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 paper describes the process of partitioning DOAs to allow a justifiable DOA structure to be achieved. The process is expressed as patterns (*Partitioning Process Patterns*) because it is inspired by the combination of common practices from both the architectural decomposition approach and the subsystems discovery technique found in several object-oriented methods. It is assumed that the process is performed in conjuction with an object-oriented method.

1 Introduction

Partitioning is needed in designing distributed object applications (DOAs) to obtain the required application components and to make the design process manageable. A good partitioning technique allows designer to discover the necessary partitions or components, particularly distributable ones. 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. 3-tier style [1]. This top-down decomposition approach can be considered as a specialisation of architectural design methods, e.g. [2], which capture the expert software architects' approach in determining the application structure. 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. Design reworking 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 of the application [3, 4]. However, this approach disregards the fact that some presumptive structures may exist as a consequence of addressing the concerns which are 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 paper proposes a partitioning technique, which 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 technique would enable less-experienced designers to utilise architectural and design patterns or styles and yet allow them to refine the resulting designs to obtain better distributed object application architecture based on the internal forces of the system requirements. The technique is expressed as patterns (*Partitioning Process Patterns*) because it codifies common practices in finding solutions to recurring partitioning problems found in both top-down decomposition and bottom-up subsystem discovery approaches. The patterns present a step-by-step view of the process, which experts would normally perform spontaneously and proficiently. The technique is also informed by the *partitioning pattern language* (see appendix A), which would allow designers to apply certain patterns or styles at the right stage. The illustration in the patterns refers to the description of the FoodMovers case (see appendix B).

2 Partitioning Process Patterns

The interdependence among the *partitioning process patterns* is illustrated by figure 1. The partitioning process is expressed in five patterns, which act upon the artifacts produced by a particular object-oriented development method, which emphasises the bottom-up discovery of subsystems [5]. Three top-down *partitioning process patterns* include the DISSECTION, IF THE SHOE FITS, WEAR IT!, and BACK TO BASICS patterns. Two further bottom-up *partitioning process patterns* include the PUT DOWN ROOTS and PUT TWO AND TWO TOGETHER patterns.

The DISSECTION pattern allows intuitive partitioning based on the concerns given from the outset. The IF THE SHOE FITS, WEAR IT! process pattern let the designers make sense of the informal structure of all the intuitive partitions discovered from the DISSECTION. Informed by such informal structure of the system, the BACK TO BASICS pattern formalise the overall structure of the system by applying relevant architectural styles or patterns. The object-oriented method is then applied to explore each of the partitions of the system in detail. The objects discovered from the object-oriented method are then reconciled in the PUT DOWN ROOTS pattern to ensure that there are no duplicate specifications. The details of inter-component communications and any concurrency that might exist in each component are explored in the PUT TWO AND TWO TOGETHER pattern. The BACK TO BASICS process pattern is reapplied to the result, for fine adjustments before the next iteration.

Figure 1 also shows the related artifacts and technique of the object-oriented method. The information from *notes and sketches*, which are obtained from the interview with the stakeholders is used to guide the DISSECTION. As a more formal form of the requirement, *narratives* (or use case scenarios) and *task model*[1] are used to apply the IF THE SHOE FITS, WEAR IT! pattern. The application architecture, as a result of the top-down partitioning, is used to direct the object-oriented analysis and design. The application architecture should be consulted by the *control analysis, behavioral analysis* and *data analysis* activities. The client-server connections among the classes determines the coupling between any two classes. An overview of the client-

[1] A task is any activity which has a goal or purpose. Typically tasks are quite coarse grained business activities that decompose into subtasks, down to the level of use cases. The task model represents the structure and the logical sequencing of the system tasks.

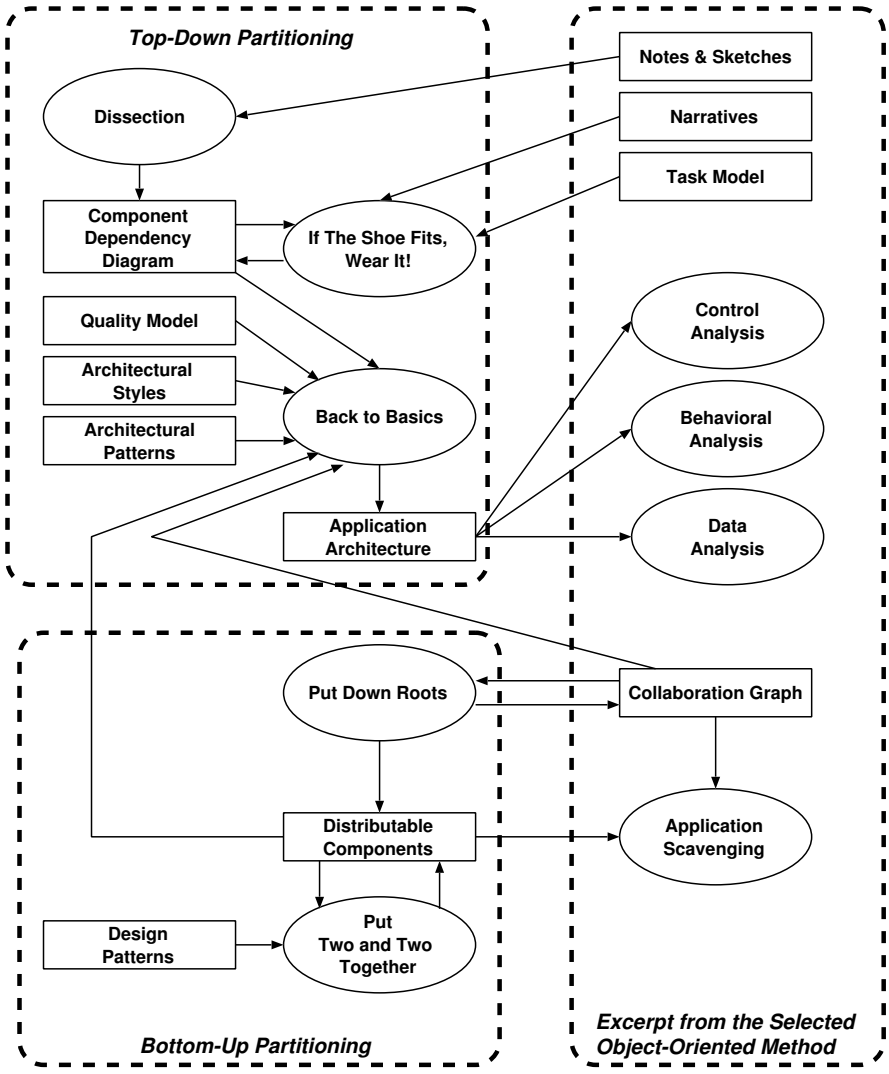

Figure 1: Interdependence of the *Partitioning Process Patterns*

server connection is represented by a *collaboration graph*. This graph is used to decide which package is the proper owner of an object in the PUT DOWN ROOTS pattern. Along with the PUT TWO AND TWO TOGETHER pattern, these two process patterns help in developing the design of distributable components. If necessary, the distributable components can then be extracted as reusable components or to form a reusable framework. This is the essence of the *application scavenging* activity.

Process Pattern 1: Dissection

Problem

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. Such partitioning should be started as early as possible.

Context

The system requirements have been explored. In the context of the object-oriented method, the requirements are represented as tasks (coarse-grained tasks to fine-grained use cases) in a structured Task Model, with associated Narratives detailing the associated stories, or scenarios. The system should now be partitioned so that designers could manage the design of the required functions and features. In other words, the use cases have yet to be grouped into packages. 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 of partitions, regardless of the actual needs of the system.
- Some systems might have some constraints regarding physical settings, technical heterogeneity, or social related issues. These indicate the elementary non-functional requirements, which are salient at such an early stage of the application development.
- Based on their previous experience, developers of the system might apply abstraction to simplify or encapsulate the complexity of the implementation.

Solution

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: IF THE SHOE FITS, WEAR IT!).

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

There are some alternative viewpoints, from which the application partitions could be discovered. These include:

- Client preference

  The client or the system owner may impose preferential groupings of the application. However, these 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 1:
  > In the FoodMovers example (see Appendix B), 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.

- 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 2:
  The analysis of the FoodMovers case shows that the 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.

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 several types of site and each type plays different role than the others. Therefore the initial groupings are determined based on these types of site.

  In case where there is only one type of site, 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 site are the same, since they play different roles from each other then the initial groupings are determined based on these roles.

  Example 3:
  From the FoodMovers example, 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 a different implication for the application implementation.

  > Example 4:
  > 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 5:
  > 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.

  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 6:
  > Some of the Foodmovers' suppliers are long-established manufacturers who 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 parti-

tion 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 by laws or regulations, one or more dedicated subsystems are identified to handle those functions, which ensure that the laws or regulations are followed accordingly.

  > Example 7:
  > 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 8:
  > There are at least two kinds of staff roles played in the warehouses, in the FoodMovers example. The 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).

Resulting Context

The groups discovered so far become the initial partitions of the application. The easily perceived system constraints have been taken into consideration in proposing these initial partitions. At this point, only the partitions of the systems that are discovered. The relationship between them have yet to be established.

Consequences

Benefits

- Allows designers to intuitively decide on initial conceptual groupings, based on the listed alternative viewpoints or on the designer's past experiences.

- Considers both functional and elementary non-functional requirements.
- Accommodate the client preferences and integrate them into the design

Liabilities

- If all the alternative viewpoints are considered, they may suggest too many initial groupings to handle
- There may be some overlap or coincident groupings found, which could cause a confusion.

Rationale

This is a pattern that represents 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. More importantly, this model has already taken notable constraints into account.

Some of these constraints suggest 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 [6] 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 IF THE SHOE FITS, WEAR IT! pattern.

Process Pattern 2: If the Shoe Fits, Wear It!

Problem

How do you arrange system partitions, such that their interdependency makes sense? Furthermore, the arrangement should contribute to a fair eventual design, which demonstrates minimal

coupling between the application partitions.

Context

The observable partitions of the system have been identified. These partitions originate 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 have yet to be established.

Forces

- Indicated by the logical sequence of the tasks, several partitions may be involved in completing one course of action within the system. This implies that there are dependencies between partitions.
- 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. It is not advised to delete the any partition just yet because further information supporting in which partition the tasks properly reside has not been determined yet.
- 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. This means analysing the *task model* as well as the *narratives*, which have been discovered so far. The analysis of the specification is performed to:

- allocate the tasks to the partitions,
- analyse the fitness of these tasks in their allocated partitions,
- establish the dependencies between partitions, and
- rearrange these tasks and partitions based on the inter-partition dependencies.

Implementation

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

1. Analyse the fitness of the tasks from the *task model* and allocate them into the most appropriate partitions. The top level tasks should be allocated first, taking the subtasks with them.

- 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 on the basis of physical settings.

This initial allocation of tasks to partitions is 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 2, indicating the allocation of some of the FoodMovers tasks into partitions.

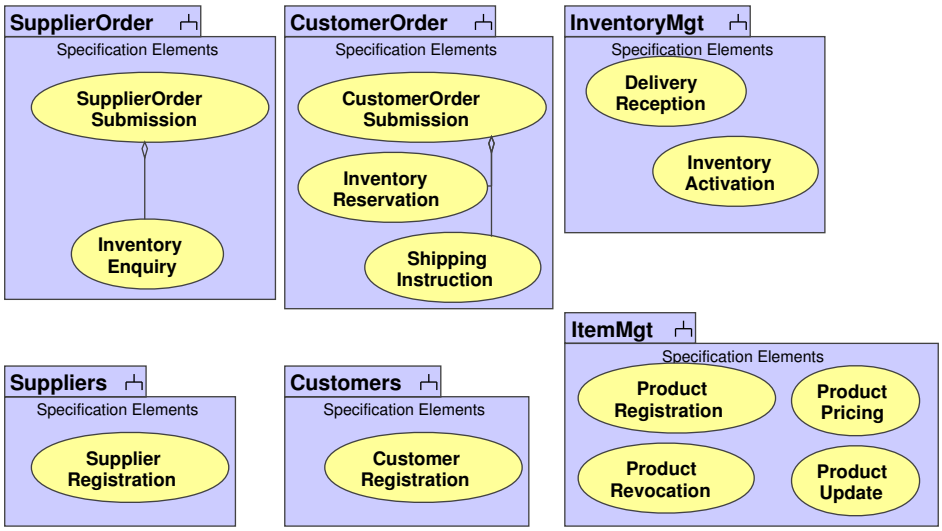

Figure 2: FoodMovers example: allocated tasks into partitions

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 logical sequence of the tasks would highlight the unsuitability of subtasks in a partition, allowing us to reallocate tasks to more appropriate partitions.

   Example 9:
   In the FoodMovers example, a partial result of the partition arrangement step can be seen in figure 3. By arranging partitions, it is clear that the Inventory Reservation and Inventory Enquiry tasks are less related to their original parititons. Therefore they are reallocated to different partitions, which have tasks that manipulate similar things, i.e. the Inventory Reservation and Inventory Enquiry tasks are put in the InventoryMgt partition.

3. Analyse the dependencies between partitions based on the Narratives. The Narratives provide a detailed description of the system's tasks. The dependency between one tasks and 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 4.

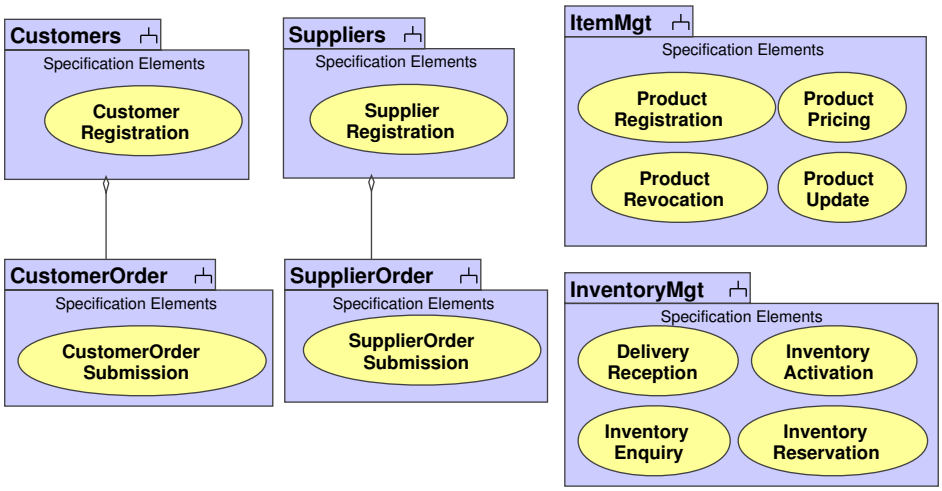

Figure 3: FoodMovers example: arranging partitions and reallocation of tasks

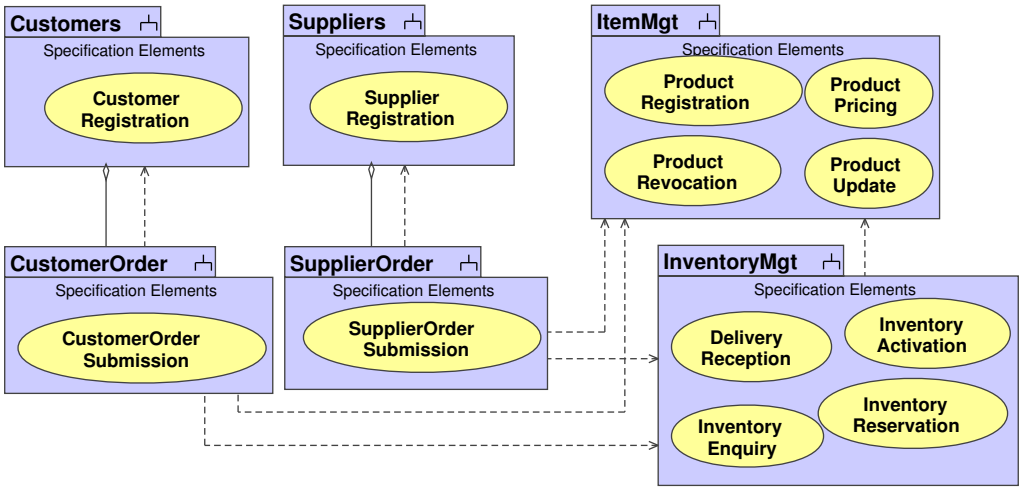

Figure 4: FoodMovers initial dependency diagram

4. The analysis of the narratives may result in such an undesirably 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 depends, is required to perform tasks in a same session as the depending task.

   (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. The notion of these dependencies is adopted from the concept of software dependency proposed by Wijegunaratne and Fernandez [1].

   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 allows 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 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. 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 5.

5. If necessary, the Partitions Dependency Diagram can be transformed into *Conceptual Architecture View* [7] or *Logical View* [8] 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. Further analysis by expert software architects and the application of architectural styles and patterns (see Process Pattern 3: BACK TO BASICS) would provide more detailed architectural design.

Resulting Context

All of the important main partitions, which are extracted from the problem domain, have been 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

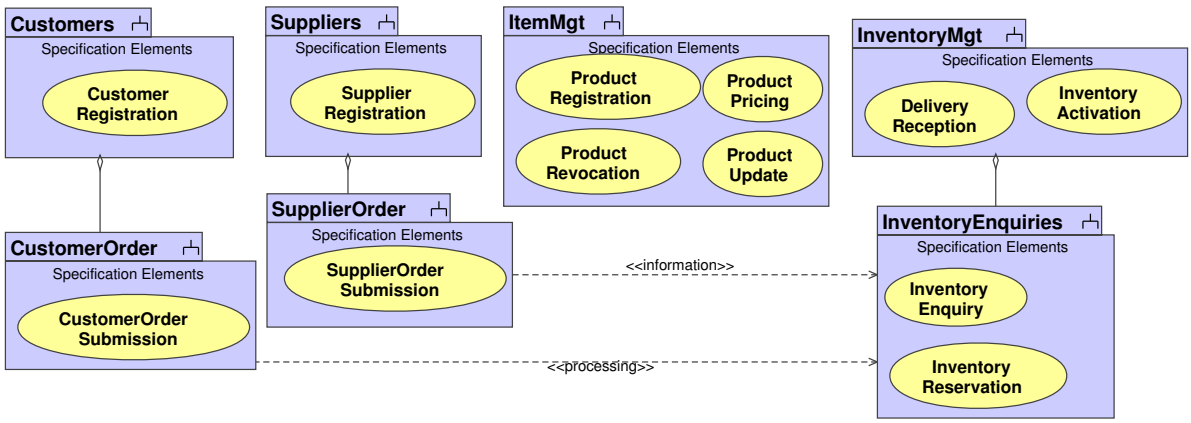

Figure 5: FoodMovers Partitions Dependency Diagram

taken various constraints into consideration, further design process can be performed, including applying architectural styles or patterns.

Consequences

Benefits

- designers can have a clear overview of the future system, with regard to the responsibilities of the partitions and the dependency between any two partitions.
- hence, designers can focus their attention on certain part of the system, allowing them to refine the under-specified partitions.

Liabilities

- the dependency analysis may result in quite complex diagram, which can cause confusion.
- the partitions are subject to refinement. Further elaboration of the tasks will reveal the collaborating objects. The associated classes may belong to different partitions from the original partitions to which the tasks are assigned.

Rationale

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 indicates the existence of a connector between the two respective partitions. Initial requirements for these connectors can be deduced from the Narratives. Data, events, or commands used by a task or sent to other tasks indicate the required interaction between the two tasks and hence between the partitions, in which the tasks reside. This narrows down the type of connectors to be used between the partitions (architectural components).

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 [9]. 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 BACK TO BASICS pattern follows this pattern or not) where small groups of developers use an object-oriented method to design or maintain the subsystems independently.

Next

The BACK TO BASICS process pattern and the *Partitioning Pattern Language*.

Process Pattern 3: Back To Basics

Problem

Having discovered the partitions, their dependencies and the system constraints, these partitions need to be arranged within the selected structure. What is the best structure for distributed object applications, into which the partitions should be arranged, and how can the 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

- There are a lot of published software patterns available. These patterns cover three different levels of abstraction: architectural patterns, design patterns, and idioms. 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. [10] 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 have yet to address the issues in the solution space. This may lead to the definition of new partitions. However, these partitions should only be invented in accordance with the design of the selected application structure.

Solution

Select the most appropriate architectural styles and patterns. Based on the description of the selected architectural styles and patterns, the required stereotypes of components of the system or subsystem are identified. The application should adapt to the applied styles/patterns by matching the discovered partitions with the components stereotypes of 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 them. Therefore the quality attributes, which are instilled within each of the styles or patterns, will be carried forward to the design of the application when we apply these styles or patterns. The quality model that has been developed beforehand could be consulted to further refine the application design. The quality model specifies the non functional requirements of the application. Consulting the quality model would confirm whether there are particular non-functional requirements, which have not been addressed. Strategies to meet these unaddressed non-functional requirements might in turn lead to the application of other styles or patterns.

Implementation

The *Partitioning Pattern Language* (see appendix A) 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) [11] 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 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 [11] 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 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 with 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.

Consequences

Benefits

- There is no need to invent new structures as existing architectural styles and patterns, and design patterns provide the structures with the required properties.
- The quality of the application is assured by applying the proven solutions (in the form of styles and patterns).
- Particularly for the design of distributed object applications, the *Partitioning Pattern Language* provide a guided choice of styles and patterns to apply.

Liabilities

- For novice designers, selecting and applying styles or patterns may become a cumbersome activity [12].

Rationale

This pattern represents the activity where the experts' experiences are 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 object-oriented analysis and 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 as further requirements of the partition. This allows each team of designers to focus on a specific part of functionality of the system.

It is likely that there would be duplication of specifications as the result of independent development of the partitions. The PUT DOWN ROOTS process pattern removes such duplication and further optimises the partitions.

Process Pattern 4: Put Down Roots

Problem

There might be the same objects that are used in more than one partitions. However, the specification of these objects, i.e. the classes, need to reside only in one partitions. How do you know which partition is the right home for a class?

Context

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

Forces

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

Solution

This pattern lets the similar classes, that reside in different partitions, be refined by reconciling their specifications. 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 other associated classes are then redistributed back to the affected partitions. Each of these replicated classes is examined further to determine the most suitable partition to which each class belongs. In the chosen partition, new interfaces are specified for the services offered by the original clases. The rest of the same classes in other partitions are replaced by this interface.

The refined classes might not always fit in the existing partitions as their responsibilites might not be compatible to the themes of the existing partitions. This leads to the invention of new partitions, on which some of the original partitions depend.

Implementation

Although so far this pattern mentions classes as the main artifact to work on, it is the roles of objects that this patterns actually deals with. The steps of reconciling the object roles are as follows:

1. The first step in reconciling objects is to look for the potential concordance indicated by the same or similar object role names in the current Collaboration Graph[2]. Although this does not guarantee to cover all the potential roles, but this is a good enough approximation. Recall that the partitions were developed based on the same set of use cases using the same dictionary. Although there might be a variance in the names given to the object roles, the developer who chose the names would have been guided by the dictionary. This means that the same concept would be named with slightly different, but related, terms. Therefore, this simple strategy yields the majority of the potential roles to be reconciled.

[2]This is a deliverable of Discovery[5]. If other object-oriented method is used, the class diagram can be the source of information.

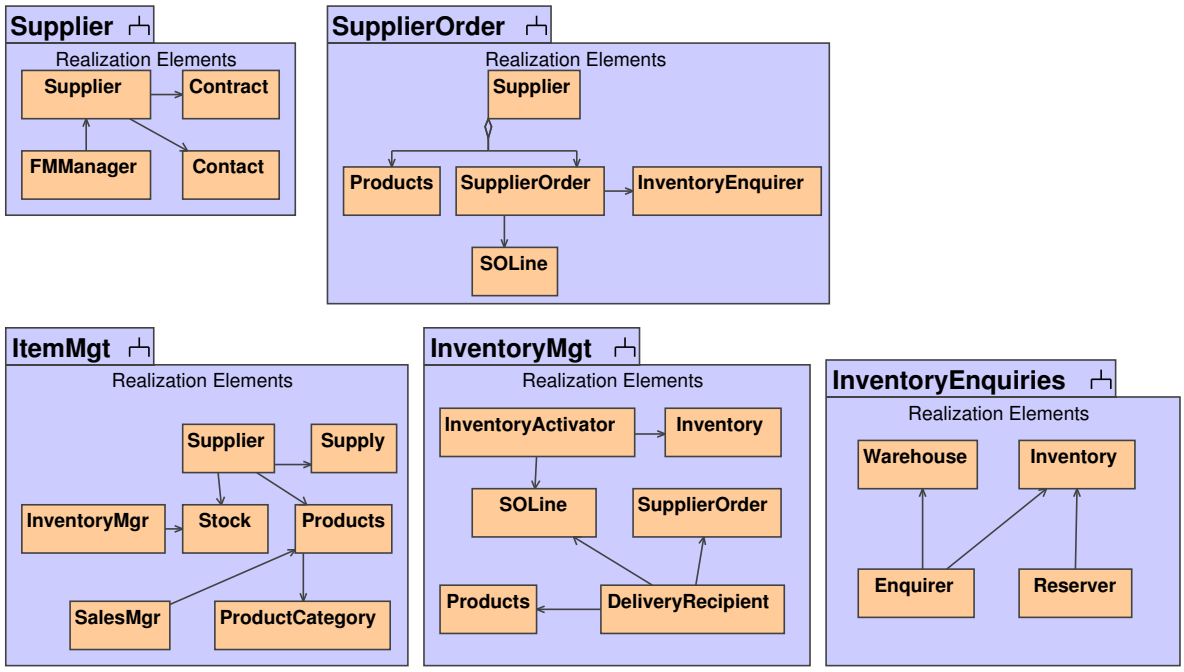

Figure 6: Example of Collaboration Graph

Example 10:
Figure 6 shows the collaboration of some of the functions of FoodMovers example. It is clear from the figure that there are several object roles that have the same name, for example: Supplier, Products, and Inventory. Others have similar names, for example InventoryEnquirer and Enquirer. These are the potential object roles which would be reconciled.

2. Once similar class names have been identified, the responsibilities of these object roles are analysed to look for the similarity of the object roles' responsibilities. The analysis results would be one of these situations:

   (a) If they have the same set of responsibilities, then it means that they are actually the same object roles. Therefore, one name is assigned to those set of object roles.

   (b) If they have different and disjoint set of responsibilities, then they are actually of different object roles. In this case, assign different names for different set of responsibilities.

   (c) If they have intersectional responsibilities or if the responsibilities of one object role is a subset of another's, then they are actually multiple roles of the same object (see also the *related patterns* section of this pattern). Further analysis is needed to determine whether or not merging all the responsibilities into one object role would cause integrity problems. If the merger does not cause any integrity problem, they could become one object role type. If the merger would likely cause integrity problems, a host object role should own the intersectional (common) responsibilities, whereas the specific responsibilities would be owned by the variant object roles.

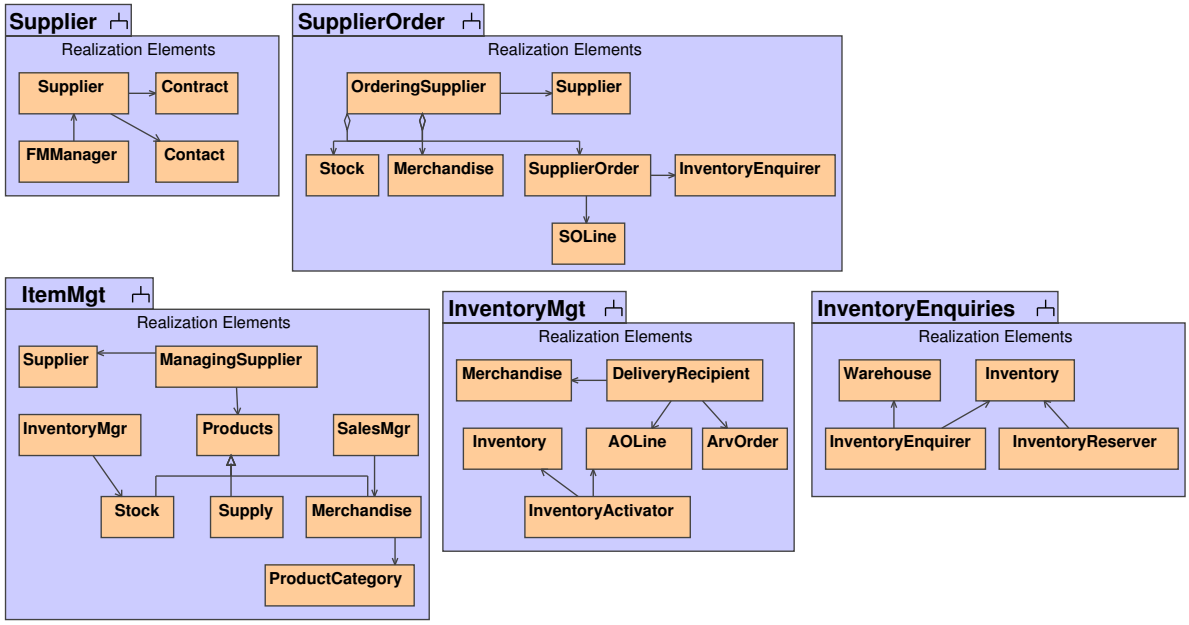

Figure 7: Alignment of Object Roles

Example 11:
Figure 7 shows that similar roles found in the collaboration graphs are actually the same, for example: InventoryEnquirer and Inventory.

The alignment of the object roles from the collaboration graphs also shows the splitting of roles. The figure shows that the analysis of the role Supplier indicates that there are three different roles of supplier: one as the holder of data of the supplier, one as the actor who actually managing the items it supplies, one as the actor who actually builds up orders. To accomodate the last two roles, two new object roles appear: ManagingSupplier and OrderingSupplier.

A more interesting alignment is shown around the roles of a product item. Originally (see figure 6, in the ItemMgt partition, there are three different roles of product: Products as the data holder of the sellable item, Stock as the stock level invariant manipulated by the inventory manager, Supply as the original data holder given by the supplier. When the SupplierOrder and the InventoryMgt partitions are taken into account, it is clear that SupplierOrder partition needs the original Stock and Products roles, whereas the InventoryMgt only needs the original Products roles. Therefore, further management of the item roles is needed, as shown in figure 7. The common responsibilities, the general data holders, are assigned to the Product role. This role has three other role subtypes: Stock (as in original Stock role), Supply (as in original Supply role) and Merchandise which now plays as the sellable products. Now, the other partitions use Merchandise, instead of the original Products.

3. When the object roles have been aligned, the existance of the object roles in each partition is revisited. This is to avoid having object roles whose responsibilities are incompatible with the purpose of the partitions. The technique for this object alignment is adopted from the development of business component proposed by Eeles [3].

   Eeles' technique proposes encapsulated business components (units of delivery) by con-

sidering business classes found in the requirements. This is done by identifying *focus classes* and then identifying a *focus group* for each focus class discovered. A focus class, together with other associated classes, forms a cohesive unit known as a focus group.These focus groups are the basis to define the distributable component classes and eventually becomes business components.

Focus classes are the specification of focus objects. Focus objects are determined by examining their characteristics. According to Eeles, a focus object:

- is meaningful in the problem domain and is typically large-grained.
- represents an abstraction that may be applicable outside the domain currently under construction.
- represents a fundamental aspect of the object model.
- is an instance of a concrete class.
- is not usually aggregated, although it can aggregate other objects.
- represent a concept which, as it is refined through the development lifecycle, will be network visible (distributable).

Using the guideline above, an object role is examined, to determine if it is a focus object or not.

4. Once an object is determined to be a focus object (role), its existence in the partition is examined. At this stage, each partition is a potential distributable component, hence each partition is a focus group. To judge whether the existence of a focus object in a partition is appropriate or not, Eeles' guidelines in defining focus groups are used.

   Eeles suggested that a focus group:

   - contains exactly one focus object
   - includes any classes that represent attributes of the focus object.
   - includes any objects that the focus object aggregates.
   - includes other objects associated with the focus object that are not themselves focus objects.
   - includes any classes from which the class of the focus object is inherited (as long as those superclass are not focus objects)

   Taking on the guidelines above, there should only be one focus object role in each partition. If there are more than one focus object role found in a partition, then it should be decided which focus object role is the most appropriate focus object role for that partition's purposes. This results in either of two situations:

   (a) A partition is split up and a new partition is invented

   The definition of the new partition is also based on the guidelines in defining focus groups suggested by Eeles.

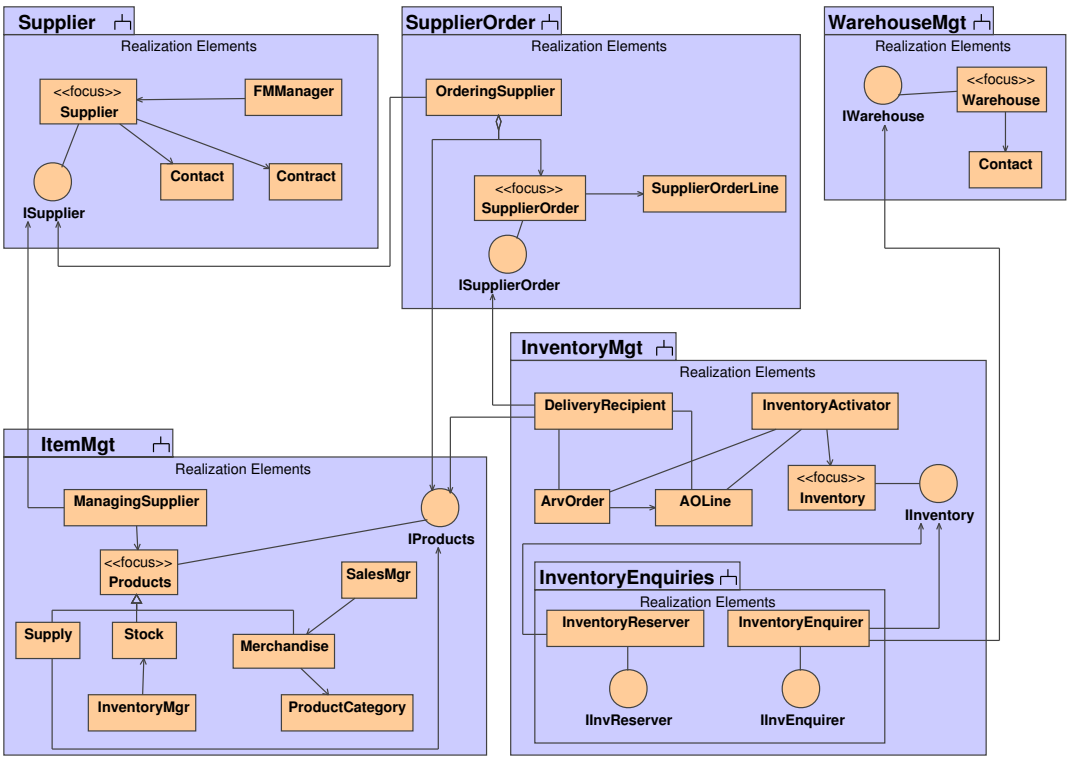

Figure 8: Distributable Components Candidates

Example 12:
Figure 8 shows that the examination on the InventoryEnquiries partition reveals the need to define new partition WarehouseMgt. This is because the original InventoryEnquiries had InventoryEnquirer as an object role, but then the Warehouse object role was found to be a focus object role as well. Since the responsibilities of the Warehouse object role is not compatible with the purpose of the original InventoryEnquiries, and that the Warehouse object role represents a business concept on its own, then the new WarehouseMgt partition is defined to handle the management of Warehouse objects. Further look into the Warehouse object shows that Contact objects are associated with the Warehouse objects. Although the responsibilities of this Contact object role is the identical with that of the Supplier partition, in this case, the Contact object role can be replicated as the objects in both partitions are disjoint.

(b) The focus object has already been defined as the proper focus object of other partitions

The client-server relationship between multiple focus object roles indicates the client-server relationship between one partition and another. Let FOR1 be a focus object in partition PAR1 and FOR2 is also a focus object found in PAR1. A client server relationship between FOR1 and FOR2 exist when FOR1 needs services from FOR2. If FOR2 has already been defined as or is judged to be the proper focus object of partition PAR2, then a client-server relationship between PAR1 and PAR2 exists. FOR2 in PAR1 becomes the interface to the proper focus object FOR2 in PAR2 to

which FOR1 is actually linked. These interfaces publicly declare the services offered by the proper focus object with the help of other object roles it is associated with.

Apart from the guidelines given by Eeles, the decision on whether a focus object role is appropriate for a particular partition is also made on the basis of compatibility of the purposes of the focus object role and the partition as a whole.

> Example 13:
> Figure 8 shows the result of the transformation of the object roles. Some of the object roles, like Supplier and Inventory, are transformed into the proper focus object roles and interfaces in their appropriate partitions, for example Supplier and InventoryMgt. Other object roles that need their services link to these interfaces.

Resulting Context

The design of the partitions has been adjusted to accomodate the refinement of the possible intersectional responsibilities of the classes. There might still be some replicated classes, but they have been designed to adapt to such replication. The process might also result in the invention of new partitions.

Consequences

Benefits

- This pattern treats the similarities between objects in a stepwise refinement.
- This pattern gives a guidance in deciding the home of each object specification.

Liabilities

- Revisiting each of the object specifications and deciding whether they are potentially be reconciled or not the could be a tedious work, as there may be very subtle differences between them.

Rationale

This patterns deals more with object roles than classes because system modelling at this stage prefer roles of objects over classes [13]. The term *class* in object modelling is defined as a declaration of a collection of methods, operations, and attributes. Thus, it defines the common implementation over a group of objects. The term role was introduced to encompass common purpose in a structure of collaborating objects within a specific *area of concerns* (the term used by OOram as business functions or phenomena). Roles are used to elicit object types. Object types is the specification of a set of objects with identical externally observable properties. Eventually, the classification of objects is merely the implementation of the external behaviour of the objects, i.e. object types, which in particular encodes the roles and the responsibilities of the objects.

Eeles' technique is adopted for it is judged to have the right viewpoint and means in determining the distributable components of the application. The main aim of Eeles' original

technique is to discover and to design business components in the context of enterprise component framework, which is inherently distributed. Eeles distinguished three types of components that the technique is primarily concerned with: embedded class, distributable components, and business components. Embedded class denotes the classes that do not have *network visibility* in a distributed setting. That means the instances of these classes cannot be addressed outside the process in which they reside. Conversely, in a distributed environment, there are some objects which have network visibility (i.e. distributable objects), such as DCOM object. These objects are instances of *distributable component* classes. A distributable component class is constructed from one or more embedded classes. These distributable components are used to implement business components. *A business component is the realisation of an autonomous business concept* [3].

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 found in different partitions. This refinement makes sure that resulting partitions are potentially true components which have distinct capabilities, services (interfaces), and boundaries.

Related Patterns

There are several ways of representing multiple object roles as suggested by Fowler [14]:

- The first case would be to lump all the roles into one object type (*Single Role Type).*
- If there were no common behavior or there were only few but negligible (in a sense that they do not cause integrity problems) common behavior, then separate role types can be declared (*Separate Role Type).*
- If separating the roles, hence duplicating the common behavior into different role types, is likely to cause integrity problems, then further object role management is needed. The most common solution is to declare the variant roles as the subtype of the main role (*Role Subtype*).
- Alternatively each variant is declared as separate role type and the common behavior is defined in the host object. The host object would consult the variant role objects when a client ask for a role's feature (*Role Object*).
- If the role is significant only when the object is related to another object, then the role can be expressed as the relationship between those objects (*Role Relationship*).

Next

The PUT TWO AND TWO TOGETHER pattern, the *Partitioning Pattern Language* and the BACK TO BASICS pattern.

Process Pattern 5: Put Two and Two Together

Problem

Based on the services requirement, the design of communication between any two communicating partitions should be performed. What are the best types of communication to be used? What are the consequences of implementing such communication on each of the partitions in terms of concurrency?

Context

Using the object-oriented method, the dynamics of objects within a partition has been explored. The details of the services needed and offered by the objects have been identified. From this, the services needed and offered by a partition as a whole could also be derived. One way of doing this was shown in the Process Pattern 4: PUT DOWN ROOTS. If this is the case, then each partition would have had the basic properties of components. These components can potentially be distributed over the network in which the system resides. However, to allow these components to communicate with each other in such a distributed setting, further design is needed to specify them as distributed components.

Forces

- The implementation of the partitions is constrained by the distributed location. Some partitions could become true distributable components, which are allocated to an identified particular location type. These components would be implemented as heavyweight processes at runtime. Other partitions would become components, whose services are invoked by the distributable components.

- Distribution issues, such that location and latency, need to be considered when designing the communication between any two distributable components. In addressing these issues, the coupling between the two components should be kept minimal.

- The objects or data being passed around may not necessarily be a simple data. Some advanced queries over one or more components may be performed to actually produce the required data. The design of the inter-component communication should minimise the size of the data passed between any two distributable components and should also take into account the creation, deletion and migration of the distributed objects that are involved.

- Essentially, the distributed object system, on which the application run, provides facilities to invoke methods of remote objects (Remote Method Invocation). This facilities are synchronous requests. If the requests can be made non-synchronous, the dependencies between any two partitions could be lowered, hence achieving minimal coupling.

- Threads (lightweight processes) might be needed to help a component in carrying out the communication with other components. In this case, the concurrency issues, such that types of communication requests between threads and resource sharing, should be addressed in designing the component.

Solution

Specify each partition to become a PAC agent by appyling Presentation-Abstraction-Control (PAC) pattern. Based on the services requirement, decide for each agent whether it should be a client, a server or both. Also decide whether the service invocations requires remote invocation or not. Then specify the appropriate facilities to reify each client-server connection required for each service. The specification is also informed by the distributable component framework to be used in the software construction.

Implementation

As indicated in process pattern 4: PUT DOWN ROOTS, each partition is essentially a candidate for distributable components. Here, such assertion is reinforced by applying the Presentation-Abstraction-Control (PAC) pattern on each partition. By applying this pattern, the current partitions are refined to ensure that each partition becomes a proper PAC agent. Proper PAC agents become components.

In parallel to the implementation steps of the PAC pattern [11], defining distributable components also involves defining several things:

1. The hierarchy of PAC agents as prescribed by the PAC patterns is defined by referring back to the location constraints and the grouping constraints of the application. This should have been established earlier, for example in process pattern 1: DISSECTION. As a result, at least one hierarchical structure of PAC agents could be determined. The PAC hierarchy helps to pinpoint which partitions become the actual distributable components and which partitions become ordinary component. A distributable component would potentially be allocated to a location type, has a network visibility, hence would be inlvolved in remote service invocations.

2. The services offered and needed by a component indicate the degree of coupling between components, which should be kept minimal. According to Constantine and Yourdon [15] the weakest coupling is data coupling. Therefore, in aiming for the minimal coupling between any two components, all the client-server relationship between any two components are examined to determine what data are to be passed between them and to minimise the size of these data.

    - Each client-server relationship between two components is examined to find out what data are passed. This is to determine the kinds of communication that are needed to implement this client-server relationship. The objective is to reduce the cost of communication by minimising the synchronous communication between two components. Therefore, any chance to avoid instant link between two components at runtime to pass data is highly sought after.

        Example 14:
        Here are examples of scenario where the passed data motivate the types of communication between two components:

        - In figure 8 the SupplierOrder component requires the ISupplier interface from the Supplier component. Further examination of these client-server relationship reveals that the only information that is required from ISupplier is the identification of the Supplier that makes

the order. The actual link at runtime between the SupplierOrder component and the Supplier component is virtually none, because the information could easily be obtained by reading the supplier database. In this case reading from a database is sufficient because the data are not changed frequently. Therefore the communication between these two components does not need to be synchronous, which also means that the coupling is reduced to minimal.

- The client-server relationship between the SupplierOrder component and the DeliveryRecipient component (discovered at point 1 above) seems very highly coupled. However, further examination reveals that in principles, the SupplierOrder component lets the DeliveryRecipient know that there was an order made so that the DeliveryRecipient can validate the actual goods that are delivered against the original order. Since there would be a time lag betwen the making of order and the delivery of goods, then an instant communication is not needed. Therefore there is no need for a link at runtime as well. The solution using a database as the previous case can be applied here. However, since there are several different warehouses to which the order would go, it might be neater to use messaging technique to send asynchronous messages to these warehouses. Using these messages, the warehouses are notified that an order has been made. The notification allows each warehouse to prepare itself to receive goods delivery without having to establish direct link with the message sender. Using one way messaging communication means that the coupling is reduced as well.
- A different scenario applies to the client-server relationship between the SupplierOrder and the InventoryEnquiry. Further examination of the relationship reveals that the SupplierOrder needs the information about the current stock level provided by the InventoryEnquiry to resume its own process of making up the order. This means that the actual link between these two partitions exists at runtime, which also means that there should be a synchronous communication between them.

- To minimise amount of data to be passed around, clients only received the data they need. These data might be only a part of the state of a server object or they might be as the result of more complex computation in the server component. Special wrapper objects are created to encapsulate the data. These wrapper objects are transmitted, rather than the whole objects. The Data Transfer Object pattern [16] is used to realise this data wrapper object and the Remote Facade pattern [16] is also used to conceal a complex invocation and minimise a number of call. The introduction of new objects would reconfigure the structure of classes in the component in which the data resides to conform to the PAC pattern. This data handling facility of a partition is designed as part of the Control component of the corresponding agent.

3. Decide the type of requests employed by the communication of all client-server relationships. In general there are two type of requests: synchronous and non-synchronous [17]. Non-synchronous requrest includes: one way request, deferred synchronous request and asynchronous requests. Distributed object systems by default prescribe applications to use the synchronous communication, enforcing a strong coupling between any two

distributed objects. To minimise coupling, synchronous requests should be made non-synchronous if at all possible. This would need a higher-level application design technique.

The *Partitioning Pattern Language* (see A) suggests the patterns to be applied to implement the communication design and, as a consequence from such design, the design of any concurrency within the components. The design patterns give proven solutions concerning the concurrency issues that may occur, particularly in implementing the non-synchronous requests.

The concurrency design implies that the design of the controller objects and active objects are specified for each distributable component. Controller objects serve as an operation manager among several objects within the component and do not necessarily have their own thread of control. Controller objects mediate objects of the Abstraction component and objects of the Presentation component of each agent. 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. As indicated above, the design of such active objects takes on the Active Object pattern. This communication and concurrency facility is also designed as part of the Control components of the involved agents.

4. The distributable component framework to be used in the software construction provides the rules to be applied and the means in implementing the agents as the actual distributable components.

Resulting Context

The actual application components have been designed. The internal elements of the components have also been designed to support the inter-component communications. This includes the implementation of the required concurrency within a component as a result of such an inter-component communication.

Consequences

Benefit

- This pattern allows stepwise refinement to the detailed distributable component design. This also means that the respective distributable objects and distributed objects are also carefully designed:
  - The selection of distributed components is guided.
  - The design of communications between any two components, i.e. distributable objects, is decided carefully.
  - The design of distributed objects that handles the communications is guided by the existing design patterns.
  - The design of concurrency is clearly driven by the design of communications and also guided by the existing design patterns.

- The design of communication and concurrency is kept in accordance with the properties of the application architecture.

Liability

- When designing the details of communication and concurrency, a PAC agent might become so complex and big. It might be further decomposed, leading to new distributable objects. However, there is no clear detection technique to know when does this happen.

Rationale

The definition of an agent given by the Presentation-Abstraction-Control pattern [11] is coincident with the concept of component. Therefore, expressing the partitions as PAC agents allows them to become components.

During the top-down partitioning, the partitions have been designed 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. This includes the design of the detailed inter-component interaction, particularly concerning the distribution and concurrency issues. Except for Gomaa's method [18], existing object-oriented methods fail to address distribution and concurrency issues formally. In practice however, designing communication and concurrency facilities are imperative in developing distributed object applications. This process pattern captures such activity and does it in respect of the required properties of the application structure.

Next

The design of the 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. The *Partitioning Pattern Language* also keeps being consulted.

3 Concluding Remarks

The five process patterns that were presented in this paper are novel, in the sense that they do more than describe the management of the process, like most other process patterns. The *Partitioning Process Patterns* represent identifiable patterns of the partitioning technique, which can be applied in pattern-directed ways, rather than in a single monolithic order. These patterns show a step-by-step technique in decomposing a system and defining the solution as a DOA. The patterns capture the impromptu activity which is usually performed by experts spontaneously, proficiently, and extemporaneously in partitioning a DOA.

The *partitioning process patterns* are complementary to the *partitioning pattern language* (Appendix A) but they are not interchangeable. The pattern language gives a choice of patterns to be applied in the distributed object application design. The language however, does not provide the sequence in which these patterns would be best applied. The *partitioning process patterns* provide an alternative of sequence of application in which the architectural styles, patterns or design patterns are carefully selected to address a specific design aspect of the DOA.

References

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

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

[3] P. Eeles, "Business Component Development," in *Software Architectures: Advances and Applications* (L. Barroca, J. Hall, and P. Hall, eds.), ch. 3, pp. 27–59, Springer, 2000.

[4] G. Low and G. Rasmussen, "Partitioning and Allocation of Objects in Distributed Application Development," *Journal of Research and Practice in Information Technology*, vol. 32, pp. 75–106, May 2000.

[5] A. J. Simons, "Object Discovery - A Process For Developing Medium-sized Applications." Tutorial 14, ECOOP 1998 Tutorials, Brussels, 1998. AITO/ACM.

[6] A. R. Silva, F. Hayes, F. Mota, N. Torres, and P. 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.

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

[8] P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, pp. 42–50, November 1995.

[9] K. Marquardt, "Patterns for the Treatment of System Dependencies," in *Proceedings of EuroPLoP 2002*, 2002.

[10] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan, "Architectural Styles, Design Patterns, and Objects," *IEEE Software*, vol. 14, pp. 43–52, January 1997.

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

[12] J. Noble, "Towards a Pattern Language for Object-Oriented Design," in *28th International Conference on Technology of Object-Oriented Languages and Systems - Pacific (TOOLS 28)*, 1998.

[13] T. Reenskaug, P. Wold, and O. A. Lehne, *Working with Objects: The OOram Software Engineering Method*. Prentice Hall, 1996.

[14] M. Fowler, "Dealing with roles." http://www.martinfowler.com, July 1997. last access: 17 May 2004.

[15] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, 1979.

[16] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

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

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

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

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

[21] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, 2004.

[22] F. Buschmann and K. Henney, "A Distributed Computing Pattern Language," in *Proceedings of EuroPLoP 2003*, 2003.

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

Appendices

A Partitioning Pattern Language

The *Partitioning Pattern Language* contains patterns that form a pattern language for the design of distributed object applications (DOAs). The language does not define any new pattern. Instead, it incorporates existing patterns from the existing collections. The inclusion of patterns into the language is judged according to their applicability in a typical DOA design. The main sources of patterns to be included are the Gang of Four patterns [19], the Pattern-Oriented Software Architecture (POSA) series [11, 20], and the work of Fowler [16] and Hohpe and Woolf [21]. The *partitioning pattern language* also includes the work of Buschmann and Henney [22], who worked on a pattern language for a distributed computing infrastructure.

The *partitioning pattern language* is not intended to replace the original description of the constituent patterns. Rather, the pattern language puts these patterns within a different perspective in applying these patterns. The name and the implementation details of these patterns are as they were originally described. When a pattern is adapted for different purposes than those of the original description, a new interpretation is given if necessary without detailing the implementation.

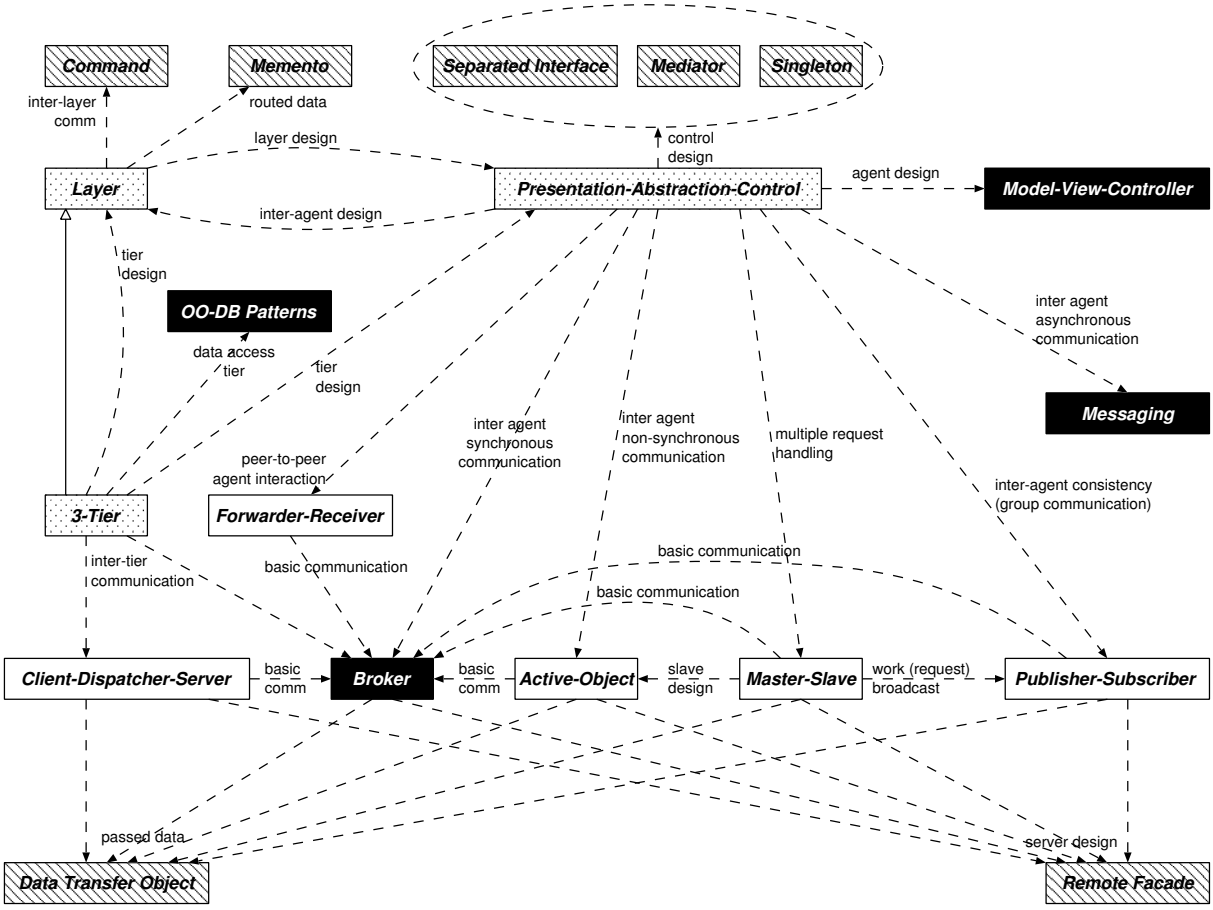

Figure 9: *Partitioning Pattern Language*

The *partitioning pattern language* highlights the selected pattern into three levels of ab-

straction: 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 DOAs. Design patterns address more general DOAs design problems, which are not necessarily architectural.

Figure 9 provides the structure of the *partitioning pattern language* using UML notation. The patterns are represented as classes. The background shading of the classes in the figure distinguishes the level of abstraction of the patterns. Patterns showed with dotted background shading are stereotyped as **architectural styles**. Patterns showed with white background are stereotyped as **architectural patterns**. Patterns showed with diagonal cross-hatching are stereotyped as **design patterns**. Patterns with black background are sterotyped as **architectural patterns** which embody the **frameworks** that can be applied directly to the applications. The application of a framework should conform to the associated architectural pattern. Take Broker for example. It is actually an architectural pattern [11] that prescribes a structure for distributed software systems. The pattern decouples components that interact by remote service invocations. This architectural pattern is 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 classes, create the client and the server stub, and use the broker provided by the framework. Therefore, the utilisation of this framework requires the design to conform with the Broker pattern.

As a pattern language, there is an implicit process to unfold the application design in the *partitioning pattern language*. This implicit process is made explicit by the *partitioning process patterns* described in main body of this paper.

B 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 [18] and Wijegunaratne and Fernandez [1]. The FoodMovers case study [23] 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.

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 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.