UMIACS-TR-91-24
CS-TR-2607

March 1991

A Reference Architecture for the Component Factory*

Victor R. Basili and Gianluigi Caldiera
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742, U.S.A.

Giovanni Cantone
Universita' di Napoli
Naples, Italy

Abstract

Software reuse can be achieved through an organization that focuses on utilization of life cycle products from previous developments. The component factory is both an example of the more general concepts of experience and domain factory and an organizational unit worth being considered independently. The critical features of such an organization are flexibility and continuous improvement. In order to achieve these features we can represent the architecture of the factory at different levels of abstraction and define a reference architecture from which specific architectures can be derived by instantiation. A reference architecture is an implementation and organization independent representation of the component factory and its environment. The paper outlines this reference architecture, discusses the instantiation process and presents some examples of specific architectures comparing them in the framework of the reference model.

*Research for this study was supported in part by NASA (Grant NSG-5123), by ONR (Grant NOOO14-87-K-0307) and by Italian CNR (Grant 89.00052.69).

1. INTRODUCTION

The issue of productivity and quality is becoming critical for the software industry at its current level of maturity. Software projects are requested to do more with less resources: deliver the required systems faster, reduce turn-around time in maintenance, increase performance reliability and security of systems. All of this implies radical changes to the way software is produced today. A straightforward solution to the problem of increasing quality and productivity can be synthesized in three goals: improve the effectiveness of the process, reduce the amount of rework, and reuse life cycle products.

The production of software using reusable components is a significant step forward for all three of those goals. The idea is to use pre-existing well designed, tested and documented elements as building blocks of programs, amplifying the programming capabilities, reducing the amount of work needed on new programs and systems, and achieving a better overall control over the production process and the quality of its products.

The possibility of assembling programs and systems from modular software units "classified by precision, robustness, time-space performance, size limits, and binding time of parameters" [McIlroy 1969] has been suggested from the beginnings of software engineering. However it has never acquired real momentum in industrial environments and software projects, despite the large amount of informal reuse already there. Reuse is a very simple concept: it means use the same thing more than once. But as far as software is concerned, it is difficult to define what is an object by itself, in isolation from its context [Freeman 1983]. We have programs, parts of programs, specifications, requirements, architectures, test cases and plans, all related to each other. Reuse of each software object implies the concurrent reuse of the objects associated with it, with a fair amount of informal information traveling with the objects. This means we need to reuse more than code. Software objects and their relationships incorporate a large amount of experience from past development activities: it is the reuse of this experience that needs to be fully incorporated into the production process of software and that makes it possible to reuse software objects [Basili and Rombach 1991].

Problems in achieving higher levels of reuse are the inability to package experience in a readily available way, to recognize which experience is appropriate for reuse, and to integrate reuse activities into the software development process. Reuse is assumed to take place totally within the context of the project development. This is difficult because the project focus is the delivery of the system; packaging of reusable experience can be, at best, a secondary focus of the project. Besides, project personnel are not always in the best position to recognize which pieces of experience are appropriate for other projects. Finally, existing process models are not defined to support and to take advantage of reuse, much less to create reusable experience. They tend to

be rigidly deterministic where, clearly, multiple process models are necessary for reusing experience and creating packaged experience for reuse.

In order to practice reuse in an effective way, an organization is needed whose main focus is to make reuse easy and effective. This implies a significant cultural change in the software industry, from a project-based frame of mind centered on the ideas and experience of project designers, to a corporate-wide one, where a portion of those ideas and experience becomes a permanent corporate asset, independent from the people who originate it. This cultural change will probably happen slowly, and a way to facilitate it is to provide an organization that is flexible enough to accept this evolution. Two characteristics stand out in this context

- *Flexibility*: the organization must be able to change its configuration without a negative impact on its performance, incrementally gaining control over the main factors affecting production.

- *Continuous improvement* (the Japanese "kaizen"): the organization must be able to learn from its own experience and to evolve towards higher levels of quality building competencies and reusing them.

This paper will present some ideas on how to design the production of software using reusable components. In particular we will show the effectiveness of a representation of the organization with different levels of abstraction in order to achieve the desired flexibility. This will lead us to the concept of reference architecture that will provide a representation of the organization as a collection of interacting parts, each one independent of the way the other ones perform their task. In this framework, methods and tools can be changed inside each one of those independent "development islands" without the need for a change in the other ones. We will outline a possible reference architecture and illustrate it with some examples.

After having automated other organizations' business and production, the software industry is facing today the problem of a substantial automation of its activities. Without mechanically translating concepts that are pertinent to very different production environments, we can say that flexible manufacturing systems combine many desirable features: modular architecture, integration of heterogeneous methods and tools, configuration and reconfiguration capabilities, wide automation under human control. Flexible manufacturing comes to age in the software industry through the definition of integrated software engineering environments if they are based on the concepts of flexibility and continuous improvement we have mentioned earlier.

The next two sections of this paper will present an organizational framework designed to make reuse happen in the most effective way. Sections 4 and 5 will discuss the representation that we propose for this framework, that uses different levels of abstraction in order to obtain a flexible and evolutionary organizational design. Section 6 will outline our methodology for deriving a particular environment from the general one, and section 7 will illustrate this derivation with some theoretical and actual examples.

2. A REUSE-ORIENTED ORGANIZATION

In order to address the problems of reuse in a comprehensive way, Basili has proposed an organizational framework that separates the project specific activities from the reuse packaging activities, with process models for supporting each of the activities [Basili 1989]. The framework defines two separate organizations: a project organization and an experience factory.

One organization is project-oriented. Its goal is to deliver the systems required by the customer. It is called the *project organization*. The other organization, called *experience factory*, has the role of monitoring and analyzing project developments, developing and packaging experience for reuse in the form of knowledge, processes, tools and products, and supplying it to the project organization upon request.

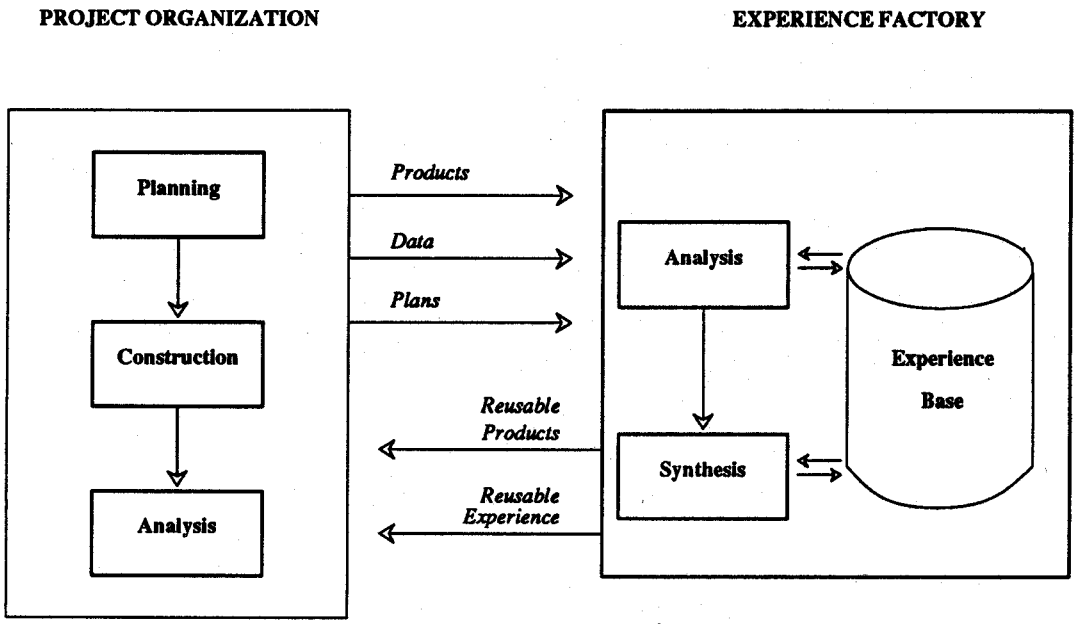
Each project in a project organization can choose its process model based upon the characteristics of the project, taking advantage of prior experience with the various process models provided by the experience factory. It can access information about prior system requirements and solutions, effective methods and tools and even available system components. Based upon access to this prior experience, the project can choose and tailor the best possible process, methods and tools. It can reuse prior products tailored to its needs.

The experience factory, conceptually represented in Figure 1, is a logical and/or physical organization that supports project developments by analyzing and synthesizing all kinds of experience, acting as a repository for such experience, and supplying that experience to various projects on demand. There are a variety of forms for packaged experience. There are, for instance,

- reusable products of the life cycle (i.e. the Ada package that creates and updates B-trees);
- equations defining the relationship between variables (e.g. Effort = a * Size ^{b});
- charts of data (e.g. Pareto chart of classes of software defects);
- management curves (e.g. product size growth over time with confidence levels);
- specific lessons learned associated with project types, phases and activities (e.g. in code inspections reading by step-wise abstraction is most effective for finding interface faults);
- models and algorithms specifying processes, methods and techniques (e.g. an SADT diagram defining Design Inspections with the reading technique as a variable dependent upon the focus and the reader perspective.

Figure 1

The Experience Factory



From this brief discussion we see that the organization of the experience factory can be divided into several sub-organizations, each one dedicated to processing a particular kind of experience.

A first level of subdivision of the experience factory can be based on the application domain [Neighbors 1989]: for each different domain we have a different *domain factory*, conceptually represented in Figure 2a, whose purpose is to define the process for producing applications within the domain, to implement the environment needed to support that process, to monitor and improve that environment and process. Examples of domains are Satellite Ground Support Software, Information Management Software for Insurance Companies, C^3 Systems, etc. The experience manipulated by the domain factory are the definition of an application domain and the experience relative to engineering within that specific domain [Caldiera 1991].

A further subdivision of the experience factory, that will be the object of the discussion in this paper, is the development and packaging of software components. This function is performed by an organization we call the *component factory*, conceptually represented in Figure 2b, which supplies software components to projects upon demand, and creates and maintains a repository of those components for future use. The experience manipulated by the component factory is the programming and application experience as it is embodied in requirements, specifications, designs, programs and associated documentation. *The software component produced and manipulated by the component factory is a collection of artifacts that provide the project organization with everything needed to integrate it in an application system and to provide life cycle support for this system.*

The separation of project organization and (experience, domain or component) factory is not as simple as it appears in our diagrams. Having one organization that designs and integrates only, and another one that develops and packages only, is an ideal picture that can present itself in many different variations. In many cases, for instance, some development will be performed in the project organization according to its needs. Therefore, the flows of data and products across the boundary are different from the ones we have shown in the figures of this section. One of the aims of this paper is to deal with this complexity, providing a rigorous framework for the representation of problems and solutions.

We will discuss these issue focusing on the concept of component factory, both as an example of the more general concept of experience factory and as an organizational unit worth being considered independently. On one hand, the component factory will be the framework used to define and discuss various organizational structures via our concept of reference architecture, which is applicable as well to the more general frameworks of domain and experience factory. On the other hand, the discussion will give us better insight into the role of the component factory as a milestone on the roadway to an industrialization of software development.

Figure 2a

The Domain Factory

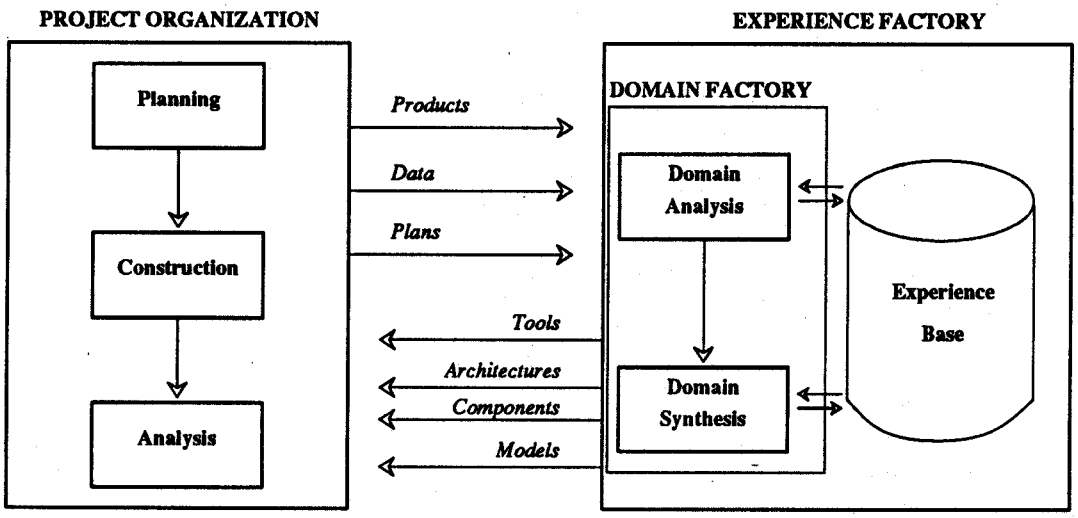
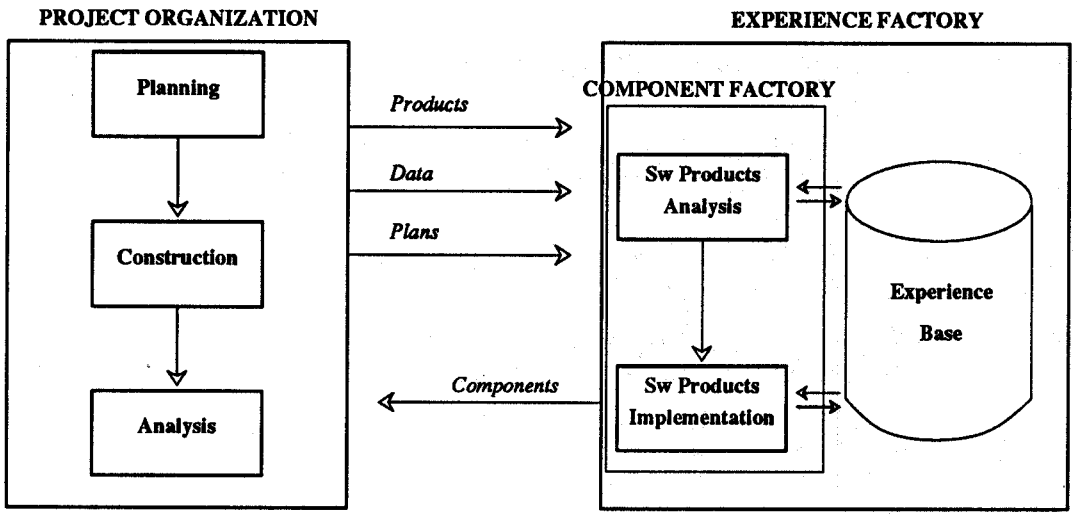


Figure 2b

The Component Factory



3. THE COMPONENT FACTORY

The concept of component factory is an extension and a redefinition of the concept of software factory, as it has evolved from the original meaning of integrated environment to the one of flexible software manufacturing [Cusumano 1989]. The major difference is that, while the software factory is thought of as an independent unit producing code and using an integrated production environment, the component factory handles every kind of code-related information and experience. The component factory is defined as a part of the experience factory, and therefore it is recognized that its potential benefits can be fully exploited only within this framework.

As noted earlier, a software component is any product in the software life cycle. We have

- Code components: objects implemented in some compilable, interpretable or executable language. This includes programs, subprograms, program fragments, macros, simple classes and objects, etc.

- Designs: objects representing function, structure and interfaces of software components (and collections) written in a language that can be formal, semi-formal, graphic, or natural. This includes structured design specifications, interface specifications, functional specifications, logical schemata, etc. In some cases designs are code components themselves.

- Collections of Code Component or Designs: objects obtained by aggregation of several functionally homogeneous code components or designs. This includes the libraries (mathematical, statistical, graphic, etc.), the collections of packages of Ada-like languages, the composite classes of object-oriented languages, the architectures of structured design techniques, etc.

- Documents: textual objects written in natural language with figures, tables and formulas to communicate information in some organized way. Hypertext objects can be, in many cases, considered in this class. This includes requirements documents, standards and policy documents, recommendations, lessons learned documents, reports from specific studies and analyses, etc.

The software component produced and maintained in the component factory is the *reusable software component* (RSC): it is a composite object made of a software component packaged with everything that is necessary to reuse and to maintain it in the future. This means very different things in different contexts, but it should include at least the code of the component, its functional specification, its context (i.e., borrowing the term from the Ada language, the list of the software components that are in some way associated with it), a full set of test cases, a classification according to a certain taxonomy and a reuser's manual [Caldiera and Basili 1990].

The project organization uses reusable software components to integrate them into the programs and the system that have been previously designed.

The capability of the component factory to make reuse happen in an efficient and reliable way is a critical element for the successful application of the reuse technology. Therefore the catalog of available components must be rich in order to reduce the chances of development from scratch, and look up must be easy.

There are three major groups of activities associated with the production of software through reusable software components

- *Use reusable software components*

  When the project organization needs a component described by a certain specification, the catalog of available components is searched:

  - if a ready-to-integrate component that matches the specification is found, the project organization uses it;
  - if a component that needs some adaptation in order to match the specification is found, the project organization uses it after the needed modifications have been applied to the component; organization;
  - if either no component is found that matches the specification or the needed adaptation is too large to be considered a simple modification of a pre-existing component, the component is ordered.

- *Develop and maintain reusable software components*

  A reusable software component enters into the production process for one of two reasons:

  - because it has been recognized as useful from a preliminary analysis of the application domains the project organizations deal with [Arango 1989];
  - because it was needed, it has been deemed "reusable", and wasn't available;

  Once the need for some component has been recognized or the component has been ordered there are three ways for the component to enter into the production process:

  - by direct development either from scratch or from pre-existing generic elementary processes [Joo 1990];
  - by direct procurement from an external source (an external repository, a vendor, etc.) [Tracz 1987];
  - by extraction and adaptation from existing programs and systems [Caldiera and Basili 1990].

In whatever way the reusable component has entered into the production process, it is

- adapted for further reuse by enhancement or generalization of its functions and structure, by fusion with other similar components, or by tailoring to expected needs;
- maintained to satisfy the evolving needs of the applications as identified by domain analysis;
- maintained to guarantee its correctness.

- *Collect and package experience from activities*

The activities of project organizations and component factory are recorded and processed in order to be preserved in a reusable form. This means they are packaged in units that are

- understandable by everybody interested in using them;
- either independent or explicitly declared and packaged with their dependencies;
- retrievable using some kind of search procedure.

The experience is formalized and translated into a *model*. Different kinds of models are produced in order to represent the knowledge the project organizations and the component factory have of products, processes, resources, quality factors, etc.:

- product models representing the observable characteristics of the software components through measures;
- process models representing the observable characteristics of the production process, its phases and states, and the measures that allow its control;
- resource and cost models representing the amount of resources allocated to, or estimated for, a set of activities and their distribution
- lessons-learned models representing the knowledge acquired from former projects and experiments.

This first summary analysis shows that the ability to anticipate future needs is critical to the efficient implementation of a reuse oriented paradigm, and to the work of the component factory in order to satisfy the requests coming from the project organization as soon as possible. Also critical is the ability to learn from past activities and to improve the service while providing it. Therefore, crucial to the component factory are creation and improvement of the models based on a methodology to systematize the learning and reuse process, and to make it more efficient. We will now briefly discuss our methodological approach to creation and improvement of models of experience.

The methodology to develop formal models of experience is provided by the *Goal/Question/Metric (GQM) approach* [Basili and Weiss 1984]. The GQM approach defines a model on three levels:

- Goal level: a goal is defined that characterizes a certain organizational intent or problem
- Question level: a set of questions is used to characterize in an operational way how a specific goal is going to be dealt with
- Metric level: a set of metrics is associated with every question in order to answer it in a quantitative way

Each GQM model has, therefore, a three-level hierarchical structure (Figure 3), but several goals can use the same question and the underlying set of metrics, and several questions can share some metrics.

The formal models developed using the GQM approach are characterized by a combination of

- Object: A model of the object that is represented by the GQM model, e.g. a process, a product, or any other kind of experience.
- Focus: A model of the particular characteristic or property of the object that the GQM model takes into account.
- Viewpoint: The perspective of the person or organization unit needing the information represented by the model.
- Purpose: The purpose of the GQM model, e.g. characterization, evaluation, prediction, motivation, improvement.
- Environment: The characteristics of the context where the analysis is performed.

The resulting GQM models are defined and constantly improved through use, learning from past experience, and translating this knowledge into changes to the model. This adds a new dimension to the characterization of a GQM model: its multiple evolving versions. The methodology that systematizes the learning and improving processes required to generate these versions is provided by the *Improvement Paradigm* [Basili 1984] and is outlined in the following steps (Figure 4b):

Figure 3

The Structure of a GQM Model

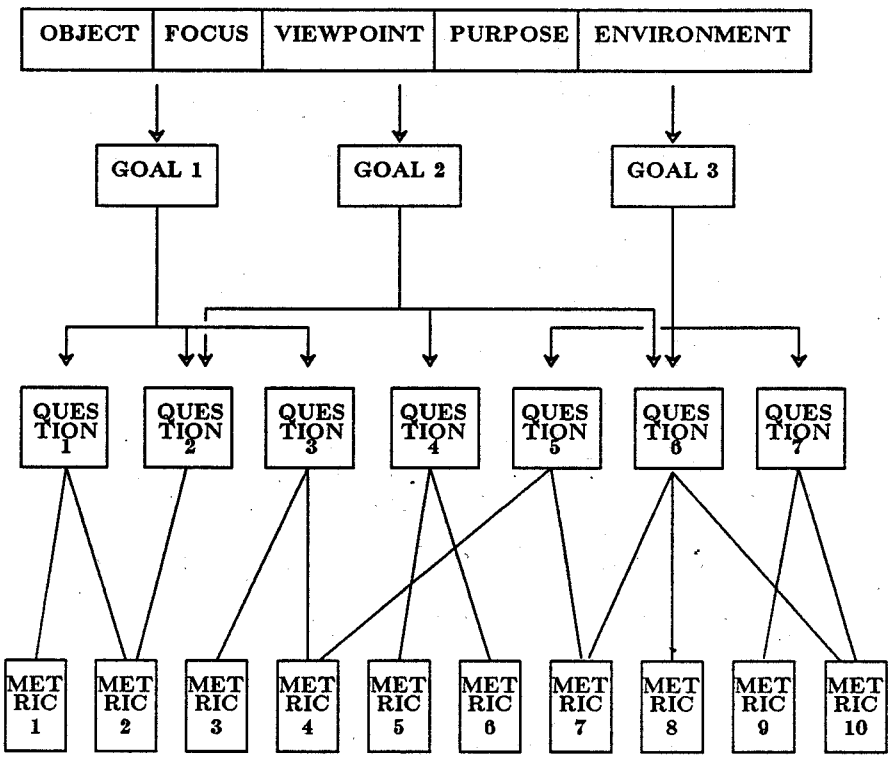


Figure 4a
The Deming Cycle

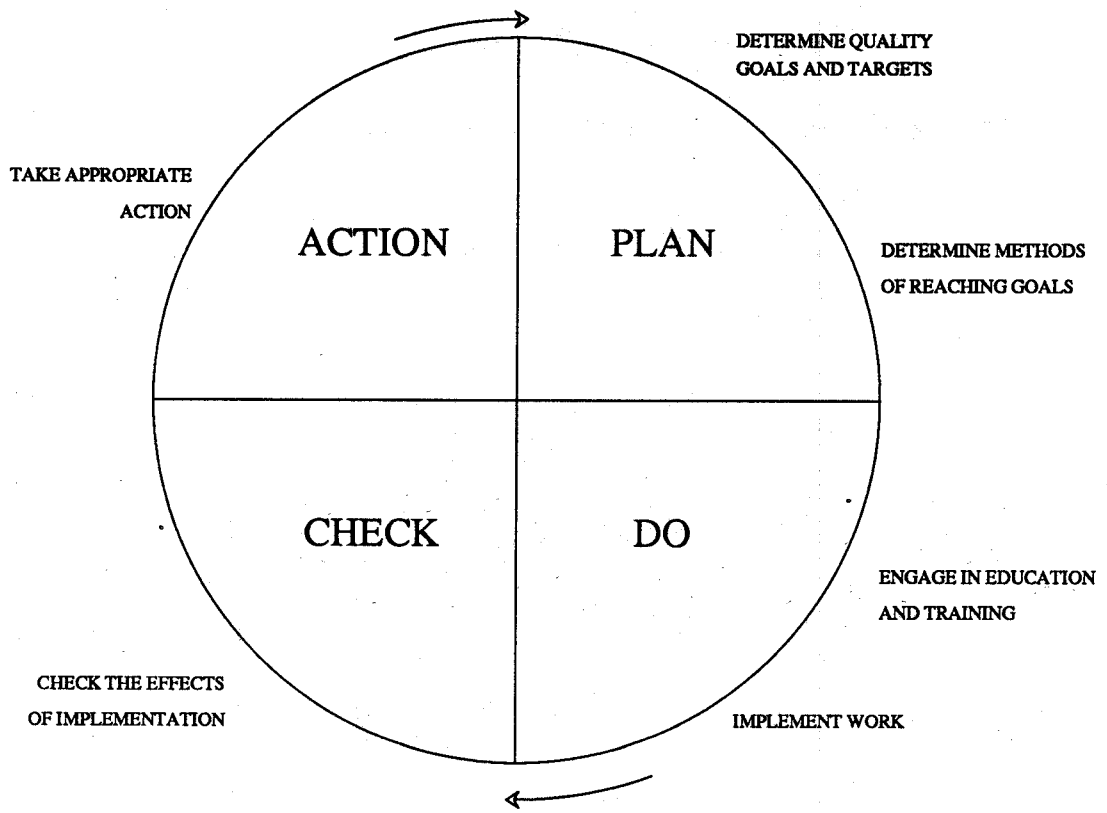
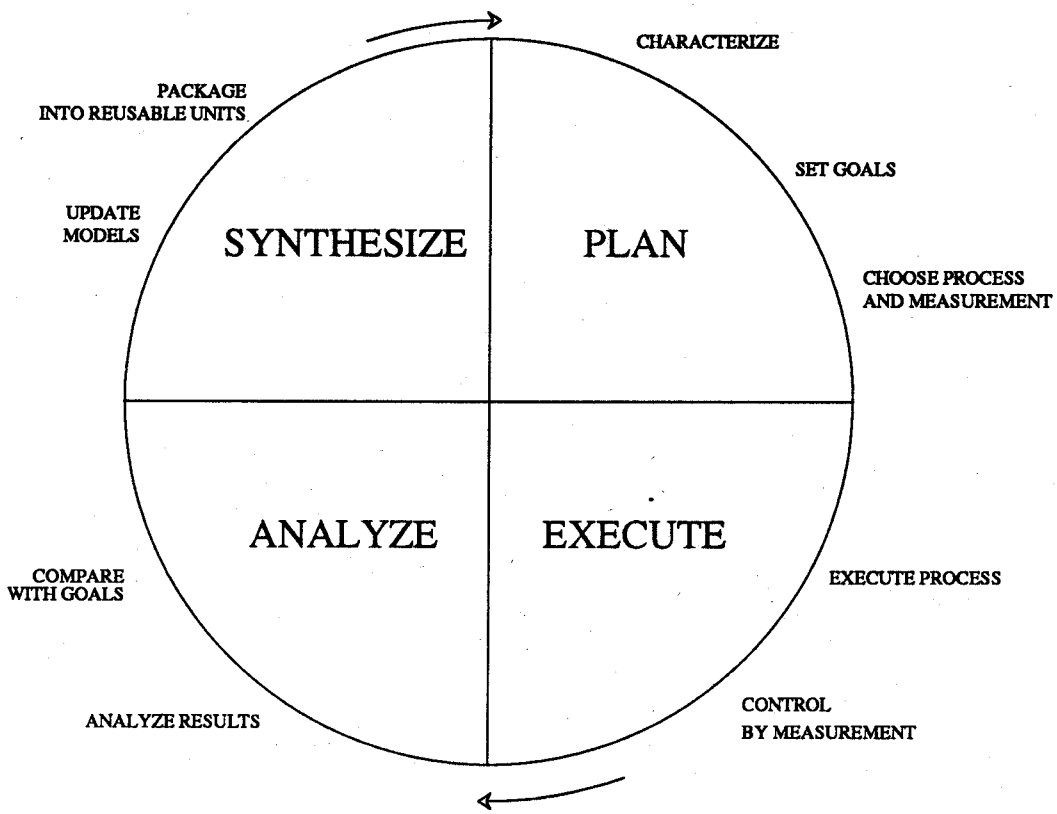


Figure 4b

The Improvement Paradigm



1. *Plan*

1.1 Characterize the activity and the environment in order to identify and isolate the relevant experience
1.2 Set the goals and refine them into a measurable form (this is done using the GQM approach)
1.3 Choose the execution process model, the supporting methods and tools, and the associated control measurement

2. *Execute* the process, control it, using the chosen measurement, and provide real-time feedback to the project organization

3. *Analyze* the results and compare them with the goals defined in the planning phase

4. *Synthesize*

4.1 Consolidate the results into updates to the formal models and to their relationships
4.2 Package the updated models into reusable units and store them for future reuse

The improvement paradigm is based upon the notion that improving the software process and product requires the continual accumulation of evaluated experiences in a form that can be effectively understood and modified into a repository of integrated models that can be accessed and modified to meet the current needs. It is an evolution of the famous Shewart-Deming Cycle [Deming 1986] Plan/Do/Check/Act (Figure 4a) in an environment in which formalization and institutionalization of experience are critical factors to the performance of the whole process.

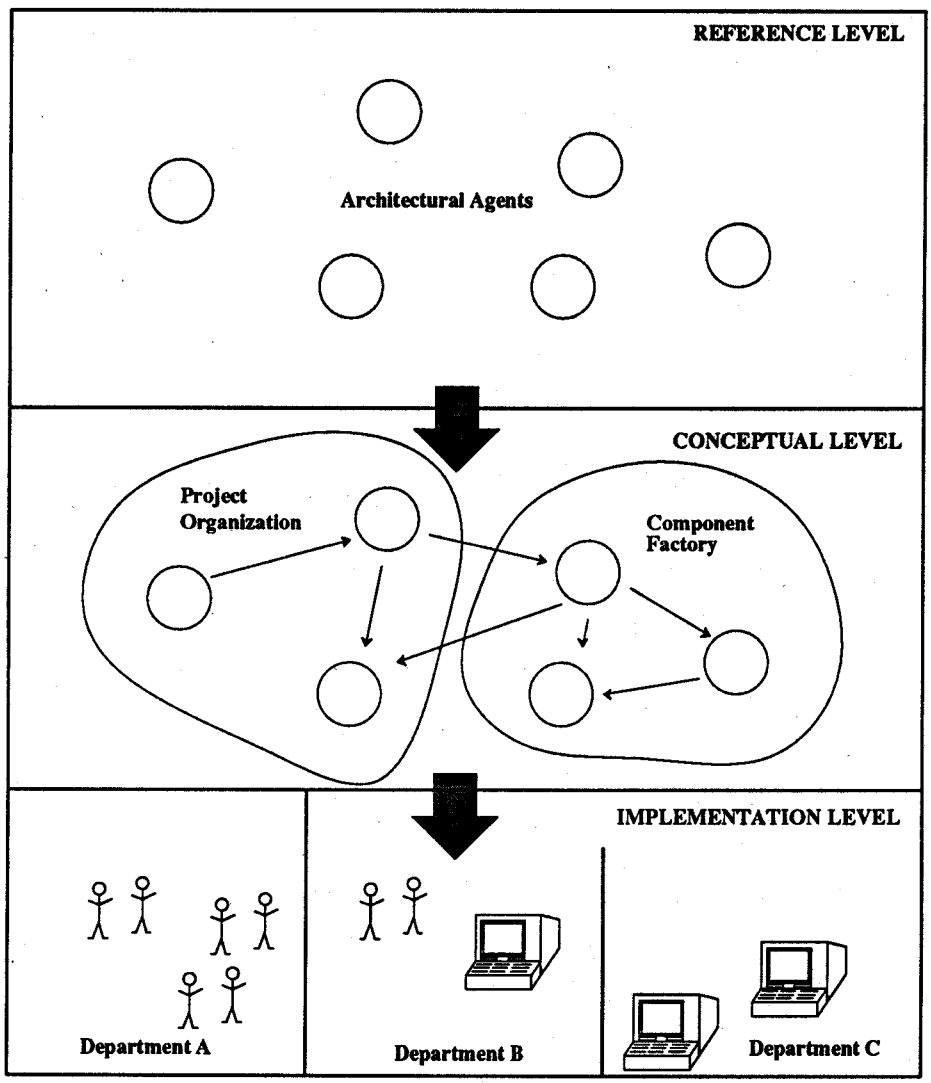
4. LEVELS OF REPRESENTATION OF A COMPONENT FACTORY

The experience factory and its specialization, the component factory, are necessarily very general organizational elements. Every environment has its characteristics and pursues certain goals by certain means different from every other one. Therefore we need different levels of abstraction in the description of the architecture of a component factory in order to introduce at the right level the specificity of each environment. The allocation of a function to an organizational unit is a first distinction, the actual implementation of some functions, for instance through automated tools is another distinction. However these choices are only variations of the paradigm of the component factory that can be captured using different levels of abstraction in representing the framework of the factory.

In this section we will discuss briefly the levels of abstraction that we want to use in order to represent the *architecture of a component factory* (Figure 5):

Figure 5

The Levels of Abstraction



- *Reference level*

At this first and more abstract level we represent the building blocks and the rules to connect them, that can be used to represent an architecture. This is not the description of a component factory but a modeling language for it. The basic building blocks are called *architectural agents* and represent the active elements performing tasks within the component factory or interacting with it. They exchange among each other software objects and messages.

Example:

In the last section we have discussed the activities associated with the production of software through reusable components and decided that a reusable component may enter into the production process by direct development, by procurement from an external source, or by extraction and adaptation from pre-existing software. Each one of these possibilities becomes a function that can be assigned to an active element of the reference level:

- an agent decides which reusable components are needed
- an agent develops reusable components
- an agent provides external off-the-shelf reusable components
- an agent extracts and re-engineers reusable components
- an agent manages the internal repository of reusable components

These agents have potential connections with other agents: for instance, the agent that develops reusable components is able to receive specifications from another agent, and the agent that decides which reusable components are needed is able to provide these specifications, but the actual connection is not specified at the reference level.

- *Conceptual level*

At this level we represent the interface of the architectural agents and the flows of data and control among them, and specify who communicates with whom, what is done in the component factory and what in the project organization. The boundary of the component factory, i.e. the line that separates it from the project organization, is defined at this level based on needs and characteristics of an organization, and can change with them.

Example:

In a possible conceptual architecture an agent that designs systems and orders components, located in the project organization, is connected with an agent that develops reusable components, located in the component factory. Specifications and designs are exchanged between these two agents.

In another conceptual architecture the agent that designs systems and orders components, still located in the project organization, communicates with an agent that coordinates the activities of component development and adaptation in the component factory. It is this agent that communicates with an agent that develops reusable components exchanging specifications and designs with it.

- *Implementation level*

At this level we define the actual implementation, both technical and organizational, of the agents and of their connections specified at conceptual level. We assign to them process and product models, synchronization and communication rules, appropriate performers (people or computers) and specify other implementation details. The mapping of the agents over the departments of the organization is included in the specifications provided at this level of abstraction.

Example:

The organization is partitioned into the following departments

- System Analysis and Deployment
- System Development
- Quality Assurance and Control
- Software Engineering Laboratory

In a possible organizational choice, the functions of component design are allocated to System Development, and the functions of system integration are divided between System Analysis and Deployment and Quality Assurance and Control. The Software Engineering Laboratory analyzes the production process, provides the models to control the activities and improves those models through experience. The Component Factory includes System Development and Software Engineering Laboratory. The process model for design is the iterative enhancement model: it starts from a group of kernel functions and delivers it to System Analysis and Deployment, and then it expands this group of functions in order to cover the whole set of requirements in successive makes of the system. The development is done in Ada using a language-sensitive structured editor and an interactive debugger.

Each one of these levels of abstraction can be divided into sub-levels by different operations applied to the agents or to their connections:

- functional decomposition: an agent is decomposed in a top-down fashion into more specific agents;

- connection decomposition or specialization: a connection between agents is decomposed into different pipelines, or the object types are refined, or synchronization rules are defined in greater detail;

The drawing of the boundary separating project organization and component factory at the conceptual level is motivated by the need for defining a formal interface between the two organizations, more formal than the one between the agents that lay inside each one. We can, for instance, imagine the boundary as a validation and verification checkpoint for the products that go across it. More generally, we can look at the couple project organization/component factory as at a user/server pair that can be used for quality measurement and improvement.

The next section will discuss in some detail the reference level and then we will deal with the problem of the instantiation of a generic architecture into a specific one.

5. THE REFERENCE ARCHITECTURE

While the reference level describes a generic architecture, the conceptual and implementation levels describe a specific one at different levels of detail. In order to obtain a specific architecture we can either design it from scratch or obtain it as an instantiation of a generic architecture. The problem with the architecture designed from scratch is that it doesn't give enough assurances it will be able to evolve in order to match the needs of the organization or to follow its evolution: designing from scratch is often focused on the needs and goals of the present organization and does not take into account the evolution of this organization.

The instantiation of a generic architecture is instead a technique that leads to more flexible architectures, because the adaptability is already in the abstraction. The specific architecture can be obtained from the abstract one through instantiation. It can be modified by altering the instantiation without changing the generic architecture, which is explicitly parametric and reusable. Besides, a generic architecture is a common denominator for comparing and evaluating different specific instantiations. It allows us to choose the architecture that is best suited for a particular organization.

The reference level provides this generic architecture. A *reference architecture for the component factory* is a description of the component factory in terms of its parts, structure and purpose, defining which parts might cooperate and to what purpose [Biemans 1986]. The interacting parts of the reference architecture are kept free from unnecessary linkages in order to keep the concerns separate and to leave this to the instantiation process. The genericness of the reference model is represented by the many possible ways of connecting those parts at the conceptual level for a specific architecture, and the many ways to map them into the implementation level.

The elements of the reference architecture, the architectural agents, are the components of the factory production process. We don't make any assumption about the way they are implemented: they can be a person, a group of people, a computer based system. The reference architecture specifies only their tasks and the necessary communication paths, leaving to particular instantiations their implementation and the specification of their nature. We can look

at the agents as islands in the component factory, whose nature and implementation can be changed according to the needs and the improvement goals of the organization, independent of other agents. This allows us to implement the tasks of an agent today with a group of people, and tomorrow with a computer-based system, introducing changes that do not impact the whole organization.

From our discussion in section 3 of the activities associated with the production of software through reusable components, we can derive a set of functional requirements for the architectural agents operating in the component factory or interact with it:

- Receive the specifications of a system and produce its design taking into account the information about existing reusable software components made available by the component factory. These functions specify the role of the *Designer* agent.

- Build the specified system according to the design using the reusable components made available by the component factory, and perform the system test in order to verify the conformance of the developed system with the requirements. These functions specify the role of the *Integrator* agent.

- Choose how to fulfill a given request for a reusable software component, based on the specification of the component, and on the information about existing reusable software components already available in the component factory. The choice between development from scratch and adaptation of an existing component is probably done according to a model of cost and time effectiveness, i.e. whether it is worth modifying an existing component or the modification would be so substantial that it is more convenient to develop a new component. These functions specify the role of the *Shopfloor Coordinator* agent.

- Develop a reusable software component according to a given specification. The development can be done from scratch or assembling pre-existing components and elementary processes. It includes the design of the component, its implementation and verification. These functions specify the role of the *Developer* agent.

- Modify a reusable software component that is "close enough" to the one described by a given specification. This can be done generalizing the existing component, tailoring it to meet a specific need, combining it with other components, etc. The modification activities include the analysis of the impact of a certain adaptation, its implementation and verification. These functions specify the role of the *Adapter* agent.

- Produce and update reusable components based on domain knowledge, extract reusable components from existing code, and generalize already existing reusable components into other reusable components. The main difference between these functions and the development ones is that these are asynchronous with respect the

production process in the project organization. These functions specify the role of the *Component Manipulator* agent.

- Develop formal models analyzing the experience developed in the component factory and in its interfaces with the project organizations. The experience that is collected and processed has different levels of formalization, according to the incremental perspective provided by the improvement paradigm. It starts with very simple models that are improved in a continuous way in order to fit better in the actual environment of the component factory. These functions specify the role of the *Experience Modeler* agent.

- Manage the collection of objects and information that is used by the component factory to store experience and products derived from its activities (experience base). In particular, this function includes the management of the repository of reusable software components, which is a subset of the experience base. Every agent, while performing its tasks, accesses the experience base either to use its contents or to record a log of its activity. The access control, the manipulation of objects and the search strategy to answer a request are the main experience base management functions. These functions specify the role of the *Experience Base Manager* agent.

- Supply commercial or public domain off-the-shelf reusable components that satisfy the specifications developed by the organization. This function specifies the role of an *External Repository Manager* agent

This list of agents represents a complete set of architectural agents that covers all the activities of the component factory and of the project organization. However, we could have defined the agents differently. For example, the reference level can be decomposed into several sub-levels, and the set of agents in the reference architecture presented here is one of these levels. Agents can be composed into larger ones or decomposed into more specific ones in a way that is very similar to the functional decomposition used in structured analysis. For instance, the Designer agent can be merged with the Shopfloor Coordinator obtaining an agent that specifies not only the components that are needed but also the way of obtaining them. Or, the Designer agent can be decomposed into a System Designer agent that performs the preliminary design of the system, and the Software Designer agent that performs the critical design.

It is also possible to expand the scope of the analysis, for instance by introducing the collection and analysis of requirements into the picture, and by specifying the agent that performs these functions in the reference architecture.

Besides a set of agents, the reference architecture contains a set of *architectural rules* that specify how the architectural agents can be configured and connected in the specific architectures derived from a reference architecture.

One set of rules deals with the presence and replication of the agents. The agents in the component factory can be unique or replicated: in the former case only one instance of the agent can be active, in the latter many instances of the agent are possibly active at the same time. For example: there might be two Designer agents that share the service provided by one Adapter agent.

Another set of rules deals with the connections between agents. The agents communicate with each other exchanging objects and experience at different levels of formalization, and cooperating towards the completion of certain tasks. This communication is realized through communication *ports*. A port is specified when we know what kind of objects can travel through that port. For instance: the Component Manipulator agent has a port through which it receives and returns reusable software components. There are data ports and control ports: the port through which the Designer agent receives the requirements for the system is a data port; the port through which it receives the process model to design the system is a control port. A port bundles several *channels*: each one is an elementary access point specifying the kinds of objects traveling through it and the direction. For instance: the port through which the Designer communicates, say, with the Shopfloor Coordinator has two channels: an output channel to send component specifications and an input channel to receive the requested reusable software components.

Ports can be mandatory, i.e. always present on the agent in one or more instances, or optional, i.e. possibly absent in certain implementations of the agent. For instance: the Developer agent has always a port through which it receives the specifications of the components it must develop and a port through which it returns the components it develops, these are mandatory ports. On the other hand, the Developer might or might not have a port to receive external off-the-shelf reusable components to be used in the development of the requested components. From the point of view of the reference architecture we don't make this choice, leaving it to the specific architecture.

We can represent our reference architecture using an Ada-like language in which the agents are task types that encompass port types in the way Ada task types encompass entries. This allows us to use the distinction between specification and body of the task type to defer the implementation of the agent to the specific architecture, and also to use Ada generics to represent certain abstractions that are specified at the conceptual level. In this representation, the architectural rules are declarative statements, incorporated in the definition of the agent they are applied to.

Without getting into the details of the representation, but to provide an example, we present a sample specification for the Developer agent

Generic

```
-- These variables are used as options in the ports that
-- are optional: their instantiation to "true" will imply
```

```
-- that the port is present.
        search_components is boolean;
        external_acquisition is boolean;
        recommended_components is boolean;

Task type Developer is

-- This is the port through which the agent receives the
-- specs and returns the components that have been
-- developed
Data port component_development
        (specs : in  component_specification,
        component : out  reusable_software_component)
end component_development;

-- This is the port for access to the internal components
-- repository: it is present if this access is permitted
Data port internal_components_acquisition
        options (search_components)
        (specs: out component_specification,
        components: in list_of_reusable_software_components)
        end internal_components_acquisition;

-- This is the port for access to external components
-- repositories: it is present if access is permitted;
-- there might be many instances of this port
-- corresponding to different repositories
Data port type external_components_acquisition
        options (external_acquisition)
        (specs: out component_specification,
        components: in list_of_reusable_software_components)
        end external_components_acquisition;

-- This port is used to receive components from another
-- agent (probably the Designer or the Shopfloor
-- Coordinator) and use them in the development of a new
-- component
Data port components_reception
        options (recommended_components)
        (components: in list_of_reusable_software_components)
        end components_reception;

-- The next two ports are used to interface with the
```

-- experience base
Data port activity_report
(current_process_data: **out** process_data,
current_product_data: **out** product_data,
current_resources_data: **out** resource_data)
end activity_report;

Control port models
(current_process_model: **in** process_model,
current_product_model: **in** product_model,
current_resources_model: **in** resource_model)
end models;

End Developer;

This *specification language* for the reference architecture is complemented by a *configuration language* whose purpose is to represent the choices made in order to instantiate the reference architecture into a specific conceptual architecture: presence and number of agents, presence and number of ports, connection of the ports.

In order to give an idea of the way this configuration language works, let's see a possible way of configuring a Developer agent. We do this by creating new tasks where

- it is specified what is the type of the agent that is being specified;
- the generic variables used as options in ports are instantiated in order to specify the presence of a port;
- instances of replicated ports are represented by declaring their port type;
- connections between agents are specified by a **connected with** statement in the specification of a port, declaring with which port of which agent it is connected.

The resulting task is conceptually represented by:

Task Developer_1 **is new**
Developer (search_components := true; external_acquisition := true; recommended_components := false)

Data port component_development
connected with Shopfloor_Coordinator_1.component_acquisition;

Data port internal_components acquisition

connected with Experience_Base_Manager_1.component_supply;

Data port repository_AAA **is** external_components_acquisition
connected with External_Repository_AAA_Manager.component_supply;

Data port repository_BBB **is** external_components_acquisition
connected with External_Repository_BBB_Manager.component_supply;

Data port activity_report
connected with Experience_Base_Manager_1.reports;

Control port models
connected with Experience_Base_Manager_1.models_out;

End Developer_1;

The implementation of the agents is specified by another language, the *implementation language*, that assigns to each task type a task body where **port** statements are associated with receive statements (like **accept** in Ada) having the same parameters.

6. INSTANTIATION OF THE REFERENCE ARCHITECTURE

The reference architecture has been defined as a collection of architectural agents and rules supported by an experience base managed by an agent. There are some degrees of freedom in this collection that are eliminated by choices made when the architecture is instantiated at the different levels of abstraction:

A. Choices at the conceptual level:

a. Boundary between component factory and project organization. In making this choice one tries to optimize reuse, on one hand, by incorporating more functions into the component factory, and to optimize customer service, on the other hand, by concentration of the appropriate activities in the project organization.

b. Presence of agents, number of agents of each type, fusion of several agents. This choice is about communication complexity: a large number of agents increases the complexity and the overhead due to communication, a small number of agents produces bottlenecks that would affect the whole organization.

c. Presence of ports and number of ports. As in case b., this choice deals with the communication complexity: a large number of ports increases the complexity of the activities of a single agent but reduces the impact of possible bottlenecks.

d. Interconnection of ports between different agents. This choice is about distribution of control: concentrating the control in a small group of agents makes planning easier, but serializes many activities that could be otherwise performed concurrently.

B. Choices at the implementation level:

a. Distribution of the agents over organizational units. This choice deals with the optimization of the already existing organization units and the smooth evolution to factory concepts. It takes into account the available resources and the historical roles of those units.

b. Implementation of the functions of the agents (the task bodies). In choosing algorithms, procedures, methods and tools one tries to achieve an organizational and technical profile that is correct, efficient and best suited to the overall mission by dealing with the available resources and technology.

Therefore, in order to design a specific component factory, we need to instantiate the reference architecture by an instantiation process based on the levels of abstraction introduced earlier (Figure 6). This instantiation process is embedded in the methodological framework of the improvement paradigm now applied to the specific architecture. The four steps of the paradigm, introduced previously in a different context, become in this context:

1. *Plan component factory*: the desired instantiation is designed based on the characteristics of the organization and on the goals to be achieved:

1.1 Characterize the activities and the environment of the current organization: production process, products, formal and informal models currently in use, software tools, standards, etc.;

1.2 Set goals and priorities for the introduction of the component factory and for its separation from the project organization: productivity, customer satisfaction, product maintenance, environment stability. The goals can be refined into questions and metrics that will be used to control the production of the component factory.

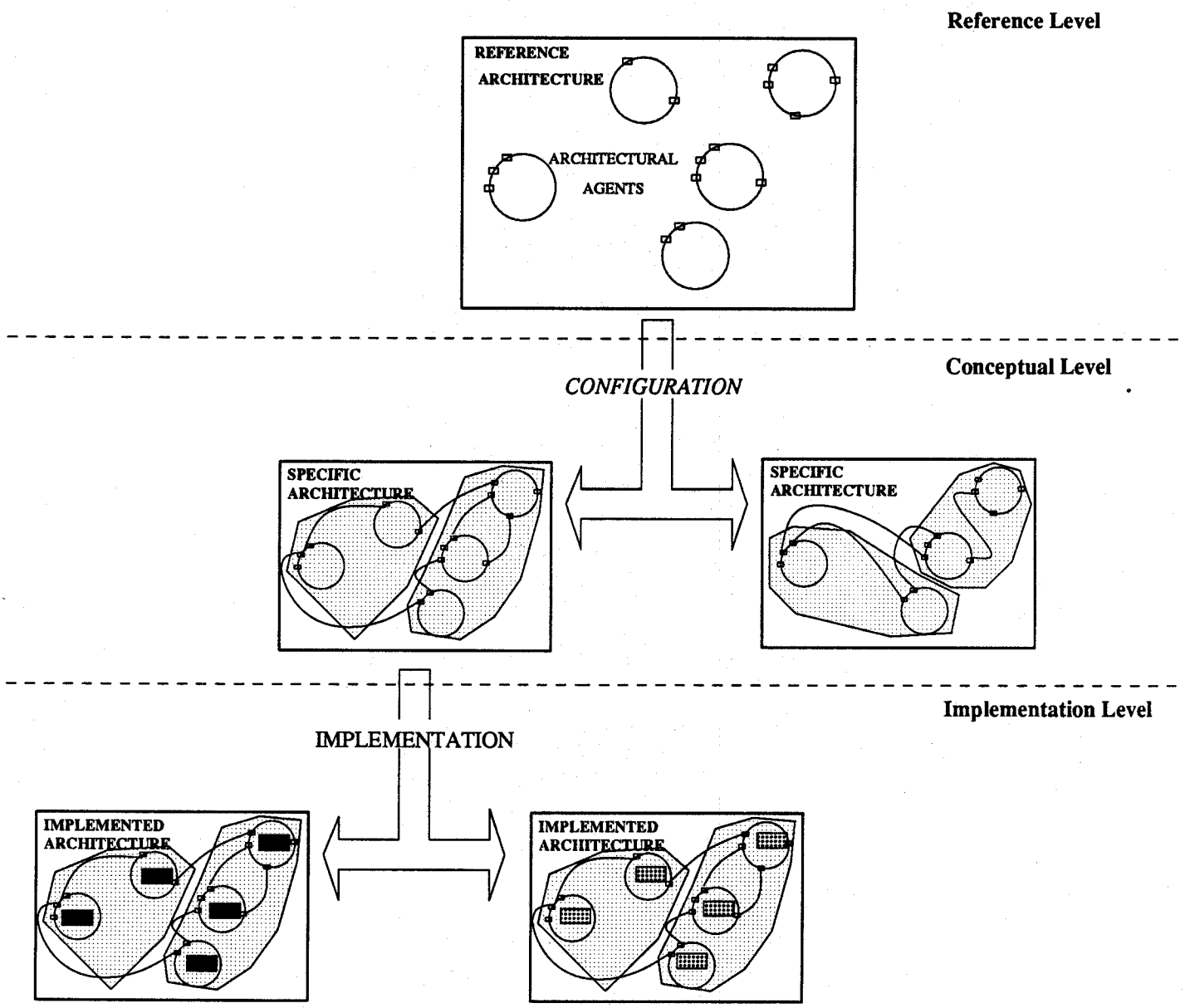
1.3 Instantiate the reference architecture into a particular component factory architecture and define the associated measurement environment:

Instantiation process

A. *Configuration of the architecture*
(reference level → conceptual level)

Figure 6

Instantiations of the Reference Architecture



A.1 Definition of the activities of the organization and mapping of those activities into specific architectural agents

A.2 Identification of the boundary of a specific factory by specifying which agent is in the project organization and which one is in the component factory.

A.3 Definition of the conceptual representation of the specific component factory by specifying the agents and connecting their ports using the configuration language.

B. *Implementation of the architecture*
(conceptual level → implementation level)

B.1 Specification of the mapping between the agents and their functions and the departments of the organization, and of the responsibilities for production control.

B.2 Definition of the implementation representation of the specific component factory by mapping agents and functions over specific units (e.g. people, automated or semi-automated tools), and specifying algorithms, protocols and process models.

2. *Produce components* for the project organizations and load products and information into the experience base:

2.1 Execute the production process using the particular architecture that has been defined;

2.2 Control the process while executing by using the measurement environment that has been defined.

3. *Analyze the results*, after a pre-established period of time, assessing the level of achievement of the goals that were behind the introduction of the component factory.

4. *Synthesize the results*

4.1 Consolidate the results of the analysis into plans for new products, models, measures, etc., or for updates for the existing ones

4.2 Package the new and updated products, models, measures into reusable units and store them for future reuse;

4.3 Modify the instantiation of the particular architecture and the measurement environment associated with it.

The crucial point of the process is the possibility, offered by the reference architecture, of modifying the particular architecture without modifying the interfaces between its building blocks. The modular structure allows configuration and reconfiguration of the processes as required by an efficient and realistic implementation of an optimizing paradigm. The evolution of the conceptual level and sub-levels is more difficult because it has impact on the implementation level, but the explicit definition of the interface types, which is part of the reference architecture, offers a certain freedom in the evolution, even at the conceptual level. Changes in the automation and organizational choices have definitely a lower impact, if they are applied to the implementation level leaving unchanged the conceptual level.

7. EXAMPLES OF COMPONENT FACTORY ARCHITECTURES

7.1 CLUSTERED AND DETACHED ARCHITECTURES

In order to illustrate the concepts of reference architecture and instantiation we can present two different conceptual architectures for the component factory.

The two architectures differ for the different role they assign to the Designer agent:

- in the first architecture the Designer coordinates all software development activities from the side of the project organization, we call it "clustered" architecture;

- in the second architecture the development activities are concentrated in the component factory under the control of the Shopfloor Coordinator agent, we call it "detached" architecture.

In a *clustered component factory architecture* (Figure 7a) every development takes place in the project organization and the role of the component factory is to perform the activities of processing and providing existing reusable software components.
The agents are assigned in the following way

Project Organization

- Designer/Integrator/ Shopfloor Coordinator
- Developer
- Adapter

Component Factory

- Component Manipulator
- Experience Modeler
- Experience Base Manager

Figure 7a

The Clustered Architecture

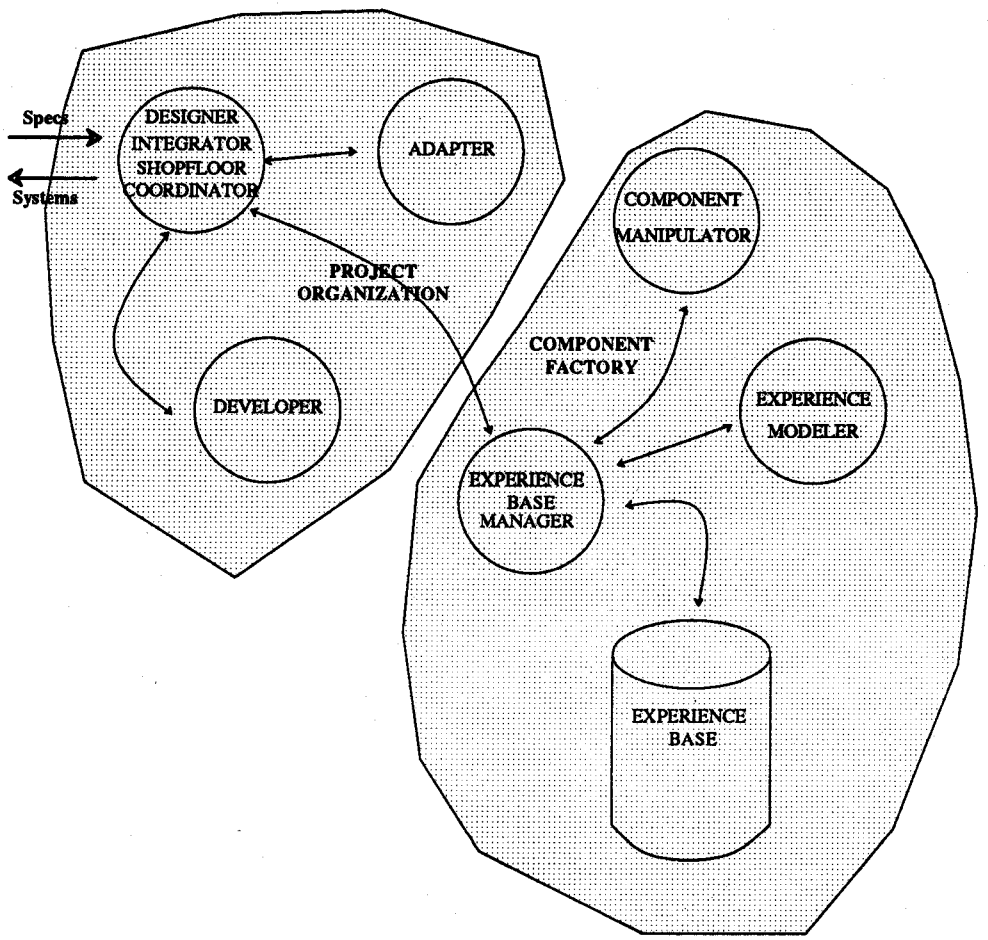
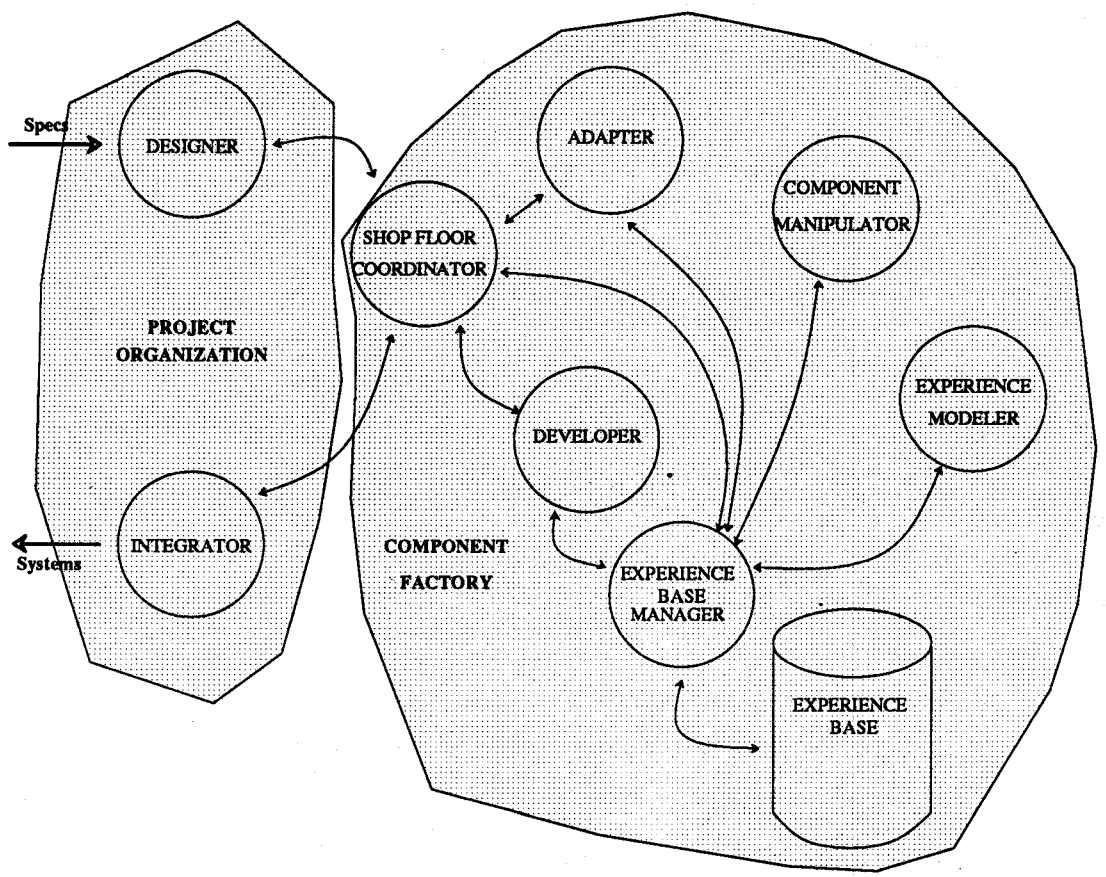


Figure 7b

The Detached Architecture



In a *detached component factory architecture* (Figure 7b)no development takes place in the project organization but only design and integration. The project organization develops its design of the system based on the information existing in the experience base and requests from the component factory all necessary developments. Then it integrates the components received from the component factory according to the design. The agents are assigned in the following way

Project Organization	**Component Factory**
· Designer	· Shopfloor Coordinator
· Integrator	· Developer
	· Adapter
	· Component Manipulator
	· Experience Modeler
	· Experience Base Manager

The activities of the agents that form the kernel of the component factory (Component Manipulator, Experience Modeler, Experience Base Manger) don't change in the two instantiations.However, the role of the component factory in the detached architecture is much more relevant because it encompasses activities that are both synchronous and asynchronous with the project organization.

An evaluation of the two instantiations can be performed using the GQM approach mentioned earlier. In order to compare the two architectures we can develop, for instance, the GQM models having as

- Object : the clustered and the detached architecture;
- Focus: characteristics such as performance, functionality and evolutionary nature;
- Viewpoint: the technical management of the organization
- Purpose: the evaluation of the architecture
- Environment: the specific organization

From these goals it is possible to derive questions and metrics that allow us to collect data to perform the comparison. Without getting into the details of a particular application, we can make the following general remarks, based on the characteristics that we have listed as focus of the evaluation:

- Performance: the level of productivity and serviceability of the project organization/component factory system.

  - In the clustered architecture, the project organization develops the components that are not available, therefore, if it has enough resources, it performs probably faster because there is less communication overhead and more pressure for their delivery. On the other hand the components developed in the framework of a project are more context dependent and this puts more load on the component factory and in particular on the Component Manipulator.

  - In the detached architecture there is more emphasis on developing general purpose components in order to serve more efficiently several project organizations: planning is easier and the optimization of resources is more effective. On the other hand there are more chances for bottlenecks and for periods of inactivity due to a lack of requests from the projects, that would affect the overall performance of the organization.

- Functionality: the conformance to the operating characteristics of an organization producing software using reusable components

  - In the clustered architecture all functions are implemented but the most critical ones are concentrated on the Designer/Integrator/Shopfloor Coordinator. This means that errors and operating failures of this agent can affect the functionality of the whole organization.

  - In the detached architecture the high modularity of functions reduces the impact of errors and failures of one agent but increases the possibility of communication errors.

- Evolutionary nature:

  - The clustered architecture is much closer to the way software is currently implemented and therefore its impact on the organization would be less drastic.

  - The detached architecture provides the component factory with enormous possibilities for adaptation and configuration making continuous improvement easier and less expensive.

The detached architecture is probably better suited for environments where the practice of reuse is somewhat formalized and mature. An organization that is just starting should probably instantiate its component factory using the clustered architecture and then, when it reaches a sufficient level of maturity and improvement with this architecture, start implementing the detached architecture to continue the improvement. The improvement paradigm, as applied to the

component factory in the last section, provides a methodology for a step-by-step approach to this implementation. In this way the organization takes advantage of the flexibility and evolutionary nature of this approach, that are among the primary benefits of reasoning in terms of instantiations of a reference architecture.

7.2 THE TOSHIBA SOFTWARE FACTORY

A further illustration of the concepts of reference architecture and instantiation comes from the analysis of a real case study.

One of the most significant accomplishments in the attempt to make software development into an industrial process is represented by the experience of Toshiba Corp. in establishing, in 1977, the Fuchu Software Factory to produce application programs for industrial process control systems [Matsumoto 1987]. In 1985 the factory employed 2,300 people and shipped software at a monthly rate of 7.2 million EASL[1] per month.

The organizational structure of Fuchu Software Factory is designed to achieve a high level of reusability (Figure 8). Projects design, implement and test the application systems reusing parts that are found in a Reusable Software Items Database. Just to give an idea of the size of an application system developed by a project, we have an average size of 4 million EASL but there are projects that go up to 21 millions EASL. The parts are made available by a Parts Manufacturing Department based on the requirements specified by a Software Parts Steering Committee made of Project people and of Parts manufacturing People. Statistics on alteration and utilization of parts are processed and maintained by the Parts Manufacturing Department.

The conceptual architecture of the Fuchu Software Factory (Figure 9) presents the replication of some agents:

- Project Organization
  - Shopfloor Coordinator: this agent performs the functions of the Software Parts Steering Committee. It is very much project oriented but some of its functions, such as planning for reuse can be identified with some functions of the Experience Modeler in the reference architecture therefore we can position it at the border between project organization and component factory.
  - Designer 1: this agent designs the application system and the components that have been deemed project specific by the Coordinator.
  - Developer 1: this agent develops the software components specified by Designer 1.

---

[1] EASL: Equivalent Assembler Source Line of code

Figure 8

The Toshiba Fuchu Software Factory

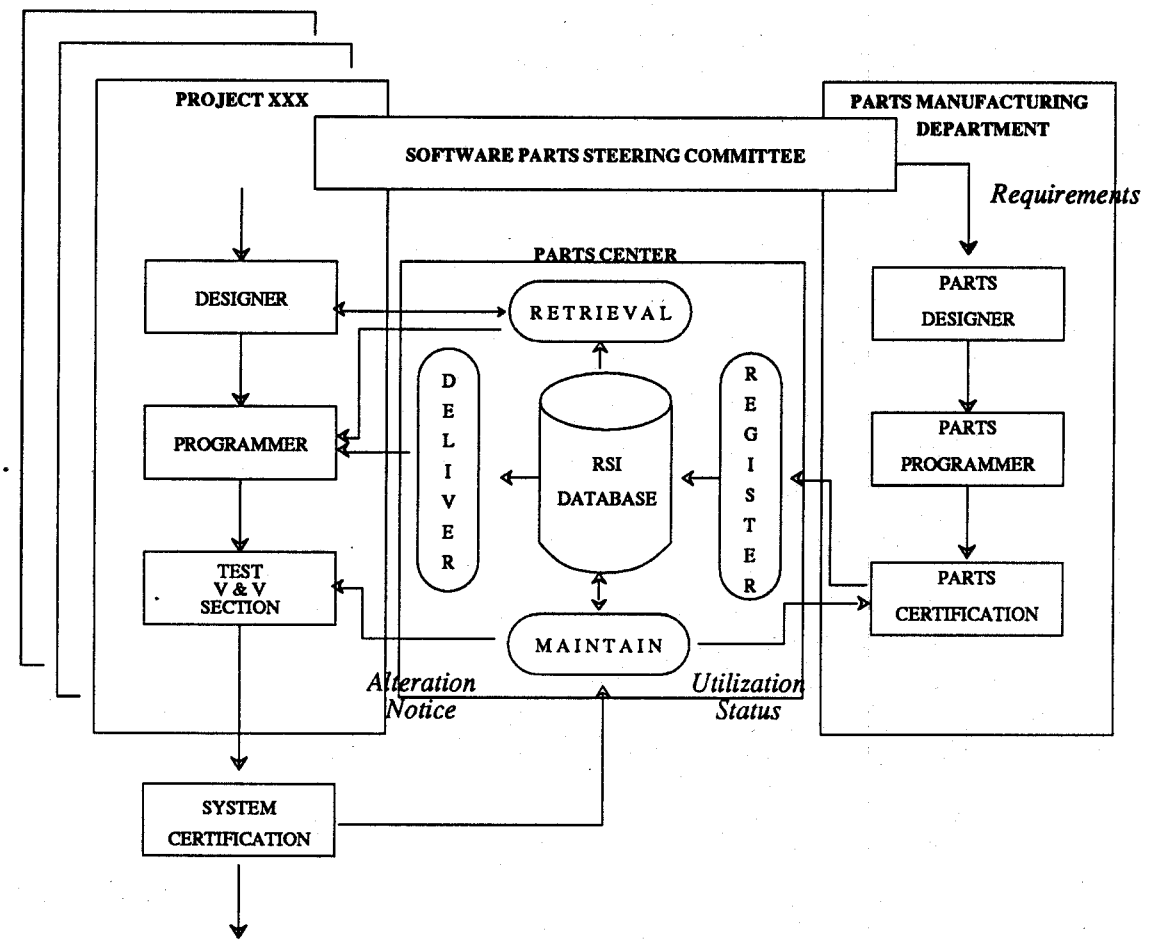
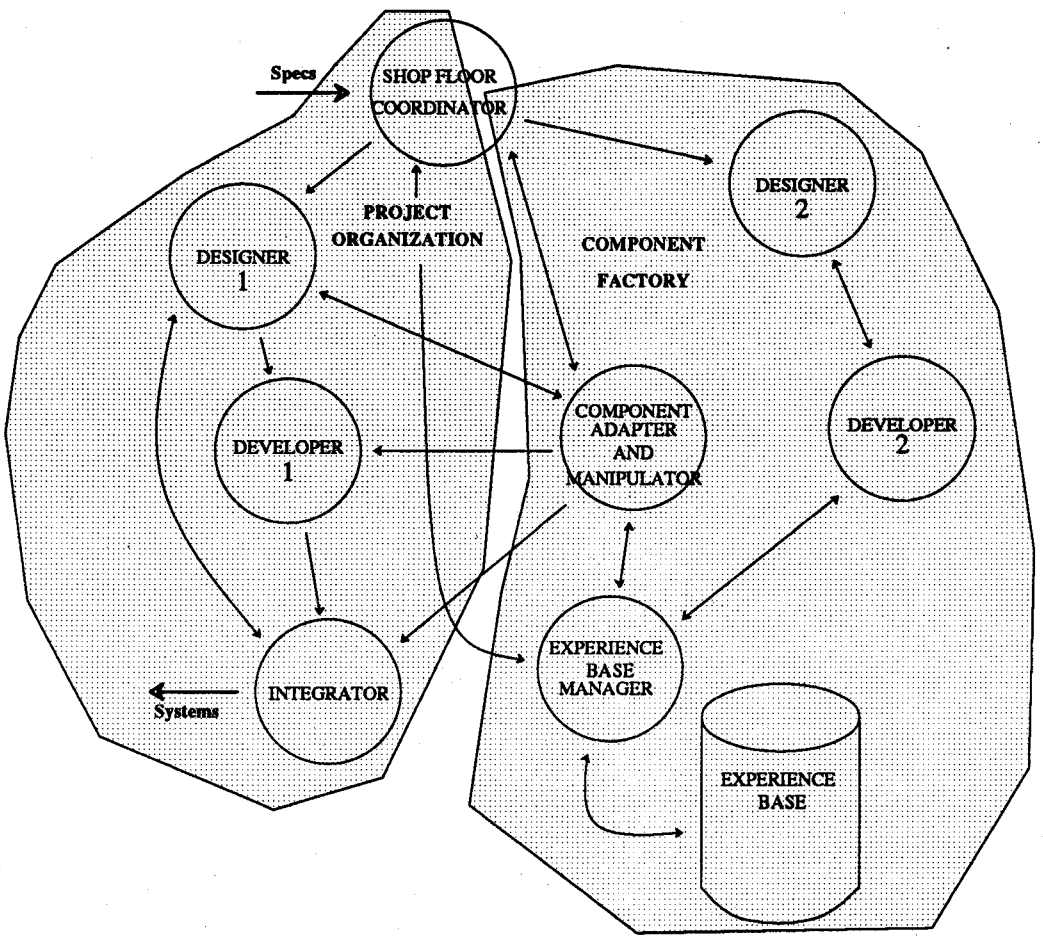


Figure 9

The Toshiba Software Factory Architecture



- Integrator: this agent assembles the system using the components received from Developer1 and from the component factory according to the design provided by Designer 1, and verifies its conformance with the requirements.

- Component Factory

  - Designer 2: this agent designs the components that have been deemed reusable by the Coordinator.

  - Developer 2: this agent develops the software components specified by Designer 2.

  - Component Adapter and Manipulator: this agent can be identified with the Parts Manufacturing Department that adapts and supplies parts under request and, based on the statistics on the utilization of the software parts contained in the database, modifies and improves those parts.

  - Experience Base Manager: this agent is in charge of the management of the Parts Center and, in particular, of the access to the Reusable Software Items Database.

A function that is not explicitly implemented in this architecture is the experience modeling function even though the factory uses state-of-the-art techniques to manage its projects. The absence of formal experience modeling is probably one of the causes for which, according to the data reported by Matsumoto, the major factor affecting productivity is the reuse of code (52.1%) while improvement of processes, techniques and tools have a less significant impact.

8. CONCLUSIONS

Flexible automation of software development combined with reuse of life cycle products seems the most promising program to solve many of the quality and productivity problems of the software industry. It is very likely that in the coming years we will see a wide and deep change in this industry, similar to the one that took place in manufacturing through the introduction of CIM (Computer Integrated Manufacturing).

The abstraction levels and the reference architecture presented in this paper are aimed at providing a framework to make both automation and reuse happen. The major benefits of this approach are

- a better understanding of reuse and of the requirements that need to be satisfied in order to implement it in a cost-effective way;

- the possibility of using a quantitative approach, based on models and metrics, in order to analyze the tradeoffs in design associated with the component factory use;

- the formalization of the analysis of the software development process and organization with a consequent enhancement of the possibilities for automation;

- the definition and the use of an evolutionary model for the improvement of a reuse-oriented production process.

One of the major problems the software industry is facing today, when using automated production support tools, like the CASE tools (application generators, analysis and design tools, configuration management systems, debuggers, etc.), are rigidity and lack of integration. These problems affect dramatically also the chances to reuse life cycle products across different projects.

Tools modeled over the reference architecture would represent a significant step towards the solution of those problems because the interfaces would be specified and standardized. An organization would have the possibility of using different set of methodologies and tools in different contexts without dramatic changes to the parts of the organization that are not affected by the specific choice. Besides, different alternatives can be analyzed and benchmarked based on the input provided by the experience base on the performance of methods and tools in similar situations.

This aspect of simulation based on historical data and formal models, is one of the most important benefits of the proposed approach and is one of the focuses of our research. The development of a complete specification for a component factory and its execution in a simulation environment using historical data as well as the study of the connection between application architecture and factory architecture will be the main goals of our future work in this field.

8. REFERENCES

[Arango 1989]

G. Arango, "Domain Analysis: From Art to Engineering Discipline," *Proceedings of the Fifth International Workshop On Software Specification and Design* (*Software Engineering Notes*, Vol. 14, No. 3), May 1989, pp. 152 - 159.

[Basili 1984]

V.R.Basili, "Quantitative Evaluation of Software Methodology", *Computer Science Technical Report Series*, University of Maryland, College Park, MD, July 1985, CS-TR-1519.

[Basili and Weiss 1984]

V.R.Basili, D.M.Weiss, "A Methodology for Collecting Valid Software Engineering Data", *IEEE Transactions on Software Engineering*, November 1984, pp. 728-738.

[Basili and Rombach 1991]
V.R.Basili, H.D.Rombach, "Support for Comprehensive Reuse", Software Engineering Journal, July 1991, (also, *Computer Science Technical Report Series*, University of Maryland, College Park, MD, February 1991, CS-TR-2606 and UMIACS-TR-91-23).

[Basili 1989]
V.R.Basili, "Software Development: A Paradigm for the Future (Keynote Address)", *Proceedings COMPSAC '89*, Orlando, FL, September 1989, pp.471-485.

[Biemans 1986]
F.Biemans, "Reference Model of Production Control Systems", in *Proceedings of IECON 86*, Milwaukee, September 29 - October 3, 1986.

[Caldiera and Basili 1991]
G.Caldiera, V.R.Basili, "Identifying and Qualifying Reusable Software Components", *IEEE Computer*, Vol.24, No.2, Feb.1991, pp.61-70.

[Caldiera 1991]
G.Caldiera, "Domain Factory and Software Reusability", Proceedings of the Software Engineering Symposium S.E.SY. 1991, Milano, Italy, May 1991.

[Cusumano 1989]
M.A.Cusumano, "The Software Factory: A Historical Interpretation", *IEEE Software*, March 1989, pp.23-30.

[Deming 1986]
W.Edwards Deming, *Out of the Crisis*, MIT Center for Advanced Engineering Study, MIT Press, Cambridge, MA, 1986

[Freeman 1983]
P.Freeman, "Reusable Software Engineering Concepts and Research Directions", *ITT Proceedings of the Workshop on Reusability in Programming*, 1983, pp.129-137.

[Joo 1990]
Bok-Gyu Joo, "Adaptation and Composition of Program Components", PhD Thesis, Department of Computer Science, University of Maryland, College Park, MD, January 1990.

[Matsumoto 1986]
Y.Matsumoto, "Management of Industrial Software Production", *IEEE Computer*, Vol.17, No.2, February 1984, p.59-72.

[Matsumoto 1987]

Y.Matsumoto, "A Software Factory: An Overall Approach to Software Production", in P.Freeman (Ed.), *Tutorial: Software Reusability*, Computer Society Press, Washington, DC, 1987, pp.155-178.

[McIlroy 1969]

M. McIlroy, "Mass Produced Software Components", Software Engineering Concepts and Techniques, *Proceedings of the NATO Conference on Software Engineering*, 1969.

[Neighbors 1989]

J.M.Neighbors, "Draco: A Method for Engineering Reusable Software Systems", in T.J.Biggerstaff and A.J.Perlis (Eds.), *Software Reusability - Volume 1: Concepts and Models*, ACM Press, New York, NY, 1989, pp.295-319.

[Tracz 1987]

W. Tracz, "Ada Reusability Efforts: A Survey of the State of the Practice," *Proceedings of the Joint Ada Conference, Fifth National Conference on Ada Technology and Washington Ada Symposium*, U.S. Army Communications-Electronics Command, Fort Monmouth, New Jersey, pp.35-44.