DSSA (Domain-Specific Software Architecture) Pedagogical Example

Will Tracz
Loral Federal Systems - Owego
tracz@lfs.loral.com

ADAGE-LOR-94-13A

3 April 1995

Abstract

A Domain-Specific Software Architecture (DSSA)[1] has been defined as:

- "an assemblage of software components, specialized for a particular type of task (domain), generalized for effective use across that domain, composed in a standardized structure (topology) effective for building successful applications" [Hay94] or, alternately
- "a context for patterns of problem elements, solution elements, and situations that define mappings between them [Hid90].

The following small example[2] illustrates these definitions as well as provides the reader with some insight into the types of processes and tools needed to support the creation and use of a DSSA.

1 Introduction

This paper describes an exemplary DSSA for a small domain (i.e., theater ticket sales). The material is presented in the logical order of its creation in the DSSA process [TC92]. Figure 1 depicts the DSSA artifacts in relationship with the individuals who use or create them.

Key "insights" appear at appropriate points throughout the text and are distinguished in this manner.

The first section describes the **domain model**[3] that was generated based on **scenarios** or "operational flows" that reflect the behavior of applications in the domain being analyzed – ticket sales. The domain model consists of:

1. scenarios,
2. domain dictionary,
3. context (block) diagram,
4. entity/relationship diagrams,
5. data flow models,
6. state transition models, and
7. object model.

The second section focuses on the **reference requirements**. Besides specifying the functional requirements identified in the domain model, the reference requirements also contain:

1. non-functional requirements,
2. design requirements, and
3. implementation requirements.

The third section describes the resulting **reference architecture** consisting of:

1. reference architecture model,
2. configuration decision tree,
3. architecture schema or design record,
4. reference architecture dependency diagram (topology),
5. component interface descriptions,
6. constraints, and
7. rationale.

The final section provides an analysis of differences between "real world" problems and this "toy" example.

For additional information on DSSA processes, the reader can refer to [TC92, CT92].

[1]This effort is sponsored by the US Department of Defense Advanced Research Projects Agency in cooperation with the US Air Force Wright Laboratory Avionics Directorate under contract F33615-91-C-1788.

[2]This paper is a condensed version of the orginal technical report – ADAGE-LOR-94-13A.

[3]For this example, all diagrams use Object Modeling Technique (OMT) symbols [RBP+91] in combination with Feature-Oriented Domain Analysis (FODA) conventions [KCH+90].

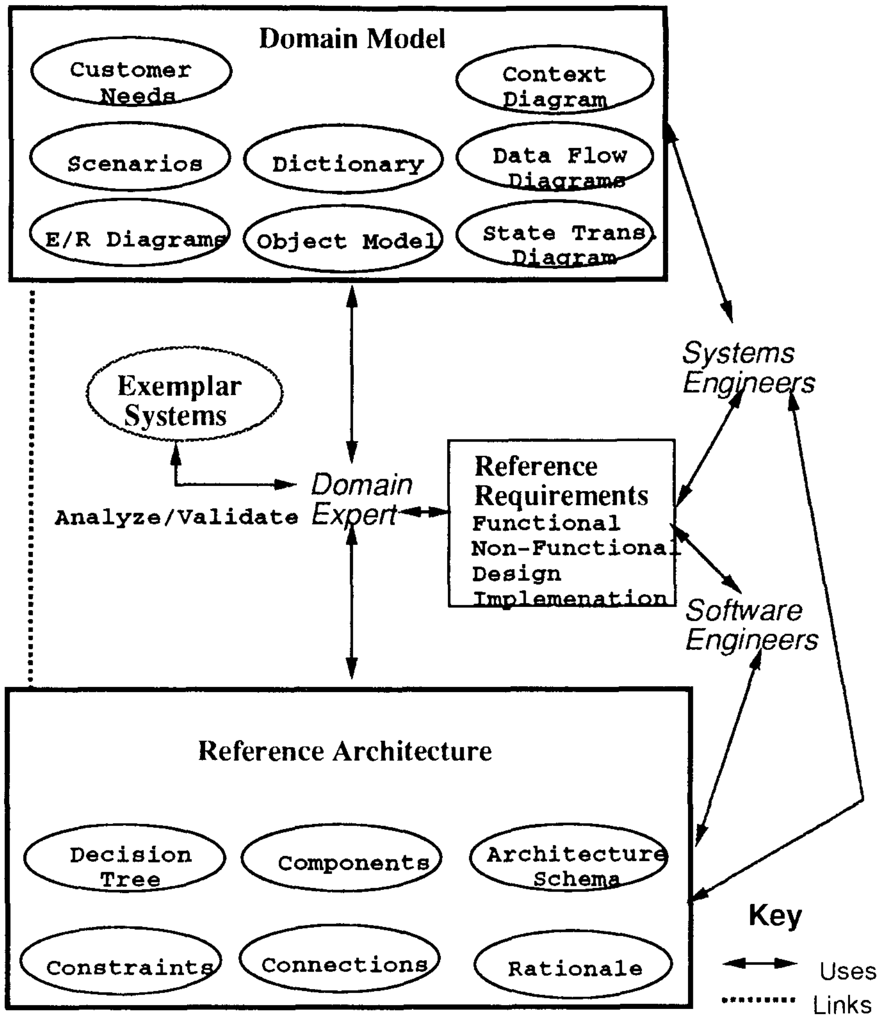

Figure 1: DSSA Artifacts

One of the insights to be gained from this example is the separation of "problem space" from "solution space" or "design space."

The domain model generally tries to characterize fully the former, while the reference architecture addresses a portion (for reasons of practicality) of the latter.

2 Domain Model

Every DSSA starts with an analysis of the application domain. This domain analysis process often involves several domain "experts" who are intimately familiar with legacy systems of this kind or other aspects of the domain of interest. It also may involve customer inputs as well as inputs from others familiar with various aspects of the application.

The purpose of a domain model is to provide to individuals who will develop or maintain applications in a domain an unambiguous[4] understanding of various aspects of the domain.

A domain analyst is like a systems analyst except instead of analyzing just one system to be developed, the domain analyst focuses on families of systems or a product line.

2.1 Customers' Needs Statement

An informal needs statement is a good place to learn to "talk the talk" of the customers. Therefore it can be valuable as a basis for the domain dictionary.

Often a customer's system requirements are first expressed informally in terms of what "needs to be done." Such an opera-

[4]One can not overly stress the importance of using consistent, unambiguous terminology throughout any system development process. That is why the domain dictionary plays a central role in the domain modeling process.

tional needs statement for the ticket sales application domain could be stated as follows.

> "I am in charge of the finances for a play that is being performed by our community theatrical group. This is a one time shot, but I think it would be nice to have a computer program to help the person taking phone and mail orders for tickets. Depending on how it works, I may want to use it for the the rest of the performances by our theatrical group.
>
> The theater we are using has reserved seats (i.e., row number, seat number). We are charging $10 for orchestra seats and $7 for seats in the balcony.
>
> We would like the program to tell us such things as: how many tickets are sold, how many are left, and how much money has been taken in. To help the ticket agent, we also would like a display of the seating arrangement that shows which seats are sold and which are available."

While there are clearly several clues on the requirements for the system, a more general model needs to be constructed to factor in all the implications of the domain and to create more general requirements.

> One important difference between DSSA requirements analysis and traditional systems requirements analysis is the emphasis on the separation of **functional requirements** from **design and implementation requirements**.
>
> In the customer's mind, these are all "requirements." but from the DSSA perspective, the functional requirements define the (problem) domain, while the design and implementation requirements constrain the design/architecture.

2.2 Scenarios

The following scenarios consist of a list of numbered, labeled scenario steps or events followed by a brief description.

2.2.1 Ticket Purchase Scenario

1. **Ask:** The customer asks the agent what seats are available.
2. **Look:** The agent enters the appropriate command into his/her terminal and relates the results to the customer (cost, section, row number, and seat number).
3. **Decide:** The customer decides what seats are desired, if any, and tells the agent.
4. **Buy:** The customer pays the agent for the tickets. The agent gives the tickets to the customer.
5. **Update:** The agent records the transaction.

2.2.2 Ticket Return Scenario

1. **Return:** The customer gives the agent tickets that are no longer needed.
2. **Refund:** The agent gives the customer money back.
3. **Update:** The agent records the transaction.

2.2.3 Ticket Exchange Scenario

1. **Ask:** The customer asks the agent what seats are available.
2. **Look:** The agent enters the appropriate command into his/her terminal and relates the results to the customer (cost, section, row number, and seat number).
3. **Decide:** The customer decides what seats are desired, if any, and tells the agent.
4. **Exchange:** The customer gives the agent the old tickets, then the agent gives the customer the new tickets.

   Depending on the price of the new tickets, the agent either collects additional money from the customer or issues a refund.
5. **Update:** The agent records the transaction.

2.2.4 Ticket Sales Analysis Scenario

1. **Stop Sales:** The sales manager enters the command to stop the sale of tickets for a particular performance.
2. **Tally:** The ticket sales program generates a report listing total sales.

2.2.5 Theater Configuration Scenario

1. **Performance Logistics:** The sales manager enters in the name, time, location, and date of the performance.
2. **Seating Arrangement:** The sales manager decides if the performance is "Reserved Seating" or "Open Seating."
3. **Theater Logistics:** If this performance is reserved seating, then the sales manager enters the number and kind of sections in the theater, what rows are in what sections, and what seats are in what rows.

   If this performance is open seating, then the sales manager enters the total number of tickets to be sold.
4. **Pricing:** The sales manager enters in the price of each ticket, determined by section and seating style.

> Scenarios are not only a good way of eliciting functional requirements, data flow, and control flow information from a customer but they also allow the analyst to get an idea of what kind of "look and feel" the system should have.

2.3 Domain Dictionary

The following "initial" version of the domain dictionary consists of commonly used words or phrases found in the scenarios and customer needs document (statement of work).

At this point in the domain analysis process the following words have been used:

Agent:	The person who interacts with the application, answers customer questions, and handles tickets and money.
Available:	The status of a seat. If a seat is available, then a ticket can be issued for it. See "Sold."
Balcony:	The farthest away and usually the least expensive seats in a theater.
Configuration:	Information describing the performance and seating style for which tickets are sold. "See Performance" and "Seating Style."
Cost:	See "Price."
Customer:	The person who interacts with the agent to inquire about, purchase, return, or exchange tickets.
Date:	The date of the performance.
Location:	The physical location of the performance.
Open Seating:	A seating style where there are no reserved seats (a ticket is good for any seat in the theater).
Orchestra:	The closest and generally the most expensive seats in a theater.
Performance:	The date, time, location, and name of a theatrical production.
Price:	The cost of a ticket.
Reserved Seating:	A seating style where a ticket allows the customer to sit in one particular seat denoted by section, row, and seat number.
Row:	The row is a sequence of numbered seats.
Sales Manager:	The person who configures a performance, closes sales, and issues status report requests.
Sales Report:	A description of how many tickets have been sold, how many are left, and how much money has been collected.
Seat:	What a ticket entitles a customer to sit in. A seat is located in a row, in a section of a theater.
Section:	A section is a sequence of named rows (of seats).
Seating Arrangement:	A display showing what seats are sold and what seats are available.
Seating Style:	Either open seating or reserved seating.
Sold:	The status of a seat indicating that a ticket has been given to a customer for that seat. See "Available."
Theater:	The place full of named sections, rows, and seats where performances are held.
Ticket:	A ticket is what the customer buys, sells, and uses to get in the door of a performance.
Total Sales:	How much money was collected.

As additional domain analysis artifacts are generated (e.g., in data flow and state transition models), the domain dictionary will evolve. In particular, the events, objects, instances, actors, and data stores labels used in various diagrams and models should be consistent with entries in the domain dictionary.

Finally, not everyone will always agree on the "right word" or the "right definition." The Domain Dictionary should store all such information for possible resolution at a later time.

2.4 Context (Block) Diagram

Figure 2 depicts the high-level data flow between the major components in the system.

Everyone has one of these, what they call it is not important. What is important is that it is **not** the software architecture.

What is important is that it shows what is in the domain and what is outside the domain.

2.5 Entity/Relationship Diagrams

The following figures (3, 4, 5, and 6) depict a portion of the entity relationship diagrams for this problem domain. There are basically two types of relationships of interest:

1. **Aggregation**: "a-part-of" relationships (denoted by a diamond symbol in figures 4 and 6, and
2. **Generalization**: "is a" relationships (denoted by a triangle in figures 3, and 5).

Note that by convention, little circles[5] (i.e., ∘) denote optional or alternative entities. For example, figure 5 shows that the sections (in a theater) can be any combination of orchestra, mezzanine, or balcony (or none).

[5]More precisely, the ∘ indicates zero or one occurrences, while the • indicates zero or many occurrences.

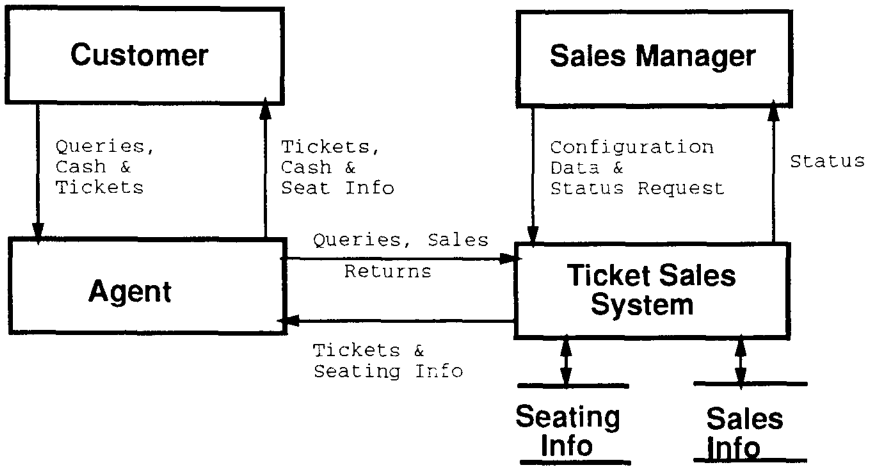

Figure 2: Context or Block Diagram

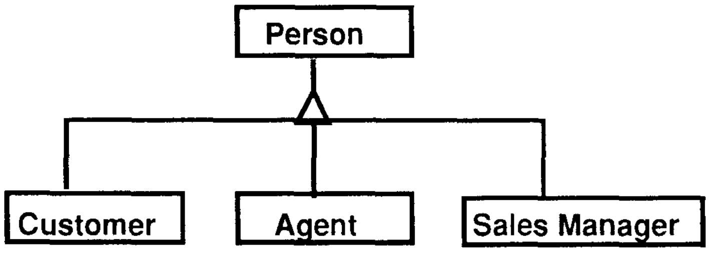

Figure 3: Person Types Taxonomy

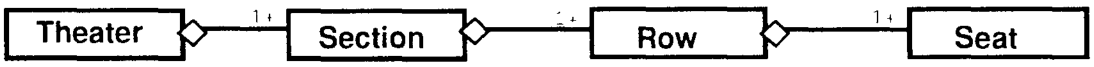

Figure 4: Theater Aggregation Hierarchy

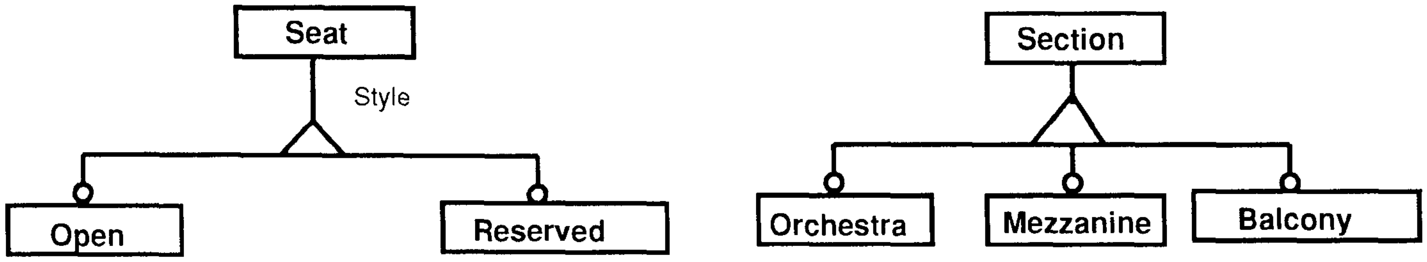

Figure 5: Seating Styles and Section Taxonomy

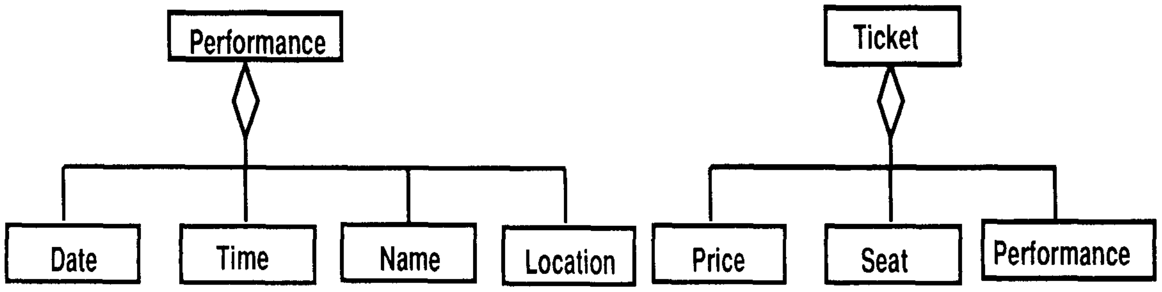

Figure 6: Performance and Ticket Aggregation

"Generalization" relationships, which show alternatives or options, factor heavily into architecture design.

One will quickly observe that the domain model is starting to become "generalized," based on these "is a" and "a-part-of" relationships.

2.6 Data Flow Model

Figures 7 and 8 represent amalgamated data flow diagrams involving the customer/agent and sales manager. Note, because this is a small example, the flow of data in figure 7 between the customer and ticket agent has been combined.

2.7 State Transition Model

Figure 9 describes the events and states that take place in this domain. One leaves the initial state when the system is configured. One enters the final state when sales are closed.

About this point in time, the domain analyst will recognize the inadequacies of the information found in the initial scenarios and domain dictionary and go back to update them accordingly.

2.8 Object Model

The object model is the first phase of component interface design. As such, it provides valuable insights into the resulting reference architecture. The domain analyst/architect should not place too much detail (over specify) on the attributes and operations exported by each object because they may unduly constrain the reference architecture.

Using an object-oriented approach to module decomposition and specification, one can easily identify the following objects (among others):

1. Seat,
2. Row (of seats),
3. Section (of rows), and
4. Theater (of sections).

The operations and attributes associated with these objects are found in the table 1[6].

2.8.1 Object Model Analysis

For all intents and purposes, a "seat" and a "ticket" are synonymous. One could argue that "ticket," " agent," "balcony," "orchestra," "money," and "seating arrangement" are also objects, but those have been deferred as implementation details. Similarly, "agent" is an implicit object in that the application system will have an interactive interface.

[6]The information in this table could have been represented as OMT Object Model Diagrams.

2.9 Generalizing the Domain Model

So far in this example the system configurability capability and optional and alternative capabilities have introduced generalizations to the initial customer "needs" statement. With minimal efforts, a domain analyst probably could factor in additional optional capabilities dealing with such things as:

- adding a matinee performance with change in ticket prices,
- generating additional reports,
- having multiple agents handle ticket sales,
- supporting mail order sales or season subscriptions, or
- choosing from pre-configured seating arrangements.

In addition, the experienced domain analyst might recognize that the nature of the problem domain is analogous to other problem domains (see Table 2) and either structure the architecture to leverage existing systems of this sort or expand the domain of applicability of the analysis to include these new domains in order to exploit a larger customer base.

3 Reference Requirements

The domain architect uses the reference requirements to drive the design of the reference architecture. The domain model defines the behavior of applications in the system through scenarios, data flow diagrams, and state transition diagrams. The next step in the DSSA process it to identify the portion of the solution space that the domain model (problem space) will map into (see figure 10).

Functional requirements are defining characteristics of the problem space. Non-functional, design, and implementation requirements are limiting characteristics (or constraints) in the solution space.

One can gain further insight into this dichotomy by examining the two kinds of requirements observed by Ruben Prieto-Diaz in *Establish Global Requirements* stage (A5113 – Stage 1.1.3) of the STARS Domain Analysis Activities [PD91]:

1. **Stable Requirements** — ones that do not change from application to application, and
2. **Variable Requirements** — ones do/might change.

Following the traditional separation of "what" from "how," one could conclude that the

- stable requirements are the "**what**" requirements and
- variable requirements are the "**how**" requirements.

The following are examples of "how" and "what" requirements:

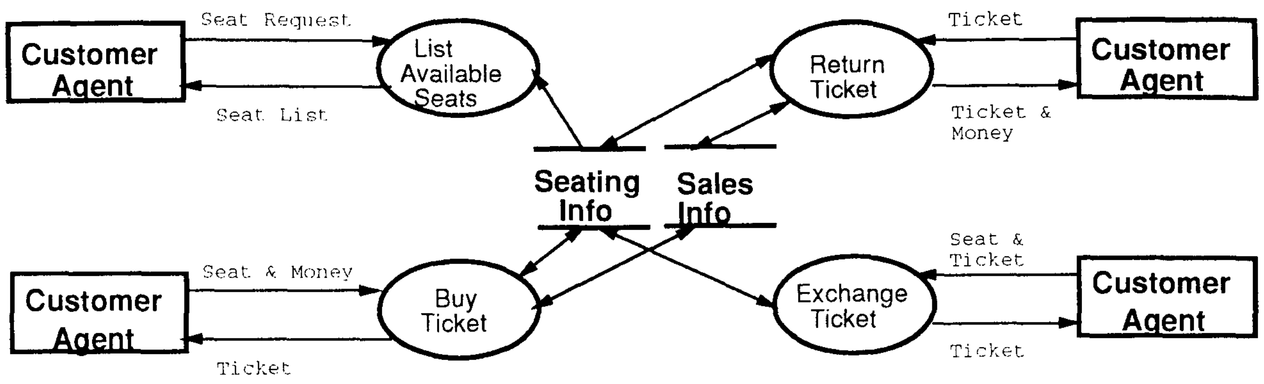

Figure 7: Data Flow Diagrams involving the Customer and Agent

Object	Attributes	Operations
Seat	Name Status (e.g., sold, available)	Sell a Seat Return a Seat Initialize a Seat
Row	Name	Number of Available Seats in Row List Available Seats in Row List Seats in Row Initialize a Row
Section	Name (e.g., orchestra, balcony)	List Rows in Section List Available Rows in Section Initialize a Section
Theater	Name Total Tickets Sold Total Tickets Unsold Total Sales	List Sections Display Seating Arrangement Initialize a Theater

Table 1: List of Objects, Operations, and Attributes

Theater Domain	Airline Domain	Library Domain	Inventory Generalization
Seat	Seat	Book	Item
Row	Row	Shelf	Room/Shelf/Bin
Section	Ticket Category	Section	Aisle or Building
Performance	Flight Number	Title	Description
Seating Arrangement	Seating Arrangement	Floorplan	Warehouse
Tickets Sold	Tickets Sold	Books on loan	Items sold
Ticket Remaining	Tickets Remaining	Books available	current inventory
Price	Price	Penalty for lateness	Cost/Item
Performance time	Flight Departure	n/a	n/a
Performance date	Flight Date	Due date	Expiration date?
Ticket Agent	Ticket Agent	Librarian	Clerk

Table 2: Comparison of Theater, Airline, Library, and Inventory Domains

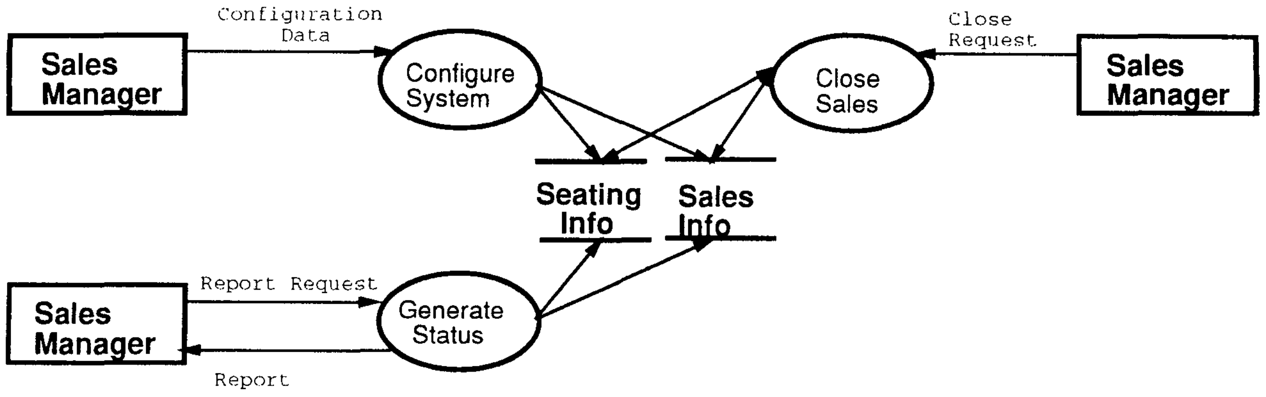

Figure 8: Data Flow Diagrams involving the Sales Manager

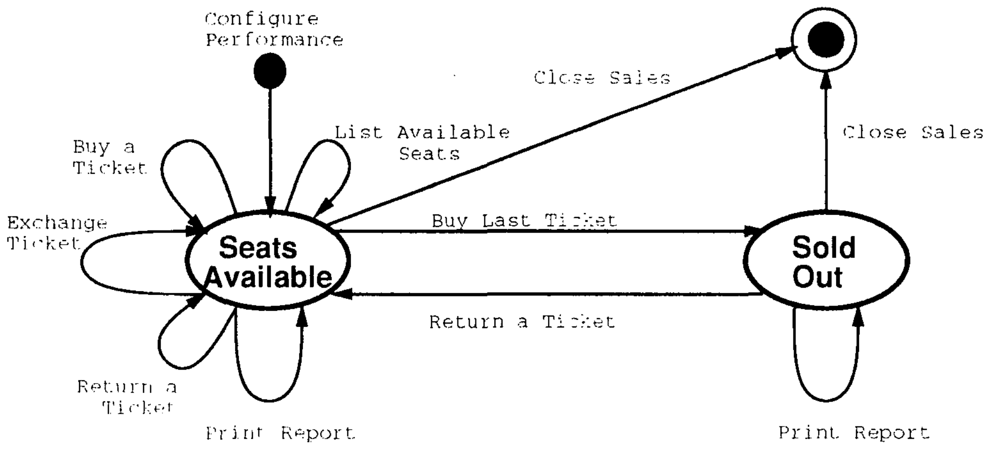

Figure 9: State Transition Diagram

- "what it does" (functional/behavioral requirement)
- "how often" (performance requirement),
- "how fast" (performance requirement),
- "how big,"
- "how accurate,"
- "how implemented" (physical requirements as well as language),
- "how delivered,"
- "how it looks" (user interface), and
- "how it works" (operational requirements (protocols to follow) or algorithmic alternatives).

The reference requirements listed in the following sub-sections have the following naming convention:

1. each requirement has a unique name or label,
2. if the requirement is optional, then the suffix "-OPT" is added to the requirement's name/label, and
3. if the requirement is an alternative, then the suffix "-ALTn" is added to the requirement's name/label (where "n" is the nth alternative).

Finally, as is to be expected, when additional functional requirements are introduced, they result in a ripple effect through previous documentation.

3.1 Functional Requirements

3.1.1 Sales Manager

Configure: The system shall allow the sales manager to enter performance information (e.g., time, date, location) of the show as well as the seating configuration of the theater (sections), and cost of the tickets.

Open Seating-ALT1: The system shall allow the sales manager to specify an open seating format for ticket sales.

Reserved Seating-Alt2: The system shall allow the sales manager to specify a reserved seating format for ticket sales.

Close Sales: The system shall allow the sales manager to halt the sale of tickets.

Request Report: The system shall allow the sales manager to request reports of current ticket sales including: number of tickets sold, number of tickets remaining, and total sales.

Reconfigure-OPT: The system shall allow the sales manager to adjust the configuration parameters

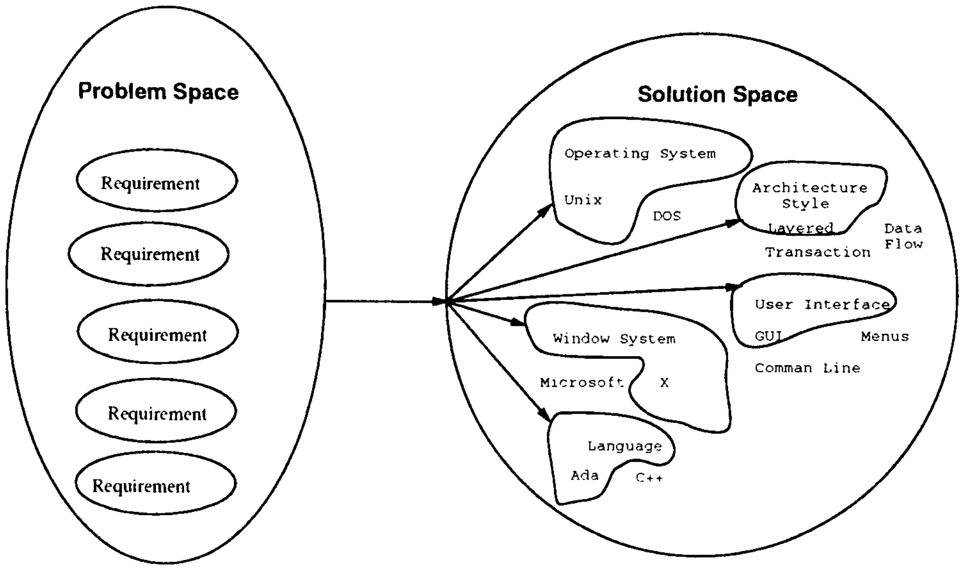

Figure 10: Mapping between Problem Space and Solution Space

once the initial information has been entered.

3.1.2 Agent

Sell: The system shall allow the agent to give the customer a ticket for a seat to a performance in exchange for payment of the cost of the ticket.

Query: The system shall display a "List of Available Seats" upon the request of the agent.

Transactions: The system shall allow the agent to record the sale, return, and exchange of tickets.

Will Call-OPT: The system shall allow the agent mark tickets as "reserved" to be picked up by the customer at the "Will Call" window.

3.2 Non-Functional Requirements

The following exemplify non-functional requirements for systems like this:

Security-OPT: The sales manager shall be the only person to configure the system.

Fault Tolerance: The system shall, in event of a power failure, not loose any ticket sale data.

Multi-user Access-OPT: The system shall support the sale of tickets by several agents at different locations.

Safety: The system shall allow only one ticket to be sold for each seat of a performance.

Response: The system shall have a response time of less than one second for each "List of Available Seats" query.

3.3 Design Requirements

The domain architect is faced with a multitude of decisions regarding design tradeoffs. The most significant design decision is the issue of **architectural style** (e.g., hierarchical/layered, transaction based, data flow, interpreter, blackboard system, etc.). The style of architecture, besides affecting performance and cost of development, will affect the interface style of the corresponding components.

Another design decision is related to **user interface style** (e.g., command line, pulldown menus, function keys, hot keys, etc.). There is a dependency between the interface style and the type of hardware and operating system that is selected.

User Interface-ALT1: The system shall provide a command line user interface.

User Interface-ALT2: The system shall provide a menu driven user interface.

User Interface-ALT3: The system shall provide a pulldown menu driven user interface.

Obviously, the reference architecture could be designed to support numerous styles of user interfaces, operating systems, etc., through the use of virtual machine interfaces and the creation of a family of plug-compatible components.

3.4 Implementation Requirements

Implementation requirements are similar to design requirements in that the analyst or domain architect needs to determine the subset of several implementation options that will drive the design of the reference architecture and implementation of its respective components.

Common implementation decisions that affect a reference architecture include:

1. **programming language:** Ada, C++, Smalltalk, Visual Basic, LISP, etc,
2. **operating systems:** Unix, DOS, VMS, OS2, NT, etc.
3. **data base and/or data structures:** Oracle, DB2, CORBA, etc.
4. **hardware platform:** PC, workstations, X-Terminals, dumb heads, etc.
5. **networking capabilities:** token ring, ATM, ethernet, etc.

The implementation requirements for the theater example include:

Language: The system shall be implemented in Ada.

Operating System ALT1: The system shall run on a Unix platform.

Operating System ALT2: The system shall run on a DOS platform.

Size: The system shall handle ticket sales of up to 2,000 seats per performance.

There is no such thing as "doing it right the first time."

These requirements do not represent the "best" requirements for this application domain but they do illustrate a point. Requirements need to evolve based on feedback.

The domain dictionary helps in developing "good" reference requirements by supporting the use of consistent and unambiguous terminology in the requirements. But, when requirements are refined, new terminology must be reflected back into the domain dictionary.

Finally, a desirable artifact of the reference requirements specification process is the capture of the rationale and interdependence between requirements (see section 4.6). Having this information will assist the application engineer understand configuration tradeoffs.

4 Reference Architecture

A reference architecture is a parameterized design that satisfies a clearly distinguished subset of the functional capabilities identified in the reference requirements within the boundaries of certain design and implementation constraints, also identified in the reference requirements[7].

[7]This definition implies that not all the requirements have to be satisfied by any one reference architecture, but that several reference architectures may exist in any one domain.

A reference architecture is more general than the design for a single system because it was engineered to be reusable, extendable, and configurable. Furthermore, significant documentation is associated with the reference architecture to provide the application engineer or maintenance programmer with enough information to easily generate new applications or modify existing ones that are based on the reference architecture.

The sub-sections that follow illustrate the types of artifacts associated with a reference architecture along with the respective documentation (see figure 1).

4.1 Reference Architecture Models

All designs start out with some simple abstraction based on an architecture style. Figure 11 reflects the overall structure of the theater ticket sales system that is being designed using a "layered" architectural style.

Figure 11 sheds no major insights because it is dealing with a simple problem at a very high level of abstraction.

One should note that such models (the most famous being the toaster model) are **not** reference architectures because they do not show the data that flows between components nor do they indicate any interfaces that exist on each component or sub-architecture.

Figure 11 does indicate that the *User Interface* can be separated out from the *Functionality.* This implies that a family of plug-compatible and separately selectable *User Interface* components could exists for subsequent integration into the desired system.

4.2 Configuration Decision Diagram

Figure 12 contains a possible decision tree for configuring a ticket sales reference architecture. One should note that for illustrative purposes, this example makes the following implicit assumption:

> configuration takes place at reference architecture instantiation time[8].

This is in contrast with having the generated system provide interactive configurability[9], (i.e., the sales manager uses a generated sales ticket sales program to specify the configuration parameters of a given performance rather than configures the reference architecture to generate a sales program for a given performance).

Therefore in configuring the reference architecture, the application engineer (sales manager in this case) would:

- choose the user interface style,

[8]This may or may not imply that the performance can be reconfigured (see **Reconfigure-OPT** requirement in section 3.1.1.

[9]This alternative is argumentatively more intuitive, though may not be desirable for safety reasons.

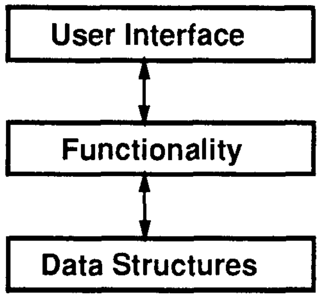

Figure 11: Simple "Layered" Reference Architecture Model

- choose the seating styles,
- specify the date, time, name, and location of the performance, and
- provide the seating arrangement (i.e., section names, rows per section, and seats per row) and price of each seat.

Additional lower-level choices (e.g., implementations of list and set packages to be used for internal data-structures) also could be selectable, if families of plug-compatible components have been provided by the domain architect.

The key insight to be gained from this section is that the design decision diagram can easily be mapped onto the reference requirements.

Therefore, configuring a system, in affect, becomes the process of selecting a subset of the reference requirements. Furthermore, technology exists (e.g., constraint-based reasoning systems) to assist the user in this configuration process preventing the specification of incomplete or incorrect systems. Finally, additional technology exists to generate applications, based on the configuration data.

4.3 Architecture Schema/Design Record

The purpose of an architecture schema or design record is to serve as a vehicle for software understanding by functioning as a collection point for knowledge about the components that make up a DSSA. In particular, the design record organizes

- *domain-specific* knowledge about components or design alternatives and
- *implementation-specific* knowledge about alternate implementations,

The primary goal of a design record is to adequately describe the components in a reference architecture such that the application engineer can make design decisions and component selections without looking at implementations. The secondary goal of a design record is to provide information that the tools in the supporting environment can use.

Two design record/architecture schemas are being used on the ARPA DSSA program: Loral Federal Systems - Owego [TSC94] and Teknowledge's [TPD+94]. The Teknowledge architecture schema for representing reference "architectures, designs, and implementations based principally on components and connections" compared to the Loral design record example presented below, provides for finer grained specification of architecture component information.

4.3.1 Loral Design Record Example

The design record data elements used by Loral Federal Systems - Owego's DSSA ADAGE (Avionic Domain Application Generation Environment) [CS93], as proposed by Scherlis [Sch90] and arranged according to phases in the software life cycle, include:

1. **name/type**,
2. **description**,
3. **reference requirements satisfied**,
4. **design structure** (data flow and control flow diagrams),
5. **design rationale**,
6. **interface and architecture specifications and dependencies**,
7. **PDL (Program Design Language) text**,
8. **implementation**,
9. **configuration and version data**, and
10. **test cases**.

    In addition to the "primary" lifecycle elements listed above, the following "secondary" elements aid in the (re-)use of the components by capturing additional information:

11. **metric data**,
12. **access rights**,
13. **search points**,
14. **catalog information**,
15. **library and DSSA links**, and
16. **hypertext paths**.

    For the avionics domain, the ADAGE design records contain the **basic** data items listed above (with some domain-specific clarifications) in addition to some DSSA-ADAGE specific items including:

17. **models**, and
18. **constraints**.

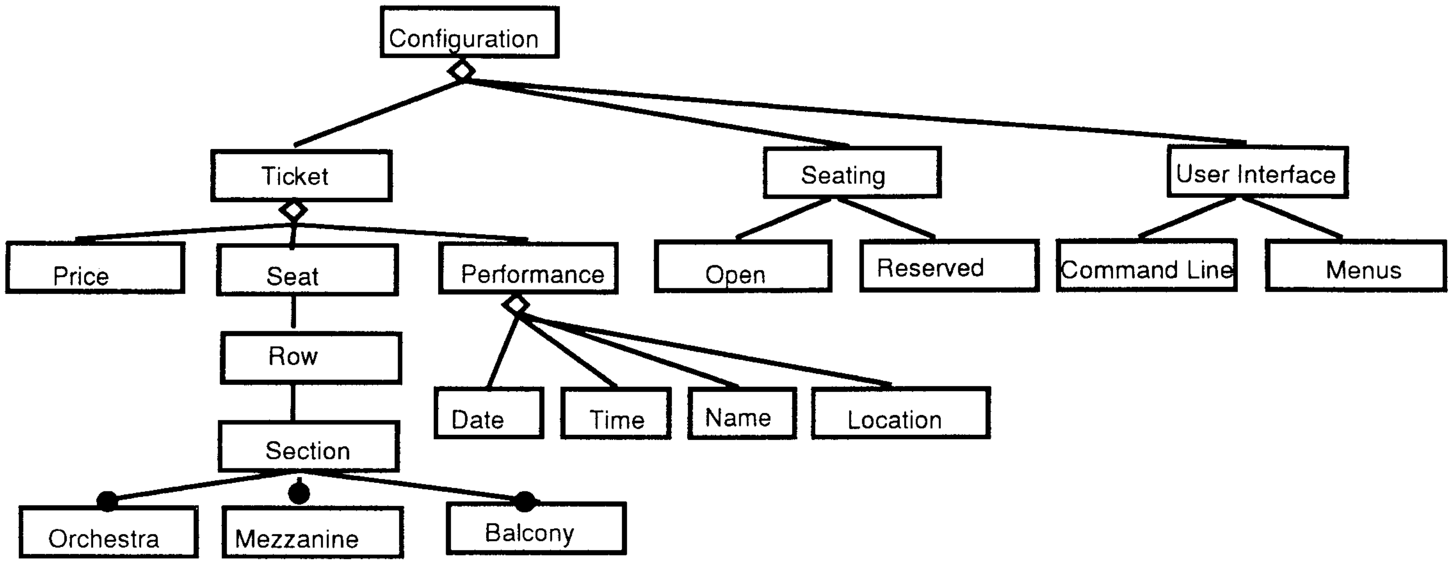

Figure 12: Design Alternative Diagram

4.4 Reference Architecture Dependency Diagram

The reference architecture dependency diagram reveals component connections at a level of granularity reflecting the architectural style chosen by the system architect. One form of the dependency diagram is a call tree. Another form is an inheritance hierarchy.

Figure 14 shows both the inheritance (horizontal structure) and control flow (vertical structure) dependencies of one reference architecture for the theater application domain.

4.5 Component Interface Descriptions

The following example uses the architecture description language LILEANNA [Tra93b, Tra93a] to describe the interfaces to components in the reference architecture. LILEANNA, by design, is a superset of Ada, which facilitates the development and integration of Ada packages into the resulting application. Appendix A contains an example illustrating how LILEANNA can be used to generate a ticket sales program, based on the reference architecture and a set of Ada packages.

4.5.1 LILEANNA Package for Theater

The generic LILEANNA *Theater* package shown is figure 15 complies with the dependency diagram (figure 14) described in the previous section except that instead of inheritance, genericity is used to "gain visibility" into the *Section* component (horizontal structure). The **needs clause** shows the vertical dependency on the *Set_Theory* component (see figure 16), which itself is parameterized by some component *Triv*.

```
generic package Theater [ S :: Section ]
       needs ( SetP :: Set_Theory [ Item :: Triv ] ) is

  type Theater; -- a set of sections of rows of seats
  type Currency;
  No_More_Sections   :    exception;
  Duplicate_Section  :    exception;

  function Total_Tickets_Sold (T : Theater ) return Natura
  function Total_Tickets_Unsold (T : Theater ) return Natu
  function Total_Sales (T : Theater ) return Currency;

  function Theater_Name ( T: Theater ) return String;
  function Is_Last_Section_in_Theater ( T : Theater;
                        S in Section; ) return Boolean;
  -- raise No_More_Sections is null section:?

  procedure Get_First_Section (T: Theater;  S: out Section
  -- raise No_More_Sections is null Section?

  procedure Get_Next_Section ( T: Theater;
                             Current_Section: in Section
                             Next_Section:   out Section
  -- raise No_More_Section if Current_Section is Last

  procedure List_Sections (T: Theater );

  procedure Display_Seating_Arrangement (T: Theater);

  procedure Initialize_a_Theater (T: in out Theater);
  -- create an object of type Theater
  -- create a set of sections & init them with unique name

end Theater;
```

Figure 14: Theater LILEANNA Generic Package

4.6 Constraints and Rationale

The final attribute of a reference architecture is the composition and configuration constraints and rationale to be used by the application engineer in the application generation process. These constraints and rationale may take the form of traditional expert system rules or they may be informal text that is

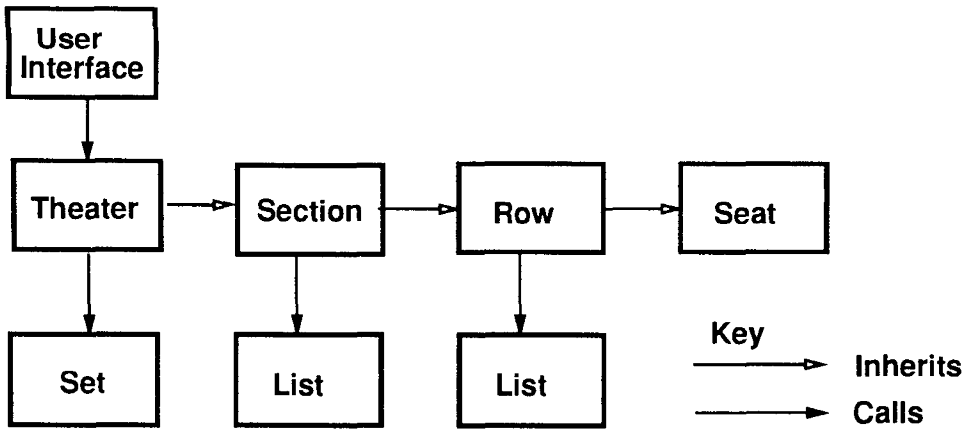

Figure 13: Reference Architecture Dependency Diagram

```
generic theory Set_Theory [ Item :: Triv ] is

  type Set;
  exception Item_Not_Found;

  function Is_in ( E: Element: S: Set ) return boolean;

  function Add ( E: Element; S: Set ) return Set;

  function Remove ( E: Element; S: Set ) return Set;
  --| not Is_In( E, S) => raise Item_Not_Found;

  function Size_of ( S: Set) return integer;
  -- see "Programming With Specifications" by David Luckham,
  --      pages 306-310 for a formal specification in Anna.
end Section;
```

Figure 15: Set Generic (Requirement) Theory

included as part of the design record or architecture schema.

Constraints indicate ranges of parameter values, relationships between parameter values or components (exclusion, dependency, etc.). Rationale may take the form of "rules of thumb" or "lessons learned" from using the reference architecture to generate various applications.

5 Real World Differences

This was a "toy example." Had this been a real example, you would have seen:

1. more vocabulary,
2. more diagrams, alternatives, requirements, etc.,
3. more time spent iterating over the domain model and reference architecture,
4. more disagreement between "experts," and
5. more complexity, which begs for tool support.

The interested reader should consult **DSSA Tool Requirements for Key Process Functions** [HRT94] for additional insight into the types of tools that assist in the DSSA process.

Furthermore, this example did not illustrate:

1. the use of OO design patterns to represent sub-architectures [GHJV94],
2. "flavors" of architecture components [Bat94] as a means of "wrapping" components to facilitate different communication protocols (connections),
3. alternative architecture styles [GS93, SG95], and
4. alternative architecture description languages [LAK+95].

Each of these topics provides additional insights to the DSSA process.

Closing Remarks

The DSSA process needs to be adapted to the culture of the domain it is being applied to. The domain model, reference requirements, and reference architecture are "best" represented using the tools and methodology that the application developers in that domain traditionally use.

Finally, because a DSSA is generalized across a product-line or family of applications, it costs approximately three times more to create [Wen94]. The expected savings can be appreciated in reduce application development costs (approximately a quarter) and reduced maintenance costs (approximately a third) based on experience in software reuse [Jon86, Tra87, Wen94].

For additional information on DSSA lessons learned, the reader should consult **Architecture-Based Acquisition and Development of Software Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program** [Hay94].

References

[Bat94] D. Batory. A Software Generator for Flavored Type Expressions. Technical Report ADAGE-UT-94-02, University of Texas at Austin, February 1994.

[CS93] L. Coglianese and R. Szymanski. DSSA-ADAGE: An Environment for Architecture-based Avionics Development. In *Proceedings of AGARD'93*, May 1993.

[CT92] L. Coglianese and W. Tracz. Architecture-Based Development Process Guidelines for Avionics Software. Technical Report ADAGE-IBM-92-02, IBM Federal Systems Company, December 1992.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J Vlissides. *Design Patterns – Microarchitectures for Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[GS93] D. Garlan and M. Shaw. An Introduction to Software Architectures. *Advances in Software Engineering and Knowledge Engineering*, I:41–49, 1993.

[Hay94] Architecture-Based Acquisition and Development of Software Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program. Technical report, Teknowledge Federal Systems, October 1994.

[Hid90] Proceedings of the Workshop on Domain-Specific Software Architectures. Technical Report CMU/SEI-88-TR-30, Software Engineering Institute, Hidden Valley, PA, July 9-12 1990.

[HRT94] R. Hayes-Roth and W. Tracz. DSSA Tool Requirements for Key Process Functions. Technical Report ADAGE-IBM-93-13B, Loral Federal Systems – Owego, October 1994. Version 3.0.

[Jon86] T.C. Jones. *Programming Productivity.* McGraw-Hill Book Company, New York, 1986.

[KCH+90] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[LAK+95] D.C. Luckham, L.M. Augustin, J.K. Kenney, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, TBD 1995.

[PD91] R. Prieto-Díaz. Reuse Library Process Model. Technical Report AD-B157091, IBM CDRL 03041-002, STARS, July 1991.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[Sch90] W.L. Scherlis. DARPA Software Technology Plan. In *Proceedings of ISTO Software Technology Community Meeting*, June 27-29 1990.

[SG95] M. Shaw and D. Garlan. *Software Architecture – Perspectives on an Emerging Discipline.* Prentice Hall, 1995.

[TC92] W. Tracz and L. Coglianese. DSSA Engineering Process Guidelines. Technical Report ADAGE-IBM-92-02A, IBM Federal Systems Company, December 1992.

[TPD+94] A. Terry, G. Papanogopoulos, M. Devito, N. Coleman, and L. Erman. An Annotated Repository Schema. Version 4.0. Technical report, Teknowledge Federal Systems, 1994.

[Tra87] W. Tracz. Software Reuse: Motivators and Inhibitors. In *Proceedings of COMPCON87*, February 1987.

[Tra93a] W. Tracz. LILEANNA: A Parameterized Programming Language. In *Proceedings of Second International Workshop on Software Reuse*, pages 66–78, March 1993.

[Tra93b] W. Tracz. Parameterized Programming in LILEANNA. In *Proceedings of ACM Symposium on Applied Computing SAC'93*, pages 77–86, February 1993.

[TSC94] W. Tracz, S. Shafer, and L. Coglianese. DSSA-ADAGE Design Records. Technical Report ADAGE-IBM-93-05A, Loral Federal Systems Company, July 1994. Version 1.1.

[Wen94] K. Wentzel. Software Reuse, Facts and Myths. In *Proceedings of 16th Annual International Conference on Software Engineering*, pages 267–273, May 16-21 1994.