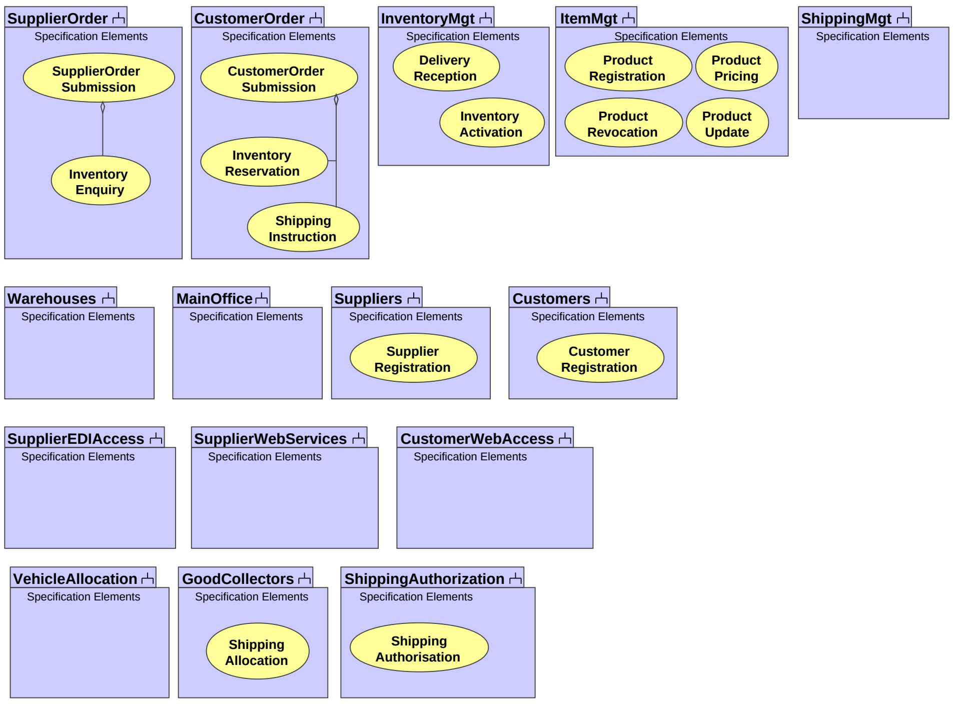

Figure 3: FoodMovers example: allocated tasks into partitions

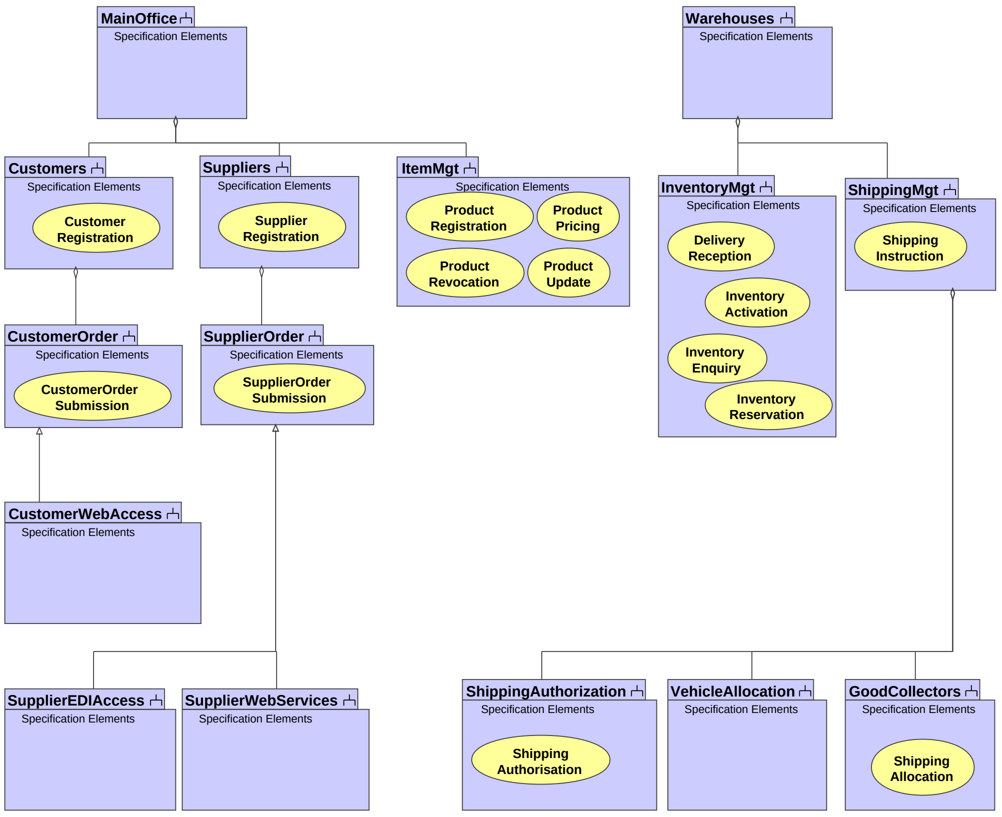

Figure 4: FoodMovers example: arranging partitions and reallocation of tasks

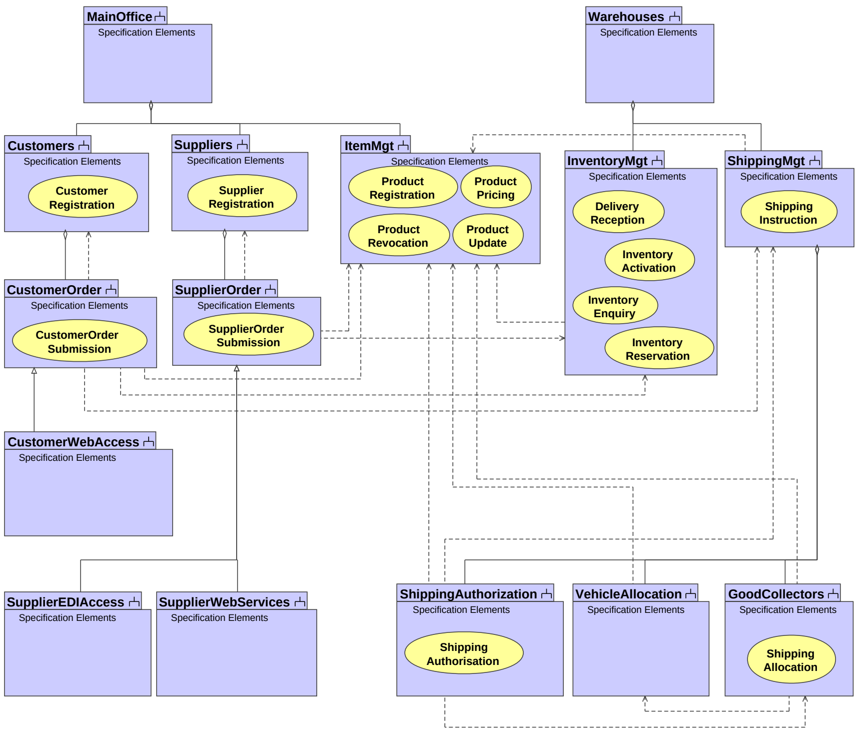

Figure 5: FoodMovers initial dependency diagram

(b) The kind of dependency that occurs. There are two types of dependency that may occur: Processing and Information dependency. Processing dependency between two partitions exists when the task in one partition delegates some of its processing functions to a task in the other partition, whereas Information dependency between two partitions exists when a task in one partition uses information produced by other tasks in the other partition.

If a dependency is an Information dependency and the depending task does not care when the task, on which it depends, produces the information, then we can hide the dependency from the diagram. The reason for this is that this kind of dependency leads to a weak coupling because it can be implemented using database or message-oriented middleware. Hiding the weak information dependency like this allow developers to obtain a cleaner diagram to work with to focus on the more complex dependencies. It is a good practice to document and save the complete dependency diagram to be revisited later when the time comes to design the database and data exchange.

Further analysis should also be taken for a partition on which several other partitions depend. When there are several partitions depend on a particular partition, this partition may have to be split up to reduce dependencies. If there are specific tasks which are the target of dependency and distinct from other tasks in that partition, then a new sub-partition could be invented to decouple these two task groups. This is to avoid unnecessary dependency to occur. The result would push tasks down the hierarchy of the partitions as low as possible. Reorganising tasks in this way will give less coupled partitions and allow these partitions to have more focused purposes. The resulting dependency diagram of the Foodmovers example is given in figure 6.

5. If necessary, the Partitions Dependency Diagram can be transformed into *Conceptual Architecture View* [HNS99] or *Logical View* [Kru95] as normally exercised in the architectural design. In this transformation, the partitions are simply translated into *architectural components*, while each dependency found in the partitioning dependency diagram (including the hidden ones) could be translated into *connectors* with further analysis. Figure 7 shows an example of the transformed FoodMovers partitions dependency diagram using the notation given by Hofmeister et.al. Further analysis by expert software architects and the application of architectural styles and patterns would provide more detailed architectural design.

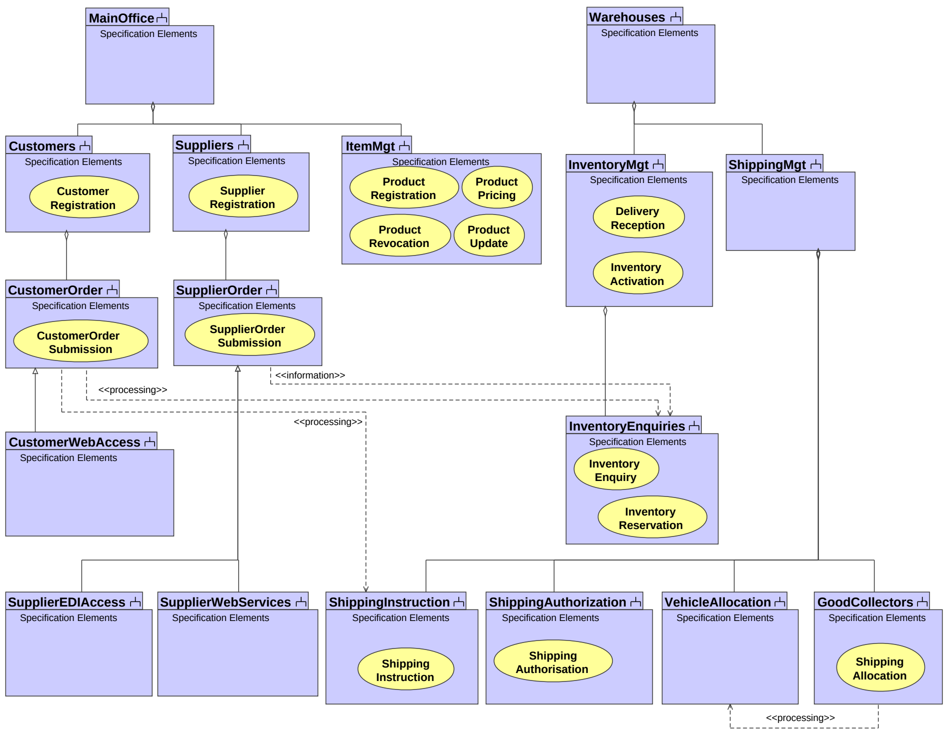

Figure 6: FoodMovers Partitions Dependency Diagram

Resulting Context All of the important main partitions, which are extracted from the problem domain, have been relatively revealed. Most importantly, the conceptual dependency between the partitions has been worked out to set out a framework, in which the minimal coupling between the partitions can be achieved in the eventual design. Based on this conceptual dependency diagram, which has taken various constraints into consideration, further design process can be performed, including applying architectural styles or patterns.

Rationale The notion of dependencies is adopted from the concept of software dependency proposed by Wijegunaratne and Fernandez [WF98]. Wijegunaratne and Fernandez specified that there are two kinds of Processing dependencies: Simple and Transactional. Simple processing dependency occurs when a component needs another component to to be performed remotely to complete its processing. Transactional processing dependency occurs when a component needs other components to be performed remotely as one single logical unit of work to complete its processing. Information dependency occurs when a component, as a result of some events in its jurisdiction conveys some information to one or more remote components, but there is no expectation of any processing associated with this information. The adoption of the notion of software dependency in this thesis does not distinguish simple and transactional processing. All processing dependency is transactional. It is simply because, we see that there is no sense in distinguishing one participant from a number of participants in transactional processings.

When it is necessary to have a formal software architecture design, then the Partitions Dependency Diagram is transformed into conceptual architectural design using the selected architectural language. So far, the information obtained from the user alone may not be enough to design the system architecture. The software architect should perform more thorough analysis of the system requirements to be able to decide on the solutions, particularly when designing the connectors. Dependency exists where there is a need to communicate, coordinate or cooperate between any two partitions. This means that such dependency is supported by a certain connection between the two partitions, which indicates the existance of a connector between the two partitions. Initial requirements for these connectors can be deduced from the Narratives. Data, events, or commands used for or required from other tasks indicate the required interaction between the tasks and hence between the partitions where the tasks reside. This narrows down the type of connectors to be used between the partitions (architectural components).

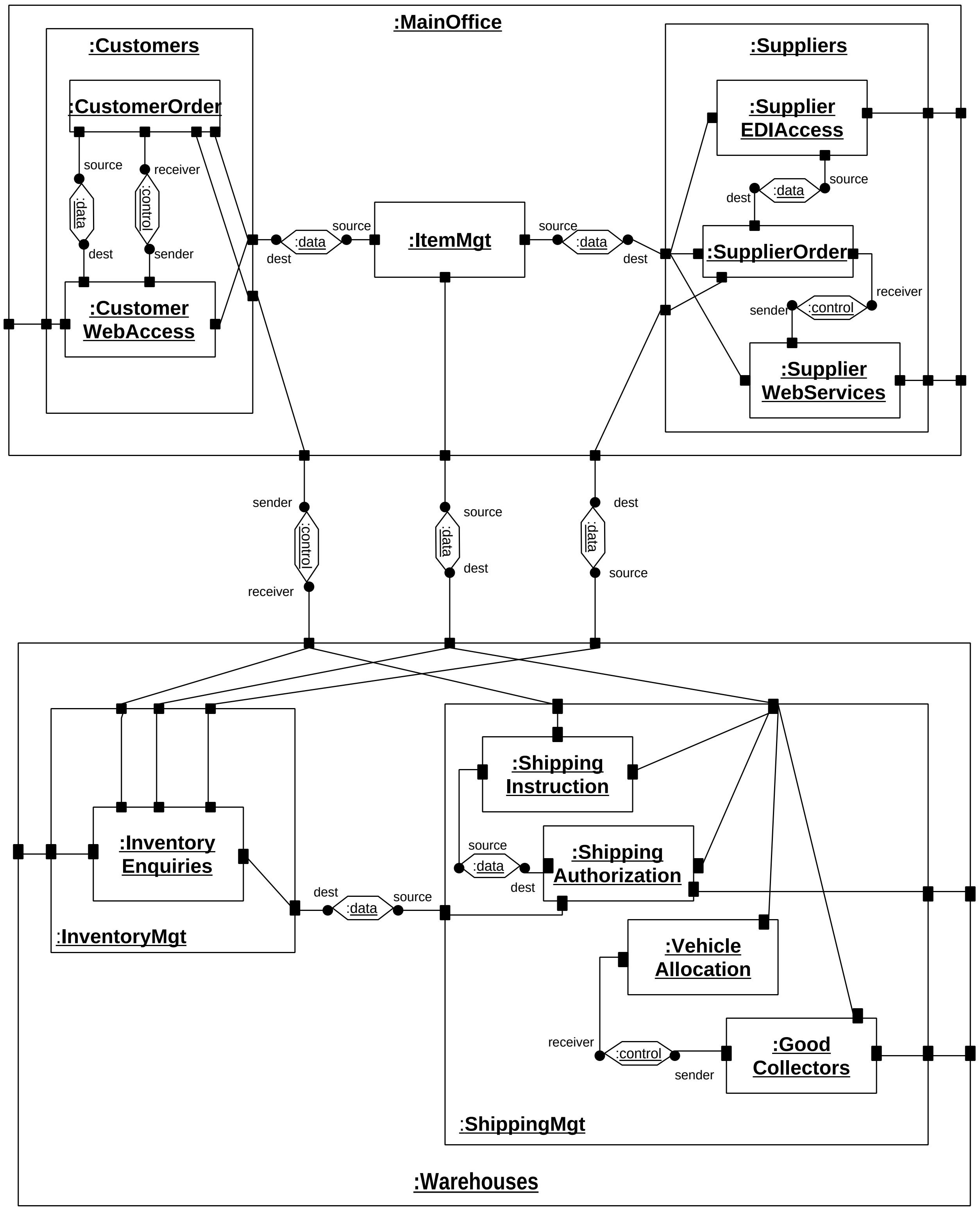

Figure 7: Example of Transformation to Conceptual Architecture View of Hofmeister et.al.

Related Patterns This pattern can be regarded as a more detailed elaboration of the *Divide et Impera* pattern, which is given as one of the patterns in the Patterns for the Treatment of System Dependencies [Mar02]. The Divide et Impera pattern suggested that to handle large and complex project, the system should be divided into distinct parts by identifying its subsystems and defining their dependency graph. The resulting context of the Divide et Impera patterns states that smaller scopes of subsystems are obtained, which can be used and maintained by small groups of developers. This coincides with the targeted resulting context of this pattern (regardless of whether the Application of Architectural Style and Patterns pattern follows this pattern or not) where small groups of developers use an object-oriented method, particularly Discovery in this case, to design or maintain the subsystems independently.

Next The Application of Architectural Style and Patterns process pattern and the Partitioning Pattern Language.

3.3.3 Process Pattern 3: Application of Architectural Style and Patterns

Problem What is the best structure, into which the partitions should be arranged and how can the already discovered partitions fit into this structure?

Context All the system partitions as well as their conceptual relationships are already revealed. However, to be able to form a real solution, a formal structure is needed. Existing architectural styles and patterns provide abstractions of general design structures, which are favoured by the experts.

Forces

- Since software patterns were popularised by Gamma, Helm, Johnson, and Vlissides [GHJV95], there are a lot of published patterns available. These patterns cover three different levels of abstraction: architectural patterns, design patterns, and idioms (also known as coding patterns). At the same time, the increasing attention to the field of software architecture leads to the establishment of architectural styles. Superficially, the architectural styles are interchangable with architectural patterns, but Monroe et. al. [MKMG97] have shown the subtle and yet important difference between the two. Considering this, designers need to carefully select the architectural styles to establish the general structure of the application and to apply the necessary architectural patterns to encode the architectural concerns.
- The partitions discovered so far are still raw. They address the problems in the domain space, but still yet to addressed the issues in the solution space. This may lead to the definition of new partitions or components. However, these partitions or components should only be invented in accordance with the design of the selected application structure.

Solution Based on the description of the applied architectural styles and patterns, the main components of the system or subsystem are identified. The application should adapt to the applied styles/patterns by matching the partitions that are already discovered with the applied styles/patterns. The matching process might require designers to invent new partitions, rearrange or merged the partitions as directed by the applied styles/patterns.

Experts define architectural styles and patterns to document solutions that exhibit particular qualities in it. Therefore some quality attributes are already instilled within each of the styles or patterns, which also are carried forward to our application design when we apply these styles or patterns. The quality model that has been developed beforehand could be consulted to further refine the application design. This quality model specifies the non functional requirements of the application. The consultation would confirm whether the particular non-functional requirements has been addressed by the applied styles or patterns. When a particular non functional requirement has not been addressed, strategies to meet the requirements should be applied. This in turn might lead to the application of other styles or patterns.

Implementation The Partitioning Pattern Language (see appendix B) provides a guided choice over many available patterns. The language indicates the typical architectural styles or patterns to be applied when the designer is addressing a specific aspect of the application.

The language delineates that a distributed object application should consist of partitions, each of which is designed as an agent based on the Presentation-Abstraction-Control (PAC) (appendix B.1 and [BMR+96]) architectural pattern. In its simplest form, there would be at most two partitions (two PAC agents) that communicate with each other in a distributed object application. The communication is carried out using the facilities given by the underlying distributed object system.

Currently, most of the commercially available distributed object systems provide the synchronous communication type for their basic communication mechanism. In particular, this mechanism takes on the Broker (appendix B.7) architectural pattern. This allows us to generalise that in its simplest form, a distributed object application would also apply the Broker architectural pattern. The client and server in this pattern would be the two PAC agents involved.

If the nature of the application is beyond a simple interaction between two partitions, it might need to be organised in layered or tiered fashion. The increasing complexity stems from different roles played by each agent or by each collection of agents.

The Layer (appendix B.2 and [BMR^{+}96]) architectural style is applied when an agent delegates its functionalities to other agents under its control or when several agents share the same functionality. The former situation causes the lower layer to be created, while the latter causes a higher layer to be created.

The 3-Tier (appendix B.3) architectural style is defined to allow us to underline the specific roles each layer plays in a particular 3-layered system. Each tier might in turn be organised as layers or a collection of PAC agents. The communication between these three tiers are in general proprietary, according to the selected product and technology. However, if necessary, the tier can communicate to other tiers using the general approach of distributed object communication using the Broker pattern.

Resulting Context At this point the high-level system structure has been carefully designed and yet the technique leaves some room to refine the structure later on. By applying architectural styles or patterns, the framework for the solution has been defined. The partitions discovered before have been assigned to the required architectural components, as specified by the styles or patterns, and the need of new partitions (or architectural components) are revealed.

Rationale This pattern represents the activity where the expert's experience is taken into account by mapping the partitions and their dependencies to the chosen architectural style or patterns.

Next Each partition can then be developed separately by different teams to discover the details of its objects and classes. Such an exploration is accomodated by the original Discovery process or any other object-oriented analysis and design method. The process may also benefit from the component design method. The properties of each partition as prescribed by the styles or patterns, in which the partition plays a role, are also taken into account and may be regarded as further requirements of the partition. This allows each team of designers to focus on a specific part of functionality of the system.

References

[BH03] Frank Buschmann and Kevlin Henney. A Distributed Computing Pattern Language. In *Proceedings of EuroPLoP 2003*, 2003.

[BMR^{+}96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John-Wiley and Sons, 1996.

[Bos00] J. Bosch. *Design and Use of Software Architectures : Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[CHHKC01] Carl K. Chang, Jane C. Huang, Shiyan Hua, and Annie Kuntzmann-Combelles. Function-Class Decomposition: A Hybrid Software Engineering Method. *IEEE Computer*, pages 87–93, December 2001.

[Eel00] P. Eeles. Business Component Development. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures: Advances and Applications*, chapter 3, pages 27–59. Springer, 2000.

[Emm00] W. Emmerich. *Engineering Distributed Object*. John-Wiley and Sons, 2000.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gom00] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Application with UML*. Addison-Wesley, 2000.

[HNS99] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 1999.

[HW04] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, 2004.

[Kru95] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[LR00] G.C. Low and G. Rasmussen. Partitioning and Allocation of Objects in Distributed Application Development. *Journal of Research and Practice in Information Technology*, 32(2):75–106, May 2000.

[Mar02] Klaus Marquardt. Patterns for the Treatment of System Dependencies. In *Proceedings of EuroPLoP 2002*, 2002.

[MKMG97] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural Styles, Design Patterns, and Objects. *IEEE Software*, 14(1):43–52, January 1997.

[SHM$^+$96] António Rito Silva, Fiona Hayes, Francisco Mota, Nino Torres, and Pedro Santos. A Pattern Language for the Perception, Design and Implementation of Distributed Application Partitioning. Workshop on Methodologies for Distributed Objects, 1996. in conjuction with OOPSLA 1996.

[Sim] Anthony J.H. Simons. *The Discovery Method for Object-Oriented Systems Development*. URL: http://www.dcs.shef.ac.uk/ ajhs/discovery.

[SSH99] Anthony J.H. Simons, Monique Snoeck, and Kitty S. Y. Hung. Using Design Patterns to Reveal the Competence of Object-Oriented Methods in Systems-Level Design. *Computer Systems Science and Engineering*, 14(6):343–352, November 1999. Special issue: Object-oriented Information Systems.

[SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume Vol. 2: Patterns for Concurrent and Networked Objects. John-Wiley & Sons, 2000.

[Sto99] Steven A. Stolper. Streamlined Design Approach Lands Mars Pathfinder. *IEEE Software*, pages 52–62, September/October 1999.

[Tra03] Brian Travis. FoodMovers: Building Distributed Applications using Microsoft Visual Studio .NET. http://msdn.microsoft.com/library/en-us/dnvsent/html/FoodMovers1.asp, November 2003.

[WF98] I. Wijegunaratne and G. Fernandez. *Distributed Applications Engineering: Building New Applications and Managing Legacy Applications with Distributed Technologies*. Springer, 1998.

Appendices

A Bottom-Up Partitioning

A.1 Principles of Bottom-Up Partitioning

The principles of the adopted bottom up approach in partitioning include: *explicitly implement non-functional requirements and concurrency analysis.* The patterns govern the explicit encoding of the non-functional requirements, particularly regarding the distribution issues. The bottom-up partitioning technique is applied when all the important classes in each partitions have been identified using the basic object-oriented modelling. The design of the distributable partitions are iteratively fed back to the Application of Architectural Styles and Patterns pattern to ensure that the detailed design conforms to the application architecture.

A.2 Bottom Up Partitioning Patterns

The bottom-up approach is represented by two patterns: Objects Synchronisation and Concurrency Analysis.

A.2.1 Process Pattern 4: Reconciliation of Objects

Problem How can the consistency between similar classes in the partitions be ensured?

Context Each of the partitions might be developed independently by different teams. It is likely that similar classes or components are discovered as parts of two or more different, independently developed, partitions. The reconciliation of the object specifications are required to allow proper distribution of object responsibilities.

Forces

- The classes or components may have the same name but serve different purposes.
- There may be classes or components which have different names but serve the same purposes.
- The intersectional classes or components which have similar purposes, may have slightly different strategies to achieve the purpose.

Solution This pattern lets the similar classes, that reside in different components, be reconciled by refining their specifications and redistributing them the affected partitions. This may cause new classes to be discovered as the result of specialising, generalising, aggregating, or even splitting the classes. The refined classes and possibly the related classes, whose objects are closely associated with the objects of the refined classes, are then distributed back to the original partitions. This may result in the replication of classes or component interfaces dispersed throughout the partitions.

As the result of generalisation and aggregation, the need of new classes might emerge. These classes might not fit in the existing partitions as their responsibilites are to manage or mediate the reconciliated classes. This leads to the invention of new partitions on which the original partitions depend.

Implementation The technique is adopted from the development of business component proposed by Eeles [Eel00].

Resulting Context The design of the partitions has been adjusted to accomodate the refinement of the intersectional classes or components. There might still be some replicated classes or components, but they have been designed to adapt to such replication. The process might also result in the invention of new partitions whose responsibility is to manage or mediate the shared classes or components.

Rationale This pattern can be thought of a refactoring process. Existing top-down techniques on partitioning assumed that the resulting partitions are final. This pattern refines the partitions discovered by the top-down partitioning by reconciling of the similar classes or components found in different partitions. This refinement makes sure that resulting partitions are true components which have distinct responsibilities and its own boundary.

Next The Concurrency Analysis pattern, the Partitioning Pattern Language and the Application of Architectural Styles and Patterns pattern.

A.2.2 Process Pattern 5: Concurrency Analysis

Problem How is the concurrency requirement of and between different partitions addressed?

Context The inner part of the partitions has been designed but the interaction among these partitions has yet to be specified. While the dynamics of objects within a partition would have been designed using the selected object-oriented methods, the dynamics of objects across the partitions has not been addressed.

Forces

- There are two forms of distributable components: as a process, or as a thread. A process runs within the boundary of a machine, whereas a thread runs within the boundary of a process. For each of the partitions, the appropriate form of distributable component should be determined.
- The cost of the forms of distributable component should be taken into account. Some trade-off may need to be taken to implement the desired property of a partitions.

Solution This pattern helps in the design of the dynamics of objects across the partitions by considering concurrency requirements. If there is no dependency between two partitions, then they are two different distributable components. If there is an information dependency between two partitions, then they potentially become two different distributable components because a weak coupling can be assigned between them. If there is a required concurrency between two objects within a partition, then the partition is broken up into two different distributable components. The first two options result in the specification of software processes, whereas the last option results in the specification of threads within a single process.

The controller objects and active objects are also determined to refine the component specification. Controller objects serve as an operation manager among several objects within the component and do not necessarily have their own thread of control. Active objects have and initiate their own thread of control and they are responsible for coordinating execution or managing the communication with (provide request to and require requests from) other components. The inter-process communication is then designed, aiming for minimal coupling between two different components. The Partitioning Pattern Language suggests the patterns to be applied in such a detailed concurrency design.

Implementation The detail specification of concurrency are very much related to the framework of the distributable components used, such as EJB.

Resulting Context The inter-process communication between any two partitions has been designed. This implies some changes to the internal design of the partitions involved, including the implementation of the necessary concurrency within a partition.

Rationale Architecturally, the partitions have been design to deal with the inter-partitions dependencies through the application of architectural styles and patterns. However, the introduction of new classes or components, which are discovered during the basic modelling with the object-oriented method requires further adjustments in the design of the detailed inter-partition interaction, particularly concerning the distribution and concurrency issues. Except for the method proposed by Gomaa [Gom00], existing object-oriented methods are lacking in addressing the distribution and concurrency issues. This process pattern addresses these issues in respect of the required properties of the application structure which has been established before.

Next The design of the distributable components are iteratively fed back to the Architectural Style and Pattern Application pattern. This is to ensure that the detailed design conforms with the application architecture such that all the application requirements and constraints are properly addressed.

B Partitioning Pattern Language

The Partitioning Pattern Language is the result of an attempt to define a pattern language which address the partitioning of distributed object applications. The Partitioning Pattern Language adopts existing patterns from existing collections. Therefore, the full detail of each pattern will not be presented but pointers to the original pattern descriptions will be given.

The inclusion of patterns into the Partitioning Pattern Language is judged according to their applicability in a typical distributed object application design. The main sources of patterns to be included are the Pattern-Oriented Software Architecture (POSA) series [BMR$^+$96, SSRB00] and the work of Fowler [Fow02]. The arrangement of some of the pattern in the partitioning pattern language also takes on the work of Buschmann and Henney [BH03], who worked on a pattern language for a distributed computing infrastructure. Unlike their work, which was at the system level of abstraction, the partitioning pattern language defines a collection of patterns to build a distributed object application that utilises the underlying distributed object system. Some patterns from the POSA are included and adapted in the partitioning pattern language by considering various ways of communication between distributed objects as identified by Emmerich [Emm00].

The patterns discussed in the literature shows that some patterns overlap with some others. They may be referred by different terms but essentially address similar problems. Such distinction may occur to emphasis different roles/purpose or to address different levels of abstraction. On the other hand, the patterns may also have the same name, but they address different problems. The difference may lie on the levels of abstraction, or it can be on different domain altogether. As we can see in figure 8. the pattern names in the Partitioning Pattern Language are the original names as they were published. However, the following description will show that some of these patterns are used at different levels of abstraction from the original intent.

The patterns selected are grouped into three levels of abstraction: Architectural Styles, Architectural Patterns, and Design Patterns. Architectural Styles direct the general structure of the application. Architectural Patterns define patterns, which embed some architectural concerns in typical distributed object applications. Design patterns address more general design problems of distributed object applications, which are not necessarily architectural.

Figure 8 provides the structure of the pattern language using UML notation. The patterns are represented as classes. The color of the classes in the figure distinguishes the level of abstraction of the pattern. The **green** patterns are **architectural styles**. The **yellow** patterns are **architectural patterns**. The **blue** patterns are **design patterns**. The **orange** patterns are **architectural patterns** which embody the **frameworks** that can be applied directly to the application. The application of the framework should conform to the associated architectural pattern. Take Broker for example. It is actually an architectural pattern [BMR$^+$96] that prescribes a structure for distributed software systems. The pattern decouples components that interact by remote service invocations. This architectural pattern has been realised as a framework of classes in distributed object systems, such as Java/RMI. The utilisation of this framework requires developers to design the client and the server, create the client and the server stub, and use the broker provided by the framework, to be able to conform with the Broker pattern.

The implicit process to unfold the application design in the language is made explicit with the partitioning process patterns described in section 3.

B.1 Presentation-Abstraction-Control

The Presentation-Abstraction-Control pattern [BMR$^+$96] is adopted to be the primary structure of partition. This patterns was originally intended for interactive systems, which can be regarded as a construct of a hierarchy of cooperating agents. Each agent is composed of presentation, abstraction, and control components. In a true interactive system, the presentation components provides the visible behaviour of the agent. The abstraction components implements the functionality and the data model of the agent. The control components links the presentation and abstraction components as well as handles the interaction with other agents.

In the agent hierarchy, the top-level agents are the core of the system, on which other agents depend. The bottom-level agents represent self-contained semantic concepts of the application domain, on which the end-user can act. The intermediate-level agents relate the agents from the two other levels. Such a hierarchy matches the Layer (section B.2) pattern, which specifies the inter-agent design.

This pattern can be generalised to other systems, which are not purely interactive. In these systems, the presentation components provide the data format required by other agents, to which an agent interacts. This viewpoint fits the purpose of partitioning.

If the agent is intended to be used to interact with the end-user, the Model-View-Controller (MVC) (section B.4)

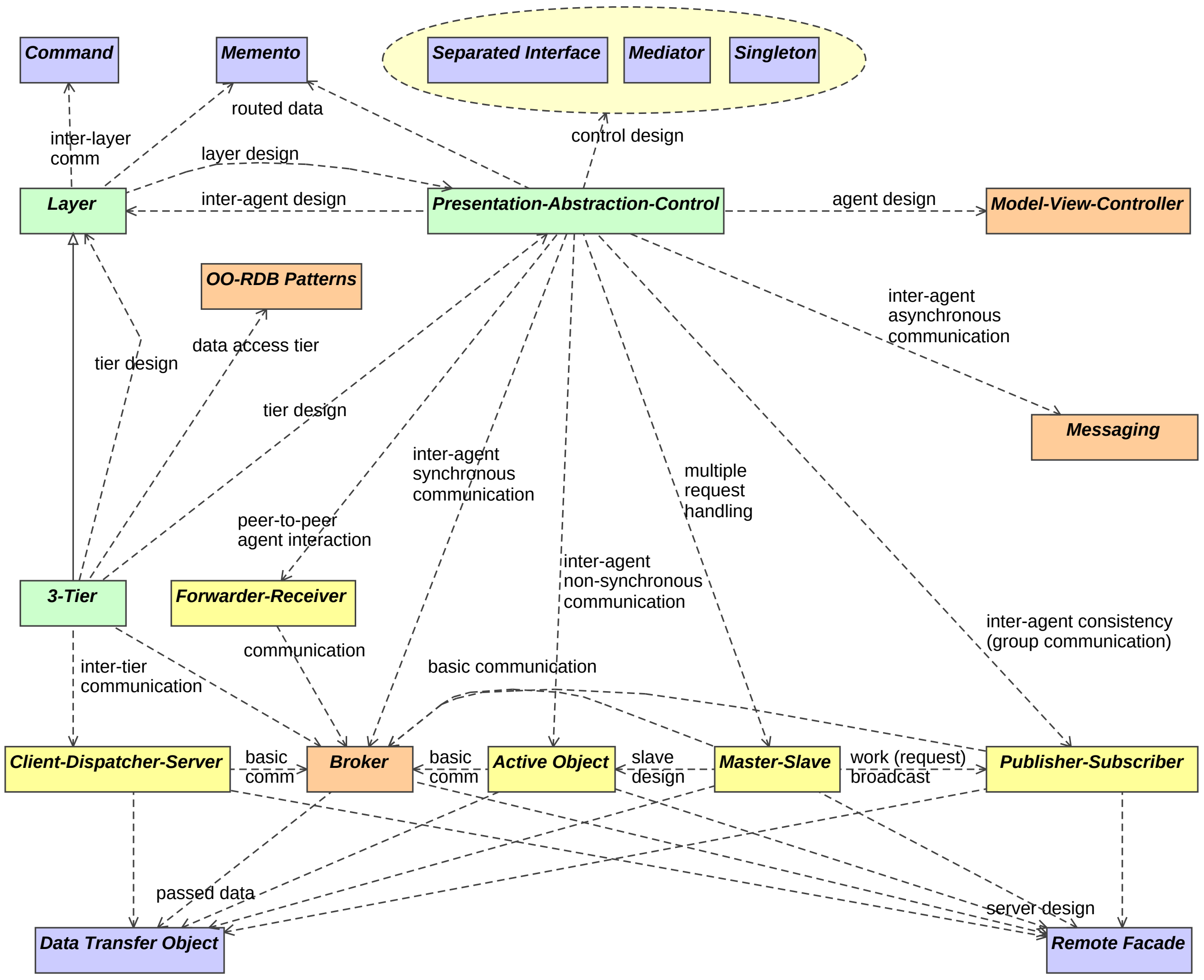

Figure 8: Partitioning Pattern Language

pattern should be applied. The Model in the MVC pattern corresponds to the abstraction components of an agent, and the view and controller parts to the presentation components [BH03].

The design of the control components makes use of the Separated Interface [Fow02], Mediator [GHJV95], Singleton [GHJV95] design patterns. The Separated Interface pattern decouples the interface from its implementation. Therefore, any changes to the implementation of a agent's functionality would not affect its relationship with other agents, with which it interacts. The Mediator pattern routes the requests from the agent's presentation to the abstraction and propagates the changes in the abstraction back to the presentation. The Mediator pattern also helps a control to route requests to and from other agents and coordinate the cooperation. To manage the routing of data, they can be encapsulated using the Memento [GHJV95] design pattern. The relationship between the PAC pattern and these other design patterns are adopted from the Distributed Computing pattern language [BH03].

In distributed object applications, the types of communication between two agents takes on the types of communication between distributed objects. The types of request synchronisation include: synchronous, non-synchronous, asynchronous communication [Emm00]. In addition to the single request, communication could also be implemented as a multiple request or a group request. The most common request type is the synchronous request, which follows the Broker (section B.7) architectural pattern. Non-synchronous requests are made by arranging the involved components according to the Active Object (section B.11) architectural pattern. Multiple requests can be sent from an agent by oberving the same principles given by the Master-Slave B.12 architectural pattern. To send group requests or to maintain the information consistency across several agents, the Publisher-Subscriber (section B.10) pattern is used. When two agents, which are considered peers, interact with each other, the Forwarder-Receiver (section B.6) architectural pattern should be employed to allow an agent to manage the information sent out and received.

B.2 Layer

The Layer [BMR$^+$96, BH03] architectural style defines the structure of application. Partitions, which are represented as PAC agents, are clustered together and arranged in different levels of abstraction. Each level forms a layer and each layer consists of one or more PAC agents. The Layer pattern prescribes that communications can only occur between two adjacent layers. To allow one layer to communicate with layers beyond the adjacent layers, instructions and data should be encapsulated in a uniform way. The Command [GHJV95] pattern is used to allow the uniform communication and the Memento [GHJV95] pattern is used to allow data to be propagated beyond the adjacent layers. The actual implementation of these patterns might be merged with the implementation of the control component of an agent within a particular layer.

B.3 3-Tier

The 3-tier pattern is a specialisation of the Layer architectural pattern. The general rules of the Layer pattern applies, but there are only 3 layers involved, each of which is called a tier. The three tiers have been variously described before, for example: Presentation - Domain - Data Source by Fowler [Fow02], or Client Software Modules - Composite Services - Data Access Servers by Wijegunaratne and Fernandez [WF98]. This specialisation of the Layer patterns specifies the pattern further with the components that should exist within a system as well as the relationship between the components. By definition, this can be categorised as an architectural style.

The style allows designers to implement each tier as a PAC agent or as layers of PAC agents depending on the application requirements. Therefore, the design of each tier should consult the PAC (section B.1) pattern.

In general, the communication between the tiers follows the rules imposed by the Layer (section B.2) pattern or the PAC (section B.1) pattern. However, in cases where the tiers reside in different sites, there could be the need for a remote communication between the tiers. In these cases, the basic communication as suggested by the Broker (section B.7) pattern applies. In more specific cases where the design of clients and servers should be abstracted away from the communication details, the Client-Dispatcher-Server (section B.8) pattern could be applied.

The implementation of the data access servers tier depends on the technology used to store the data. The majority of systems right now uses relational database technology as this technology is the most mature data storing technology. When implementing data access tier using the relational database technology, the patterns for Object-oriented to Relational Database mapping (section B.5) apply.

B.4 Model-View-Controller

The Model-View-Controller pattern is adopted purely from its original specification in POSA1 [BMR$^+$96]. In the application level of abstraction, where in general the implementation of the solution utilises the provided technologies, the

MVC pattern is enforced by the applied framework. Examples of this framework are Microsoft MFC, or Java Swings. The existence of this pattern in the pattern language directs the designers to apply the rules and provided implementations of the MVC framework.

B.5 OO-RDB Patterns

The OO-RDB patterns is a system of patterns that should be applied to map object-oriented model to relational database as the model's persistent storage. This system is adopted purely from its original specification by Fowler [Fow02]. The system consists of four subsystem: Data Source Architectural Patterns, Object-Relational Structural Patterns, Object-Relational Behavioral Patterns, and Object-Relational Metadata Mapping Patterns. The existence of the OO-RDB Patterns pattern in the pattern language directs the designers to apply the system of patterns specified by Fowler.

B.6 Forwarder-Receiver

The Forwarder-Receiver pattern was originally specified in POSA1 [BMR+96]. The original pattern specifies the transparency of communication between two system components which interact in a peer-to-peer fashion. The structure of the original pattern specifies that there are Forwarder and Receiver classes whose responsibilities are to actually deliver, marshal, unmarshal, and receive messages, using the underlying IPC mechanism.

The pattern in this pattern language lifts the level of abstraction such that the Forwarder and Receiver classes are not responsible for the actual message marshaling and unmarshalling anymore. Instead they deliver and receive messages or data using the provided communication framework of the underlying distributed object systems. Therefore, the pattern use the Broker (see section B.7) framework, such as Java/RMI, to engage the communication between peers.

B.7 Broker

The Broker architectural pattern, which was originally specified in POSA1 [BMR+96], specifies the structure of distributed system with components that interact by remote service invocations. The importance of this pattern is emphasised in the pattern language for distributed computing [BH03], by declaring this pattern to be one of the two entry points of the language. In parallel with the pattern language for distributed computing, this pattern language also treats the Broker pattern as the core element, with which the basic communication is implemented.

In many of the commercially available distributed object systems, which underlie distributed object applications, the Broker pattern has been implemented as communication frameworks. The framework encapsulates the details of the remote service invocations so that the details of remove service invocations can be abstracted away from the application design.

The use of the framework directs the designers to express the design in such a way that it conforms to the Broker architectural pattern. This means that application designers need to design the client and server components, and with the help of the framework generate the client and server proxy components. Because of the communication overhead, each call to perform the remote service invocation is costly. Therefore, the server components should be designed by composing the necessary services in such away that the number of remote invocations to the servers could be kept minimal. Furthermore, the interface, through which the services are invoked should be decoupled from their implementation so that changes to implementation do not affect the way the services are invoked. To allow such a server implementation the Remote Facade [Fow02] pattern is used.

In distributed object applications data that are passed remotely from servers to clients might be a composite or an extraction of the actual data held by the object model. To allow efficient data passing, these data should be packaged in such a way that the communication calls can be reduced. To implement this, the Data Transfer Object [Fow02] pattern is used.

B.8 Client-Dispatcher-Server

The Client-Dispatcher-Server design pattern was orginally specified in POSA1 [BMR+96]. This is another communication design pattern which introduces the dispatcher component between the client and the server to provide location transparency by way of a name server and to hide the details of the establishment of the communication connection between clients and servers. The importance of this pattern in distributing computing is emphasised in the pattern language for distributed computing [BH03], as the other entry points of the language beside the Broker pattern.

Taking on the principles of the pattern, the partitioning pattern language introduces this pattern at the application level, moving up the level of abstraction at which the original Client-Dispatcher-Server pattern was intended. This pattern can be used to implement the inter-tier communication, particularly between the client software modules tier and the composite services tier (see section B.3). While the broker framework provides the basic synchronous communication, the dispatcher component receives the requests from the client components, looks up for the actual services that have been registered before, and facilitate the clients and the servers to communicate directly. As with the basic communication using the broker architecture, the Remote Facade [Fow02] and the Data Transfer Object [Fow02] patterns can be used to represent the server design and the data to be passed through the communication channel.

B.9 Messaging

One of the ways in which the coupling between two partitions could be reduced is to use the asynchronous requests for the communication. This type of request synchronisation can be implemented using message queues [WF98, Emm00]. The clients put the request as a message in a queue, which can be collected by the servers whenever it is available to process it synchronously. The result would be put back in the message queue by the server to be collected by the client later on. This way there is no need for the client to wait for the server to give back the processing result.

The implementation of such a message queue facility is provided by commercially available Message-Oriented Middlewares (MOMs). Some implementations of the message queue are also provided as extensions to the existing distributed object systems, for example JMS (Java Message Services) can be used with other distributed Java facilities.

The Messaging style is included in the partitioning pattern language because its specification and related architectural patterns described by Hohpe and Woolf [HW04] explain the principles of the message queue described above. The actual application might only involve using the provided message-oriented facilities or middleware. However, this also means that the application should be designed to conform to this architectural styles and patterns.

The Data Transfer Object [Fow02] patterns can also be used to represent the data to be processed or returned through the communication channel. The Remote Facade [Fow02] can be used to specify the services required of the servers.

B.10 Publisher-Subscriber

When a partition, which is designed as a PAC agent (see B.1), needs to make a request to a group of other partitions/agents, then the group request [Emm00] is sent out. To enable the partition/agent to send a group request in one go, the Publisher-Subscriber [BMR$^+$96] pattern can be used. Using this pattern, the server should be registered first as the subscriber of the request and the client would be the publisher of the request. The use of this pattern in distributed application has also been established [BH03], particularly to maintain the consistency of states among the agents involved in the PAC system.

The implementation of the actual request made uses the provided broker framework (section B.7). The Remote Facade [Fow02] pattern is particularly useful to represent the uniform design of the servers, where the actual implementation of the servers may vary as the partitions involved also vary. The Data Transfer Object [Fow02] pattern is used to represent the concise design of data to be passed around.

B.11 Active Object

There are times when the synchronous communication is not acceptable for the required service level of the application. This is when the clients cannot wait until the results of the service are given back by the servers. In this case, non-synchronous request should be made instead by the client.

There are two types of non-synchronous request: one-way request and deferred synchronous request [Emm00]. One-way request is made when the client takes control back immediately after the request is sent to the server and the client and the server do not synchronise. Deferred synchronisation request is made when the client takes control back immediately after the request is sent to the server and polls the results later on.

The non-synchronous requests can be implemented using threads and the already provided synchronous request mechanism provided by the broker framework (section B.7). To implement the one-way request, the client simply needs to initiate a new thread and let this thread do the actual request while the client can continue on. To implement the deferred synchronisation request, the Active Object [SSRB00] pattern can be applied. The Data Transfer Object [Fow02] pattern is used to represent the concise design of data to be passed. The Remote Facade [Fow02] pattern is used to represent the design of the server.

B.12 Master-Slave

In some cases, simple unicast request is not enough. In these cases, an atomic request consists of multiple requests which need to be managed. To implement such multiple request communication like this threads and the basic synchronous mechanism provided by the broker (see B.7) framework are used. The Master-Slave [BMR$^+$96] pattern is adapted. Originally, this pattern is specified to support parallel computation where a master component distributes works to identical slave components and works out the computation result based on the results given back by these slaves. In the multiple request context, the slaves may not be identical, but the remaining principles apply. The master component sent the requests to the intended *slaves* and later on it collects and combines the final result from the results given back by these *slaves*. In this context, these slaves are actually the servers. The Data Transfer Object [Fow02] pattern is used to represent the concise design of the results to be passed. The Remote Facade [Fow02] pattern is used to represent the design of the server.