COMPUTER ENGINEERING SERIES

COBOL Software Modernization

From Principles to Implementation with the BLU AGE® Method

Franck Barbier
Jean-Luc Recoussine

ISTE WILEY

COBOL Software Modernization

Series Editor
Jean-Charles Pomerol

COBOL Software Modernization

From Principles to Implementation with the BLU AGE® Method

Franck Barbier
Jean-Luc Recoussine

ISTE
WILEY

First published 2015 in Great Britain and the United States by ISTE Ltd and John Wiley & Sons, Inc.

ISTE Ltd
27-37 St George's Road
London SW19 4EU
UK

www.iste.co.uk

John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030
USA

www.wiley.com

Library of Congress Control Number: 2014955859

British Library Cataloguing-in-Publication Data
A CIP record for this book is available from the British Library
ISBN 978-1-84821-760-7

Contents

Acknowledgments

The BLU AGE method and tool have been developed for more than 10 years with a significant investment in terms of involved researchers and engineers, as well as money. The authors of this book would like to thank all contributors and administrative enablers.

The BLU AGE method has been partly funded by the European Commission through the ReMiCS project (www.remics.eu), contract number 257793, within the 7th Framework Program.

The authors also wish to thank the following people (in alphabetical order) who provided contributions, ideas, feedback, etc., so that this book could become a reality: Christian Champagne, Olivier Le Goaer and Alexis Henry.

Note from Franck Barbier

As books are long odysseys, their elaboration is not only linked to technical thinking at work places. Seeking harmony in life, philosophy and deep exchanges with "profound people" are, among other sources of inspiration, strong factors of idea regeneration, stimulation. In this spirit, I strongly thank great thinkers (and thus indirect contributors), philosophers (?), inspiring the people of this book: Sophie (with infinite love!), Vincent (when bicycling and canyoning!), Bruno J.

(within (too many?) long coffee breaks) and Bruno P. (within (definitely too many!) long body building exercises). Their presence, outreach, humanity simply, etc. help me a lot.

Acronyms

ADL	Architecture Description Language
ADM	Architecture Driven Modernization
API	Application Programming Interface
ASTM	Abstract Syntax Tree Metamodel
B2B	Business to Business
B2C	Business to Customer
BLU AGE	BLU Application GEnerator
BNF	Backus-Naur Form
BPMN	Business Process Model and Notation
BSP	BLU AGE Shared Plugin
CASE	Computer-Aided Software Engineering
CICS	Customer Information Control System
COBOL	Common Business-Oriented Language
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
CRUD	Create, Read, Update, Delete
DAO	Data Access Object

DSMLs	Domain-Specific Modeling Languages
DTD	Document Type Definition
EAR	Enterprise Java Archive
EJB	Enterprise JavaBeans
EMF	Eclipse Modeling Framework
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
ForTran	Formula Translation
FUML	Semantics of a Foundational Subset for Executable UML
HQL	Hibernate Query Language
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
IDL	Interface Description Language
IT	Information Technology
JAAS	Java Authentication and Authorization Service
JAR	Java Archive
Java EE	Java Enterprise Edition
JBI	Java Bus Integration
JNI	Java Native Interface
JMS	Java Message Service
JPA	Java Persistence API
JSL	Job Specification Language
JSF	JavaServer Faces
JSON	JavaScript Object Notation
JTA	Java Transaction API
JTS	Java Transaction Service

JVM	Java Virtual Machine
KDM	Knowledge Discovery Metamodel
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MOF	Meta Object Facility
MVC	Model-View-Controller
NIH	Not-Invented-Here
OCL	Object Constraint Language
OLTP	On Line Transaction Processing
OMG	Object Management Group
PaaS	Platform as a Service
PDMs	Platform Description Models
PIMs	Platform-Independent Models
POJO	Plain Old Java Object
PSMs	Platform-Specific Models
QoS	Quality of Service
SaaS	Software as a Service
SASTM	Specialized ASTM
SCXML	State Chart XML
SBVR	Semantics of Business Vocabulary and Rules
SEI	Software Engineering Institute
SLA	Service-Level Agreement
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SME	Small and Medium Enterprise
SQL	Structured Query Language

UDDI	Universal Description Discovery and Integration
UI	User Interface
UML	Unified Modeling Language
WAR	Web Java Archive
WS-BPEL	Web Services Business Process Execution Language
WS-Choreography	Web Service Choreography
WSDL	Web Service Description Language
XMI	XML Model Interchange

Introduction

The digital economy is expanding faster and faster. This results from recurrent advances in information technology (IT). There is a virtuous circle such that, in turn, more and more (often unpredictable) innovative usages boost IT. These usages are social and, in a broad sense, economical. More generally, the impact of IT on business is immense nowadays.

In this dynamical context, two decades ago software became the premier economy sector in terms of revenue. Substantial overturning occurred: the progress and multiplication of operating systems (LINUX, Windows, OS X, etc.) and associated product lines (e.g. mobile variants), the development and increasing re(use) of open-source software, outsourcing in developing countries, etc. United States and Europe tried to keep their advanced positioning through "differential software engineering": inventing new programming languages like Java or C#, new software development approaches like agile development, *model-driven development* (MDD), new software architecture paradigms like *service-oriented architecture* (SOA) and related middleware like *Java Enterprise Edition* (Java EE) or .NET and new computing paradigms like mobile computing, cloud computing or Internet computing as the superset of all modern computing paradigms.

However, a great paradox persists, as technological entry costs, human involvement, the acquisition of new technologies and take-up

initiatives, etc., are not easily and straightforwardly controllable. In other words, the great majority of people and teams in software development continue to use "old" technologies. First, a very good reason for this is that information systems on the top of these cannot be thrown overboard. Second, development hides maintenance. Challenges relate to software evolution not to software creation, especially from scratch. Seacord in [SEA 02] highlights this point as follows: "For large enterprise systems, a strategy of design *for evolvability* is a need. *This approach does not distinguish between development and maintenance; maintenance is simply continued product development*".

Therefore, there is a natural temporal gap between the emergence of any high tech software. and its daily use with total conviction and ensured return on investment.

In this sector, *COmmon Business-Oriented Language* (COBOL) remains a representative programming language. Namely, in 1997, 310 billion lines of software were in use and more than 220 billion lines were in COBOL (source: Wikipedia). Beyond this, five billion lines of new COBOL were developed every year, nowadays leading to an incommensurable mass of code. In fact, most enterprise software today is based on legacy technologies because COBOL had this ever-contested lead role. More recently, programming in (newer) COBOL also continues to have an impact on software architecture due to COBOL's inevitable "adaptation" to the Internet. Unfortunately, what also comes up with older COBOL is a set of specific infrastructures (computers called "mainframe computers" or "mainframes" for short), proprietary operating systems and middleware platforms, for instance *Customer Information Control System* (CICS); COBOL professionals who have a very particular background, culture, state of mind, etc., were/are also a great component of the overall COBOL influence on today's running software.

As observed, COBOL is thus not a closed world without any link to the Internet computing in general. That is the reason why going on with COBOL is always technically possible. Strategically, COBOL software evolution strongly depends upon proprietary solutions that

limit interoperability in reasonable costs, innovation and liberty in general, to better adapt information systems to business. There is indeed an increasing demand for reactivity: IT, information systems must leverage the business instead of, as it often happened in the past, being a source of inertia. Most data and applications are business-critical. Rapid changes in business require software and information systems with higher adaptation capabilities. With this new reality, it is not certain that COBOL computing is the strategic track to be followed.

Nowadays, the worldwide COBOL offer is "modern". COBOL may run on any operating system with seamless Internet integration. In spite of this modernity, the COBOL offer attempts to create a technological continuity between the legacy COBOL code and that newly produced. Furthermore, it also intends to keep cultures, practices or states of mind as-is; these issues are deeply discussed and strongly called into question in this book. More precisely, there is a renewal of expectations for IT users and stakeholders, *software as a service* (SaaS) especially, that may conflict with, not COBOL as a tool, but COBOL as a vehicle for old-fashioned ideas.

This book, *COBOL Software Modernization*, is not a front-end attack against COBOL. The reason for this is that COBOL has brought a lot of value to the initial integration of business in computers. With the Internet in particular, the way doing business has, however, changed in a radical manner that highlights COBOL as a very debatable technological solution. Accordingly, this book puts forward the idea of software modernization, in general, which may benefit from two proven technologies: MDD and SOA. This book strives to create the glue between the two in a ready-to-use "development + maintenance" technological framework.

I.1. Behind software modernization is "modernization": the car metaphor

Changing is not a natural way of being. People tend to be conservative. However, modernization without motivation and

expected progress makes no sense. Changing is merely a fear of people in everyday life. Changing contexts, environments and practices is also disturbing. To that extent, software has the particularity of accentuating this phenomenon through the incessant appearance of "new" technologies. By analogy, one may, however, wonder why I would change my 1960s car for a 21st-Century (electrical, hybrid, etc.) vehicle?

Rationale for change might be:

– cost savings (oil consumption);

– sustainability (pollution);

– safety (traffic rules have changed, encountered cars are "modern", (young) driver behaviors are not the same, etc.);

– technology issues: car parts are no longer available or they must be fabricated in a (costly) tailored way; mechanics are few or would likely be retiring. In short, obsolescence problems a rise on an exponential scale;

– today's route burdens, traffic jams contradict with the driving spirit of cars from the 1960s;

– etc.

Arguments for no change are: keeping the driving style, the driving emotion, the driving needs and sensations 1960s cars provide and passionate drivers' desire.

In business, any trade-off between "change" and "no change" often relies on survival. In software, old technologies often possess interesting features like robustness or stability, which may compensate benefits announced by high technologies' evangelists and promoters.

The car metaphor leads us not to view modernization as a panacea, but as an opportunity. Inserting an electrical engine into a 1960s car is definitely not the solution. Nonetheless, why not an electrical car provided that driving requirements are assured? Novelty and innovation in such a car might bring out unimagined pleasure. Why not?

In this car metaphor, 1960s car drivers are "legacy people" in IT. Driving style, emotion, needs or sensations match their business intelligence engraved for a long time in their applications. Modernization thus amounts to the adaptation of the contemporary (to-be-bought) car to their company's driving spirit instead of the contrary. Business intelligence pervades the soul of legacy people... technology is just a means.

This book is above all the end-to-end integration of human concerns in software modernization, considering modernization at large as a real vocation. In that area, COBOL has influenced people beyond technology. In the car metaphor, COBOL is probably one the most successfully adopted car product lines of the 1960s.

I.2. COBOL

The authors of this book were COBOL programmers. However, for a long time, they have changed to *object orientation* (OO) in general, and Java in particular. The history of programming languages and their design shows that, with time, psychological relations between languages and developers emerge. Developers adhere to languages, adopt them and eventually love (defend) them. In this scenario, there was/is an affective link between COBOL and COBOL programmers. The understanding of this intimate relationship is an important issue when subscribing to the idea that COBOL-based information systems are dinosaurs whose life is unsuitable in today's IT.

In fact, what are the reproaches against COBOL? Its historical programming style is probably far from the canon of modern code structuring. In this context, the common understanding of the expression "structured programming" is mostly associated with the class of Pascal languages: Ada and later OO languages that offer the top-level vision of "structuring". Is COBOL "structured"? Yes, undoubtedly it is. Beyond, it has/keeps sexy shapes, i.e. the language has fairly intelligible constructs. There is a clear organization of a program in divisions: *IDENTIFICATION*, *ENVIRONMENT*, *DATA* and *PROCEDURE* (see also Figure 9.13). Moreover, statements can

be considered as more explicit compared to ordinary languages. For example, *ADD*, *SUBTRACT*, *MULTIPLY*, *DIVIDE* or simply *MOVE* are the keywords that lead to smart code: *add 1 to counter* is indisputably more comprehensive than *counter*++ in Java. Labels are also appropriate means for cutting programs into well-delimited pieces even if they sometimes invite programmers to use global variables and/or *GOTO* constructs badly.

Three decades ago, Barnes in [BAR 84] already wrote, talking about Formula Translation (FORTRAN) and COBOL: "(…) these languages were not modern…". This modernism is the hidden part of the iceberg. In reality, the rule is always the same and applies for any language: the quality of programs greatly depends upon the dexterity of programmers. Typically, we may have ill-structured Java programs when programmers make ill-advised use of, for instance, inheritance as a programming support. To that extent, moving applications from an aging language to a modern language, must be neither a dogma nor an intrinsic strategy. In short, COBOL merits every respect, because it possesses all the required means to produce well-structured code. The question is then why, when analyzing COBOL code portfolios in organizations, COBOL programs are spaghetti dishes? Three causes are fairly well known:

1) From a focus on code portfolios, the drawback comes from the fact that a considerable quantity of the existing COBOL code has been produced by people without academic knowledge on computing. Many of them came from third-party functional sectors. Namely, bookkeepers, storekeepers, etc. became developers at the time when education was not able to provide enough trained personnel in computer science in general.

2) More importantly, in the past, software architectures were standardized by a data-driven approach, while today's SOA style views code mass production as pejorative compared to the necessity of software componentization and the need for fitting Internet computing architectures, like cloud computing frameworks. There is a net challenge about COBOL software maintenance, later developed in this book, which calls for architecture-driven modernization [ULR 10].

Roughly speaking, everybody is now convinced that COBOL software is more expensive, in terms of evolution especially. This was formally proven when programs had to cross the year 2000 (also known as Y2K).

3) In practice, COBOL does not favor abstraction at all. For example, data formats are totally rough using the "X" sign for alphanumeric data while "9" is used for numerical data. Worse still, the same raw data may be assigned to different variables with different formats for different usages using the *REDEFINES* clause. For example, coding years of dates with the "XX" or "99" data format thus precluded the crossing of 2000. Practically, "00"is a value conforming to "XX" or "99". Semantically, "00" is interpreted in non-modernized programs as the year 1900 instead of 2000. More openly, Lientz and Swanson in [LIE 80] told us that the breakdown of maintenance costs is such that 17.6% of maintenance efforts result from changes in data formats. So, out of many other factors, the absence of abstraction facilities in COBOL is a certain source of maintenance budget overrun.

Despite these three COBOL pitfalls, this book's spirit is not the definitive conviction of COBOL as being solely responsible for the expensive construction, operation and support of today's existing information systems. We may acknowledge that COBOL was a good means to spread business intelligence into information systems. In this line of reasoning, the native business-oriented nature of COBOL is a source of propitious inspiration when modernizing information systems toward newer technologies. In parallel, regarding the new deal of the digital economy, it is broadly accepted that COBOL and its surrounding technologies (mainframes, etc.) are not the best tools to address dynamicity issues: information systems must change according to a real-time scale. This leads us to view software evolution, even adaptation at runtime, as a renewed challenge in terms of software management. In short, what is good in COBOL (its business facet and culture) must, in a conceptual way, be retained when modernizing.

I.3. Why the Cloud?

Discussing COBOL modernization toward SOA obliges us to pay attention to cloud computing. How do we keep our head on our shoulders when reading software marketing reports and surveys that proclaim "the Cloud"! Who has not heard about cloud computing? Nobody! In the same line of reasoning, everybody probably has a hazy comprehension of this expression. Eventually, will cloud computing mean "fog computing"? Perhaps it will if no precaution is taken. This feeling comes from an exaggerated media hype without, for average IT professionals, the possibility of having the time to digest concepts, reorganize their own ideas (demystifying buzzwords, having trustworthy references to well-explained experiments etc.) and simply testing the Cloud as a not-so-different way of using computers and networks for business.

In terms of money savings, cloud computing promises a lot. Typically, computing infrastructure pooling is a key concern of IT managers. Intuitively, cloud computing is at least understand able as an appropriate solution for this. What else? A lot of computing paradigms behind the Cloud are confirmed, even reinforced, namely SaaS: "Put simply cloud computing is the infrastructural paradigm shift that enables the ascension of SaaS" [MCF 12].

I.4. Legacy2Cloud

We comment later on in this book on the Cloud and SaaS in relation to many other cloud-related notions. We strongly believe in the Cloud's power as, in the past, we trusted COBOL as an originally ingenious language for business. This book capitalizes on the better of the two through the *Legacy2Cloud* paradigm. This is a jump from a world with high skill on business-oriented programming to a world of ever-seen flexibility provided by computer/network infrastructures (adaptability, elasticity, reactivity, etc). Beyond this vision are a lot of tricky technical issues to be addressed to actually enable such a jump.

Is this extraordinary jump realistic? In this scope, in [MCF 12], it is written:

> "*Legacy applications* As a general rule, older applications do not make good candidates. That is because legacy applications tend to be rigidly coded, with outdated programming constructs, lots of references hard-wired into the code, and a reliance on a single (usually massive) database. There is no easy way to transform such an application into a flexible, agile cloud service, so you are most often better off leaving such applications *on the ground*."

It is also written:

> "*Modern applications* All things considered, newer applications that use modern programming techniques and architectures are well suited to a cloud migration. This is particularly true of multi-tiered applications that are based on the web and the Internet standards and that use multiple, distributed databases."

When reading this text extract, the answer to "Is this extraordinary jump realistic?"would be: "No, it is not." It is indeed observed that only already modern applications may go to the Cloud. Such an opinion misses the fact that, on a massive scale, such applications do not really exist, i.e. there are not many. For example, the move of a Java EE well-formed application to the Cloud, even if it may require some days (even weeks) of energy, is not the core of the game. Pragmatically, why the Cloud if the latter is only concerned with "the nobility of software applications"? Within such a vision, the Cloud's take up might be dramatically slowed down. Instead, the real possibility of skipping intermediate shapes (i.e. applications that are "modern" without being cloud-based) is the Holy Grail, this book's challenge. The resulting question is about the elaboration of a method and a supporting tool that are able to accomplish COBOL software modernization in an end-to-end and seamless way.

I.5. Human weight on successful modernization

One key concern of this book is the integration of people in modernizing activities. Once convinced of transforming COBOL applications into Java EE for instance, the modernization method must greatly take care of business analysts, project managers, software architects, developers and so on to succeed. The culture, the background, and the professional life of these people were/are so different. They probably come from computer prehistory, but their know-how, their knowledge of their organization (institution, agency, firm, *Small and Medium Enterprise* (SME), etc.), and their control on business, are all of inestimable value. So, modernizing COBOL software is above all a collaborative work to extract business nuggets: business logic, i.e. data semantics, rules, functionalities, etc. In this context, skilled "legacy people", high technology coaches and powerful *Computer-Aided Software Engineering* (CASE) tools are required.

I.6. This book's structure

This book proposes a reflection on software modernization to align old information systems with current Internet-centric computing paradigms, mainly SOA and the Cloud along with highlighting popular middleware platforms, namely Java EE[1], .NET, Spring, etc.

The book aims at addressing business and technical issues from Chapter 1 to Chapter 7. Practical insights into a ready-to-use method and tool are in Chapter 8 and Chapter 9. In this scope, this book comments on the BLU AGE method and tool used in large-scale projects in the USA and in Europe for varied business domains: healthcare, retail, transportation, tourism, energy, manpower, government, etc.

1 Java EE is a standard encompassing a family of compliant application servers: Apache TomEE, GlassFish, JBoss, etc.

1

Software Modernization: a Business Vision

1.1. Software-based business

As of today, there is a great paradigm shift. In past decades, software was the unavoidable way to "automate business" in the logic of cost and time savings, productivity, and better quality in product and service delivery. More recently, "software became recognized not just as an automation tool but more broadly as a strategy for providing products and services not yet offered" [FAV 11]. In other words, nowadays, software is a non-removable part of products and services. Software may be embedded in a car, for instance, leading to attractive functionalities (assisted parking). Another example could be a jewel reseller who is able to provide online authenticity certificates for its products through its accession to a trustable international organization in charge of regulating such certificates (respect of laws sale tracking, etc.). In both cases, software delivers some business added value.

Companies whose primary activity is selling software are reputed to provide intangible goods [POP 11]. The distinction between these and other companies is tending to disappear. Car manufacturers of the future will thus, instead of selling "a car", sell "a computer" and hardware/software interoperating with an engine, a chassis, an interior, a steering wheel, etc. Jewelers will probably be in a similar situation due to the irreversible interpenetration between the Internet and business activities.

The shift is the fact that the business model of "modern companies" is changing, critically relying on software. In this context, transforming car engineers into software engineers would be a huge challenge, or, in the opposite way, a very bad idea; this is the same for jewelers. So, new business models have to be invented to tame software.

From a software engineering viewpoint, we mean it is important to build software differently and beyond this to have software evolution under control because of proliferation. In this line of reasoning, most of the classical software providers still suffer from handmade practices. Introducing these practices in non-software companies might be a nightmare. Software divisions of future companies will include software builders/maintainers or not. In the negative case, at least, business analysts and innovators will constitute these divisions to offer differentiating, and thus competitive, goods and/or services. Finally, stand-alone software will no longer exist to the benefit of cooperative pervasive (more or less big) software components irrigated by the Internet.

1.2. Information-driven business

The value coming from software is the computed information. Forthcoming software-based business models must then focus on information-as-a-revenue and try to diminish the costs generated by software creation, maintenance and utilization.

Today's entrepreneurship success is thus strongly ruled by information. As an immediate result, organizations (companies, administrations, etc.) continuously grow their dependency upon information and thus information technology (IT).

In this context, business processes increasingly rely on high-end information: undisruptive availability, liveliness, sharpness, easy digestion (even "digestibility"), rich semantics and creation of meaningful knowledge from computed information.

Business processes are powered by information systems whose criticality, optimality and dynamicity, i.e. efficiency in short, are key concerns of business analysts, software project managers, software architects and software developers. These people think about and maintain applications on a daily basis, which are edges of an ill-delimited graph, even imbroglio, of information channels (hardware and software). Over the years, nobody has the global overview of this graph. Worse still, everybody wonders why this graph does not collapse as a paper castle built from a card game. The rule of the game is now clear: the crash of the information graph is the straightforward bankruptcy of the organization.

Figure 1.1 shows a common vision of information and information systems in organizations. On the right-hand side, the *computer layer* not only goes on providing operating means for business automation, but it must also be a booster.

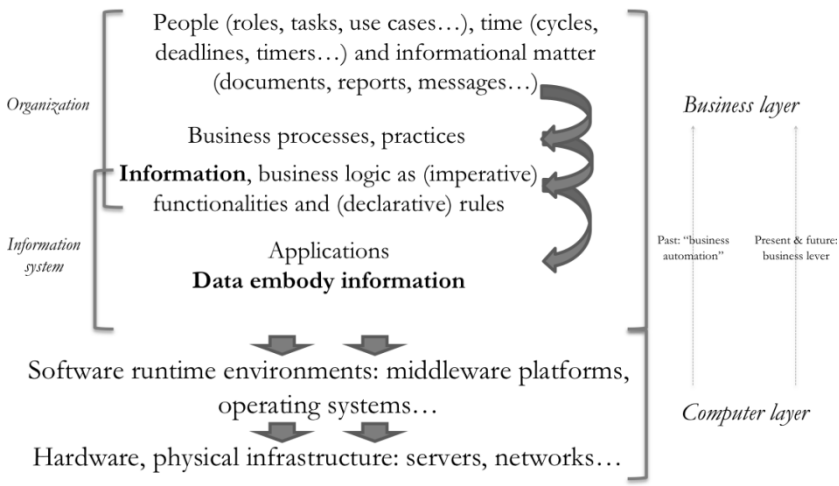


Figure 1.1. *Information as an ever-rising value in organizations*

1.2.1. *Adaptation to business*

Information systems are an abstract view (Figure 1.1, left-hand side) of software applications and databases, middleware platforms, operating systems and related hardware (servers, mainframes, personal devices and computers) and infrastructures (power feeding, server cooling, networks, both local area networks and wide area networks, etc.). In essence, information systems constitute a logical view, which focuses on the immaterial assets of computer environments: information, its structuring, organization, production and delivery means.

As a metaphor, information systems are similar to a set of services offered by a town: municipal libraries, book loans, magazine consultation, buses, green car renting, kindergarten children's entertainment events, etc., with related synchronicity, e.g. bus schedules fit to libraries' working hours, children's entertainment events, etc. In such a context, town citizens do not care about librarian and bus driver salaries, fuel in buses, libraries' heating, etc.

In this line of reasoning, it has always been tempting, even healthy, to isolate information from its physical implementation. This approach aims at better considering information-as-a-service. Instinctively, information consumers do not pay attention to computing environments being hardware or software.

Designers of information systems thus have the permanent difficulty of guaranteeing and maintaining high-quality services wrapping information processing. The difficulty mainly lies in hiding intrinsic problems from piled (hardware and software) layers and recurrent failures. As an analogy, a bus drivers' strike would probably diminish the quality of the town's services to citizens.

For a long time, the ideas of architecture and urbanization have taken a prominent place in IT. It is important to notice that we consider architecture or urbanization of information systems in a logical way. As discussed before, information systems are mind views while in practice bits move about within circuits. Thus,

information pieces, building blocks, etc., have virtual connections, links, etc., whose awareness is a key aspect of information management at large. Architecture is related to software that powers information systems, while urbanization is a macroscopic wrapper including information channels, forms, circulation, restitution, etc. Both urbanization and architecture act as a basis for, respectively, information systems and applications. Cartographic representations of these (sample in Figure 1.2) can be made more or less explicit, depending on their rational nature. Rationality aims in essence at controlling useless complexity.

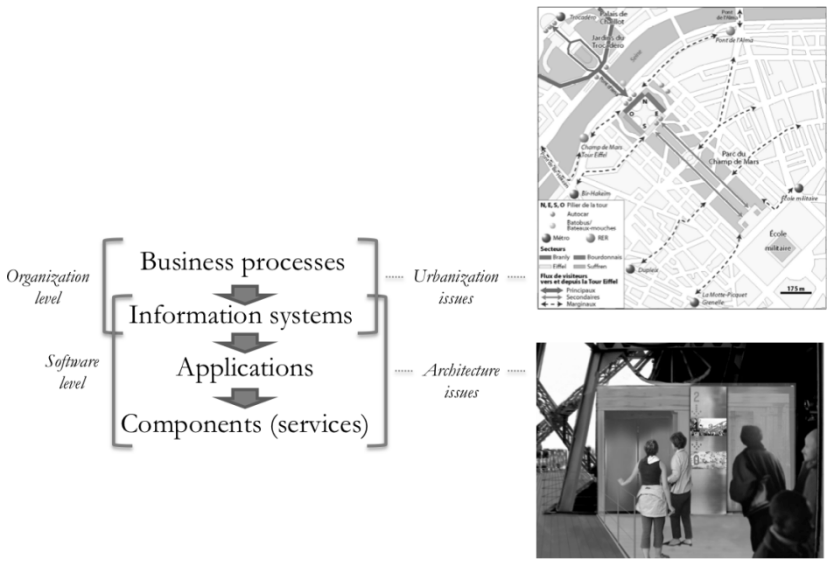


Figure 1.2. *Urbanization and architecture*

As an analogy, Figure 1.2 shows the case of tourist flow management for the Eiffel Tower. Urbanization (right-hand side, top of Figure 1.2) copes with transportation infrastructure in connection with tourist visit routes and coarse-grained throughputs. Architecture (right-hand side, bottom of Figure 1.2) is concerned with "solutions" (e.g. signage). Information boards about visit routes are components of a chosen architecture to perform tourist flow management "at runtime". Services rely on components, for instance, displaying on

boards the next times of bus, boat, subway, etc., arrivals at closer transportation stations.

Gradually, information system designers face newer challenges. While architectures take time to become optimal, nowadays, they are expected to be/become variation-prone. Today's economical contexts (globalization, trend reversals boosted by the Internet and consumers' zapping) call for changing business: practices and processes at the organizational level. At the underlying level, information and logic engraved in information systems are surely subject to modifications as well. While organizations may require business practices and processes to rapidly adapt, information systems do not have the same latency. Re-architecturing is above all an offline activity. Fortunately, not all business adaptations involve re-architecturing, but information systems must be thought by designers to cushion business shocks: that is the new deal.

Returning to the case of tourist flow management for the Eiffel Tower, re-architecturing could be the review of the existing information systems for dealing with sporadic phenomena (e.g. cold/heat waves) or frequent events (e.g. sport shows), which may increase or decrease the presence of tourists. The case of a heat wave may, for instance, call for fit-like-a-glove services: boat traffic and arrivals to the Seine river embankments have increase to allow people to refresh themselves on the water when departing or arriving. In other words, customers will prefer boats to the detriment of subways, buses, etc.

So, nowadays, since architecture variability cannot be ignored, information systems should gain more flexibility. Typically, architecture components must, on demand, collaborate in a different way and/or extend collaborations with third-party components often unknown at design time. As mentioned in the introduction, service-oriented architecture (SOA) is a solution principle, but a lot of progress is expected in this research field.

In [BAT 14], it is especially revealed that acting on architectures is often infeasible due to excessive complexity. The difficulty to sort out

business logic from this complexity is high. Instability of architectures (the contrary of variation-prone) is thus the phenomenon when interventions in architectures' inner workings generate long periods before recovering stability.

As an overview, business pressure is such that information systems must demonstrate a kind of real-time evolvability. In this scenario, attenuating the adherence between information systems (as the immaterial value of organizations) and computer facilities (both hardware and software) seems to be a perpetually renewed challenge. The well-known weakness of information systems is their poor reactivity in terms of requirement adaptation while, in contrast, today's business is subject to very frequent variations, even shocks. In other words, long-term strategies related to information management poorly comply with volatile short-term business activities.

1.3. The case of tourism industry

The sector of tourism is indicative of the increasing and inescapable intertwining between IT and business. Gallo and Krupka in [GAL 08] argue "(…) travel companies will face the need to introduce *in-depth changes* to their business strategies in order to adapt to the changes affecting their customers. (…) The development of new products and services and the adaptation of the offer to global customer trends require *a great deal of innovation*" (emphasis ours). In reality, as in many other sectors, tourism to a great extent relies on IT to support this innovation. Nonetheless, IT can also be a source of possible setback when companies are slowed down by rigid information systems.

As an illustration, Figure 1.3 shows what might be an economical process whose aim is the customization of travel offers for new customer profiles, namely singles. Invariably, the creation of new business services leads to new software services (and their tricky connection with what is existing). At the bottom of Figure 1.3, software evolution is caught in a cost vise. Two contradicting requirement streams drive changes: innovation in scope and daily

business. Experience especially shows that change implementation is a source of regression. Namely, one may observe what follows: what works perfectly at a given time after months of effort can spontaneously become out of order. As an illustration, the addition of new services for singles is both an extension and a modification (coupling with the existing architecture's components). To get the job done well, modification may call for "adaptation" in existing components. Afterward, these do not serve the daily business (unexpected failures) while they did before. The expected innovation and its associated revenues may then be significantly penalized by the impossibility of driving software evolution in a timely manner under controlled costs.

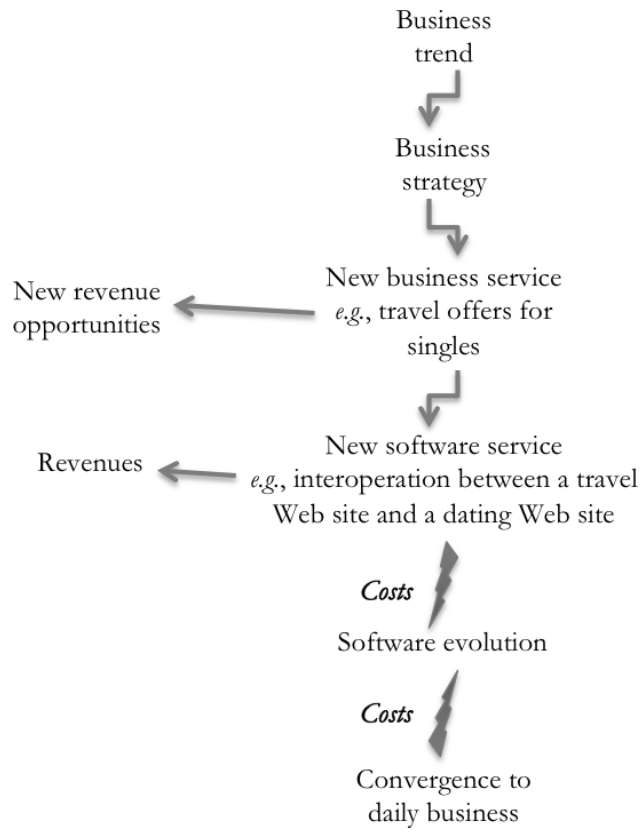


Figure 1.3. *IT and software evolution positioning in the fluctuating tourism industry*

In all business sectors, in people's minds, IT is often rightly considered as an aspirator of financial resources. That is true when IT is no longer observed as a business developer. Moreover, people outside the IT world do not understand why IT is costly (a euphemism) while, *par excellence*, it is the technical field where competition is fiercer, innovations are bigger and, accordingly, costs linked to hardware/software parts (e.g. open-source software libraries) are increasingly lower.

Regarding the tourism industry, it should benefit from both the Internet (as an ever unbound marketplace) and IT advances, which together reshape the Internet-based possibilities of doing business. Nonetheless, over the years, the tourism industry has been unsettled by the Internet, which created an excessive, even confusing, offer with an exacerbated competition. In fact, the globalization of tourism business diminishes sales margin, relying on adaptive information systems not to miss pioneering revenue opportunities.

New players, new deals, new rules of the game, etc., appear in quasi-real-time. The paradox is that IT makes possible this liveliness, while software applications must accordingly behave differently to cushion new business events. Ultimately, this leads us to ask developers to change code and in the worst case to reformat software architectures. The latter is both a source of stress and risk and, unfortunately, software crash before reaching a new stable situation, which, in turn, does not meet the very last business expectations. This infernal circle can only be broken with flexible software frameworks.

Tourism players, such as hotel chains, tour operators, tourism agencies/organisms/consortia, transporters and car rental companies, are involved in both business to customer (B2C) and business to business (B2B) commerce. For instance, hotel chains may buy excursions from tour operators while the latter buy bedrooms from these chains.

New players are, for instance, health centers because a confirming trend is the fact that customers associate travels with the possibility of care: dental care, plastic surgery, fitness, etc. Another trend is the possibility of collaborating with real estate agents, which can supply

different kinds of accommodation, and thus multiply the types of lodging on offer.

New deals can be joint and/or bulk purchasing, subcontracting, product/service sharing, partnership with price comparison Websites, etc.

The new rules of the game are, for instance, the fact that end customers include implicit concerns when ordering travels. These are security, sustainability, privacy, responsible tourism, etc. In the best case, such values might be transformed into paying services, which probably require collaboration with specialists. In the worst case, these values may be in contradiction to cheap offers.

Intuitively, from a software viewpoint, it turns out that, *a minima*, tourism applications must be able to exchange data. Beyond this, we may simply imagine, for example, the connection between a health care center software and a travel management platform to book and arrange care stays within touristic stays. This link is similar to service interoperation between travel and dating Websites in Figure 1.3. Each business adaptation case would probably lead to a specific software technical problem. Reasoning case-by-case results in numerous induced problems whose piling is inevitable and resolution is very long.

For software experts, SOA, later discussed in this book, is an appropriate approach for organizing software so that interoperability succeeds beyond data: applications may evolve incrementally through new services (i.e. functionalities) and/or new service composition. In the case of the mentioned travel management platforming, we should have the possibility of easily, straightforwardly and transparently calling for services accessible from the healthcare center software, provided that the latter has been thought up, designed and equipped with interoperability abilities, say, secure Web services since medical data require more privacy.

Beyond the excitement provided by the Internet, there is an actual potentiality for IT to favor reactivity in business. More generally, IT

and information systems must be the springboard for business adaptation in shorter and shorter cycles.

In the common business-oriented language (COBOL) world, this vision is a myth. In the Internet and cloud computing worlds, using Java platforms/technologies in particular, SOA is a technical reality. Nonetheless, from a business perspective, SOA often remains a (later reachable) goal: no company, in the tourism sector in particular, has developed such appealing adaptation capabilities to absorb very high business fluctuations. In very rare cases, only software aims at changing. In effect, business processes around applications also have to mutate in involving users differently, modifying usages (roles, tasks, documents, frequencies, etc.). Mutations generate natural inertia, which is most of the time incompatible with the time slots required to have software applications that instantly suit requirement fluctuations.

So, software modernization, with a focus on COBOL, is not only a technical issue to be addressed. There is a crucial need to have enough reactivity in business processes, information systems and software applications/components that simply help rapid development/ maintenance. There is a challenge in moving legacy systems to renewed ones. This challenge especially amounts to, as much as possible, separating business concerns from technical constraints.

More generally, the top of Figure 1.4 shows that IT may sometimes be a hindrance when rigidity in information systems prevents any kind of adaptation. Beyond technical issues and the particular case of COBOL, the idea of software modernization is thus the progressive erasing of such rigidity.

1.4. IT progress acceleration

Theoretically, IT progresses are the source of inexorable improvements in the functioning of information systems. Empirically, this position statement is false. It turns out that the migration of any information system, or information systems part (applications, components, services, etc.) from one “legacy” technology to a

"modern" technology, may be, without experience feedback and expertise, a nightmare, never mind the costs.

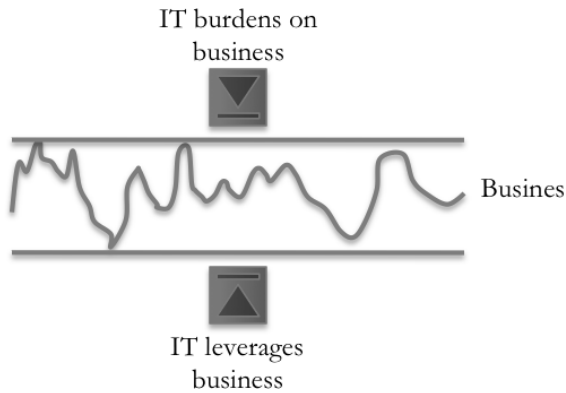


Figure 1.4. *IT may be both a business stimulator and a brake*

Broadly speaking, high tech may be viewed as a lure promoted by "evangelists" who never, in the past, present and future, used/use/will experience the high tech they have built and promote aloud. Lies about high tech are in essence its masked inefficiency due to its (natural) intrinsic lack of maturity, weak testing, poor adoption and small-scale utilization. High tech with maturity, representative experimentations, rich feedbacks and lessons learned, etc., is actually no longer high tech.

From a business viewpoint, high tech is of little interest if it does not address business issues. Results are not necessarily immediate and tangible. If they are deferred, the high tech implementation method must, however, provide guarantees in time; tangible progresses through returns on investment especially must occur after a certain period of time. Observable cases are rare; they are often accompanied by debatable numerical data and statistics. However, there is a poor communication on absolute failures whose counting and deep analysis is thus illusory. This context does not favor experience exchanges in an impartial manner and contributes to freeze the opinion of legacy people: they are still negative about software (unjustified) "sophistication" at large.

For COBOL professionals, model-driven development (MDD), SOA, agile software development, cloud computing, etc., may thus be considered as high tech whose maturity needs to be proved first. The specificity of business sectors, companies and well-isolated activities may also be an argument against going out of legacy contexts. The adhesion of people is the rule. Feasibility surveys and studies are helpful; proofs of concept are essential to convince COBOL people. High tech inventors are respectable in their recognized role for IT progress acceleration, but they generally have a tight vision on business.

Roughly speaking, there are at least two very different coarse-grained categories of computer professionals: business application builders and generic software providers. The latter develop open-source software, commercial off-the-shelf (COTS) products, etc., and act as suppliers for the former. The former meet end-users that are difficult to synchronize with domain requirements. The main task of application builders is to push and acquire information to and from the reality of organizations. This task does leave time for integrating new software technologies in application development frameworks: for instance, switching to an object-oriented programming language, adopting and setting up an agile software development method, etc. So, pragmatism is as follows: high tech has to be thought like any science contribution: effectively shared and beneficial for humanity; if not it will be forgotten or postponed to the next century.

1.5. Legacy world

The legacy (software) world is the sum of legacy technologies, legacy information systems, legacy applications and “legacy people”, anything apparently aged but still delivering the expected business services in time and quality – is this a paradox? Not really… Behind the “legacy” term is probably a lot of expertise, long experience, background and wisdom. Briefly, summarizing “legacy” as something pejorative is often misplaced.

In [BAT 14], there is a very recent interesting summary on interviews of IT practitioners about their own perception of legacy

software systems and their possible modernization. Recognized qualities are: "(76.7%) business-critical, (52.8%) proven technology, (52.3%) reliable system and (24.4%) performance". This directly confirms the idea that "legacy" conveys positive values. In contrast, it is concomitantly agreed that strong factors impose modernization: "(1) high maintenance costs, (2) lack of knowledge, (3) to remain agile to change and (4) prone to failures". Discerning readers may detect questioning contradictions in this survey, for instance, how a legacy system may at the same time be a "reliable system" and "prone to failures"? In fact, legacy systems are diverse in nature: people share common characteristics like "something aged", but they may disagree about criteria like "reliability" above.

In this line of reasoning, the qualification of a software system as "legacy" is not systematically linked to a programming language such as COBOL: "more than half of the informants do not agree that the programming language is a determining factor for a system to be legacy, while the rest were in agreement" [BAT 14]. However, half of the respondents' legacy systems are known to be built on top of COBOL.

In [NAS 08, p. 6], another survey on legacy systems is given: "inability to be adequately supported, maintained, or enhanced" (82.8% of interviewees) is the premium discriminating criterion for "legacy".

Evolvability, or more precisely, the proven absence of this potentiality is, on the spot, what better qualifies "legacy". In relation to our prior analysis on the new business deal, the question is: "why such a significant concern about evolvability?". The answer that comes is the same: "inability to meet business needs or system not agile enough to continually meet the challenging needs of the organization" [NAS 08, p. 6] (79.3% of interviewees). This criterion is not independent of the first criterion, since evolvability mainly results from business need fluctuations. The interesting word in the previous text extract is "continually". IT practitioners no longer view maintenance as discrete, but as inevitably continuous. Driving changes without break periods has definitely become "the job".

1.5.1. *Exiting the legacy world*

In fact, economical considerations prevail over technical ones; this is mostly true because the latter are induced from the former. More precisely, the ratio between the immaterial value of information systems and the cost of ownership and the technical debt associated with these systems is a balance indicator. The worry factor, i.e. decrease in this ratio, is the door to software modernization.

As discussed previously, information systems are the core source of information-as-a-revenue, but they may behave like an old car whose oil consumption is no longer consistent with the essential services to be delivered: transport from point to point. In this metaphor, using a public bus is similar to replacing a legacy software system by a COTS software package.

So, exiting the legacy world is just a breaking point with respect to the inadmissible deviation of financial indicators. Software modernization then becomes an actual concern before being an obsession. Nonetheless, in all surveys, people never claimed that filling the gap with the newest technologies is the motivation behind software modernization.

Strategically, organizations want to avoid technological silos like developing solutions based on an evident isolation with the Internet, cloud computing, etc. Beyond this, using these newer technologies may be unacceptable, particularly because of entry costs. In [NAS 08, p. 12], “funding” has rank 5 (in a scale of 1–5 with 1 being “not challenging” and 5 being “extremely challenging”). This criterion is recognized as the first major obstacle for modernization. In these times of crisis, budget constraints drastically limit the spectrum of candidate methods for modernization.

Returning to the metaphor of the energy-consuming car, modernization is not just the replacement without awareness of the old car by a cost-saving car. In other words, a smooth ride, for example, can be an existing practice to be kept. More generally, old car usages are probably associated with the best cost-saving practices: best-

known circuits, shortcuts, car sharing, etc. We mean the immaterial value of information systems is nothing but the business value buried in computer memories and storages (information), as well as programs (functions and rules, logic in short). Only modernization methods based on a solid extraction and an intelligible reconsolidation of this business value make sense. This approach may attenuate the "funding" disease through the fact that modernization is first and foremost porting business intelligence from one target to another.

In this spirit, high tech and legacy technologies are not opponents. They are just different means, whose appropriateness is strictly linked to different time slots. Accordingly, we may then write that any high tech is the legacy technology of tomorrow. This strongly confirms that modernization methods cannot be proposed in terms of point-to-point technology mapping and transfer. The consistent and complete expression of legacy systems, once ported, independent of new technologies, is thus so vital.

1.5.2. *Legacy world professionals*

IT is strongly characterized by mutations, which apparently and permanently call for "people brain updates" in terms of acquired knowledge, technology comprehension and so on. There is, understandably, a natural reluctance to follow up these mutation cycles, which are numerous, frequent, but sometimes volatile and unjustified. Worse, they are sometimes just hype. Being open-minded must *a priori* be the rule in IT, but experience shows that most of the worldwide software development stakeholders have no professional time to devote to IT news, in terms of knowledge enhancing especially. Beyond this, the volume of technology releases (products, versions, application programming interface (API), standards, even paradigms, etc.) is simply too huge.

We cannot, without any nuance, talk about the inability of people to apprehend technology jumps; it is certainly only a matter of time. In this context, software modernization is either an opportunity to invest in new knowledge or it may be viewed as the end of

"tranquility". As mentioned above, people share the positive opinion on legacy software systems as being "reliable". Behind the "reliable" word is the fact that any long professional life with only one line-of-product concern (e.g. COBOL) is a sure way to converge to "reliable" systems, i.e. we must read here: "systems with full controllability". More generally, technology capitalization contradicts high tech. The source of stability and full control of information systems relies on keeping old technologies under long-term utilization despite the fashions.

Software modernization is in essence the moment that has been pushed away for a long time. Technology jumps in non-chosen moments are then problematic because they are human-centric. Several psychological and cultural barriers may strongly slow down the process to move forward. By translating software to ill-known technological targets, feelings such as creativity vanishing, loss of control, being software robots, etc., may increase for individuals or groups.

There is another source of trouble in IT. The persisting craftsmanship in IT is the consequence (or perhaps the cause) of the not-invented-here (NIH) syndrome. In effect, software reuse might be considered as a semi-failure at the beginning of the 2010s, while the origin of the software crisis was put forward in the 1960s. People persist in considering that they build so-specific software. Any the software from outside is, in this scenario, suspicious. This is both true and false. This is false because, as a counterexample, any new employee after learning periods must be able to play a significant role in existing software evolution. This is true because the proximity with end-users is irreplaceable. To that extent, software outsourcing is nowadays identified for certain types of software only.

Another key human factor of software modernization is the "graying" of IT staff. Employee retirement is an everyday event in organizations. A driver for software modernization is then often this human factor. However, beyond the loss of human (technical) resources is the loss of business intelligence. Indeed, computing wrongly remains a technical discipline omitting the *raison d'être* of an information system as being the nervous system of organizations.

Organization management is preponderant. As already discussed, the careers of IT people close to retirement are almost always based on an economical background. Software modernization in this scenario is the true opportunity to mine this business intelligence before retirement.

1.6. Conclusions

A justified criticism against IT is the fact that it was created to assist organizations in management and business, but in increasing the number of applications, information systems tend to become incoercible. We mean, in an organization, IT components (hardware + software) not only become more complex but also tend to exist to only feed each other. Keeping IT and business converging is an everyday battle, which calls for more and more effort, means and money. IT people are skeptical about newer technologies because they do not actually deliver what they promise. Non-IT people do not understand why previous important investments in computing infrastructures do not solve problems in a timely manner. These people only want to relate to information systems from the surface. On the opposite side, IT people cannot easily argue that information systems' inner workings are very difficult to monitor and manage.

Honestly speaking, IT people are overwhelmed. They cannot step back. They suffer IT. Concretely, in COBOL for instance, programs come from nowhere. Technically, each appears as yet-another-retaining-wall. From the business viewpoint, over the years, IT components have begun to look like patches whose direct positive impact on the business is often imperceptible. In this context, legacy information systems are naturally guilty. Why then modernize with the risk of standing still?

This depressing vision contradicts news in IT magazines, Web blogs, great-fanfare announcements, success stories, etc. Indeed, this chapter shows that a new deal may exist: software is no longer the means for information processing; it is the source of extended business through the idea of information-as-a-revenue and that of

service in SOA. With the Internet's unfinished culmination, there is indisputably a paradigm switch. Software and information are no longer only helpers or boosters; they are "the value". Precisely, services as consumer goods and services as software artifacts become increasingly less distinctive. Intentionally, SOA and the Cloud are the up-to-date software supports to favor such a convergence. The case of the travel industry is representative through the endless opportunity to develop and sell new services. In such a revolution, software is componentized; components are business-related and pervasive including high availability and strong dependability.

Because business without people is meaningless, this chapter also mentions that revolutions, even though technological, cannot ignore people's aspirations, cultures, experience, know-how, etc. COBOL software modernization arises in line with this healthy observation.

2

Software Modernization: Technical Environment

2.1. Legacy system

Until now, we have singled out an intuitive idea of what a "legacy system" really is. Being massively constituted of Common Business-Oriented Language (COBOL) applications does not qualify a given information system as "legacy". NASCIO in [NAS 08, p. 2] proposes the following definition: "A *Legacy System* is not solely defined by the age of IT systems (e.g. 20 years) as there are many systems that were designed for continued upgrades, but the term also focuses on elements such as "supportability", "risk" and "agility", including the availability of software and hardware support, and the ability to acquire either *internal* or *outsourced* staffing, equipment or technical support for the system in question. The term may also describe the system's inability to adequately support "line-of-business" requirements or meet expectations for use of modern technologies, such as workflow, instant messaging (IM) and user interface".

In this definition, the age of the legacy system plays a great role, but this criterion is not enough. In [ORA 08], it is highlighted that two other factors play a greater role: "(…) that "agility" and "adaptability" top the list of business drivers prompting the modernization of legacy

systems[1]". As written in Chapter 1, legacy systems were designed where change was the exception, not the rule. Today's business perpetual oscillations call for mechanisms to conduct recurrent (small or medium) changes in information systems, ultimately leading to the possibility of revising applications in time-to-market and cost-effective compatible cycles. Precisely, "agility" and "adaptability" mean the potential to be agile and adaptable. So, behind the idea of software modernization is primarily the idea to make "agility" and "adaptability" tangible in the modernized systems, whatever the modern technology concretely used.

In short, the term "legacy" both provides a positive and negative idea. The positive is the idea of heritage of business know-how. The negative is the fact that this know-how is engraved in technology in such a way that the legacy system's daily functioning penalizes the business.

2.2. Modernization

In the literature on legacy systems, several words refer to the transition from outdated systems to newer ones: modernization, replacement, migration, renovation, recasting, revamping, etc. It is thus important to first sort out this word list.

Simply speaking, since systems have lifecycles they have to die someday anyway. The motivation behind transition is their evident business value while their evolution is complex and costly, even no longer supportable. This business value may be difficult to measure and obtain. For example, a COBOL program generating a report from several flat (often odd) files keeps a business value through the fact people continue to read the report. They in particular have the possibility of bringing with them the report at different (professional or not) places. However, in a world of mobility, we may imagine the availability of the report's data on smartphones and tablets, with probably more digested presentations/interpretations: charts,

1 Ranking of this concern in [NAS 08, p. 9] is 4 (a high attributed value) on a scale of 1–5.

consolidated indicators, etc. Transition to novelty is thus above all an opportunity to make a thorough inventory of business practices. In other words, suppressing applications like this report editing application is a kind of modernization.

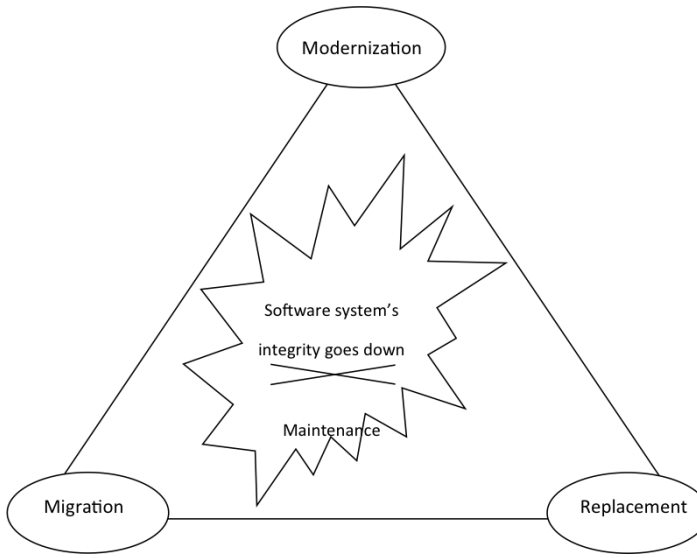


Figure 2.1. *The dilemma between modernization, migration and replacement*

In [COM 00], there is a typology of software system evolution between maintenance, replacement and modernization (white-box and black-box). Maintenance is characterized as having limitations when it no longer conforms to its initial mission ("(…) bug correction and small functional enhancements (…)"). Namely, if the software system's integrity cannot be guaranteed then traditional maintenance has to be questioned. This is the case when a sum of changes is uncontrollable in the sense that malfunctioning results from the sum and cannot be explained via a well-isolated change. The system's integrity is violated because all changes are like drugs whose concomitant ingestion creates a new disease.

Replacement or modernization are substitutes, but Comella-Dorda *et al.*'s survey misses the notion of migration, which has a great resonance in industry through today's modernization tools. Figure 2.1 shows the appearing dilemma when maintenance is no longer the solution.

2.2.1. *Replacement*

Replacement is the radical abandonment of the old system to build a new one from scratch. In the best case, the old system is a source of inspiration, but in many circumstances, the total absence of documentation, knowledge or informed people prevents such an inspiration. Independent of business concerns, the legacy system often cannot be extensible, worse, it can be "untouchable": adding a few COBOL statements at any place in the code is a sure crash followed by several weeks of repair. Such an operating mode for maintenance is unrealistic in a professional framework. As an illustration, the French billing application for landline telephony is a COBOL dinosaur that was subject to a couple of strict-replacement attempts: each led to a failure. Replacement is in essence highly risky due to some empiricism, i.e. undefined methods to re-engineer the business expertise. For examples, billing rules may have many exceptional cases imposed by aged, but still applicable, regulatory clauses coming from (forgotten) agreements, contracts, laws, etc. In such a case, replacement imposes extreme-value requirements' engineering actions with unpredictable results.

Later on, replacement includes an overlap phase where the old and the new (completed) system run, for comparison purposes, in parallel. The latter must be as functional and robust as the old one. There is a risk of degraded service. For business-critical applications like billing, this may correspond to non-encashment or litigation costs on bills, a nightmare for a company. This remark encompasses the vivid need of carrying out the full testing of the new system with respect to the old one. The question is to what extent the older may serve as a reference to measure and establish that the newer offers equivalent

functionalities, even the same quality of service. We come back later on to this crucial issue.

2.2.2. *Migration*

Different from "replacement", "migration" covers either a lightweight or heavyweight code transcription [SEL 03].

"Lightweight" is especially the case when one moves from an obsolete COBOL dialect (e.g. COBOL Pacbase whose maintenance by IBM is no longer supported) to a "modern" COBOL. Here, "modern" means that we guarantee that the generated COBOL code is actually surrounded by perennial (efficient) maintenance tools. Common (soft) cases are when organizations only want to move non-maintainable COBOL, a bottleneck, to something, which again becomes evolvable. Another use case is a COBOL-to-COBOL solution, which mostly consists of addressing architectural issues, i.e. moving the code from mainframes to platforms with distribution capabilities or, more frequently, adopting the Web three-tier application style. We may also carry out the migration to another programming language, say C#, with the necessity to fit the transcribed programs to the new platform constraints, in this case, .NET.

In fact, when the initial code leads to no significant restructuring when observed in the new target (language and/or platform), this is "lightweight". In this case, most data structures remain as is. Unfortunately, the output code remains cryptic; it is thus still subject to sizeable long-term maintenance costs. In this line of reasoning, migration to object-oriented COBOL might be a ticking time bomb if existing data structures are transformed into classes in a one-to-one mapping approach. We mean that generating ill-structured OO programs is possible when we do not, as expected, dogmatically apply OO principles (encapsulation, inheritance-polymorphism, exception handling, etc.).

There are always optimization opportunities when programming for a target platform/technology. Moving to object-oriented COBOL makes sense only if the reuse of classes in libraries, for instance,

classes in a persistence-dedicated library, is effective. Most of the time, migration misses this possibility or, in the other extreme, recreates a too much excessive adherence to the target platform/technology. This is the case when the transcribed code includes many abstruse platform/technology details.

As a comparison, the migration can be qualified as “heavyweight” when the programs are looked into thoroughly. For instance, varied concerns on data (unexplained dispersion, unjustified replication, low access performance, etc.) may involve a language-to-language transcription in concomitance with sizeable code reorganizations. These challenges exist in relation to the utilization of new data supports (e.g. migration from flat files to SQL-like databases). Typically, the introduction of data access objects, as proxies between computations and data stores, becomes useful to separate the data semantics from data codifications. This naturally leads to a broader review of the existing data structures. Another kind of “heavyweight” migration is pure code refactoring when the code needs reshaping for further reuse. Hybrid approaches apply of course.

The key feature of migration, being lightweight or heavyweight, is the fact that the primary concerns are a technology-to-technology focus. Both lightweight and heavyweight transcriptions have the risk of being offered by a technology provider who proposes her/his “future legacy proprietary technology” as is the case with object-oriented COBOL. There effectively exist contemporary COBOL technologies, which comply with the Internet, distribution, service computing even cloud computing. The key drawback of “migration” is the fact that issues are tackled through, solely, a technical angle. Technology-to-technology encompassing language-to-language transcription is thus a lure when there is no serious attempt to distinguish between the technology facets and the business logic.

Figure 2.2 illustrates the risk associated with migration: it is surely the direct transcription of, not only code and data, but the imbroglio between the two as well. Put simply, the existing chaos is ported from a legacy to a modern technology, so what?

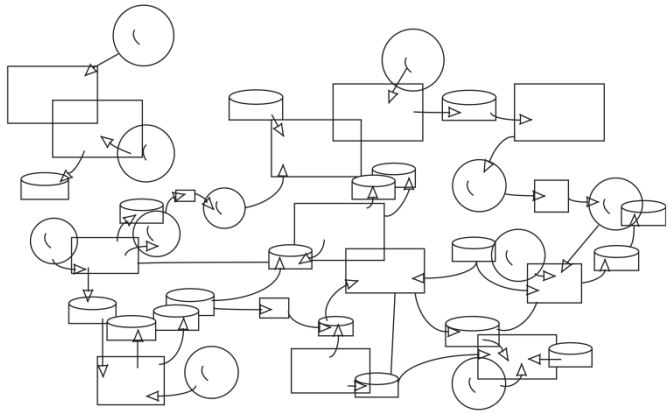


Figure 2.2. *COBOL software jungle as candidate for migration*

2.2.3. *Modernization versus migration*

"Modernization" as promoted in this book is twofold. Black-box modernization (a.k.a. "renovation") amounts to repainting applications. Legacy systems are analyzed through their inputs and outputs. From this analysis, "black-box modernization is often based on wrapping. Wrapping consists of surrounding the legacy system with a software layer that hides the unwanted complexity of the old system and exports a modern interface" [COM 00]. In this case, applications have to smell, to feel "modern", but there are no real changes, consequently, no progress either.

Black-box modernization raises the problem of adding layers without a clear limit and thus adding unwanted sophistication, a certain source of future complexity. Besides, stacking layers is often risky in terms of performance comprehension and effectiveness. Black-box modernization may also be, contrary to COBOL-to-object-oriented COBOL transcription, the surrounding of old COBOL within object-oriented COBOL. There are thus many ways of hiding a legacy system within a modern appearance.

"White-box modernization requires *an initial reverse engineering process to gain an understanding of the internal system operation*.

Components of the system and their relationships are identified, and a representation of the system *at a higher level of abstraction* is produced" [CHI 90].

White-box modernization is the method defended in this book. In this sense, reverse engineering acts on the legacy system to create knowledge on its inner workings. However, the resulting representation from abstraction must ultimately be an expression of the business logic engraved in the old system. The core goal of white-box modernization is then a technology-neutral representation of the legacy system. Having the ability to list the system's components and their semantic relationships (e.g. "includes", "calls", "occurs before", etc.) gives knowledge on the old architecture. Nonetheless, this view has *no* high value if it does not enlighten us on the way this architecture serves the business. From experience, the old architecture is fully off topic in the forthcoming modernized system. As detailed in this book, the extraction of knowledge in the legacy system is a multiphase process, each phase delivering representations mixing business and technology system properties. This mixing progressively decreases in the course of reverse engineering. A white-box modernization process is thus based on an iterative method separating one-block views in related viewpoints. The code view is the first nugget extracted. In a simplified line of reasoning, an architecture viewpoint and a business viewpoint might be deduced from this code view.

For example, Customer Information Control System (CICS) calls in COBOL are eminent parts of the application's architecture:

– one-digit precision:

  IF … THEN MOVE 1 TO precision.

– two-digit precision:

  ELSE MOVE 2 TO precision.

  END-IF.

– 'Currency' program call:

  EXEC CICS LINK PROGRAM('Currency').

From the code perspective, these COBOL lines are a statement suite. From the architectural viewpoint, this is a (calling) link to a packaged program named *Currency*. From a business viewpoint, this informs us that the business logic interweaves with currency conversion functionalities in two different ways, i.e. the business logic has two calculation precision rules: 1 or 2 digits. Unfortunately, we have to be aware that the structuring level of average COBOL programs is significantly lower, compared to the code above. This results in duplicated business functions and rules, whose codification is not uniform at all. There is a huge need to reconsolidate these. As a comparison, recall that, in a migration process, the business logic is deemed to be immutable.

2.2.4. *The superiority of white-box modernization*

An open issue about modernization is the capitalization of the business value. To that extent, choosing between replacement, migration, black-box or white-box modernization favors in any case the, possibly wide, (re-)visitation of a legacy system, a rarely encountered occasion for properly (re-)expressing its business value. However, modernization methods differ in power on that concern.

For example, in [COM 00], they sketch the ever-topical "Functional (Logic) Modernization" of an anonymous legacy application by means of the encapsulation (wrapping) of the business data and logic with the help of the *Enterprise JavaBeans* (EJB) technology, the core computing support of Java EE. Java EE had many advantages: vendor-neutrality, cloud-compliance, durability, being a worldwide standard, having a broad support offer, being a recognized, proven and widespread technology… Encapsulating a legacy application (a kind of black-box modernization) using the EJB technology or something equivalent is a bad idea despite the listed advantages of EJB.

Looking at the same problem with migration would lead us to replace the legacy code by EJB code without significant revisions. In effect, migration does not really address the following issues: what is the buried business value? How can we restructure and/or

re-architecture? With migration, there is a risk to reinvent the wheel by implementing, for instance, *several times* an EJB component offering currency conversion functions.

Differently, pure white-box modernization may demonstrate, through abstraction, the need for such calculations without any assumption on how these may be supported at runtime. As an illustration, appropriately, a remote Web service may ensure these calculations once and for all, in all code places calling such functions. As a summary, white-box modernization does not systematically imply redevelopment. Besides, white-box modernization is the only way to have enlightened opinions, e.g. to decide suppressions. This is a very key issue of modernization: simplification as a springboard of easier maintenance.

In practice, white-box modernization is the approach that requires a pivot representation of the legacy system, both free from the outdated and targeted (up-to-date) technologies of interest. Another open issue about white-box modernization is to keep only valuable things, even make them much more simple in the interest of evolvability. So, white-box modernization revises systems to always remain evolvable. This opinion is confirmed in [SEA 02]: “Before systems can be evolved, they must be evolvable. Transforming legacy systems *to the point where evolvable software development again makes sense* is accomplished through legacy system modernization”. In this book’s vision, this declaration of course excludes black-box modernization as an appropriate solution.

As an overview, modernizing a system in a white-box manner is above all the action to make it evolvable for its entire lifecycle instead of straightforwardly moving it to a current technology (migration) to make it, as soon as possible, operating. Frequently, business functions benefit from being rationalized (removed, merged, split, enhanced, etc.). Such maintenance actions must only occur on the pivot representation promoted by white-box modernization. We show in Chapter 7 in particular how model-driven development (MDD) supports this idea.

2.3. Software engineering principles underpinning modernization

Chikofsky *et al.* in [CHI 90] formally defines three key expressions that refer to principles used in software modernization: "reverse engineering", "design recovery" and "re-engineering". "Reverse engineering in and of itself does *not* involve changing the subject system (…) It is a process of examination, not change or replication." Based on this characterization, replacement may possibly rely on reverse engineering while migration and white-box modernization must necessarily rely on it. Design recovery is a kind of reverse engineering process in which information on the legacy system is produced not only from the code, but from other sources: documentation, experienced people, etc. Beyond this, the produced information is both observations and deductions. As shown before, the CICS-based call to a *Currency* program is surrounded by some code on a "precision" global variable. The whole code is a business clause: currency conversions vary from one logic ("precision" = 1 digit) to another ("precision" = 2 digits). The extraction of such a business rule is impossible without semantic interpretation. Moreover, further analysis is required outside the scope of this simple code to formally detect and formalize the full business rule, which leads to "precision" = 1 digit or "precision" = 2 digits.

From the observation that reverse engineering and design recovery are read-based processes, re-engineering, instead, may be viewed as a write-based process: "Re-engineering (…) is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form" [CHI 90]. This ever-topical vision stresses the distinction between the modernized system in a new form and its deferred implementation in a specific process: forward engineering. This vision clearly affirms the dichotomy between migration and white-box modernization through, for the latter, a pivot representation before the reconstitution. In this scope, end-to-end modernization is such that reverse engineering produces software artifacts, which are well-prepared for forward engineering. We show in the rest of this book that UML (standing for Unified Modeling Language) models, due to their neutral nature, are good candidates for supporting pivot representations between reverse and forward.

For intuitive comprehension, Figures 2.3 and 2.4 sketch the MDD principles behind, respectively, reverse and forward engineering. For example, the recovered (deliberately simplistic) model at the bottom of Figure 2.3 does not refer to the plastic matter used for the vintage car's dashboard. Nowadays, this matter is probably unrecyclable. In short, the model at the bottom of Figure 2.3 is nothing but abstraction in action. In their very deep nature, models leave us the possibility of forgetting what is/becomes worthless.

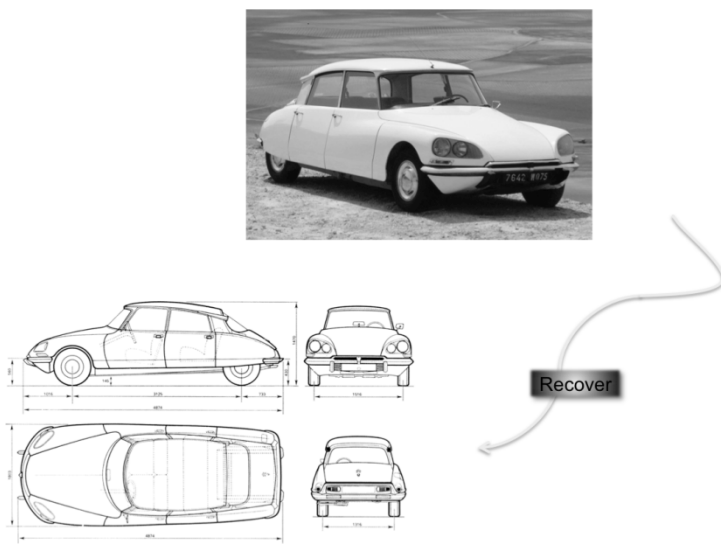


Figure 2.3. *Reverse engineering as a re-industrializing metaphor of a vintage car (pictures are taken from autoautomobiles.narod.ru)*

The recovered model is a more or less complete basis for creating a modernized car. In this case, the business logic is the car's lines, an often-encountered style, a source of prior success, "the value". The model perfectly reflects this value.

Figure 2.4 is a more common case of modeling. Car engineers are premier users of MDD. The generated product from the model, even robotized, is subject to a long run. However, everybody agrees in the car industry that models allow product line management, supply chain rationalization, deferred assembly for late customization and more.

What is intangible in Figure 2.4 is a service-based approach. Conceptually speaking, it is awkward to convince car manufacturers that a car is a computing cluster with a middleware platform, both surrounded by mechanical elements: engine, chassis, interior, etc. (see also Chapter 1). This cultural rupture is a strong means of integrating mechanical/electronic/software components in order to differentiate a car product line from competitors. Car SOA is then the idea of easily pluggable services: park assistance, car-to-car communication, etc. The latter component is a palpable incarnation of the link to Internet computing.

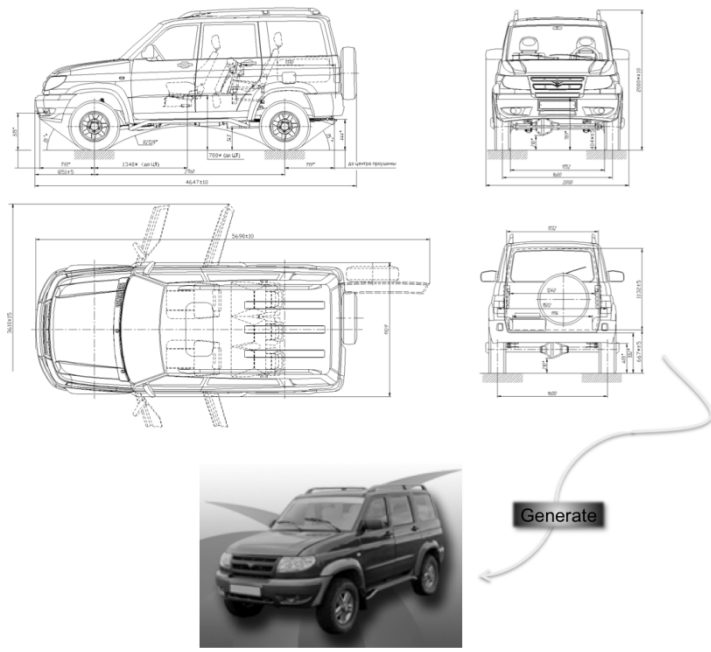


Figure 2.4. *Forward engineering as car computer-aided design (pictures are taken from autoautomobiles.narod.ru)*

2.3.1. *Re-engineering in action*

Nowadays, reverse and forward engineering have a consensus in terms of both definition and practice. To that extent, there exists a plethora of re-engineering methods and tools. In virtually all of these,

there is an assumption that the legacy code is the primary matter. In this context, reverse engineering always amounts to producing a syntactical code modeling, which is an instance of a *Backus-Naur Form* (BNF). A BNF is a metalanguage, a set of (meta)-words and grammar rules. Existing code is divided into terminals and non-terminal pieces. For example, a given "IF … THEN … ELSE … END-IF" COBOL occurrence in the legacy code obeys the "IF CONTROL STRUCTURE" decomposable element of the meta-language. The grammar rules tell us, in a formal way, how elements may be composed. Typically, "ELSE" clauses can only be part of "IF CONTROL STRUCTURE" elements, "ELSE" clauses are optional in "IF CONTROL STRUCTURE" elements, etc. In essence, in the existing code, each "IF … THEN … ELSE … END-IF" occurrence respects these rules. Parsing the legacy code is thus above all a classification of its tokens and a (superficial) comprehension of its organization. This organization has no rationale because, over years, maintainer "strokes" have eroded it. Another analogy is to view maintainers as firemen. They save people (the business), but fire hoses may damage houses much more than the contained fire. This damage is ill-formed code organizations. In the end, there are no direct means to explain why this organization is as it is. This is similar to dismantling a terrorist bomb. Bomb parts are unknown; they are connected in such a weird way to lower as much as possible any comprehensibility.

As an illustration, returning to currency conversion functionalities in some existing COBOL code, there are a lot of chances for these functionalities to be duplicated, dispersed and in multiple code patterns. Recall that the prior COBOL code, which is a well-modularized CICS call to a *Currency* program, is the exception not the rule in terms of adequate structuring.

BNF-oriented models of legacy systems are thus only the beginnings of long stories. The trickiest issue is the move from code representations to representations with "sense". How semantic representations may be shown off through viewpoints on code (architecture, business logic, marked transactions, potential errors,

etc.) is the challenge. Multiple representations in space and time are therefore required.

In time, this corresponds to the discretization of the re-engineering process. For example, one representation includes references to the operating system and/or runtime middleware (e.g. CICS). At the next step, the calculated representation is free from these adherences. In this context, the subject legacy technology is gradually erased. Since re-engineering encompasses the alteration and later "reconstitution" of the legacy system in another form, representations aim at being enriched at a given time. As underlined above, white-box modernization is a boon for carrying out modifications. These may be refactoring to create a new system, which is really evolvable. Recall that migration has no focus on creating a new system, whose future maintenance is facilitated. Modifications may also be concerned with additions: added functionalities and so on.

In space, a good strategy is the management of representations, which are perspectives or projections of others (a.k.a. viewpoints). What is needed is traceability in general, both in time and space. A representation of the legacy system architecture is a more or less accurate form; it calls for adequate metalanguages. Indeed, software architecture as a self-contained domain has its own concerns, which impose meta-words: component, connector, assembly, deployment, service, etc. In other words, we cannot represent a given software architecture with a BNF-oriented model. In another domain like business logic, we need other concepts: business rule, business function, etc. As shown later in Chapter 7 of this book (Model-Driven Software Modernization), many standards have been elaborated from the *Object Management Group* (OMG) in particular. The *Semantics of Business Vocabulary and Business Rules* (SBVR) metalanguage is one of these standards; the *Knowledge Discovery Metamodel* (KDM) metalanguage, which overlaps with SBVR and owns another part totally dedicated to software architecture, is another one.

The major difficulty of the re-engineering process is the management of consistent relationships between proliferating representations that have dedicated roles. Ultimately, we expect the

code generation of the modernized system in line with the target (contemporary) technology: this is the implementation representation resulting from forward engineering. In any case, please note again that the higher-value representation is that named "pivot"; it is intermediate and above all free from both the aging and fresh technologies chosen at modernization time.

So, "reverse engineering", "design recovery" and "re-engineering", which includes "forward engineering", are principles to be applied in software modernization. However, there are related issues, which may ruin any re-engineering plan. These are volumes (and thus scalability issues in re-engineering), measures (particularly testing) and (re)-integration (revamped systems must be re-injected in stationary environments).

2.3.2. *Re-engineering challenges*

Chapter 1 of this book enumerates many challenges, obstacles, brakes, etc. that are financial, human or managerial. Once the modernization decision has been established, today's technical contexts show that the size of the programs/applications, in the COBOL world especially, the necessity of measuring (in fact, proving) the acquired advantages of the new system compared to the old one, and the articulation of what has been modernized with what remains in the same state, are critical issues. In other words, "intelligent" (versus "naïve") re-engineering is not just a canonical process as sketched in [CHI 90]. It is a set of methods and best practices to cope with scalability, testability and integrability. From experience, all kinds of models of legacy systems are heavyweight software artifacts when managed in specialized tools. Considering design recovery for example, inferring information from model parsing can be doomed to fail because models are numerous, big and strongly interlaced.

Modernization methods without tools are unrealistic. Reproduction, systematization and agglomeration of microscopic modernization actions must be offered in tools regardless of the size of software artifacts. In this scope, the openness of tools is also

important for tailoring methods and practices; this happens in relation to variations of technical contexts, stakeholders' expectations, and constraints in general.

Testing is a representative case. The only way for measuring a certain quality of the new system is testing. How is testing material represented? Possibly, when the extraction of the testing material occurs? Is the formulation of test cases and scenarios automated? Is it achieved by means of a uniform formalism? How are test cases and scenarios executed against the modernized system? How do test failures have an impact on the finalization of the modernized system before deployment in production? etc.

A re-engineering process for professional software modernization then calls for many advances in software engineering at large. The question is what could be the federating approach, which allows us to tackle so many problems in a uniform way (languages for representations, modernization well-founded actions for representation calculations).

To close, (re)-integration is the reconnection of what has been produced and certified in terms of expected qualities. Statically, this can be the quality of the new code with respect to code quality standards: future maintenance is no longer expensive. Dynamically, this can be the performance of the new system in interaction with an other information system's pieces. To anticipate, it could also be useful to have models of runtime environments to prepare (re)-integration.

2.4. Conclusions

Software modernization may be understood with in a variable-geometry sense. There are effective techniques behind software modernization intentions: reverse engineering, forward engineering, etc. This chapter lays down the bases for software modernization through "models". In relation to the business concerns from Chapter 1, this chapter puts under the spotlight models in the spirit of MDD. Namely, models are the means for making the business logic emerge

from legacy systems. Under no circumstance, must modernization be thought of only in terms of technology-to-technology transfer. Accordingly, white-box modernization, despite the fact it calls for more sophisticated methods and tools, is the way to exhibit the sole interesting value engraved in legacy systems:

– the business logic (data semantics, functions, rules),

– and the way applications empower business practices and processes. After modernization, we may then aim at knowing how the renewed applications might better support these as well.

Modernization is a unique boon to reconcile IT with business, provided that source or target technologies (programming language, middleware platform, data storage system, etc.) do not interfere with the reflection behind modernization: the move from a breathless system to a service-based system. This gap is linked to massive COBOL code bases and data as discussed and characterized in the next chapter. The expected jump is of course impressive, but it is worthwhile from a business perspective. Applications in companies are more or less business-critical. Possibilities are numerous for candidate experimentations.

3

Status of COBOL Legacy Applications

As written at the beginning of this book, throwing out COBOL cannot be considered as a serious and sufficient motivation. Well-structured, modular COBOL programs exist; they may probably remain as-is for many years. Besides, as noticed in this book's introduction, millions, even billions, of COBOL lines of code are still produced each year. We may imagine that the majority of them intrinsically have no legacy status in the sense that they obey contemporary computing: distribution (Web three-tier architecture style, service-oriented architecture (SOA), etc.) including mobile computing, service computing or cloud computing, object-orientation, component-based development with associated benefits, reuse especially. Nonetheless, the biggest volume of COBOL code (around 90%) comes from the 1970s, 1980s, 1990s… it is degenerate. Decades of maintenance have resulted in the situation that nobody knows what this code really does. More precisely, nobody is able to explain, in retrospect, why "algorithms" and thus execution flows follow a given path rather than any other.

Paradoxically, programs and applications are often fine-tuned and highly optimized with regard to their running environments, mainly mainframes and customer information control systems (CICS) as favorite middleware. Globally, COBOL is recognized as doing the expected job in "survival conditions". Practically, when there are bugs, these are often known and circumvented with the means at hand.

This is laborious, but it is compensated for by an intimate knowledge of programs, applications and their functioning.

As an illustration, here is a true story. Looking at some aged code in a company, we found around 20 blank lines in a source file. This space was specifically designed for copying/pasting 20 lines of code coming from another ancillary file. After discussion, the person in charge told us that the initial program, with 20 blank lines, may sometimes get in correct results, even crash. In this case, the 20 lines of code are injected in place of the free space; the program is re-executed without errors; the 20 lines of code are then removed to re-obtain the free space. The program with free space is later executed several times without any problem until the next round. In this science fiction scene, nobody, including the person in charge, was able to explain the rationale behind this "uncommon" code manipulation. On the contrary, everybody simply claims: it works.

The evoked degeneration of COBOL programs and applications is the fact a non-negligible amount of COBOL code is just an addition of patches for bugs perceived in other parts of the COBOL code. We mean, when deficiencies arise, there is a trend to add new programs to tame these deficiencies instead of first analyzing their local source and next applying radical changes. This is not an informed choice. It is most of the time impossible to intervene at some code places without creating a strong destabilization of program chains.

In this context, in numerous organizations, subcontracted maintenance by third-party companies is most of the time only keeping programs and applications afloat. Indeed, practice and experience show that programs and applications may neither decrease nor increase in functionalities while maintenance costs explode. Concretely, new programs are inserted in program chains for repairs, which mostly consist of the production of new (intermediate) files having different access types, different data formats and dependencies. Subcontractors may have an interest in supporting such a kind of evolution. Applications embodied by program chains become ever more complex. This approach applies on an exponential scale, leading to non-understandable logic, being technical (e.g. the prior 20 blank lines) or business.

3.1. OLTP versus batch programs

In COBOL, there is a traditional dichotomy between batch programs and transaction processing (TP) or on-line transaction processing (OLTP) programs. For young programmers, these two notions make no actual sense. OLTP programs are just “classical” programs in the current world. The word “transaction” most likely refers to “real-time” data reading and writing in relation to immediately visualizable results. A transaction is also *a priori* concerned with the idea of something, which is directly interpretable with respect to the business: billing, shipping, hiring, accounting… or, more precisely, any subactivity of these.

In modern computing, programs transform data in databases within transactions. Commit or rollback actions on data depend upon consistent changes on these data. For example, an ATM withdrawal must not lead to a data insertion in a database table (bank account debit) if the cash dispenser goes down. Simply speaking, a transaction is (this is most of the time also true for OLTP programs) associated with a consistent suite of business actions. Any failure when executing an action is a failure of the suite: a cancellation is required through “rollback”. In contrast, no failure at all leads to “commit”. In this context, a transaction manager ideally is a technical service (e.g. Java Transaction Service or JTS in Java) in a middleware that powers transactions. At the origin, COBOL programs did not systemically rely on a true transaction manager as JTS. So, the OLTP acronym does not imply, word for word, transaction management as characterized in modern computing. In fact, OLTP programs are above all interactive programs in the sense that end users are behind screens when executing them. On the contrary, batch programs operate without interaction with end users. That is the simple key difference.

So, in COBOL, transactions are effectively business-oriented, but, unfortunately, they are excessively coupled with screen inputs/outputs and thus they are mainly user-oriented. In modern computing, transactions are also business-oriented, but they are detached from presentation issues. From the Model-View-Controller (MVC) programming principle, transactions belong to the Model side and not to the View side (presentation). Considering distribution concerns in

general and three-tier architecture issues in particular, transactions are then associated with deployable components shared between applications while COBOL OLTP programs require their own stuff: transactions cannot be shared at the middleware level.

With Java Transaction API (JTA) for instance, this supposes the implementation of coherent business action suites as standalone software components, a very rarely encountered case in COBOL. Beyond this, transactions play a central role in modern enterprise computing. The accentuation of distribution poses significant problems in coordination of distributed transactions especially. In modern enterprise computing, transactions have to better match business processes, which are services that are partly dived into everywhere. Commit actions thus depend upon error-free execution for the involved (remote) services. As for rollback actions, they are key for error recovery management, provided that external services may notify and deliver rich information on failures; they also must have internal fault recovery capabilities like, for instance, fail soft mode functioning. As a summary, transactions in the spirit of OLTP programs are somehow far from today's transaction management.

3.2. Mainframes

The weight of mainframes in the malformation of COBOL programs can be discussed through the idea of "vertical computing". In other words, under the hypothesis that programs consume resources, mainframes impose a concentration of these resources (files especially) on a single machine. This approach is highly centralized compared to distribution, which is at the core of the Internet. Mainframes are computers that greatly favor sequential processing. Over the years, this reflects the progressive construction of a homemade culture in terms of software design experience and expertise, software architecture style, ways of thinking and thus designing programs in general.

By curiosity, scientific computing with Formula Translation (FORTRAN) has followed a totally different path with the use of massively parallel machines. A side effect is that COBOL programs

on mainframes often deliver good performance in terms of speed when facing high volumes. This results from intense customization and optimization based on the consideration that resources are very close, permanently available and above all unshared. As an analogy with programming, this context is similar to an old-fashioned program in which all variables are global. Software engineering has demonstrated that such a program is the worst form of programming about the impossibility of controlling undesired side effects.

In COBOL, resources tend to be multiplied (files especially) and arranged for a single type of usage only. Usages are heterogeneous, so requiring dedicated programs; this is again the source of program proliferation. The penalizing counterpart is code intelligibility, which is low due to high adherence to data format. Maintenance issues are often non-shareable with people outside the closed circle of initial designers/programmers. Subcontractors as the persons in charge of maintenance have total control and thus have the opportunity to exclude new incomers concerning big portions of the existing COBOL code. As to the future, porting such programs to other types of computers will probably lead to control loss (unpredictable performance, random reliability) if no redesign occurs.

3.3. Data-driven design

In the COBOL dimension, the absence or weakness of network infrastructures is the reason why the pervasiveness of data cannot be an assumption at the time of software design. Of course, COBOL programs, old or less old, run in network infrastructures. Nonetheless, the COBOL background culture is not inspired by full exploitation of such infrastructures.

Internet computing is in essence the remote access and processing of data at large. In COBOL, replication and next dispersion are mechanisms for supporting data pervasiveness. However, there is a very poor, or completely absent, data consistency management associated with replication.

So, the COBOL-oriented organization of data in storage supports drastically influences the way algorithms and thus programs have to be thought of and thus run. In other words, this organization always generates the risk of slowing down calculations. To reduce this risk, programs are data-driven, or more precisely, data format-driven. Accordingly, to design applications whose response time is compatible with end users' expectations, say, at most three seconds to obtain results on screens, programs are created that are tortuous in their actions, e.g. they may change/expect special data shapes before any processing. Applications as program chains become complex with "weird" programs whose business intelligibility becomes void.

Interactions with users in OLTP programs tend to be shortened to smooth the load between OLTP programs. A well-known side effect is the absolute necessity to postpone some second-level processing to other moments: batch programs.

3.4. COBOL degeneration principle

Taking the example of a voting system, people have to register if they intend to vote. The capture of data by officers for voters occurs in the opening hours by means of a P1 OLTP program. Entered data are written in a raw style in an F1 file whose organization is sequential (Figure 3.1).

The verification and validation of data occurs for the night with the help of a P2 batch program. This batch has F1 and F2 as inputs. The latter is a file recording the list of "birth places"; it has a direct access mechanism based on a ZIP code that is hashed to retrieve the government-compliant location of a given birth place in the file. The main role of P2 is the production of an F3 file comprising the list of erroneous data records in F1, namely the entered voting people with inconsistent and/or suspicious birthplaces. We may imagine many other Pi batch programs for any other kind of checking. There is also possibly the need for another P3 batch program, which recreates from F1 (raw data with errors) and F3 (detected errors linked to birth places), a file (clean data) named F1+. A business process may be

such that dedicated officers at daily hours have to get in touch with people having unusual birthplaces (F3 file).

In modern computing, it is more natural to mix the data capture and the data checking in a single interactive application. Design principles are different so that data availability, access and processing are (secondary) separated concerns. In other words, thinking about the application and architectural issues must not be parasitized by heavyweight data constraints: formats, organizations (sorted or not, replicated or not…), locations and so on. Skeptical people may believe that data problems in COBOL are similar to those in competing technologies. Of course, modern applications have data problems that have to be solved at design time. Universal concepts like Data Access Objects (DAOs) promoted by persistence frameworks like Microsoft DAOs, Hibernate Plain Old Java Objects (POJOs) or Java Persistence API (JPA) Entity Beans allow the design of applications without any coupling with data supports.

In an aging COBOL approach, we observe that "vertical computing" defers many calculations (data checking in the example) to periods in which the computing power is underused, during the night in particular. In the example, the execution of P1 during opening hours prevents the execution of P2, Pi… at the same time to offer the necessary computing power to OLTP programs, e.g. P1.

In terms of business criticality, this fabricates applications as highly sequential, and thus fragile, chains of programs (Figure 3.1). Any grain of sand in the gears, during the night especially when batch programs operate, may be a nightmare for the business. It is tempting to solidify these chains by creating rescue files and/or programs, e.g. sorting F1 (the file of raw data on voting people) with multiple criteria. In such a logic, the sorted file named F1++ (sorted raw data) may on demand replace F1+ (data expurgated of erroneous records) when batch programs do not give the expected results (F1+) early in the morning.

This logic is endless. This logic is irreversible software degeneration.

3.5. COBOL pitfalls

This accumulation of software matter in general gives rise to sizeable information systems whose internal/external layout becomes intelligible with strong difficulty (see again Figure 2.2 (principle) and Figure 3.1 (sample)). As a comparison, in a modern application, data capture and checking are certainly concomitant. They can deliver a set of proper, but incomplete, data if an execution suspension occurs. In other words, differently from COBOL, there are few cycle constraints considering the chains of programs from days to nights and from nights to days.

COBOL programming has a direct side effect. Many COBOL programs have no immediate business impact and value. They are just data pre- or post-processing (sorting, consolidating/merging, splitting… data) to (re)-reformat data so that "nobler" programs may operate with good performance conditions. These are often OLTP programs attached to screens and users while batch programs run for the night in critical job chains. There are consequently a lot of intermediate sizeable files/databases to manage data (contextual) views as inputs and outputs of programs.

So, batch programs themselves call for new batch programs to have upstream and downstream well-prepared data. As an illustration, a first batch program may be in charge of data aggregation for a second one while there is also the need for a third one in charge of immediate disaggregation. In this context, there is a proliferation of anti-business programs, i.e. technical issues are addressed through other invented technical issues; technique serves technique.

Other COBOL shortcomings are the fact that data duplication (or replication) is the rule, not the exception. There also exists a dispersion of the business logic, worst, a total dilution. Typically, data structures proliferate so that they have no possible interpretation in terms of business. Initially, we may have a single "patient" data structure in a healthcare software application. Over the years, there may have been 10 or more close "patient" versions. Frequently, such versions may lead to us having, for instance, a "length" field to state if the "patient" record carries little or much information. This "length" field makes no sense from the business logic viewpoint. Furthermore,

there is no protection of the data format and no real control of format alteration. Encapsulation in programming has for a long time been a paradigm for such a protection. Even if object-oriented COBOL in essence supports the encapsulation principle, only a totally negligible part of COBOL programs applies it.

More generally, even though "modern COBOL" offers computing environments that are able to tackle software engineering issues of the 21st Century with some probable efficiency, COBOL is first and foremost a state of mind. Distribution (resource sharing especially through pooling), parallelization through message programming, abstraction through data format detachment and so on, are never COBOL reflexes.

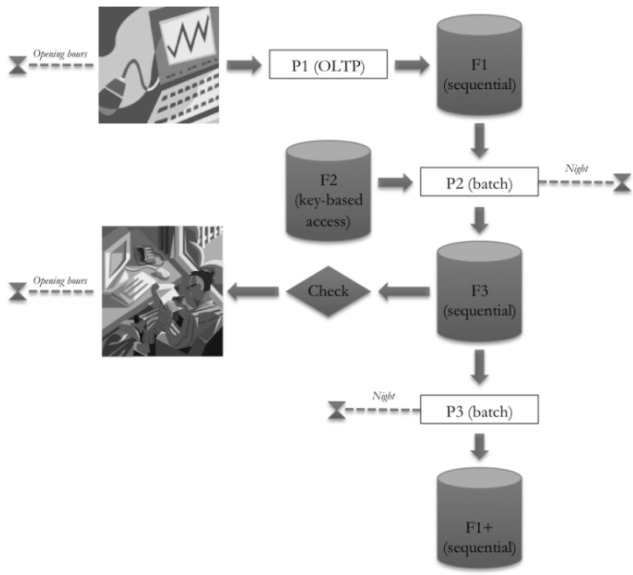


Figure 3.1. *Voting system as simplified COBOL program chain*

3.6. Middleware for COBOL

For the sake of survival, the fragility of COBOL applications is alleviated by middleware whose key role is the management of program chains. As an illustration, a middleware like CICS is in

charge of program chaining and coordination especially in case of fault recovery. Again, this augments the adherence with computing infrastructures since CICS is itself dedicated to mainframe computing. Despite many recognized qualities, such a middleware confines applications in a vicious circle.

Let us come back to the notion of software modernization. Finally, is there a way to move out this circle? This question is a particularly hot topic for batch programs, whose prior characterization seems to show that they have to vanish at the time of modernization.

In fact, COBOL software modernization can be essentially viewed as a problem of computing platform/machine. Why not virtualize mainframes? Why not have middleware platforms like CICS in a Java fashion style? The first question is discussed in section 5.5.1. The second is topical through the recent definition of a support for batch programming in Java, which is named JSR 352.

To that extent, in [VIG 13] batch programs are characterized as follows:

> "Batch processing is a pervasive workload pattern, expressed by a distinct application organization and execution model. It is found across virtually every industry, applied to such tasks as statement generation, bank postings, risk evaluation, credit score calculation, inventory management, portfolio optimization, and on and on. Nearly any bulk processing task from any business sector is a candidate for batch processing.
>
> Batch processing is typified by bulk-oriented, non-interactive, background execution. Frequently long running, it may be data or computationally intensive, execute sequentially or in parallel, and may be initiated through various invocation models, including *ad hoc*, scheduled, and on-demand.
>
> Batch applications have common requirements, including logging, checkpointing, and parallelization. Batch workloads have common requirements, especially operational control, which allow for initiation of, and

> interaction with, batch instances; such interactions include stop and restart."

This text extract is an exact reproduction of the COBOL batch philosophy and associated distinctiveness. It is an incentive to write batch programs in Java, inside or outside a place of preoccupation about COBOL-to-Java modernization.

Outside: why write Java batch programs instead Java EE applications? Scanning blogs on the Internet, this question comes up with no response. The positive contribution of JSR 352 is the support for a true transaction manager ("checkpointing" above) and the Job Specification Language (JSL) that rightly enriches this novel technology with the model-based spirit. Another evident advantage of JSR 352 is its seamless integration with Java EE at large.

Inside: this means that JSR 352 only exists to port COBOL batch programs. From platform/machine vendors, this strategic offer makes sense to develop business on associated services like third-party development and maintenance of Java batch programs. For COBOL professionals, this also makes sense if we do not want to revise architectures and to extract business logic. JSR 352 for COBOL-to-Java modernization can be qualified as "lightweight" compared to the white-box modernization promoted in this book.

3.7. Moving COBOL OLTP/batch programs to Java

People normatively consider that COBOL is the language of business and Java is the language of the Internet. We may straightforwardly conclude that Java is, before the release of JSR 352, inappropriate for "batch computing". Batch computing remains of course an uncommon notion in the Internet world. There is also a more subtle difference: the existing software architecture in which COBOL programs operate is markedly different from that of Java programs. Here, we exclude recent COBOL programs, which are well-structured, even object-oriented, modular (component/service-based) and connectable to the Internet. Instead, we look again at the quasi-infinite set of legacy programs and applications with a focus on batch programs: a greater part of the COBOL legacy world.

As underlined in section 2.2 of Chapter 2, COBOL-to-Java translation makes no sense at the code-to-code level (migration). Here, Java is just an adequate representative standard to demonstrate what has to be eliminated from the COBOL matter. As a comparison, JSR 352 is not code-to-code; this is because it is stressing platform issues. However, the profound assumption of JSR 352 prevents modernization from more open approaches like, for instance, COBOL/CICS-to-C#/.NET; something possible with this book's method and tool (see Chapter 8).

What is hidden behind JSR 352 is the fact that COBOL batch programs work together with OLTP programs. In a necessarily holistic approach, a move to JSR 352 imposes a coordinated move of OLTP programs whose (retrofittable) execution target is not JSR 352 but the common Java EE. How can we then operate this concomitant translation with two distinctive modernization logics?

The lack of maturity of JSR 352 cannot allow us to draw too many rapid conclusions. Ignoring JSR 352, moving COBOL batch programs in the same way OLTP programs are transformed into Java EE components or applications is absolutely unrealistic. A proof: for example, let us consider a batch program which processes an input file of 50 million data records in a sequential way. Data is extracted by means of different data masks (COBOL REDEFINES clause): the same data in the file is assigned to multiple variables, several times, to populate diverse instances from different data structures in the COBOL code. A memory cache may contain a big data block until the next reading of the next contiguous block (sequential access). On mainframes, such processing is common and above all rapid. The batch program records the computed data within an output file.

Applying this method from tables and relationships in a database is irrelevant. If table structures match data masks, an exorbitant number of DAOs and associated Structured Query Language (SQL) requests are necessary. In the worst case, one data mask in COBOL leads to one table, which itself leads to one DAO in Java. Relationships between tables that correspond to dependencies between

DAOs, are another probable source of slowdown. Despite caching or pooling mechanisms that are natural in Java, processing “one record” might lead to a couple of SQL SELECT statement executions. Under the hypothesis of 1 ms, such a processing lasts more than 13 h while the same on a mainframe surely takes less than one hour. Considering relational and object-oriented databases, data extraction does not depend at all upon rigid data organization in relational tables. This organization benefits from being simplified as much as possible. Simulations of the COBOL REDEFINES clause, only when useful, may rely on other appropriate SQL constructs, namely jointures, views (CREATE VIEW statement), etc.

3.8. COBOL is not a friend of Java, and vice versa

So, there is no direct relevant mapping between COBOL processing style and Java style as Internet computing reference. Beyond batch programs, OLTP programs cannot be directly translated from COBOL to Java. This is also not just a problem of software architecture since, as already written, some COBOL software matter has to be eliminated; this is especially true for architectures as backbones of COBOL programs.

Technology renewal and progress have led to the possibility of distribution of program pieces leading to objects, components, services, etc. This is the opposite of monolithic organizations of COBOL programs, which cannot be dismantled. Worse, COBOL programs are involved in rigid processing chains as “jobs”. These chains embody architectures. COBOL programs are irremovable elements in these (almost frozen) architectures. In fact, the key difference is that COBOL programs are constructed with strong adherence to running environments, including hardware (mainframes, etc.) and software (CICS, etc.), while Java offers greater flexibility.

Beyond tangible IT advances, there is a conceptual gap between COBOL and Java. Namely, object-orientation promotes abstraction and more precisely encapsulation. Java code and reuse are such that data format alterations (encapsulated in types) do not have to generate

significant maintenance. Instead, COBOL programs are primarily designed to satisfy data format constraints. In other words, the way data is organized in flat files (sequential access or key-based access, being indexed or not) or databases (having a hierarchical, network or relational underlying model), strongly influences COBOL program shapes.

3.9. Spaghetti code

We draw the evident conclusion that most of the COBOL matter greatly benefits from being re-engineered. Nonetheless, Figure 3.2 shows from personal statistics that 15% of COBOL (right-hand side) is maintenance-prone and Internet-compatible while another 15% (left-hand side) is dead for modernization. This latter package is spaghetti code and cannot be re-engineered at all.

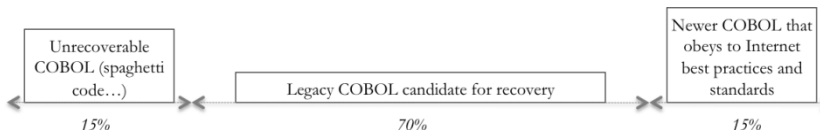


Figure 3.2. *Candidate and non-candidate COBOL applications for reverse engineering*

As an illustration of spaghetti code, many COBOL programs are concerned by reporting, i.e. the generation of leaf files that aim at only being sent to printers. In organizations' business processes, these paper-based information sets play a great role. Their absence may greatly disturb organizations' functioning and/or interaction with customers, suppliers, etc. Reports can be contracts, dashboards, legal documents or whatever. Their opportunistic replacement by other media, even technically possible, is first and foremost a matter or business process recast and consequent software adaptation, something often unrealistic in terms of effort, energy, risk, short-term strategy and consequently cost, time, etc.

If replacement decision is taken, any reporting application may often benefit from being replaced by Commercial Off-The-Shelf

(COTS) software provided that the upstream data is structured for that purpose. At this technical level, this may only correspond to data modernization, for instance moving from flat files with control characters (return, space, tabulation, etc.) to XML. Anyway, data modernization is a complex task because structuring hides data extraction, (re)-consolidation and more.

So, even though modernization solutions exist from a technical viewpoint, one interesting feature to be measured is the intrinsic complexity of COBOL batch programs making up reporting applications. Moreover, what is for each program its incorporated value in terms of business assets? We mean dealing with control characters (return, space, tabulation, etc.), cursors, odd data formats and so on is probably far from business function and rule management.

3.9.1. *Spaghetti code sample*

In this section, we comment on a precise case in which there is no business logic at all in the COBOL program because the control flow is only governed by cursors and characters (their types: control or meaningful data) in, at the same time, the input and output files processed by the said program.

In Figure 3.3, we depict the execution flow of a COBOL code portion named BEHANDLE (825 lines of code) coming out of 20.585 total lines of code (including both DATA DIVISION and PROCEDURE DIVISION) of a reporting program. This program alone does not constitute the full reporting application. Indeed, as usual, many pre- and post-processing programs apply transformation on data to make the reporting application's program chain more "fluid" (see "vertical computing" notion above).

BEHANDLE is massively relying on GOTO, both for forward (see label on the top right hand side of Figure 3.3) and backward jumps. The third kind of arrow/flow (see again label on the top right hand side of Figure 3.3) embody the fact that no flow diversion occurs, i.e. there is no systematic GOTO branching (a kind of deactivation) just before a labeled statement.

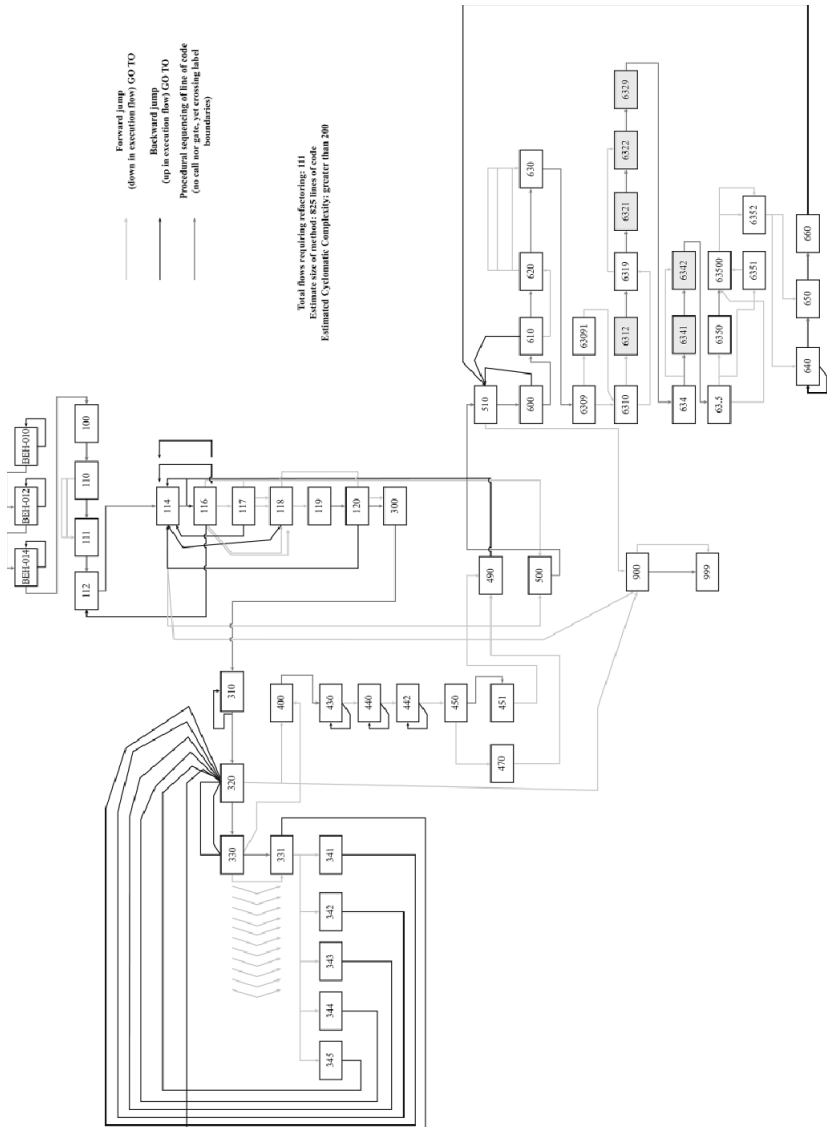


Figure 3.3. *Cyclomatic complexity analysis of a COBOL program portion*

Put simply, COBOL labels normally play the role of start sections of code blocks. However, these blocks are not really well delimited. In effect, the program flow is such that it sometimes enters into a block, not from a jump to the block's start label, say BEH-012 or BEH-014 (Figure 3.3, top of figure), but from the immediate prior statement. In fact, blocks have no marked end as strict "return" statements in common programming languages. In Figure 3.3, we may be in the BEH-012 execution flow and we may go on with BEH-014 (no flow diversion, i.e. the flow goes on while there is a label marking a different block). In terms of algorithmic logic, there is no knowledge and trace about the way we entered into BEH-12 (through a jump or not).

This inevitably results in spaghetti code whose refactoring is highly tricky. From the well-known GOTO refactoring algorithm proposed in [ERO 94], from 825 lines of code, 111 execution flows are discovered as candidates to be refactored. In [ERO 94], it is in particular explained that refactoring, i.e. GOTO replacement with IF or WHILE programming constructs forces the merging of blocks of code into one "procedure".

Applying this algorithm on the BEHANDLE case shows no convergence: the algorithm goes on looping. With manual intervention, the refactored BEHANDLE section comes up with only one service. Its volume is estimated around 825 lines of code and its cyclomatic complexity is 226 (see en.wikipedia.org/wiki/Cyclomatic_complexity).

If the refactoring algorithm was successful, the resulting restructured code would not be maintainable in a modernized version. It is agreed in the software industry to preserve service complexity under 10. Complexity 226 is simply not acceptable (density of conditional statements would be 1 conditional statement every 3 to 4 lines of code in the refactored code with individual services each close to 1,000 lines of code).

This definitely illustrates that 15% of the worldwide COBOL code is not a candidate at all for any modernization.

3.9.2. *Code comprehension*

Although technology-to-technology modernization of this specific COBOL case is proven inappropriate, Figure 3.4 demonstrates interests and advantages linked to a model-driven approach to ease code comprehension. This is a UML dynamical model (UML Activity Diagrams) that graphically expresses the control flow of the program portion.

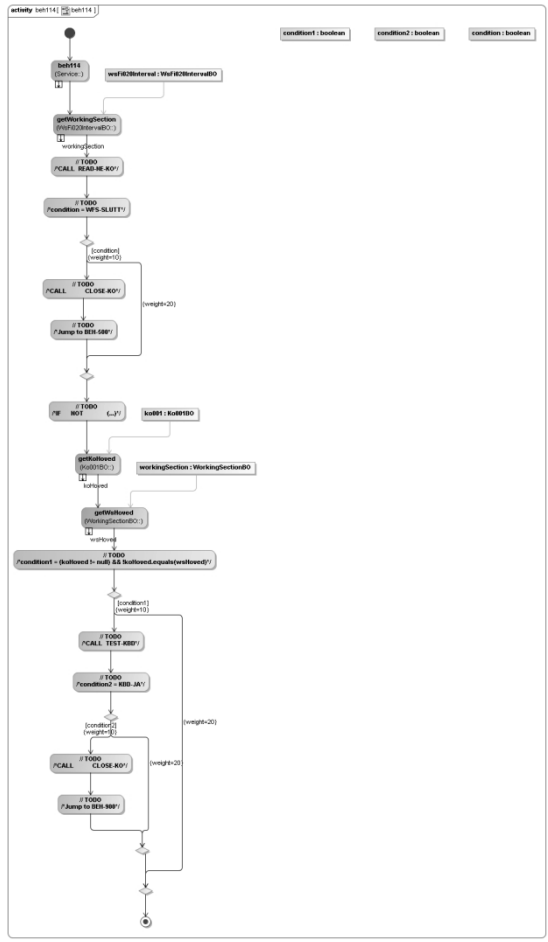


Figure 3.4. *UML representation of a COBOL program portion (subset of what appears in Figure 3.3)*

White-box modernization is then also a solution for concentrating in models the knowledge extracted from the source code. This may help in having a synthesized overview of key parts of the reporting application in order to estimate, for instance, the challenges and inherent investments about any redevelopment from scratch versus buying COTS software.

3.10. No longer COBOL?

The software market is such that nobody is able to anticipate the end of COBOL. We think that COBOL, in its newer offers, will tend be increasingly confined to market niches. The history and life of programming languages are a matter of humor. Retrospectively, the uptake of languages has nothing to do with rationality and is more due to fashion, unexplained attractiveness or even fascination.

For example, why use Java, C# or Python while C++, Objective C, Eiffel or Smalltalk sufficed. Languages convey brand images (Apple – Objective C, Microsoft – Basic/C#...). As for COBOL, it has always conveyed enterprise computing.

This book's criticisms of COBOL are only concerned with legacy COBOL code, which by definition has not been built with recent COBOL environments and tools supporting modern computing. COBOL in itself is not guilty. To that extent, on-the-ground experience shows us an indisputable know-how on business requirements' engineering in COBOL organizations. While technical practices have bastardized COBOL, COBOL through its "B" member letter has revealed, built and consolidated a unique philosophy of enterprise computing.

Despite the increasing segmentation of computing devices in parallel with the same increasing concentration of Internet data/application servers boosted by cloud computing, country competitiveness relies on a differential know-how on enterprise computing. In our opinion, the control of operating systems, middleware platforms, including cloud infrastructures and platforms, is an obvious source of country competitiveness, through today's

leaders especially. Nonetheless, the forthcoming revenues potentially brought out by enterprise applications on the top of the Cloud are also huge. The COBOL business spirit is thus still influencing the economical area of enterprise software systems. So, developing enterprise applications with COBOL, clones or more serious challengers like Java, requires a great amount innovation in present and future.

3.11. Conclusions

In this chapter, we observe a kind of fairly dramatic situation for most COBOL software. The case of COBOL-to-Java is an illustration of the move from single-machine (mainframe) computing to the Internet, an ever-growing number of interconnected and collaborative computing resources. Going into further technical details, the dichotomy between batch and OLTP programs has a great role and significance in the COBOL maintenance problems. This two-angle analysis is fundamental for modernization. Despite technical solutions like the JSR 352 in the Java world or probable equivalent solutions for .NET, the Cloud…, the overall challenge remains architectural issues: architectures must be totally recast.

Another point is recovering the business logic, a phenomenon that can only occur through deep code investigation. The fact that 15% of the COBOL code is definitely lost (spaghetti code) and that 15% is compliant with contemporary standards and practices, leads us to consider a huge amount of 70% of current applications as candidates for modernization. As it happens, billions and billions of lines of code may fall behind with regard to the acceleration of the Internet, as unified computing platform and infrastructure.

4

Service-Oriented Architecture (SOA)

Software architecture is a scientific and technical discipline in which stakeholders try to explain, reason about and, above all, formalize the ways of software elements' dependence on each other to create applications and information systems. The varied nature and the (functional and non-functional) properties of software elements, as well as their relationships, make software architectures complex.

Software architecture issues have rapidly evolved because of distributed systems (including Internet computing) in which software elements are not located in one single place. Service-oriented architecture (SOA) is borne out of this observation and has been enshrined in cloud computing.

4.1. Software architecture *versus* information system urbanization

To simplify, applications have internal organizations, which are considered as low-level structures. Information systems have applications as non-exclusive constituents. The organization of these constituents is a kind of high-level structure. Information systems feed business processes with information. The articulation between information systems and business processes is a macroscopic view or "urbanization". Architecture issues in information technology (IT)

thus encompass the search for the best structuring of applications and information systems (see Figure 1.2).

Conventionally and for clarity, we reserve the expression “software architecture” to applications, while “urbanization” is the preferred term at the information system level.

In terms of whole–part relationships, components (viewed as services at runtime) are constituents of applications while applications are constituents of information systems. However, relationships are diverse in nature; they may have various semantics. Applications may be linked by spatial dependencies (e.g. obligation of execution on the same machine), temporal dependencies (e.g. obligation of execution in sequence), functional, QoS-based (QoS standing for “Quality of Service”) or any other kind. In this scenario, core challenges in IT-centric architecture are the production of cartographic views and viewpoints on information systems/ applications and the ability to control them: behavior prediction, anticipated evolution, dynamic reconfiguration, etc.

4.2. Software architecture evolution

In organizations, growing volumes of data and programs multiply the number and the nature of links between parts of information systems and applications, not to mention as their physical backbone: personal computers, servers, cables, network equipment, etc. In short, the evolution of computing environments has a strong, direct impact on the evolution of urbanization plans and software, architectures. Any control on these factors is thus rapidly hampered by additions/modifications of elements compared to the initial design.

As a comparison, a legacy information system is like the traffic infrastructure of a city. For example, we may consider the narrowness of streets as a cultural heritage and an esthetic asset for tourism. We may be embarrassed by this narrowness for tramway circulation. Maintenance of these streets may be a dilemma when we want to keep their beauty and concomitantly search for their convenience for

tramways (power feeding equipment, lane integrity requirement, etc.). Cities have not been built in a modular way so that their elements, at any instant, interact differently to readily revive new urbanizations. In this environment, tramway circulation is an application with a frozen architecture. For instance, tramways' width cannot shrink from deformable elements to meet narrowness requirements. Over the years, the only way to create some convergence between contradictory requirements is to make urbanizations/applications more sophisticated, for example, a facility is installed to hide some aerial power feeding equipment, and consequently make streets persistently esthetically pleasing.

Nonetheless, the comparison to IT has to stop here because links in software are not material, but electronic. So, both at the information system and application levels, we expect software constitutive elements with, in number and nature, supple links. As a result, in the age of Internet ultra-connectivity, software architectures can no longer be thought of as blocks and sub-blocks only designed for a single long-term invariable purpose. Surprisingly, links in software, though immaterial, are not flexible enough to revise architectures in an easy and straightforward manner. This is effectively observable for contemporary software and disastrous for legacy software.

4.3. COBOL own style of software architecture

Looking at common business-oriented language (COBOL), the problem is that it does not have at its core the architectural paradigm on its own: COBOL systems are monolithic, i.e. versatile blocks do not exist in space and time. Still worse, such blocks are not removable. Let us come back to the city metaphor. This corresponds to the impossibility of suppressing the facility that hides some aerial power feeding equipment at the time when some land-based power feeding equipment is being substituted for. Although this scenario seems implausible in today's city urbanization management, it is the daily life with COBOL.

In practice, in an organization having COBOL as core language, software elements are generally stand-alone programs (in terms of

autonomous execution) with possible (shared) subprograms. Program execution is planned and piloted at the machine level (mainframe) within dedicated runtime middleware (e.g. customer information control system (CICS)). Time after time, files, databases, reports, programs, their commonalities, disparities, (functional or temporal, i.e. at design time or at runtime) dependencies, versions and configurations, etc., create an imbroglio of software matter (see Figure 2.2). Legacy COBOL information systems and applications generate computing environments similar to medieval cities.

Proliferation of software matter in general calls for some rationalization, which corresponds at least to the ability to produce cartographic intelligible representations of applications and information systems. The possibility, even the great difficulty, of drawing these representations gives a degree of chaos. In COBOL, only such primary representations may leverage software architecture comprehension: the unique path to modernization.

As a justification, Figures 3.3 and 3.4 show this need through the internal representation of programs. At an outer level, Figure 3.1 shows the same need at the architecture level, but the view is only "organic". In effect, participating elements such as files, programs, flows and users (even machines) are depicted. In COBOL, software architectures can rarely be drawn in a logical way, i.e. independently of the physical (underlying) installation. Changes in architectures are thus direct changes in physical elements and their relationships, something almost impossible to put into practice in short cycles.

Contrary to COBOL, Figure 4.1 shows the advantage of drawing attention to architectures in a logical way and thus exhibiting "logical" components. This is in particular the (partial) functional description in Unified Modeling Language (UML) (Component Diagrams) of what could be a general-purpose *Currency* component, which provides currency conversion functionalities. It is made up of several business objects such as content (Money, etc.) and interfaces (MoneyFormatter, etc.) such as external visibility.

As discussed in section 4.4, architectures are developed from expressing how components interact independently of their

assignment to physical elements. For the latter issue, UML typically proposes Deployment Diagrams, which allow representations somewhat "close" to that in Figure 3.1 for COBOL. Before having deployment views, pure architectural views are necessary provided that software is really componentized. Again, the CICS call to a COBOL *Currency* program in section 2.2.3 is not the rule, but the exception; it is the perfect counter-example of what the situation of most COBOL software at this time actually is. Legacy COBOL software is not componentized at all. Therefore, models such as the one shown in Figure 4.1 are science fiction in COBOL.

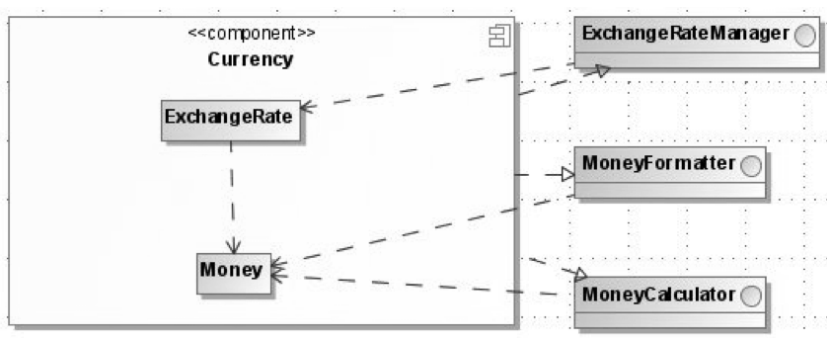


Figure 4.1. *Currency business component (partial specification)*

So, in modern software development, engineers require modeling languages like UML or more focused languages (e.g. architecture description language (ADL)) to express views and, possibly, points of views when focusing on specific urbanization/architecture problems.

From an intuitive graphical formalism, Figure 4.1, at a small scale, shows in UML how we may theoretically control urbanizations/architectures by accurately describing how logical software assets in applications relate to each other in spatial, temporal, functional, etc., relationships. In the COBOL world, urbanizations/architectures quickly decline as illustrated in the example in Figure 3.1: files, programs, flows, etc., multiply and

spread. Atomic functional ensembles are totally diluted and lost in code that primarily satisfies technology constraints; the business requirements are shredded.

In COBOL, the degeneration of software architectures comes from two main factors:

1) The partitioned development of applications in which requirements are pushed into applications in a vertical manner. There is no sharing based on an organization's domains and subdomains. As an example, the need for currency conversion facilities in several applications would surely result in numerous requirements' interpretations from the requirements' engineering phase to the implementation phase.

2) As noted before, nobody is really able to think about software architectures without a strong adherence to technology. Over the years, from COBOL to Java, architecture styles have changed according to paradigm shifts: functions (procedures, routines, etc.), packages, objects/classes, components, services, etc. For a long time, modularity reigns. Beyond this, advances in the field of software architecture have put forward "abstraction" embodied by modeling and model-driven development (MDD).

This book does not aim at "pointing the finger" on COBOL as a unique culprit. Both points 1 and 2 remain complex issues when addressed with the help of novel paradigms/technologies. Despite the maturity and widespread availability of these paradigms/technologies in the era of Web applications, namely SOA and cloud computing, software architecture remains a hot topic.

4.4. The one-way road to SOA

Returning to the city metaphor, the Internet acts as a global village of software elements, its inhabitants. Global reasoning (opposed to "local reasoning") is due to modern software development. Designing software and thus software architectures in a systemic way contrasts

with COBOL divide-to-conquer style[1]. As much as possible, software architectures must be viewed as assemblies of pre-existing and/or commercial off-the-shelf (COTS) elements instead of being invented from ground zero to alleviate piling problems.

Software architectures make no real sense if we ignore that they participate and contribute to the digital world. What is donated by the Internet, that is, an ever-seen communication and computing infrastructure, must lead, in turn, to an equivalent "return of favor". Beyond this, as discussed in Chapter 1, the Internet is a source of innovation and related profit in business through not previously imagined trading services.

Including the word "service", SOA is first and foremost a state of mind. SOA is, in particular, twofold. Its business facet is an ever-encountered opportunity to design information systems and applications in a manner that boosts the business instead of simply viewing software as the banal automation of information processing. This point is discussed in detail in Chapter 1. SOA's technical facet is what we can precisely understand behind the word "service", especially in terms of technological impacts and implications in daily software development. In fact, the SOA paradigm has become popular with the emergence and large take up of Web Services, but this standard is not the single support for SOA.

In people's minds, "service" is something billable. It is no coincidence that "service" comes from the telecommunications field. For a very long time, telecom operators have sold services to their clients. The extraordinary convergence of computing and communication triggers by the Internet has made "service" as the natural core concept of Internet computing. Mobile computing, cloud computing, etc., are nothing else than service-oriented computing paradigms and technologies.

1 René Descartes' famous "Discours de la méthode", which gave rise to the Cartesian approach for problem resolution, opposed to systemic (or system-based) approaches.

4.5. Characterization of SOA

4.5.1. *Preliminary note*

In this book, for clarity, we use the expression “software part” or “software element” in a free way, i.e. with open semantics. However, both the words “component” and “service” have an agreed significance. In general, components/services have composability features while parts/elements do not. This clarification helps us, from a software engineering viewpoint, to establish that components/services, implicitly and systematically, induce maintainability, reusability, even reliability, in software development and execution.

4.5.2. *From objects to components and services*

SOA is an architecture style based on the extreme componentization of software. Using the words “componentization”, “component”, “composition” and “composability” (the potential to be composed) is not just anecdotal. The very nature of software components is as follows: “components *are for* composition” (our emphasis) [SZY 02]. In other words, software components are software parts while all software parts are not components, especially when these parts cannot be composed. There are two famous definitions of the notion of “component. The first definition is: “a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties” [SZY 02].

The second definition is: “a software component is an implementation, in software, of some functionality. It is reused as-is in different applications, and accessed via an application-programming interface. It may, but need not be, sold as a commercial product. A software component is generally implemented by and for a particular component technology” [SEI 00].

There is no broad difference between the notions of “component” and “service”: “services are different from components in that they

require a service provider. A service is an instance-level concept – where such instances can be component instances" [SZY 02].

From the *Java EE 6 Tutorial*, the idea of "service" is evoked as follows: "on the conceptual level, a service is a software component provided through a network-accessible endpoint".

Components and services philosophically derive from objects in object-oriented programming. In this logic, the three notions have common features. In effect, the technical idea of easier composition fits the economical idea of reuse reinforced by object-oriented programming in the 1980s. As objects, components and services are reusable software assets (Figure 4.2).

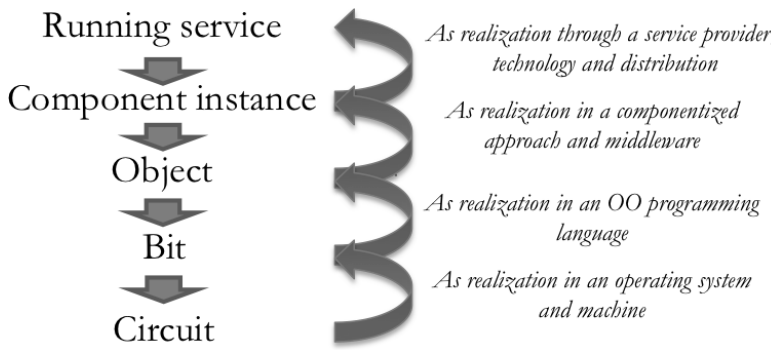


Figure 4.2. *Objects versus components versus services*

4.5.3. *Type versus instance*

Both objects and components encompass design time and runtime notions. At design time, objects are "types" or "classes", while they are called "instances" at runtime. Component instances are runtime components. There is no specific expression/term for component types at design time. Conventionally, the term "component" most likely means "component type" instead of "component instance". In the type-instance dichotomy, services are runtime notions only, as pointed out by Szyperski's definition above.

4.5.4. *Distribution concerns*

While the notion of service strictly correlates to distribution concerns ("network-accessible endpoint" in *Java EE 6 Tutorial* definition above), those of object and component are debatable. However, the key difference between an object and a component is that the latter is a unit of deployment (Szyperski's definition above). More precisely, this means that components comprise not only code but also configuration data. Components are configurable. Deployment is the operation of installing components in their running platform. At deployment time, configuration data are transformed into values used by the platform to manage the components' functioning. For example, the Enterprise JavaBeans (EJB) technology supports security configuration data, transaction management configuration data, etc. This approach facilitates the externalization of non-functional aspects outside the code. Proceeding this way is an added value for components compared to "obsolete" object-orientation, i.e. programs that refer to resources (e.g. an Internet Protocol (IP) address) in a hard-coded manner.

In this context, service computing and SOA are the extreme vision of application distribution with corollary facets, such as resource mobility and resource virtualization. These notions are discussed in further detail in section 5.5.

4.5.5. *Functional grouping*

As objects, components and services gather some consistent functionality in one piece through encapsulation. All three paradigms must adhere to this founding principle, which is often, unfortunately, a forgotten best practice. We mean that it is possible to find software with ill-formed objects, components or services. For example, an EJB component that offers two business functions, one for computing a shipping plan and the other for computing a stock inventory, is malformed. The two functions refer to two business activities that are (in terms in computing semantics) far from each other.

We strongly believe that services accentuate this need for business proximity. As underlined in Chapter 1, the notion of "service" in the computing sense has an intimate link with that of "service" (opposed to "product") in the commercial sense. In this logic, a service in an SOA application must gather a set of ready-to-business functional assets.

Otherwise, all three paradigms rely on interfaces or access points, which are the "visible part of the iceberg". This means that objects, components and services consist of an implementation part (submerged part of the iceberg) and an interface part (surface part of the iceberg). For users, the implementation is hidden because it has adherences to the underlying computing environment. Accordingly, usages expressed through interfaces' calls do not refer to the implementation part. This low coupling is the key for successful evolution in reasonable cost and time. Typically, Web Services are written in various programming languages (PHP, Java, C#, etc.) while calling them in programs does not impose any technical knowledge on any particular language.

In short, this development philosophy is a radically opposite framework compared to that of COBOL. More subtly, we may consider that modern technologies have been eliminating the handicaps of COBOL or any aged deficient technology.

4.5.6. *Granularity*

Although in object-oriented programming people may refer to "objects" and "components", components are considered as larger pieces of software compared to objects. In extreme cases, sizeable modules (billing, shipping, stock inventory, etc.) in enterprise resource planning (ERP) software packages, for instance, may also be called "components" provided that they demonstrate composition aptitudes.

Due to this difference in size, components and services exhibit composition operators, which operate at the architecture level while objects play at the programming level only.

Historically, components and services are more recent notions than objects: 1990s for components and 2000s for services. All three paradigms relate to each other as follows: objects implement components, which, in turn, implement services. We may illustrate this chain via a Java class that powers an EJB component which itself realizes at runtime a Web Service (also see section 5.1). In this special case, an EJB is probably made up of a Java class, several Java interfaces, Java annotations and/or XML files to define its configuration (deployment) properties.

4.5.7. *Technology-centrism*

Objects, components and services are often technology-centric. In practice, composition capabilities are not innate characteristics. When enclosed in a technology, say Java, objects may only be composed by means of composition operators offered by the technology. This principle applies to EJBs as component archetypes or to Web Services as having the role of the most well-known service illustrators.

All the three examples of technological frameworks have imposed formats, which drastically facilitate composition. As an illustration, asynchronous collaboration in EJB amounts to using *Message Driven Beans* as proxies between "functional" components requiring asynchronous exchanges. In EJB, this way by which asynchronous composition issues are addressed is also known as a composition pattern of this technological framework.

Outside a technology, composition and thus reuse are methodical preoccupations of software teams with the risk of more random results. We mean that one may imagine various "theoretical" composition operators. However, the way these operators may be implemented can be very open, possibly leading to heterogeneous solutions in terms of effective realization in applications.

For example, Figure 4.3 shows in UML some composition whose implementation is potentially subject to many variations, depending on the target technology for implementation. The model in Figure 4.3 is technology-independent, but it leaves much latitude in the method,

which establishes how the UML-provided (bubble) and required (semi-circle) interfaces may be embodied in a given programming language and component/service technology.

So, typically, teams use ADL, UML or any other modeling language that supports conceptual composition constructs. The chained implementation may strongly benefit from being codified through, for instance, the definition of a consensual code pattern shared by all team's developers to avoid implementation heterogeneity.

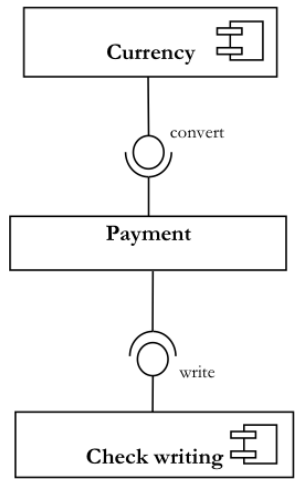


Figure 4.3. *Check writing component is assembled with Currency component via Payment*

UML offers at a conceptual level other composition operators such as containment and delegation. Again, this wealth is in contrast to the need for interpretation of these operators at implementation time. Technology-specific models may also be derived from UML: they are then lowly abstract to refer to the technology of interest (Web Services in Figure 4.4). In this case, the implementation method, which becomes systematic and more efficient, paradoxically lets fewer flexibility and creativity with regard to the sought SOA style. Beyond this, technology-centrism may be a parasitic way of thinking. Indeed, all objects are not in Java, all components are not EJBs and all

services are not Web Services. Viewing SOA as a technology-free architectural style is thus essential.

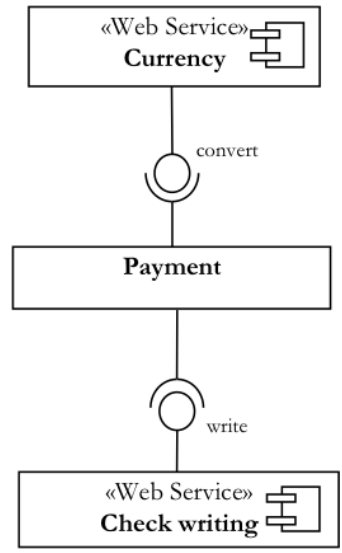


Figure 4.4. *Technology-independent model in Figure 4.3 is transformed into a technology-specific model (Web Services)*

4.5.8. *Composition at design time (… is definitely modeling)*

So, as discussed, composition is *the* founding principle of component-based development and service computing. In this context, we have to make a distinction between composition at design time and composition at runtime. The latter is also known as dynamic composition, which coexists with the idea of, for instance, "service discovery", "service mediation" or "semantic interoperability" (see section 4.5.9).

Roughly speaking, in a broad sense, application designs arise from the identification and definition of service exchanges. So, at the design stage, it is important to formalize how functionalities interoperate to meet the applications' overall functional requirements.

Composition at the design stage is generally based on the description of applications' architectures with modeling languages that may vary in scope and purpose. For example, the *Payment* entity in Figure 4.3 is designed as an aggregator of the *Currency* and *Check writing* services. The latter is able to generate checks from varied bank accounts in foreign countries.

The UML component diagram in Figure 4.3 is both static, functional (convert and write provided interfaces are exhibited) and implementation-free (one does not know how these two interfaces might be implemented). The model in Figure 4.4 is a slight variation of the underlying technology: Web Services. It is technology-specific, but does not bring out much added value.

Composition details may be numerous depending on the desired accuracy. To make models more precise, we have in particular to move from static descriptions (Figures 4.3 and 4.4) to dynamical descriptions (Figures 4.5 and 4.6).

The Payment business process in Figure 4.5 integrates the *Currency* and *Check writing* services in a workflow logic. The business process model and notation (BPMN) formalism is used. BPMN is recognized as a language devoted to the modeling of organization functioning instead of stressing software's inner workings. The model in Figure 4.5 is, by definition, represented in a technology-neutral way. UML activity diagrams are based on a very similar formalism (many examples are given in Chapter 9) with, compared to BPMN, additional constructs and a more seamless link with other UML diagrams: *class diagrams*, *component diagrams*, *deployment diagrams,* etc.

In BPMN, control flows are ruled by events (e.g. bubbles as start and end events in Figure 4.5) and data flows for inputs and outputs of "works" (also known as activities or simply functions such as *convert* and *write*).

The degree of composition precision may call for more specific modeling languages with, inevitably, the risk to adhere to a technology and implementation concerns. For example, "service

orchestration" promoted by the *Web Services Business Process Execution Language* (WS-BPEL) is a modeling language devoted to the Web Services technology.

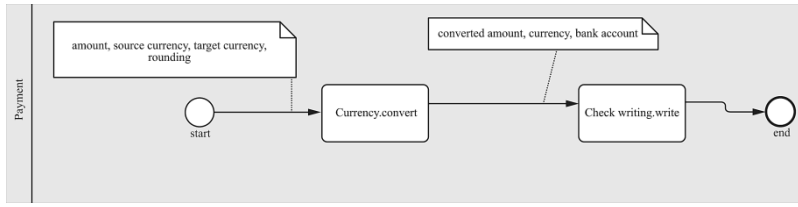


Figure 4.5. *Articulation between the Currency and Check writing activities in a BPMN Payment process*

In Figure 4.6, we show how any transnational financial application supporting a *Payment* process formerly requests a *Currency* service before requesting a *Check writing* service. The model in Figure 4.6 does not properly respect the WS-BPEL syntax, but it shows at a fine-grained specification level details such as service invocations (arrows pointing right and left), assignments (equals sign), data reception (arrow pointing right) and data reply (arrow pointing left). The *Payment* process specification in particular points out sequencing and parallelization. As an illustration, retrieving *bank account* for the *write* function of the *Check writing* Web service is parallel to the call of the *convert* function of the *Currency* Web service.

WS-BPEL capitalizes on the capabilities of the Web Services technology, which automatically lets us much latitude in implementation detail description. In this scenario, WS-BPEL models may be associated with effective predeployed Web services. WS-BPEL models may consequently be deployed on (and then executed by) a BPEL engine, for example *Apache Orchestration Director Engine*.

In Figures 4.3–4.6, the SOA spirit is embodied by the permanent availability and thus runtime reusability of the *Currency* and *Check writing* software components/(Web) services. Other business

processes may be instrumented by equivalent designs and thus models. Other components/services may be grafted onto this architecture. For example, we may envisage the addition of *Check printing* and *Check posting* services.

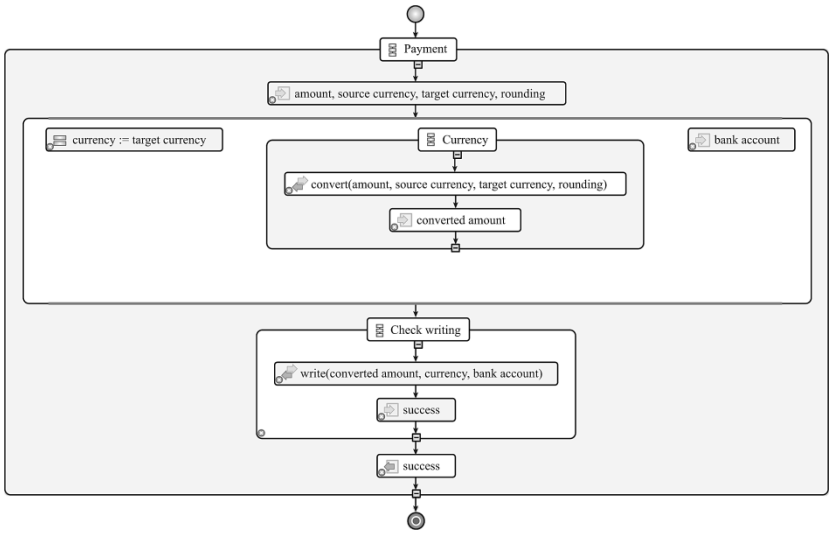


Figure 4.6. *WS-BPEL model enhancing (toward implementation) the UML model in Figure 4.4 and the BPMN model in Figure 4.5*

So, in SOA, services are therefore instances of components in execution with the following advantages:

– sharing: services are in essence runtime reusable components. Client applications or third-party components do not need their own copy of components in their image, i.e. components' code is not embedded in the package. So, in the world of software components, SOA refers to applications that do not necessarily "internally" own their functionalities. We may go toward an extreme vision of software reuse if services are shared over the Web, the case of Web Services;

– client applications or third-party components have the ability to be connected with services through normalized (transparent) access, exchange protocols (e.g. Simple Object Access Protocol (SOAP) over Hypertext Transfer Protocol (HTTP) for Web Services). Services are

"everywhere" provided that network connectivity exists. This confirms that cloud computing has the definitive culmination of SOA, but this also raises the evident problem of service disruption and subsequent fault recovery in SOA-centric business processes and applications;

– services expose features in terms of both functionality and quality (e.g. performance guarantee and availability continuity). This has created the idea of *Service-Level Agreement* (SLA) in which services are constrained by a certain degree of quality due to the fact that they are chargeable. We briefly outline in Chapter 5, with the help of the EJB technology, the way SLA may be managed from service provider viewpoint;

– service composition may have varied forms from high abstraction (UML, BPMN, etc.) to implementation details and technology strong adherence (WS-BPEL, etc.). Other mechanisms of expressing service composition at implementation level also exist. Typically, WS-BPEL favors a centralized architecture (also known as orchestration) for service composition while choreography (a decentralized architecture) is another composition paradigm particularly based on the *Web Service Choreography* (WS-Choreography) modeling language. SOA is thus manifold within or out of the circle of the Web Services technology. SOA is a family of architecture styles, which lets room for creativity and application functioning optimization, in the Cloud in particular.

The counterpart of these four recognized strengths is the risk of excessive dependence of applications on external (even uncontrollable) service provisioning and delivery. In an ideal world, services can be dynamically replaced by each other. In this context, functional contracts with (possibly new) callees must still be respected. Replacement can be motivated either by functional or QoS insufficiencies. Applying such architectural changes at the design stage amounts to changing models and next implementations to reflect model changes in terms of new service usages (application programming interface (API), locations, providers, utilization costs, etc. may change accordingly). Such a cycle is again maintenance with inevitable inertia, tuning, efforts and costs. Direct adaptation at

runtime without service disruption aims at avoiding this cycle; it is based on runtime composition.

4.5.9. *Composition at runtime*

SOA results in viewing applications as service assemblies. An induced vision is the idea of pluggable, connectable items like newer musicians (musician addition or musician substitution), who are able to participate in orchestras without, as much as possible, long and intense rehearsal.

Assembly at the design stage is comparable to the hiring, training, validation and effective integration of these newer musicians as true complements and sources of better music. Application design is the well-formed articulation and coordination of these complements in orchestras. Models in MDD play these roles: provided and required interfaces in UML, workflows in BPMN or executable processes in WS-BPEL are constructs for expressing these articulation and coordination.

Assembly at runtime is, for example, the fact that composition thought at the design stage does not lead to frozen architectures at runtime: new services (for instance, using service directories with the *Universal Description Discovery and Integration* (UDDI) standard for Web Services) can be discovered to replace deficient services. To that extent, service mediation is the use of a mediator service, which hides and manages the services in charge of delivering the required functions with the required QoS, in reasonable cost and time, etc.

Of course, with service-stable interfaces, dynamic composition can be viewed as the transparent replacement of interfaces' implementations: it is the "invisible" code running behind interfaces resulting from implementations' encapsulation. In practice, dynamic composition is more research concern of today than an industrial possibility and thus realistic strategy. To that extent, this book does not emphasize the idea of dynamic composition. In effect, COBOL heritage and modernization come up with existing monolithic stiff architectures. This rigidity often hides a useful stability when COBOL

information systems and applications face business disturbances. The gap from legacy COBOL to SOA is so important that findings may only emerge from maturity in SOA. In our opinion, dynamic composition is a conceptual step too far for COBOL practitioners at this time.

4.6. Conclusions

This chapter shows that SOA is a computing philosophy on its own. SOA is above all a conceptual architectural style, which takes a stance opposite to COBOL totally monolithic style of architecture. SOA has prominent concerns: reusability based on easy and straightforward composability, service provision and delivery in quality (security, etc.) and capacity (scalability, etc.).

Componentization as a founding principle of service computing is well relayed by models. In effect, models may greatly help the formalization of components and interactions. Given an SOA technology like Web Services, models are natively present at the heart of the technology with the WS-BPEL or WS-Choreography modeling languages. Since services may not be linked to a specific technology, models like UML (even BPMN) models may also greatly help the design of SOA samples.

To avoid COBOL historical pitfalls, SOA is the evident recipient of COBOL software modernization in a renewed business context for software in general.

5

SOA in Action

In several points of this book, we are interested in currency conversion facilities as part of applications of organizations that require financial computations at large.

We initially showed in section 2.2.3 an external call in COBOL to a *Currency* program. This middleware-based call (Customer Information Control System (CICS) middleware) is characterized by a sound modularity with the exception of the "precision" variable, which seems to be a global variable shared by the caller and the callee, a recognized bad style of programming. We also note that this modularity is rare in existing common business-oriented language (COBOL) applications. The very challenge of COBOL software modernization is not only dealing with this modularity, but it is also to tackle COBOL currency conversion facilities that are deeply immersed, excessively engraved and surely dispersed in many places of the total code of numerous legacy applications. The purpose of this chapter is to characterize the model-driven development (MDD) method that has mechanisms to extract such a knowledge as: the existing currency conversion functionalities, the business rules that govern their functioning, and the business processes fed by these.

More generally, the remaining chapters of this book are a toolkit for code mining, analysis, interpretation and recast (reverse engineering). It is the move from diluted currency conversion facilities

in COBOL to a compact unified modeling language (UML) model with equivalent riches and semantics in terms of computation. The move to SOA is the role of forward engineering.

In section 4.4 of Chapter 4, we draw the necessary conclusion of componentization. *Currency* must be a component on its own, meaning, by definition, an ability to be composed with other components. As an illustration, a *Check writing* component is introduced as a collaborator of *Currency* in a *Payment* business process. Compositions may be described in a loosely coupled way (first UML model in Figure 4.3). Going into further detail leads to make data and control flows explicit in service collaboration by means of business process model and notation (BPMN). Technologically independent models may be insufficient; we then want to link models to a given technology, Web services for example (second UML model, and Web services business process execution language (WS-BPEL) model, see Figures 4.4 and 4.6 respectively).

In the logic of modernization, to overcome COBOL sins, we indisputably converge to the idea of making *Currency* both specified (documentation) and operational (runtime image). To have something consistent and complete, all "lost + found" COBOL statements in many source code files aim at providing much knowledge on functionality completeness, utilization scope before disappearing to the benefit of a well-isolated component, for instance, a Web service. However, as shown below, there are many technological alternatives beyond Web services.

Modernization amounts to detecting and setting boundaries for these COBOL statements; further analysis and interpretation are required to establish the level of detail about all of the existing currency conversion facilities. In practice, a *Currency* component ensures currency conversions according to different configurable elements:

– precision (1 digit, 2 digits, more);

– rounding (ceiling, floor, etc.);

– conversion rate update frequency (low, high, etc.): for example, currency trading requires multiple updates of conversion rates for one short period.

Orthogonally, QoS features are also of great importance: availability, performance, security, fault recovery, etc., beyond the fact that SOA is fundamentally the search for a coherent organization of functionalities in self-contained, well-isolated modules. Once built, these modules may also be endowed with QoS features to implement service-level agreement (SLA) policies.

The fact that *Currency* must be incorporable into any application (of course, ours, but why not into applications belonging to people outside our organization?) calls for a kind of generic thinking on what a *Currency* component/service may look like. We show that standards exist later, in section 6.4: a formal characterization (again, a model) of *Currency* as a computerized object. These standards are suitable guides for homemade design; they may also play the role of requirements' documents when we want to buy, rent or freely use a commercial off-the-shelf (COTS) *Currency* component/service. In all cases, not reinventing the wheel is the rule.

The interesting point is that models are both implementation bootstraps and comprehension/communication means. Models as graphical or compact textual representations liberate us from programming code tainted of too many useless details, even if, ultimately, code is "the end of the tunnel".

5.1. Service as materialized component

Requiring a currency conversion component ensues from analysis of many legacy code pieces and possibility/opportunity of refactoring these in a single (composable and thus reusable) entity. A rough approach steers us to acquire some new code. This can be subcontracted fabrication or simple download (source code, binary code, etc.). The component can also be constructed in house with inevitable maintenance issues.

As shown later in Chapter 7, before any kind of implementation, models help us to get a better understanding of *Currency* as a unified set of functionalities (the idea of "functional grouping" in prior chapter). Models are also maintenance entry points when we decide to generate *Currency* code from a model.

In all of these cases, it exists a challenge in terms of maintenance and reuse. Choosing an external component is "design with reuse"; outsourcing imposes lightweight maintenance issues: open-source software contribution or maintenance management (versioning plans, tests, deliveries, etc.) with subcontractors. In-house fabrication is "design for reuse". In this second case, *Currency* must be formatted to fit the required services of *Payment* and, indirectly, *Check writing*. *Currency* must also be designed for meeting forthcoming requirements expressed by currently unknown client components. More generally, "design for reuse" raises the problem of unanticipated utilization. Concretely, from an economical viewpoint, should we invest on components whose (possible) postponed utilization is unknown? Design for reuse covers tactics like component higher genericity (and thus configurability, customization, etc.), component portability, technology independency, etc.

In terms of human resources, constructing component libraries at large is a fully fledged job. The comparison with COBOL is amazing in the sense that SOA is not only a problem of architectural style but also of service production economy with human facets (specialized jobs, skills, teams, etc.). COBOL software modernization toward SOA is hence, above all, an intellectual gap in the broad field of software development.

Models are implementation springboards, comprehension/ communication supports and also roadmaps to drive choices, decisions, orientations, etc., when modernization prompts for transformation strategies when reshaping applications.

In the *Currency* example, once materialized, it can be packaged and later deployed as a stand-alone service or with a set of closely related services.

In Java EE, for instance, *Currency* may be embodied as an EJB component. It is a *Stateless Session Bean*, which is, by definition, a fully functional component in the sense it has no remanence: callers share it, from one call to another, without the possibility of relying on the component's states[1].

In the Java code below, there is a clear distinction between the component's interface named as *Currency* and the component implementation named as *CurrencyImplementation*:

```
@javax.ejb.Stateless
@javax.ejb.Remote(Currency.class)
public class CurrencyImplementation implements Currency
{ …
```

In EJB, the (non-exclusive) choice between a "local interface", a "remote interface" (prior choice) and "Web service" is a question of access degree. In this context, we may add to the component another visibility, that of a Web service:

```
@javax.ejb.Stateless
@javax.ejb.Remote(Currency.class)
@javax.jws.WebService(serviceName = "Currency")
public class CurrencyImplementation implements Currency { …
```

The functional view of this code is the offering of the convert function in the component's interface:

```
@javax.ejb.Remote
public interface Currency {
        double convert(double amount, Currency
        source_currency, Currency target_currency,
        RoundingType rounding /*, etc.*/);
}
```

The QoS view aims to support SLA. For example, credentials may be assigned to a role, e.g., *"FranckBarbier"* below, to control and limit the access to the *convert* function:

1 Note that this is the original "theoretical" definition of a service, which in essence has no persisting state.

```
@javax.ejb.Stateless
@javax.ejb.Remote(Currency.class)
@javax.jws.WebService(serviceName = “Currency”)
@javax.annotation.security.DeclareRoles(“FranckBarbier” ,
“Jean-LucRecoussine”)
public class CurrencyImplementation implements Currency {
        @javax.annotation.security.RolesAllowed(“Franck
        Barbier”)
        double convert(double amount, Currency
        source_currency, Currency target_currency,
        RoundingType rounding /*, etc.*/) { …
```

Setting up values for performance attributes (through load balancing administration from a Java application server console) is also possible, but it occurs by breaking the compatibility with EJB, namely the boldface annotations below are product-dependent (GlassFish Java EE server from Oracle):

```
@javax.ejb.Stateless
@javax.ejb.Remote(Currency.class)
@javax.jws.WebService(serviceName = “Currency”)
@javax.annotation.security.DeclareRoles(“FranckBarbier” ,
“Jean-LucRecoussine”)
@StatelessDeployment(maxInstances=10,
minInstances=5)
@StatelessDeployment(poolCacheTimeout=30) // default
is 60
public class CurrencyImplementation implements Currency {
        @javax.annotation.security.RolesAllowed(“FranckBar
        bier”)
        double convert(double amount, Currency
        source_currency, Currency target_currency,
        RoundingType rounding /*, etc.*/) {…
```

Performance is constrained by the limited space and time allocated to the Java objects “realizing” the *Currency* service at runtime (see again Figure 4.2). No more than 10 instances is the number that establishes the low-throughput rate for software clients of *Currency*. Instances live no longer than 30 s. To summarize, the service is configured for low performance. This may correspond to favoring a greater performance of the other deployed and running services. In

terms of service provisioning and delivery, this may also amount to a free service. In other words, free services are often provided with lower power, while high-end versions are being reserved in another deployed module for paying customers.

This EJB example perfectly illustrates the importance of careful design: beyond coding functionalities (*convert* function and probably other client-friendly functions), there is a need for configurability. This example is realistic, not a dynamic composition. Design choices are heavyweight. We especially show that there is a gap to obtain the above EJB code from the models in Figures 4.3–4.6. To solve this problem, in Chapter 6, we enter into further details to first sketch and next precisely state the method, which enables a complete transformation/modernization, i.e. all code artifacts are obtained from models.

What do we demonstrate? With EJB, we sketch *Currency* as a homemade component. Several functional problems remain like acquiring exchange rates in a timely fashion (adequate rate update frequency). We show how to set up QoS features. We also underlie that code production inevitably generates postponed maintenance.

Enterprise JavaBeans (EJB) is a powerful technology to bring SOA to life. In the logic of COBOL modernization, COBOL code with currency conversion aspects will move to UML models and from these, EJB source code will be generated. This realistic scenario relies on findings presented in the remaining chapters of this book.

5.2. Service as Internet resource

The previous section shows a kind of “private SOA” in which many SOA components are in-house pieces of software even if some of them are probably externally acquired (open-source, outsourced or fully packaged like COTS components). They have also different formats from source code to binary software.

However, the Internet is above all an infinite marketplace (and the biggest SOA incarnation) of computing resources, including, of

course, services. Websites like www.programmableweb.com are a source of bargain, shopping and simply doing business. On-demand services are found there and, more generally, everywhere on the Internet.

In this section, instead of incorporating a *Currency* service into the application that supports the *Payment* business process, the followed SOA strategy is reusing *Currency* as a ready-to-use Internet computing resource. In other words, we experiment from the Web existing running services that are able to provide the desired currency conversion functionalities. We shift the problem from heavyweight development to connecting with something external. The main risk is a loss of control with respect to the application(s) currently using these externalized runtime functionalities.

From the modernization viewpoint, a model of *Currency* and its imposed (design with reuse) and potential (design for reuse) interactions, with, for instance, *Check writing*, may lead to match the legacy COBOL code to an immaterial component. It means that we can be in a situation in which we cannot match this legacy code to a materialized *Stateless Session Bean* as done before with EJB. Instead, this corresponds to a running service over the Web, which takes over the set of required currency conversion functionalities.

Obtaining something internally or externally does not matter when we pay attention to all types of impacts. Having a component as internal or external does not create a great difference: in both cases, this is subcontracted fabrication from requirements. In effect, in-house code production, third-party code acquisition and running service connection are three alternatives; they share the fact that computation requirements must be formally extracted and exposed by mining the COBOL code base.

In this context, the specification of functional requirements (again, the model) plays a great role in the management of risks in cases in which the externalized service shows, after some utilization, some “weaknesses”. In other words, the expected functional sophistication (precision, rounding, rate update frequency, etc.) is a first guide if we

have to switch from a service provider to a new one when problems arise.

The second guide is the expected SLA (security through mandatory encryption (Hypertext Transfer Protocol Secure (HTTPS)), availability greater than 99% for, say, 1,000 calls per day to the service, etc.). The provider, beyond "announcements", must guarantee the SLA. In other words, means must exist to precisely measure the expected SLA. In this line of reasoning, as discussed below, cloud providers offer application administration support in a programmatic way. Programmers may develop lateral administration programs (platforms as a service (PaaS) and infrastructure as a service (IaaS) levels) to better control applications in the Cloud.

The third issue is interoperability with third-party technologies. For example, the paying service (see section 5.2.1 below) can also be used freely from a compatibility constraint about the Web Services technology; it indeed works as a simple "data feed" service, i.e. simple requests and responses over HTTP, even HTTPS, are conveyed. In particular, this service may return *JavaScript Object Notation* (JSON) objects for readily processing results in Web pages using JavaScript.

There is then a functional quality of the subject service and, in parallel, the traditional QoS. For the former quality, many details are of importance when choosing the service and the provider. Typically, currency conversion facilities are a business service or domain-specific service, the domain being finance. Obeying national laws, bank standards, rules, regulations, etc., is another key facet of functional quality. A finance sub-domain like currency trading may also be interested in enhanced (functional and/or non-functional) features. We expect, from a chargeable service, great sophistication beyond "technique serving business".

5.2.1. *Pay-per-use service*

There are plenty of URLs from which currency conversion functionalities are accessible and consumable.

At the time of writing this book, *fx.currencysystem.com/webservices/CurrencyServer5.asmx* is the documentation of a professional paying Web service. It is a Simple Object Access protocol (SOAP) Web service over HTTP or HTTPS. It offers a lot of functionalities, including legal information that is often of great importance in finance. In Java, the *convert* function is simply called as follows:

```
@WebServiceRef(wsdlLocation = "META-
INF/wsdl/fx.currencysystem.com/webservices/CurrencyServer
5.asmx.wsdl")
com.currencysystem.webservices.currencyserver.CurrencySe
rver service;
…
com.currencysystem.webservices.currencyserver.CurrencySe
rverSoap port = service.getCurrencyServerSoap12();
Object o = port.convert(licenseKey, fromCurrency,
toCurrency, amount, rounding, format, returnRate, time,
type);
```

The signature of the *convert* function is in essence the degree of sophistication (and thus functional quality) of the requested service. The *licenseKey* parameter reveals the paying nature of the call.

The *convert* function is documented in the *Web service description language* (WSDL) specification:

```
<wsdl:documentation>
Currency Server – An exchange rate information and currency
conversion Web service.
</wsdl:documentation>
…
<s:element name="Convert">
<s:complexType>
<s:sequence>
        <s:element minOccurs="0" maxOccurs="1"
        name="licenseKey" type="s:string"/>
        <s:element minOccurs="0" maxOccurs="1"
        name="fromCurrency" type="s:string"/>
        <s:element minOccurs="0" maxOccurs="1"
        name="toCurrency" type="s:string"/>
```

```
            <s:element minOccurs="1" maxOccurs="1"
            name="amount" type="s:double"/><s:element
            minOccurs="1" maxOccurs="1" name="rounding"
            type="s:boolean"/>
            <s:element minOccurs="0" maxOccurs="1"
            name="format" type="s:string"/>
            <s:element minOccurs="1" maxOccurs="1"
            name="returnRate" type="tns:curncsrvReturnRate"/>
            <s:element minOccurs="0" maxOccurs="1"
            name="time" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1"
            name="type" type="s:string"/>
    </s:sequence>
    </s:complexType>
    </s:element>
```

In this documentation, we may, for example, observe that the rounding is just a *Boolean* option. The client application must then take charge of precision/rounding management by setting the rounding parameter to *false* and later applying a local policy (business rule).

As a professional service, the functional sophistication is high via, for example, the possibility of setting up, before calling *convert*, the *Foreign Exchange* (FOREX, the market of currency trading) feed source by means of other Web services offered by this site.

What do we demonstrate? There is no magic. Paying services offer a large range of functionalities and thus sophistication. However, there is no QoS observability and controllability for this chargeable service. As a Web service, this service fully complies with the SOA spirit. As a professional service, this service is provided with many other services, which, in general, avoid code enhancement (not true for rounding issues) beyond simple calls.

5.2.2. *Free service*

Otherwise, at the time we are writing this book, www.restfulwebservices.net/service.aspx?ID=2 is the documentation of a simplified free service. As minimal sophistication, only the

conversion rate may be captured for deferred conversion in the caller program:

```
@WebServiceRef(wsdlLocation = "META-
INF/wsdl/www.restfulwebservices.net/rest/CurrencyService.sv
c.wsdl")
net.restfulwebservices.servicecontracts.rest._2008._01.Curre
ncyService service;
…
net.restfulwebservices.servicecontracts.rest._2008._01.ICurre
ncyService port =
service.getWebHttpBindingICurrencyService();
Object o = port.getConversionRate(parameters);
```

The *parameters* object sent at request time simply has the *setFromCurrency* and *setToCurrency* functions to, in return, get the right exchange rate. In contrast with the paying Web service, many professional capabilities are missing like the temporal value of the rate, which plays a very important role in, for instance, currency trading applications. As a consequence, conversion code is part of the caller program; it is a source of (undesired) extra maintenance.

What do we demonstrate? That, again, there is no magic. Free services have poor features. In terms of maintenance, wrapping code is necessary (conversions themselves), as is calling other services to get any other (missing) useful information. We may, for example, require currencies in International Organization for Standardization (ISO)-compliant formats for any kind of displaying or checking. As a Web service, this service also complies with SOA, but with disappointing functional support.

5.2.3. *Data feed service*

As outlined many times in this book, it is important to be aware that Web Services are just an instantiation of SOA. So, it exists over the Web or in information systems of organizations, service-oriented architectures that are compliant with other standards or, with no standard at all; they are the proprietary solutions. The telecom

industry, for instance, may greatly benefit from the SOA paradigm without any necessary link to Web Services as an imposed technology.

A simple method of obtaining currency conversions is sending and receiving data based on a data feed service. As an illustration, we take the example of a currency conversion service (*openexchangerates.org*) that does not obey the Web Services standard, but it is capable of returning converted values in appropriate formats, JSON in this case. This service is described in greater details here: *openexchangerates.org/documentation*. Another key feature of this service is the fact that some basic features are free while sophisticated ones are chargeable.

This data feed service perfectly responds to currency conversion requirements. For example, free features include the fact that exchange rates are updated every 10 min. In contrast, paying features are, for instance, communication in secure mode with encryption (HTTPS).

The homemade *convert* Java function below just accesses to the *latest.json* object, which, free of charge, returns all of the available (effectively updated) conversion rates:

```
public static double convert(String licenseKey, String
fromCurrency, String toCurrency, double amount) throws
java.net.MalformedURLException, java.io.IOException {
        java.net.URL url = new
        java.net.URL("http://openexchangerates.org/api/latest
        .json" + "?app_id=" + licenseKey);
        java.net.URLConnection connection =
        (java.net.URLConnection) url.openConnection();
        if (connection != null) {
        javax.json.stream.JsonParserFactory factory =
        javax.json.Json.createParserFactory(null);
        javax.json.stream.JsonParser parser =
        factory.createParser(connection.getInputStream());
        … // homemade code conversion here required
```

As for the free service, conversion code is part of the caller program: a source of future maintenance. The fact that JSON is used

also forces us to deal with this technology and its application programming interface (API). Any switch from JSON to a competitor may then also hinder future maintenance.

Interestingly, the paying part of this data feed service may directly return the conversion result through replacing *latest.json* by *convert*/etc:

```
public static double convert(String licenseKey, String
fromCurrency, String toCurrency, double amount) throws
java.net.MalformedURLException, java.io.IOException {
        java.net.URL url = new
        java.net.URL("https://openexchangerates.org/api/convert/" + String.valueOf(amount) + "/" +
        fromCurrency + "/" + toCurrency + "?app_id=" +
        licenseKey);
        javax.net.ssl.HttpsURLConnection connection =
        (javax.net.ssl.HttpsURLConnection)
        url.openConnection();
        … // no code conversion here required, result from
        service call has to be immediately consumed only
```

What do we demonstrate? That after deciding to reuse currency conversion functionalities over Internet (by opposition to in-house solutions like EJBs), problems remain in choosing the right solution and extrapolating consequences in terms of forthcoming maintenance especially. We show that services in SOA differ in technologies (Web Services, data feed services over Internet, organization internal services with EJBs, etc.), in protocols (RESTful Web Services, SOAP Web Services, etc.), in surrounding technologies (JSON, etc.), etc.

SOA is the antithesis of COBOL, but it raises its own problems. What concretely emerges through these four samples of services in SOA, is the strong need of an intermediate level to abstract technology details at early design time, which may correspond to business matter consolidation resulting from COBOL mining. At the beginning of this chapter, we only briefly discuss *Currency* in a modeling logic. Going further in-depth, Chapter 6 revisits *Currency* as a model. In the

meantime, we complement this thought on SOA by introducing a kind of SOA heaven.

5.3. High-end SOA

SOA is a great progress in information technology (IT), but generates new problems on its own. A SOA-centric organization must install a kind of meta-architecture to set up, deploy and administrate a set of non-homogeneous services, be they local, remote, homemade, third-party, etc. As an analogy, there is a resemblance to CICS in COBOL, which naturally imposes a computing infrastructure "style" totally linked to COBOL programming "peculiarities" with, it should be recalled, time after time, very undesired side effects.

In SOA, such a meta-architecture is named as an *enterprise service bus* (ESB). *Oracle service bus* (OSB) or *BizTalk* from Microsoft are products in this area. No vendor lock-in products exist like *OpenESB*. In the Java world, they are mostly based on the *Java Business Integration* (JBI) standard.

Figure 5.1 shows that the main purpose of an ESB is to transparently satisfy the constraints posed by effective technologies used for designing services. This is typically the ease of the management of multiple protocols to access to and exchange with the services visible on the bus.

For example, Figure 5.1 shows the interrogation of services with SOAP, while these services are based on C/C+ through the *Java Native Interface* (JNI), which basically allows the integration C/C++ libraries and code in Java. What we basically expect from an ESB is, in this case, hiding C/C++, JNI and, above all, any adaptation code that fits the required protocol between the service consumer and the service producer.

As a comparison, the example of a data feed service in previous sections leads to specific Java code. It corresponds to using the Java APIs and libraries to cope with HTTP/HTTPS and the JSON format. An ESB would primarily mask this technology dependency.

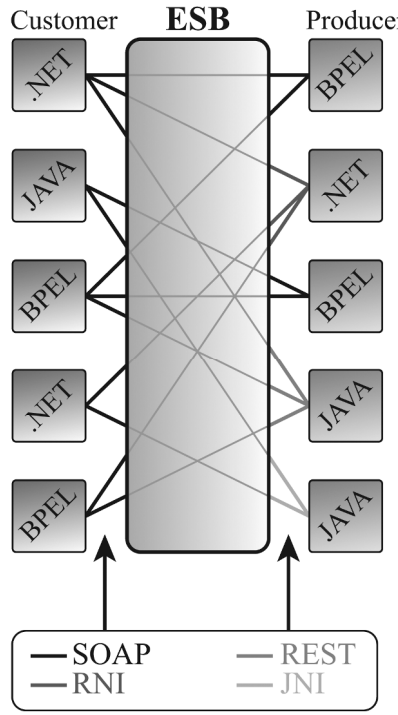


Figure 5.1. *ESB overview (image taken from Wikipedia)*

ESB-based information systems are very advanced middleware. Organizations with such a service-oriented computing infrastructure are rare. Further analysis shows that this kind of system might be appropriate for IT-based inter-organization business partnership, for instance, among car manufacturers, car dealers and car parts and spare parts' suppliers.

However, having for an organization an ESB cannot be detached from concomitant thinking about a cloud computing strategy (section 5.5 below). Indeed, at the core of an ESB, there are middleware-based (underlying) services like a service repository (with, for instance, service naming, service updating, etc.), facilities and management. An ESB is then a heavyweight computing infrastructure in the sense that it is very close to a private cloud.

5.4. SOA challenges

In an intuitive way, this chapter presents SOA as a panacea. This is justified with regard to COBOL and its so-awaited modernization. However, precautions are required in a SOA strategy. Before the implementation of an ESB or a private cloud, to make SOA real, there are hands-on considerations to outline:

– interface types and protocol issues: as shown in examples, services are designed from technologies and, accordingly, themselves generate usage constraints, their access in particular through interfaces and protocols. Consuming code outside the scope of an ESB then always leads us to rub against technologies' roughness. For example, the fact that the data feed currency conversion service above is not aligned with the two most used Web Services access protocols (SOAP and RESTful) is embarrassing in a logic of long-term maintenance.

– substitutability issue: services are components. Components are easily and straightforwardly composable in particular because of interchangeability of implementations. In contrast, interfaces are substitutable with more difficulty. We provided four ready-to-use solutions (one in-house solution with EJB, two existing Web Services over the Web and one JSON-based data feed service). Creating a design environment with the necessary flexibility to timely replace any of them by another is a true challenge. This remains a kind of magic at this time, apart from models as contracts about requirements and the possibility of generating the consuming code independently of the specificities of each service.

– functional sophistication issue: to anticipate evolving requirements, it may be appropriate to choose a currency conversion service with a lot of functions, and a lot of parameters for each function. This is the case of the paying Web service above. The dilemma is paying for unused functionalities versus paying less or nothing. In the latter case, the risk of costly maintenance at the time, where functional requirements increase, is high. Such an anticipation is beneficial if the initial choice is confirmed as the good one on a long-term scale. There is then no maintenance coming from the need to reuse another service, or the satisfaction of odd technology constraints in general (new API, new access protocol, new security issues, etc.).

– sustainability issue: the great risk of a service is its sustainability. What is the guarantee behind the longevity of its service provider, the continuous quality-driven support of the service or its necessary versioning? Choice of software, be it package software or components, is not a new concept. Similar to COBOL, in-house software raises the same issues. Loss of knowledge, know-how, skill, etc., on software may occur for both internal and external software. The case of services is characterized with fewer side effects. Indeed, the low coupling resulting from componentization may greatly help the management of software non-sustainability, even if adaptations would probably be inevitable. The question is only the amplitude of these adaptations and the generated costs.

– pricing issue: as an illustration, the paying Web service above has a utilization cost of 295 use for 1 year, with 10 accesses/day. Each organization has analytical accounting that allows decision-making with respect to in-house fabrication versus outsourcing versus buying services over the Web or elsewhere. Software acquisition or rejection is also ruled by non-functional concerns such as robustness, response time, etc.

– QoS issue: QoS cannot be ignored. For example, service security, "availability" mostly, may be a discriminating criterion for buying a service. SOA comes up with SLA support to better meet QoS requirements and control, at runtime especially, if the said quality is actually observable. We sketch above, in our own implementation of a currency conversion Web service, how SLA may be instrumented from the supplier viewpoint.

– interoperability issue: it is highly important. It is like shopping in a hypermarket: one hopes that COTS-dried lasagna brands finely accommodate with COTS tomato sauces, grated cheeses, various ground meats, etc. The application that needs currency conversion functionalities may comparatively evolve through an enhanced *Payment* business process: currency, check writing, check printing and check posting. The postponed search for check printing and check posting facilities must not lead to difficult (probably late) assembling with currency and check writing already deployed and in use.

5.5. The Cloud

It is difficult to write something about the Cloud without introducing more confusion. To that extent, we strongly advise the reading of McFedries' book [MCF 12]. This book develops efforts to make the Cloud as simple as possible; it is a kind of dream in the cloud literature "ocean".

Going to the Cloud is for most organizations a question of long-term strategy via a one-way road. Naively, we may wonder if there is a real choice with regard to the rapid evolution of Internet computing. To be in sync with this feeling, this section tries to discuss the Cloud from a fresh and innocent perspective. With this in mind, when considering COBOL software modernization, we preferably discuss the cultural gap between COBOL programming and cloud computing, in particular the whys and wherefores. Moving legacy COBOL applications to the Cloud is appealing, provided that there is a suitable technology (see in Chapter 8). In other words, a two-step move (legacy to (non-cloud) modern, modern to cloud) might be considered both laborious and tedious.

More opportunistically, if such a modernization technology completely allows the neutral description (through UML models) of the business logic and the functional requirements of any legacy application (reverse engineering), then generating the equivalent modernized application to the Cloud (forward engineering) is a source of net progresses. Concretely, the move from COBOL to Java EE or .NET is compatible with the Cloud since these two well-known computing environments[2] may be managed as PaaS in the Cloud[3]. Ultimately, we think that everything is a matter of API. It means the IaaS–PaaS–software as a service (SaaS) trinity has made physical machines, namely servers, more "transparent". Computing infrastructures and platforms through virtualization has become programmable in terms of both provisioning and deprovisioning,

2 For the sake of clarity, note that the expressions "middleware", "application server", "platform" and "virtual machine" are used with the same semantics.
3 The limitation for .NET is the single choice of Microsoft Azure PaaS offer.

administration, (re)configuration and more. The programmable Web has leveraged a never-seen possibility and its contingent opportunity.

As discussed throughout this book, the COBOL software modernization spirit inevitably converges to a SOA/SaaS problem. As underlined, "models" aim at hiding the PaaS and IaaS layers, apart from models like *UML Deployment Diagrams* that aim to deal with associations between logical components and their distribution and assignment to logical and/or physical platforms/infrastructures.

5.5.1. *COBOL in the Cloud*

Deployment of COBOL applications in the Cloud is theoretically possible through virtualization of physical servers, operating systems and middleware platforms. Furthermore, COBOL supporters propose effective solutions. Nonetheless, the theoretical facet of "COBOL in the Cloud" results from past outright failures. Legacy COBOL code has to be, most of the time, significantly altered to move to the Cloud or no change occurs. In the former case, this is the logic of COBOL-to-COBOL modernization with few advances. In the latter case, this makes no sense to keep architectures as monolithic, while the Cloud is the idealistic vision and incarnation of SOA.

In everyday practice, the current "COBOL in the Cloud" market is oriented toward the deployment of young COBOL applications, which are discussed in a way tolerating the Cloud's principles. Obviously, the "COBOL in the Cloud" market is sustained by historical COBOL supporters, so as not to break the COBOL way of life. However, young COBOL, compared to legacy COBOL, is not the big deal. What is misleading is the inevitable rupture in the way of thinking: thinking COBOL is not compatible with the Cloud, and more precisely, with SOA.

Even though the "COBOL in the Cloud" principle makes sense in terms of sketchy technical solutions, it is difficult to bypass the SOA principle if we really hope to gain value. The connectivity between the aged COBOL and the cloudy COBOL is only a matter of proprietary solutions with long-term dependency and hence lock-in.

The Cloud is also a conceptual framework and its consequential opportunity is the possibility of talking about software in a very different way. Moreover, business must govern software while COBOL is, in spite of itself, the eminent representative of the contrary. As mentioned in the beginning of this book, service computing is a state of mind: business innovation through services replaces “business automation”.

5.5.2. *Computing is just resource consumption*

As observed in the prior lines of this book, COBOL programming is a state of mind that culturally takes into account resource management in a very unwise way. Programs, applications and information systems use resources. Their design tends to work out resource usages (location, access, load, release, etc.) in the way that software is totally rigidified. Consequently, having programs with details on resources and usages generates sizeable and tricky maintenance at the time these resources and their potential (novel) usages vary in quantity, capacity, nature and capability; the everyday life of IT departments.

As an illustration, a suppliers’ database is a resource-providing persistence facility. Legacy programs like COBOL programs may refer to it through a set of named files, their access method (sequential, indexed or direct through code hashing), etc. At worst, encoded bytes in files may have many “exotic” senses: “end of data”, “next supplier category”, etc. In some cases, programs are using relational databases instead of files, where we may find programs, provider names of databases, IP addresses, port numbers, etc., as hard-wired data: a source of inflexibility when resources and/or usages aim to change.

Modern programming is first of all dealing with resources and usages in a logical way. Physical characteristics of resources are set up outside programs and defined within administration tasks. For example, administrating a Java EE application server instance (a running middleware) amounts to setting up and configuring resources (e.g. naming with name-based access mechanisms), parameterizing

potential usages through properties and values assigned to resources (e.g. resource connection pooling with minimal and maximal sizes of pools). In such a context, programs (components or modules, i.e. JAR, WAR or EAR Java bytecode files) handle logical resources, which are mapped to real (physical) elements at the middleware level. Programs use resources in a transparent way, but they must pay attention to resource deficiencies or failures, typically reaching the maximal size of a resource pool. As a direct consequence, a Java EE application has to catch middleware-oriented exceptions to preserve its integrity and functioning from resource malfunctioning, overloading, temporal unavailability, etc. Note, again, that section 5.1 illustrates this problem through the assignment of performance intervals to deployment parameters.

More generally, there is a well-separated effort on resource sharing, pooling, saving, etc., in Java EE compared to COBOL. In other words, resource management is an explicit task on its own in Java EE, while a COBOL-preferred approach provides resource monopolization for a given usage: “vertical computing” model (see section 3.2). Besides, resource management in COBOL is fully implicit behind non-compact and unintelligible code.

Keeping Java EE as an example, administration covers load balancing, the possibility of assigning components or modules to other machines, and being physical or virtual like a Java EE virtual machine. In the Java EE computing framework, there is thus management of computing infrastructures (starts and stops of (virtual or not) application servers) and computing platforms. Each instance of a Java EE platform may have its own characteristics in terms of declared resources and anticipated usages. The difference with COBOL is the clear separation between software development/maintenance that is fully detached from resource administration concerns. Nonetheless, organizations may need big (on-site) Java EE computing frameworks whose daily administration is both critical and considerable in terms of energy expenditure. This is the COBOL mainframe syndrome: COBOL programming habits and style lead to complex and big computing infrastructures and platforms, which are often dedicated to specific treatments. In order to

avoid falling back into the same trap, we may notice that Java EE tends to decrease maintenance efforts and increase reuse possibilities through a component/service-based approach. In contrast, in COBOL, programs become complicated due to new requirements ("normal" evolution flowing from business). Complication calls for infrastructure/platform adaptation. Adaptation imposes program reshaping (new odd intermediate files, new OLTP or batch programs as pre- and/or post-processing, etc.). This infernal circle has no end, leading to the situations described throughout this book.

5.5.3. *Cloud computing is also resource consumption, but…*

Since programs use resources, the Cloud is the realm within which programs may consider resources as unlimited in quantity and capacity. These two last notions merge because the Cloud hides, at the upper level, the knowledge that programs might have on resources. This is scalability, but, by their very deep nature, Java EE or .NET computing frameworks already address scalability issues. So, what is the key difference? In the Cloud, scalability is associated with elasticity.

In the Cloud, resources are both "logical" and "virtual" in the sense that they are invisible and, above all, immaterial. In principle, infrastructures and platforms are off-wall (the idea of "public cloud"). These are the notions of IaaS and PaaS. McFedries in [MCF 12] states the difference between the public cloud in which resources are shared with multiple organizations (multi-tenant model) and the private cloud (single-tenant model) that corresponds to the Java EE situation above described. They are also hybrid situations like having a private cloud hosted by a third-party cloud provider.

In the Java EE example, or from a private (locally-hosted) cloud, we cannot avoid resource setting-up, configuring and usage planning, and thus sizing. Elasticity is thus the infinite possibility of extending resources. No matter their quantity or their individual capacity, the cloud provider is intended to offer scalability beyond what is envisaged at the administration level in Java EE or .NET computing frameworks. Furthermore, administration tasks like, for instance, duplicating a Java EE virtual machine for accepting some extra load,

can be carried out on the fly. Such tasks can also be performed programmatically, a more accurate characterization of what IaaS and PaaS actually are. As an illustration, we may have our disposal statements to launch a Java EE platform, which is mounted with the Struts Web presentation library and API while, by default, the *JavaServer Faces* (JSF) Web presentation framework is offered.

The principle of resource virtualization plays an important role in the sense that resource consumers are detached from resource administration at large. Never mind how cloud providers add more hardware (servers, routers, cables, etc.); the deal is that they response in real time to infrastructure/platform needs. This remark makes us come back to the SLA concept: paid services oblige cloud providers to deliver a certain QoS. For example, reliability in the Cloud is mostly "availability" of the overall bought computing system. SLA may then, for instance, set availability to 98%. Cloud providers propose API to supervise, measure, even control (i.e. act on) application behaviors in the Cloud. Again, this strengthens the understanding of what IaaS and PaaS really are.

COBOL and the Cloud may be viewed as two different hermetic universes in the sense that, when reasoning on resource mutualization, that which the Cloud has been invented for, while COBOL is the historical incarnation and representative of resource monopolization.

5.5.4. *Everything as a service*

In the Cloud, the *Everything as a Service* (EaaS) precept applies. This book has no political goals other than viewing cloud computing as the preferred support of SOA. What is particularly challenging with the Cloud is the ratio between the quality of the offered services in comparison with their pay-per-use pricing, provided that many other critical issues are properly addressed: security, privacy, law, regulation, etc.

Put simply, a company having a Java EE computing environment is able to support SOA at the PaaS and SaaS levels only. Setting up a private cloud or choosing a public cloud provides the addition of IaaS

facilities. This point is important in COBOL since infrastructures embodied by mainframes are the basic obstacle for information system evolution. Indeed, in COBOL, sizing, along with downsizing, infrastructures and their management, are carried out in a physical manner, generating inertia, latency, etc. In contrast, IaaS does the same logically/virtually. Note that downsizing is quite impossible in COBOL. The major advantage of the Cloud is thus paying less when resource usages decrease.

Services differ in nature through two categories: technical services and business services. Technical services are those offered at the IaaS and PaaS levels; business services are those offered at the SaaS level. In the Cloud perspective, IT divisions become service brokers for the business needs. In other words, IT divisions have no more excuse of being monopolized by the management of infrastructures and platforms, the COBOL drain on budget. Accordingly, these can focus on business needs, their best support and automation in third-party computing infrastructures and platforms as well.

Behind the principle of cloud provider is the ability to provision services at, for instance, the PaaS level:

– naming services;

– database management services;

– messaging services.

PaaS services are nothing else than more or less sophisticated, middleware services, as those offered by Java EE, for example. In the Cloud problematic, IT departments may intervene, for instance, at the PaaS, regardless of the underlying infrastructure: the cloud provider offers platforms and their management in a transparent way (the cloud user has no knowledge on the IaaS layer).

More interestingly, the cloud user may access business services at the SaaS level, with or without taking over the responsibility and management of the PaaS and/or IaaS layers. At the SaaS level, the point is the dilemma between the in-house construction of business

services (extracted from the COBOL legacy matter in particular) and the connection with existing services:

– currency conversion services;

– check writing services;

– geographical location services;

– SMS, MMS and phone calls services.

5.5.5. *SOA in the Cloud*

Committing to the Cloud generates crucial challenges at the IaaS and PaaS. In Chapter 5 of [MCF 12], the author discusses the pros and cons of cloud computing, in IaaS and PaaS especially. For instance, the physical location of data in terms of country law and regulation, data server owner, data management policy, etc., generates preoccupations on data security and privacy.

However, from a business viewpoint, the key issue remains SaaS. In addition, from the COBOL software modernization perspective, interests are the way the business value engraved in COBOL may be ported in a world of pervasive services, regardless of the chosen IaaS and PaaS frameworks. Again, the avoidance of technology adherence cannot lead us to assimilate COBOL software modernization as a unique problem of infrastructures and platforms, even though the Cloud is by definition the antithesis of COBOL-like IT.

Roughly speaking, the Cloud put forward IaaS and PaaS as the today's means for diminishing IT costs in general. Cost and effort savings let the opportunity for IT divisions to stress SOA.

In the developing jungle of services, the discovering, choice, evaluation and validation along with the verification of interoperability with pre-selected and reused (external or in-house) services (see again the beginning of this chapter) are the constituents of a new software development approach. Interoperability refers to service orchestration, even choreography: how services may be

composed with each other in a seamless way. In this scope, we showed that models, being expressed in BPMN, UML or WS-BPEL, are extremely useful to design this composition.

SOA in the Cloud remains *a minima* a tough task and a new competency field for software engineers. Indeed, the current SaaS offer is pretty confusing when analyzing available services over the Web. For instance, billing services appear as coarse-grain services: understanding of functionalities through their accurate delimitation (i.e. what they really do and do not). Implications in daily use may be cumbersome because such services aim at being integrated in mission-critical applications.

As an illustration, remind each version (SOAP or RESTful Web service versus non-standardized data feed service based on JSON) of the *Currency* service. Each version's functionalities are well bounded, but it is extremely difficult to have a unified and synthetic view of all versions, as tried in Chapter 6 from a modeling perspective.

5.5.6. *The cloud counterparts*

The cloud vision in general may be considered as both idyllic and misplaced for computer veterans. Ironically, in [MCF 12], the author comments on a Gartner Group's "hype cycle" that shows a perceived disillusionment since 2010, while the peak of (likely arbitrary) expectations and "joy" was in 2009. In other words, some large-scale cloud experiences will soon lead to failures with resulting lessons learned, cleared pitfalls, etc.

The Cloud is just the result of the maturity of Internet computing. In practice, resources are transparently accessible without specific installation, but with the exception of continuous Internet connectivity with adequate throughput: a significant risk in all cases because of dependency at large.

A defeatist attitude, for COBOL people in particular, is to reject the Cloud. Nonetheless, the Cloud is just the infrastructural backbone

of SOA and SOA is the vector of new business through software. This affirmed economical trend cannot be rejected.

So, the Cloud brings out non-surprising issues as follows:

– money: reducing IT budgets motivates leaving COBOL. The Cloud's entry costs are significant in terms of requirements on trained people, acquired expertise. As for any technology adoption, expenditure peaks may be incompatible with available IT budgets and hypothetic returns on investments. At worst, daily functioning costs may be out of control, unexplainable, non-transparent, etc., a nightmare. The appealing "pay-per-use" principle hides the strong need of perfectly knowing what is really and accurately used between the resource provider and user.

– technology: dependency through vendor lock-in, poor control on security, performance, data location with risks of loss, violation, unavailability, illegality, etc. All of these issues are potential risks. What is funny is the fact that the COBOL spirit (i.e. all-is-made-in-house) may remain "the happy path". In other words, cloud computing has not yet demonstrated that these risks can be addressed in a cost-effective and timely way. The ugly face of IT is indeed technology instability, volatility, obsolescence, downgrading, etc. The Cloud is unfortunately not evading these. The problem of abstraction, for example, is really essential. We have to be able to design information systems and applications that last without being prisoner of technology peculiarities. The Cloud tackles this with API like *jclouds*, for instance, in the Java world. However, MDD, as discussed all along this book, has also a huge role to play in the abstraction challenge.

5.6. Conclusions

Behind SOA, there is an extreme intellectual switch: instead of being a prominent cost sector for the business, why not consider "SOA in action" as the greatest opportunity for IT departments to become significant sources of (unimagined) revenues in organizations? Indeed, as highlighted in Chapter 1, the degree of innovation resulting from successful SOA is a huge progress to create value beyond the simple fulfillment of initial demands. This

revolutionary idea makes us returning to the travel management domain (section 1.3 in Chapter 1). In this domain, travel companies have the possibility of conquering markets both from the business side and software side: software may become profitable beyond their own (internal) usages. In other words, why not to sell our own software services provided that these are componentized and free from specificities? Only usages have to be application-specific.

From currency conversion facilities resulting from COBOL modernization, we develop in this chapter a SOA-modernized application with *Currency*, *Check writing*, etc. The necessity and, later, the opportunity of developing, deploying and running a *Currency* service, a *Check writing* service (even a *Check posting* service), etc., really challenge us from a business viewpoint. Only a missing link remains: a trivialized computing infrastructure in which everybody on this planet may share and exchange these services in a more or less transparent way.

In this line of reasoning, the Cloud comes up. Nonetheless, naively, are SOA and the Cloud the legacy approaches of the future? The degeneration of COBOL came from, when facing volumes and increasing complexity, the impossibility of supervising and controlling portfolios of applications making up information systems. SOA in the Cloud is the most promising framework because it is initially thought and designed to correct legacy IT as COBOL IT. But it is no miracle: SOA in the Cloud, outside MDD, makes no sense to tame complexity.

6

Model-Driven Development (MDD)

Professional software development eventually always leads us to choose effective technologies, vendors, products, versions and so on. To be free of these is truly challenging. Controlling the side effects of technologies is more reasonable; it is the essence of Model-Driven Development (MDD).

By definition, software engineering is conceptual. For example, algorithms exist outside the scope of their operational formulation in programming languages. Doing the same at a larger scale, i.e. for information systems and applications, relies on models and languages for expressing these models. Even though this approach may satisfy engineers, users are not able to capture the deeper meaning of software without screens, keyboards, mouses, etc. Models are images of ideas in minds, but running software makes these images "playful" and thus catchable for users. The temptation to rush to technologies is then just the desire of users, the software's clients. Engineers often have the same desire to rapidly have something concrete in their hands.

MDD aims at creating interfaces between ideas of software and their incarnation in technologies. The key reason is the huge difficulty or, sometimes, the simple impossibility of revisiting these ideas once engraved in software.

In this spirit, degeneration of Common Business-Oriented Language (COBOL) in particular came from the expansion and intensive use of various odd COBOL dialects. In this context,

beyond simple compilers, the need for rich Integrated Development Environments (IDEs) devoted to COBOL[1] was a daily reality. In addition, history also showed the need for very specific execution environments, Customer Information Control System (CICS) and mainframes typically. Over the years, such a pragmatic orientation became a lock-in: software ideas no longer remained ideas, but became COBOL circumlocutions, i.e. impoverished and inextricable ideas in limited-scope jargon.

6.1. Why MDD?

In a nutshell, software artifacts as outputs of software development processes are representations of "business requirements" in "technologies' languages". In this scope, maintenance amounts to adapting information systems, applications and programs to constraining technical/technological conditions (90% of time) instead of promptly reacting to business changes with agility (10% of time). The former rate contributes to the bad, but proven, reputation of IT as a budgetary ogre. The latter rate excludes the serious listening to users' desires and, consequently, also contributes to the (likely justified) vision of Information Technology (IT) as, often, an autistic department in organizations.

To imagine new software development paradigms to be in a position to neutralize the impacts of technologies on agile changes has made, two decades ago, MDD industrially realistic. Primarily, MDD is the production of software artifacts at large in formats and forms that do not refer to technologies. MDD is thus the COBOL antivision. Beyond this observation, without contradiction or paradox, MDD may also be viewed as a fighter against newer technologies. These, similar to what COBOL did in the past, tend to trap us in their peculiarities.

Typically, not to make the same mistake again, we must pay attention to the fact that the Cloud is materialized by a profusion of offers, and thus Application Programming Interfaces (APIs) that may

1 For instance, Pacbase COBOL from IBM, with its surrounding IDE, is a COBOL macrolanguage, whose obsolescence has been officially programmed in 2015.

lead us to a COBOL-like chaos in a short time. That would happen in possible relation to the lock-in syndrome: code dived into these APIs is frozen once and for all. On balance, COBOL-trapping is real life. Cloud-trapping? Go away.

The core principle behind COBOL software modernization is, therefore, the expression of software systems neither in COBOL languages/macrolanguages nor in Service-Oriented Architecture (SOA)/Cloud counterparts. Models must first and foremost be business-related, and consequently technology-free.

6.2. Models, intuitively

A roadmap is a model of the road infrastructure of a region. Asking questions (e.g. “How far by car is city A from city B?”) on this infrastructure may be achieved with the help of the map only. The counterparty is, not all questions can be answered due to the map: “What is the width of the road at this location?” comes with no response.

From this sample, we characterize a model as an abstraction: a model emphasizes some details to the detriment of others. Be careful, this incompleteness can sometimes be penalizing. So, why are we interested in this? What follows: omitting details lowers complexity. The “separation of concerns” principle applies. Out-of-the-scope details are parasites. In practice, some categories of details are often contextually irrelevant. In other words, the topical nature of details varies in space and time.

Returning to the road infrastructure, we may imagine another roadmap with road widths for trucks (and without distances to keep the map readable). The two roadmaps are two different models, because they stress two different detail categories. The key issue is to keep in mind that the inter-relation between the two natively results from the fact that they represent the same road infrastructure. In effect, truck drivers may be hindered by the manipulation of two paper roadmaps in their cabin. Fortunately, the Internet and tablets now exist to simplify model manipulation of everyday objects and ideas.

As an analogy, MDD is the organization of "similar matter" in software development processes. In software, models are representations of software artifacts and their relationships. Practically, representations are embodied by means of languages understandable by IT professionals: entity-relationship modeling, state modeling, workflow modeling, etc. State modeling, for instance, is the fact that a variable of type "natural number" has two possible states: zero or positive. What is abstracted is all of the possible values behind "positive".

6.3. Models, formally

MDD gurus and communities have highlighted the "model" term in a manner that is nothing short of a hold-up. In epistemology, model is the property of many sciences, which share a common sense of "model" and make their own declination(s). Honor to whom honor is due, mathematics develops models (e.g. natural numbers), instantiates them with representatives (e.g. 0, 1, 2, etc.) and characterizes them with laws (e.g. addition of two natural numbers is a natural number).

Returning to the classroom, models define things in intention while *in extenso* definitions have the drawback of big volumes and thus poor (human) manipulation. In psychology, behavior models are the cases of psychological deviances; they are especially useful for constructing a disease typology. In computer science, data types are models of variables; they are useful for type checking. Compilers control canonical usages of variables according to their type and thus reject those that are unsupported by this type. Nonetheless, *in extenso* definitions are the privilege of computers. Ultimately, software applications are images (models) of real-world things in masses: for example, there are as many customers in a company's database as the number of customers of this company. Epistemologically, models are objects for reasoning.

MDD brings out the following added value: models are computable (with regard to the calculation theory) in the sense they are transformable in transformation chains; they may also be

executable. The latter point is so special that it is discussed in section 6.8 of this chapter.

6.4. Models as computerized objects

Prior discussions on currency conversion functionalities as a self-contained *Currency* service in SOA showed four code samples (Chapter 5). Each provided a set of technological capabilities, limitations and constraints: one is a SOAP Web Service while the second is not; it does not follow a common standard. In short, none of the four favors a conceptual approach, i.e. a comprehension and a characterization free from technologies. The four proposed forms are in essence impoverished by the capabilities/limitations/constraints of the implementation support. For example, the JavaScript Object Notation (JSON) exchange format (second sample that does not comply with the Web Services standard) may be noising for comprehension, i.e. we cannot understand the whys and wherefores without knowing the JSON technology.

In this book, at the same time, we tried to describe the idea of a general-purpose *Currency* service in natural language. The natural language is recognized as a creator of probable dissonant interpretations. This is especially true when many people share the topic of interest for long periods. Typically, the turnover of people in companies creates some discontinuity in the daily management of business concepts. New incomers acquire knowledge through readings, exchanges, experimentations, etc. Nonetheless, uniform comprehension and the interpretation of concepts among all stakeholders constitute a permanent battle.

So, a *model* of currency conversion functionalities is a form that aims at diminishing the interpretation latitude of any mother language and erasing the adherence to a technological support linked to any machine jargon. To deliver this model, a modeling language is required. As an illustration, IDL (Interface Description Language) is a lightweight modeling language. This textual language is programming language-free. It simply allows the possibility of precisely describing the provided interface (offered functionalities) of a *Currency*

component (service) without referring to constructs of COBOL, C++, Java, Smalltalk or whatever. If we compare IDL and Unified Modeling Language™ (UML), the former is fairly close to low computing layers (middleware), while UML is closer to humans (requirements' engineering, analysis, design, etc.).

So, IDL provides us with the opportunity to have a conceptual viewpoint on what a general-purpose *Currency* service should actually be.

The core expression of the *Currency* service in IDL lays down three business objects (or values) and five interfaces with the following syntax and style:

```
value Currency;
value Money;
value ExchangeRate;
interface StateIdManager;
interface CurrencyBook;
interface ExchangeRateManager;
interface MoneyCalculator;
interface MoneyFormatter;
```

Figure 6.1 is an overview of these business concepts, their unidirectional dependencies as well. The *CosObject Identity*: *IdentifiableObject* is a facility, which is abstracted from a third-party middleware standard named Common Object Request Broker Architecture[2] (CORBA). Otherwise:

> "The CurrencyBook maintains a group of currencies. It is used by the MoneyFormatter to retrieve the currency symbol and by the MoneyCalculator to retrieve the base currency when converting to base currency.

2 In spite of its "old age", CORBA implementations (that in Java especially) are often used as a backbone of Java EE application servers.

> The Exchange Rate Manager maintains exchange rates. It is used by the Money Calculator to retrieve an Exchange Rate to exchange Money.
>
> The Money Calculator is a utility used for performing money arithmetic. It supports a standard set of operations for arithmetic calculations and additional state operations to support rounding rules, precision settings, and conversion rules. These state settings are saved on a per-client basis. The Money Calculator uses the Currency Book to retrieve the base currency and the Exchange Rate Manager to retrieve an appropriate Exchange Rate. The Money Calculator takes Money as parameters.
>
> There is a Money Formatter class utility used for parsing and formatting money into strings. The formatter is dependent upon state settings and therefore the identifier is used for all operations to identify the application client. The Money Formatter takes Money as parameters. The Money Formatter uses Currency to retrieve the symbol." [OMG 00]

In fact, the complete IDL specification in [OMG 00] has the responsibility to make the prior explanations in natural language only complementary, even optional. Figure 6.1 is also a redundant (graphical) model to contribute to the characterization a general-purpose *Currency* service. Ultimately, what is sought is a consistent and complete expression (in IDL) that is independent of technologies.

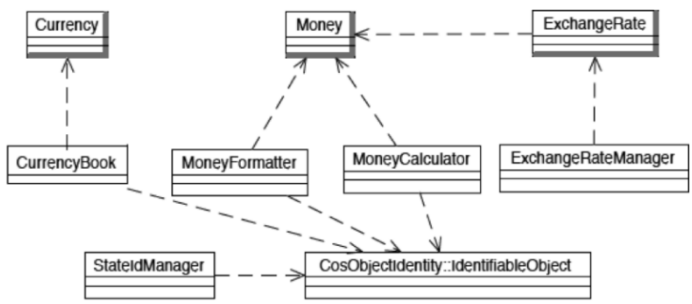


Figure 6.1. *IDL core specification (model) of a general-purpose Currency service from [OMG 00]*

In this logic, the *Currency* service model includes, for instance, the necessity of rounding capabilities through a dedicated type:

```
enum RoundingType { ROUND_DOWN, ROUND_UP,
ROUND_FLOOR, ROUND_CEILING, DONT_ROUND };
```

Rounding types are used by the *MoneyCalculator* interface. For example, this interface exposes two functionalities using this type as follows:

```
RoundingType getRounding(in
CosObjectIdentity::IdentifiableObject stateIdentifier) raises
(FbcException);
void setRounding(in CosObjectIdentity::IdentifiableObject
stateIdentifier, in RoundingType roundingFlag) raises
(FbcException);
```

So, we obtain a vision of what a "consensual" *Currency* service should be. Such a consensus is readily reached with the help of a specification instead of a specific implementation like the four code samples Chapter 5. As a result, we may reason about this model, for instance:

– Having this model as a discussion and exchange basis between software engineers and the software's clients. Typically, using a *Currency* service in a currency trading application requires high precision in conversions while using it in a smartphone application for tourists changes the deal. Does this IDL specification have the ability to support this functional variation?

– This model may be a benchmark to compare and evaluate preimplemented services as the samples in Chapter 5. For example, do these service implementations offer enough capabilities and functionalities with respect to the model playing the role of referential?

– All or parts of this model can be a contract between requirements' engineers and developers in the case of a homemade implementation. By definition, a specification is a guide for implementation.

Models require significant intellectual investments (e.g. learning IDL) and efforts (building and/or understanding models) from software engineers. This can be the same for average users when software engineers want to validate implementation choices against users' requirements formally represented in models.

In the beginning of this chapter, we underlined the temptation to quickly make ideas live in software prototypes instead of reasoning on contemplative models. We mean, models may underlie a negative feeling in the sense that MDD is not always felt as a source of productivity in software development processes. This is both true and false.

As for "true", MDD has probably been slowed down by its insufficient maturity, in tools especially. MDD projects with failures occured in the 2000s, but today's acquired experience has completely changed the deal in the 2010s.

As for "false", the computerized nature of models is their transformation capability. Practically, the IDL specification above is not directly executable and thus cannot deliver the expected service at runtime in terms of ready-to-use converted numbers in varied currencies. However, this specification is still a computerized object in the sense that it may be transformed into COBOL, C++, Java, Smalltalk, etc. code. Concretely, for each programming language, IDL is equipped with a set mapping rules in order to translate IDL text to code. Taking the case of Java, the IDL2Java tool acts as a simple model transformation engine. This tool derives an IDL specification (model) into Java code. However, only stubs may be generated. The way programmers have to power interfaces of the *Currency* service is left open. The IDL *Currency* service specification is thus a partial image (again, an abstraction) of what is intended to be running later once the "powering" code finished.

Anecdotally, we may notice that this specification may be derived from COBOL. This is simply because an IDL-to-COBOL standard exists [OMG 99]. Having used it would surely lead to proper COBOL with, later on, meaningless modernization. By contrast, the existing legacy COBOL has not been produced by means of models in general

and IDL in particular. So, an interesting idea behind the move from IDL to Java, IDL to COBOL, etc., is the possibility of taking a step back. Having at one's disposal an IDL specification allows the regeneration to a new runtime target. Model transformation in particular and MDD in general boost a kind of healthy uniformity in the way runtime software artifacts are produced. Do not forget that COBOL has favored the opposite: an exponential heterogeneity of code patterns, styles and thus samples.

6.5. Model-based productivity

Regarding the *Currency* service look and feel in IDL, we observed that the specification is endowed with numerous comments in natural language. As already underlined, a model is not a panacea. It is only useful in special conditions. Expressing ideas and objects in the mother language remains a solid source of riches, subtleties. Unfortunately, since computers do not yet understand natural language, intermediate languages are required. In this line of reasoning, the IDL *Currency* service specification plays the role of a contract, an agreement, i.e. a set of prescriptive clauses such as those found in any legal document. The other advantage is the transformability of the model. Without seamless gateways to runtime environments, models are only documentations (the notion of "contemplative models" above). In this latter case, their intrinsic value is effectively debatable for timely software development. Again, beyond transformation, the executable nature of models is a very topical issue to be addressed, a springboard for model-based productivity.

6.6. Openness through standards

Models in their very deep nature promote openness, opposed to lock-in in particular. Since models are expressed in languages that are neither programming languages nor natural languages, MDD can be viewed as a sound trade-off. In such a context, opting for a modeling language is a major challenge: IDL, UML, Business Process Model and Notation (BPMN), Web Services Business Process Execution Language (WS-BPEL), etc. The great majority of them are normative.

For instance, UML and its underlying XML declination are both ISO and Object Management Group (OMG) standards. As a metaphor, models may, more easily, cross the forthcoming software innovation decades compared to runtime technologies in general. In other words, software artifacts in the form of models are subject to stronger transformability compared to their incarnation in effective operational technologies. This is just a question of degree.

Openness is in particular achieved through XML as an agreed universal metalanguage and its nearly infinite processing capabilities. The fact that the quasi-totality of the existing modeling languages is based on the XML DTD (Document Type Definition) mechanism is a guarantee of long-term sustainability. For example, XML Model Interchange (XMI) is the XML DTD for UML. Any UML discourse (model) is an XML file, which syntactically conforms to XMI rules. This corresponds to the way UML language pieces may create UML sentences and thus models.

In theory, any software artifact can be described in an XML declination (DTD) and transformable into another declination. This transformation is easily and straightforwardly programmable, but it is only syntactical. In conjunction with this, we sketch below the principle of model transformation with semantic issues: how “senses” in models (e.g. an abstract description of a business service like *Currency*) may lead, by means of a transformation, to a realization of this service in a target technology?

Reverting back to legacy COBOL, what would be the situation of a company having at its disposal its entire information system(s) in a consolidated XML form? For example, this would correspond to the (dreamed of) availability of a consistent and complete description of its business data in XMI (the UML model), independently of data dispersion and/or replication, hard-wired COBOL formats in numerous weird scattered files (the operational realization of this UML model). This availability is nothing else than the possibility of generating (by model transformation) a new implementation toward a newer technology. Honestly speaking, companies do not have models (or documentations) of legacy systems because the energy required to

maintain and synchronize systems in production and their images (models, documentations, etc.) is equivalent to that of the Big Bang.

6.6.1. *Model-Driven Architecture (MDA)*

What is proposed by means of MDD is the prioritization of models as first-class and central ways of expressing software artifacts at large (data types, programs, components, services, architectures, etc.). Runtime incarnations of these are results of transformations only.

Applying MDD in an orthodox way views maintenance at the model level only. The engineering cycle, ideally, is such that changes in models are passed on to the runtime material generated from models in forward engineering activities. In the past, many MDD frameworks failed in trying to keep models and their runtime images synchronous. Direct modifications in the code break synchronicity.

The idealistic vision of MDD is embodied in another standard named Model-Driven Architecture (MDA). In this vision, code no longer exists. In other words, modeling is coding whether models are executable or not.

MDA is the weaving of Platform-Independent Models (PIMs or business-related models above) with Platform Description Models (PDMs) to produce Platform-Specific Models (PSMs). PDMs and PSMs raise contradictions: they are “models”, but they are not totally technology-free. No matter this peculiarity, PIMs are the more precious commodity. PIMs neglect technology details to the benefit of business information. MDD processes PIMs tinted with PDM elements to compute PSMs by transformation. These processes are discrete due to the fact that many intermediate phases are required to derive PIMs into runtime images.

Chapter 7 shows that PIMs may also be computed within transformations of legacy matter, namely COBOL code, again using other standards: Abstract Syntax Tree Metamodel (ASTM) and Knowledge Discovery Metamodel (KDM). Figure 6.2 sums up this vision: PDM elements incorporated into PIMs lead to PSMs: these

reflect technological choices with higher and higher precision ("Platform description model dimension" in Figure 6.2).

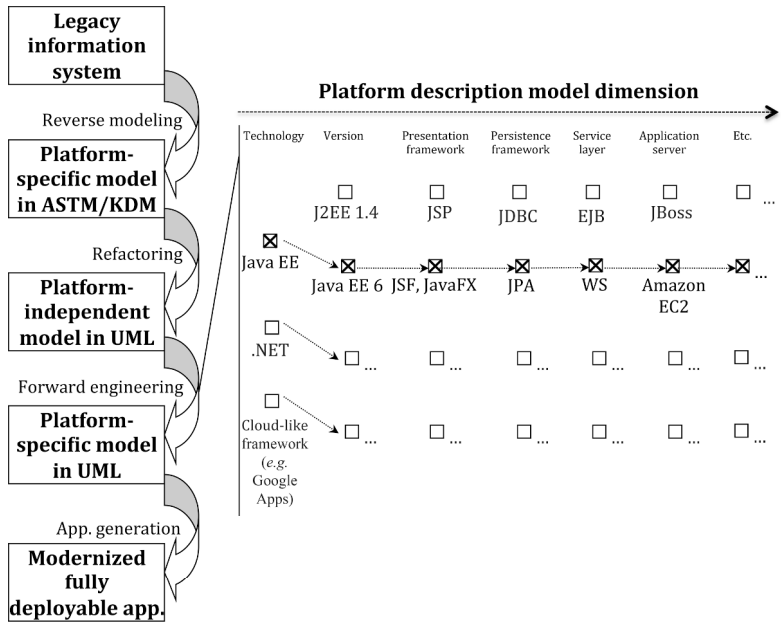


Figure 6.2. *MDA illustration with upstream reverse engineering*

To sum up, MDA may include both reverse and forward engineering through models as sole computerized objects.

6.7. Models and people

Any sincere discussion on MDD leads us to think about the intimate relationship between people and models, Love? Hate? What else? Model experts and skilled practitioners know that MDD in organizations is a cultural shock. Beyond models, technology take-up is not a natural state of mind. Some people prefer closed worlds, repetitions and comfort in general. Investments to get technologies under control are important, but fruitful. The introduction of MDD is

nothing other than a new cycle of intellectual and technical investments without, *a priori*, assurance on returns.

So, the black side of "the MDD force" may be soften with the following arguments:

– From the prehistory of computing, IT guys have been modelers. Everybody is happy to use "advanced" programming languages that prevent the use of bits or bytes when programming machines. Assembler was the first means to abstract bits and bytes. Later, more abstract languages emerged. In short, coding is modeling, programming languages are modeling languages and, finally, code is an operational model. Therefore, switching to models is not climbing Everest.

– Complexity in software is often the result of unjustified considerable volumes of data, data formats and supports, code, programs, etc. This is the dichotomy between accidental complexity and intrinsic complexity. The former can be avoided in more disciplined software development processes. Of course, MDD proposes such a discipline with more or less codified approaches like MDA. Accordingly, a cultural and intellectual adaptation is mandatory; this does not imply losing landmarks.

– Dealing graphically with software is not new. Graphical formalisms have invaded the software world with analysis and design methods from the 1970s especially: Information Engineering by James Martin, Structured Method by Ed Yourdon, etc. Even if modeling may also be textual (IDL above), modeling is mostly the drawing of software artifacts (see again the models from Figures 4.3 to 4.6), provided that drawing constructs obey composition rules: the grammar of the (graphical) modeling language. Putting aside graphics, and therefore models, is nonsense in modern computing. Models reinforce not only graphical approaches of software but also offer the automated processing of graphical software artifacts, the true novelty.

In short, in our opinion, modern computing without models is science fiction. Proliferation of technologies is natural; it is a source of progress and renewal. This proliferation is above all beneficial

provided that extant software is under control. Models are serious candidates for this control.

Criticisms from MDD detractors are of course acceptable. The best reaction is to transform Sir Winston Churchill's famous maxim on democracy: "MDD is the worst form of software development, except for all those other forms that have been tried from time to time."

6.8. Metamodeling

Over the years, in many sciences, metamodeling was the overlapping zone between a given science and philosophy. In MDD, metamodeling has become true engineering through the elaboration of metamodels and Domain-Specific Modeling Languages (DSMLs), in specialized tools especially. The Eclipse Modeling Framework or EMF [STE 08] has made metamodeling popular due to its Ecore metamodeling language.

Metamodeling plays a central role in COBOL software modernization. Metamodels and DSMLs, being homemade or standards like KDM, enable the reconstitution of COBOL matter as meaningful processable models.

6.8.1. *Metamodeling, put simply*

Metamodeling is the circular application of modeling. For example, "Currency" as a business concept is a model of US $, £, €, ¥, etc. In the IDL specification above, "Currency" is an instance of "Value", i.e. "Value" is the model (more precisely, the type of) of "Currency". By transitivity, "Value" is the metamodel of US $, £, €, ¥, etc. Circularly, what is the model of "Value"? IDL does not answer this question. In contrast, UML owns a root element named *Class* as follows (Figure 6.3):

– *Class* is the model of itself (M3 level).

– All UML elements are direct (M2 level) or indirect (M1 and M0 levels) instances of *Class*.

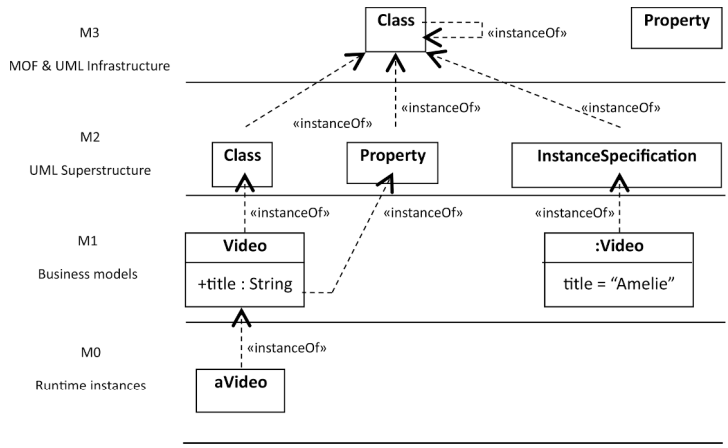


Figure 6.3. *UML metamodeling kernel*

Metamodeling is the expression and recording of metadata to be later accessible and processable for discovering information. A user guide is an informal metamodel in the sense that it may depict a data structure while this structure has only been coded to instantiate variables conforming to it.

Reflection, introspection, intercession, etc. are other words which are very close to metamodeling. Technically, in MDD, without metamodeling, model transformation makes no sense. Figure 6.4 is a metaphoric overview of metamodeling as a pyramid.

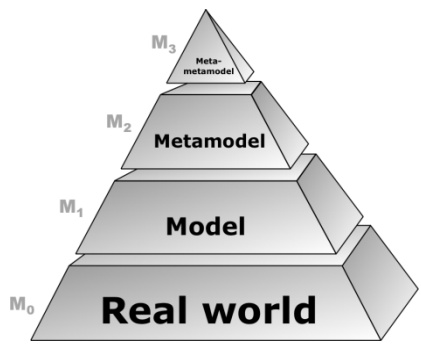


Figure 6.4. *The metamodeling pyramid*

6.9. Model transformation

A model transformation is an algorithm that expresses how instances of metatypes in a metamodel are mapped and moved to "something equivalent" in another metamodel. Without the availability of source and target metamodels, transformation algorithms cannot actually be described. When implemented, transformations are classical programs even if model transformation languages and programs mostly cope with metaelements. In EMF/Ecore, Java is an appropriate transformation language because EMF/Ecore provides metaelements in Java in a transparent way.

6.10. Model transformation by example

We briefly mentioned the fact that IDL offers mapping rules to COBOL. For example, IDL interfaces are mapped to an opaque pointer type in COBOL, as illustrated in Figure 6.5.

```
// IDL:
interface MoneyCalculator {
...
};
// COBOL:
01 MoneyCalculator POINTER.
```

Figure 6.5. *Simplistic metamodeling/model transformation approach in the IDL-to-COBOL fashion*

In Figure 6.5, on the COBOL side, *MoneyCalculator* is an instance of *POINTER* (which is itself instance of COBOL type). *POINTER* is thus the model (a.k.a. type) of *MoneyCalculator*. By deduction, *POINTER* is also the metamodel of any running instance (or concretization at runtime: interface + COBOL implementation details and peculiarities) of *MoneyCalculator*.

In short, we invent a DSML on the top of COBOL in which *POINTER* is a word of this language. Concomitantly, *Interface* is a word of IDL. Put simply, the principle of model transformation is just translation: *Interface* in IDL must be translated into *POINTER* in COBOL. The complication of model transformation code in MDD relies on grammatical constructions of the source language that aim at being translated into semantically equivalent constructions in the target language. By definition, metamodels fix grammars.

6.11. From contemplative to executable models

There are three nested purposes that can be attributed to models:

1) The inner purpose is documentation. Models are informal specifications that express software systems better than code. As abstractions, they naturally highlight key properties to the detriment of meaningless details. They are boosters for brainstorming (e.g. requirement elicitation or refutation), idea communication and thus ideal supports for team-based software development in general.

2) An encompassing purpose is application fabrication, basically code generation from models. In common practice, only model templates can be generated since models do not comprise all application details. While this surely helps, there is a significant trend in industry to expand the generated code with the necessary details (Figure 6.6). As a result, models and code are desynchronized because people tend to let models fall by the wayside: too much energy is required to maintain both (often sizeable) models and code bases as synchronized[3]. Cases (1) and (2) mimic contemplative models (i.e. informative but incomplete models for code generation in particular). Case (1) is also a perfect illustration of code generation from IDL. This is also true for broad-spectrum modeling languages, of course UML, but BPMN

3 Practice especially shows that some modeling tools and IDEs do not support this synchronization in a satisfactory way.

(organizations' functioning modeling). BPMN addresses modeling issues at a macroscopic level and thus beyond software, while UML is devoted to software.

3) The outer purpose is model execution. Execution cannot actually rely on informality compared to models as documentation. Models must then no longer be contemplative, but executable. There is an operational semantics (constraints, rules, exceptions, etc.), which accurately tells us how to move from one model state to another in a discrete way.

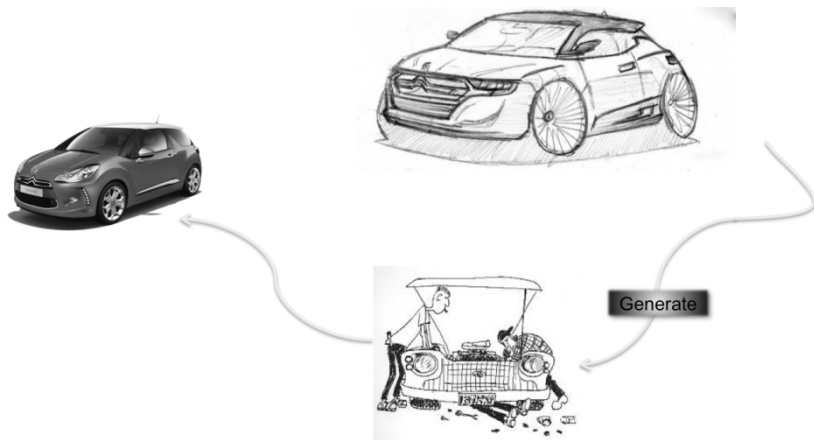


Figure 6.6. *What happens in MDD when generation is incomplete (pictures are taken from autoautomobiles.narod.ru)*

6.12. Model execution in action

Endogenous transformations are the computation of a target model from a source model, both conforming to the same metamodel. Exogenous transformations are the remaining cases (e.g. IDL-to-COBOL). In this spirit, model executability (the potential to be executed) may be based on endogenous transformations. Execution steps are then viewed as transformation steps like a processor tick, which executes machine statements at each tick.

At design time, model execution is typically simulation of model evolution. For example, in Figure 6.7, cardinalities in a UML class diagram pose evolution constraints on an executed object diagram conforming to the class diagram; it is an instance of the class diagram. Execution is useful to check a model as something well formed. Large class models may in effect have contradictory constraints that can be detected by simulation. Execution can also be used for validation against requirements. Requirement emitters may observe live execution (animation) to better understand what implies models that they build.

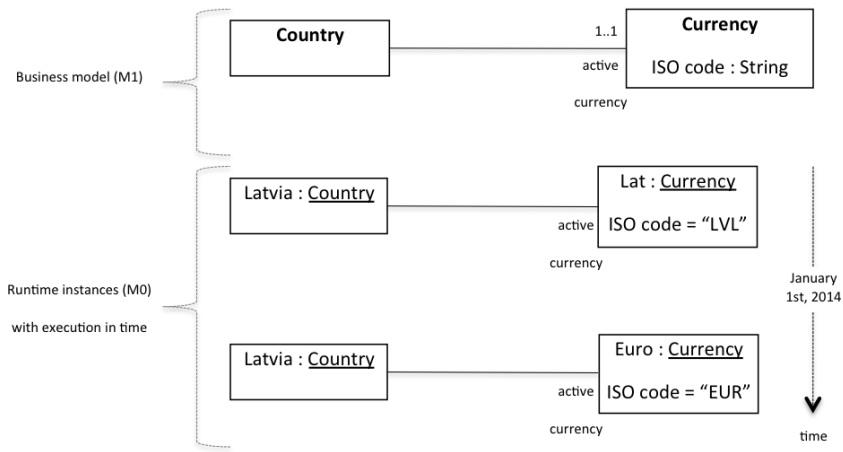


Figure 6.7. *Class diagram (top) with linked object diagram execution (bottom)*

At runtime, model execution is the fact that the transformation engine is similar to a virtual machine (an engine). Model constructs are interpretable in real time, similarly to Java bytecode processed by a Java Virtual Machine (JVM).

Modeling languages are not directly executable apart from operating some amendments in the language's structure and semantics. For example, Riehle *et al.* in [RIE 01] describe a UML virtual machine, to make UML executable. This approach is

proprietary. To address this issue at a larger scale, executable languages like Semantics of a Foundational Subset for Executable UML Models (FUMLs) are released by standardization bodies (OMG for FUML). State Chart XML (SCXML), a W3C standard, is another illustration.

Beyond this, the support for metamodeling and model transformation in an IDE like Eclipse with its EMF/Ecore component has made EMF-based modeling languages somehow executable.

The remainder of this book demonstrates why model execution is becoming a newly higher key concern of MDD. The core goal of modernization is among many other things to reveal the dynamics of the business logic: functions, control and data flows. The business logic is especially split into a declarative form (mapping to classes, relationships, OCL constraints, etc., in UML) and an imperative form to represent this dynamics. So, UML Activity Diagrams or other kinds of diagrams (Sequence, Collaboration or State Machine Diagrams) benefit from having execution properties to really allow us to have the opportunity and power to free up the overall legacy business logic. Beyond this, these dynamic representations are the trustworthy images of application behaviors to be implemented at forward engineering time.

6.13. Toward Domain-Specific Modeling Languages (DSMLs)

To overtake the problem of lacunas in existing modeling languages, there is a possibility of either creating new languages or extending one. The latter approach is supported in UML by creating UML profiles through the stereotyping mechanism.

In Figure 6.8, we may both observe the profile design and its application. The former is an engineering activity totally dissociated from the software development course. The latter allows the marking of elements when building business models. In the example, a *Currency* component is marked with the *Session bean* stereotype having the *Stateless* tagged value. The single interest of these marks (posed stereotypes and tagged values) is their processing in model transformation to guide, for instance, code generation. In other words,

the model at the bottom of Figure 6.8 is likely the source of the code in section 5.1 (Chapter 5).

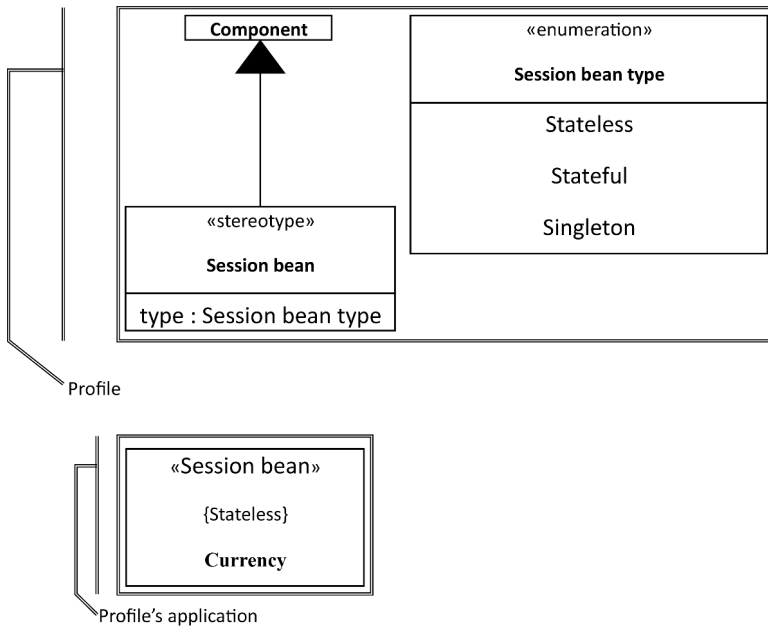


Figure 6.8. *Example of profile (top) and its application (bottom)*

Creating new languages in the DSML spirit may appear appealing, but this hides the risk of uncontrolled proliferation: the syndrome of one-problem-requires-one-language. Most of the time, DSMLs are rooted from the core of UML called the UML Infrastructure or the MOF (Meta Object Facility). Of course, EMF/Ecore is the dreamed of environment to either invent DSMLs or build profiles since EMF/Ecore is reputed to implement the MOF.

For example, Figure 6.9 shows a small piece of KDM. KDM is a DSML whose domain is legacy systems and more precisely the reverse engineering of legacy systems. In Figure 6.9, constituting

metatypes of KDM are introduced by inheritance (white triangles). Because KDM is on the top of the MOF, *ComputationalObject* is itself linked by inheritance to metatypes belonging to the MOF (not illustrated in Figure 6.9). Instead of profiles that use UML extension relationships (Figure 6.8, black plain triangle), DSMLs use inheritance and more generally have first-class metatypes as their own content.

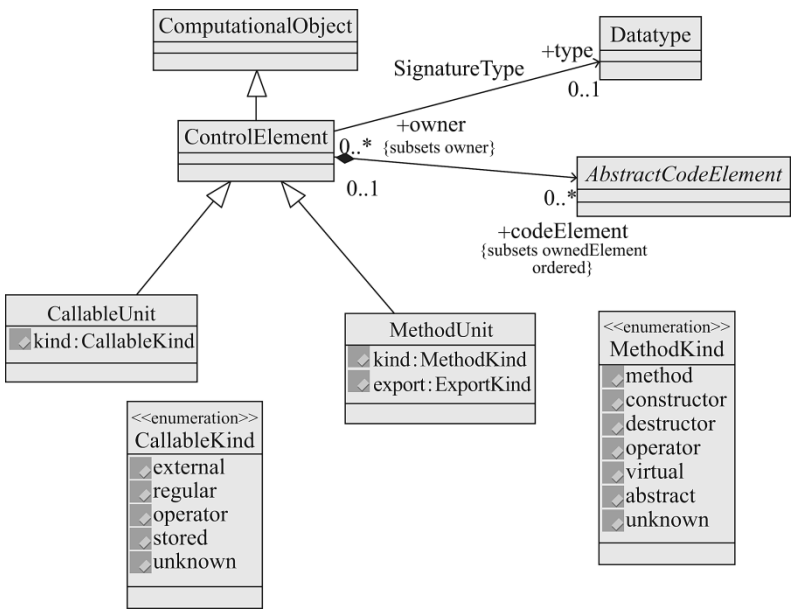


Figure 6.9. *KDM as a UML-rooted DSML*

In summary, few languages have no UML roots. However, "profile" versus "DSML" is a permanent embarrassing question. They are different in the sense that a profile does not introduce new language constructs through new metatypes, metarelationships and constraints on these. In contrast, designing a DSML is a more ambitious task to produce something consistent and complete. Moreover, due to the universal nature of UML a mapping between the

DSML and UML is often an obligation that causes difficulties. It is like a translator: the translation of a sentence in one language rarely keeps the same shape in another, thus leading, for DSML, to many semantic juggling.

Another manner of distinguishing a profile from a DSML is simply the fact that a profile is a lightweight extension of UML while a DSML has the ambition, despite having UML roots, to become a self-contained modeling language. This is the case for KDM.

As a summary, a profile has poorer semantics and thus less expression power outside the scope of UML, but it is easily processable due to the widespread nature of UML. A DSML is in essence close to a domain and its concepts. In the world of modernization, we might imagine the creation of a DSML for each legacy technology (COBOL ANSI), each variant (COBOL dialect), each version, etc.

6.14. Conclusions

MDD is a well-established software development technology, which is recognized as mature (through tool offering, e.g. EMF) and profitable. Expressing software systems with models is a safe way to truly support evolvability. Models may indeed evolve from users' requirements and innovation in general. There is a separation of concerns with curative maintenance (e.g. bug removal), which has to occur at transformation time. This is also an adequate method for dealing with scalability. Applications are naturally enlarging. This common and natural phenomenon in business often reveals the limits of COBOL. Evolution is brutally blocked; the excessive intertwining of business issues and technological issues does not allow code transformation or intervention anymore.

In MDD, the weaving of PIMs with PDMs is the process of generating the final application in a given technology and associated programming language (Figure 6.2). An unsuspected advantage of

models is the adaptability to future not-yet-released technologies. In a long-term perspective, this is really helpful and embraces the opposite of COBOL. For example, models may be initially bound to a .NET platform with C# implementation before moving to a Java EE server in the Cloud. Business-based evolution is then, in this motion, totally orthogonal.

7

Model-Driven Software Modernization

As stated at the beginning of this book, software modernization at large encompasses several different approaches, but white-box modernization is in our opinion the most fruitful approach to run end-to-end modernization processes that target ambitious software restructuring toward SOA and the cloud. More precisely, as shown in Chapter 4, the powerful idea of componentization would probably eliminate candidate modernization processes, that are not able to restructure old applications in services.

In this context, for a long time, software engineering has offered a toolbox with a wide range of principles for re-engineering: software auditing, code refactoring, decompilation of binary software, etc. More recently, model-driven development (MDD) has taken a stronger positioning in software modernization through the idea of Architecture Driven Modernization (ADM) from the Object Management Group (OMG). ADM is a software modernization framework that benefits from being built on the top of various standards including Unified Modeling Language™ (UML). Modernization vendors may adhere to these standards to break the technical dependency of modernization buyers upon vendors' solutions. This chapter concisely describes this framework before giving a turnkey instantiation of this framework in the remaining chapters of this book.

In a caricatural way, modernization is code-to-code. This is the ultimate reality. However, modernization's clients expect more, in terms of business progress especially. No matter what the source and the target code, the greater concern is the fact that the modernized application has gained quality, beyond technical quality especially.

Consequently, *a modernization method must exist independently of the source and the target technologies in general*, including the source and target programming language. While COBOL-to-Java is the most encountered case, COBOL-to-COBOL is also a possibility. For this latter case, there are plenty of tools that support the transformation of non-maintainable COBOL to maintainable COBOL. A subset of these tools can also address bulk architecture issues provided that the targeted COBOL is surrounded with a kind of Internet-compliant technology. For example, we may imagine cloud-based applications in COBOL for the Windows Azure PaaS. Nonetheless, transforming legacy COBOL applications toward this cloud technology, is as arduous as moving to Java EE or to any other PaaS offer. Again, COBOL software modernization is not a technology-to-technology issue. Technology-to-technology approaches make us blind to business challenges raised by modernization. Indeed, in technology-to-technology approaches, where is the business logic outside its expression in COBOL? The narrow-minded nature of a modernization process precludes having higher recast opportunities. Typically, properly moving to SOA and the cloud cannot result from close processes. What does this mean? What is extracted when mining the source code must only lead to "(…) a representation of the system at a higher level of abstraction (…)" as mentioned in [CHI 90] when characterizing white-box modernization.

Of course, models in the MDD spirit act as this representation. A key advantage of models is their intrinsic independency of providers of modernization solutions. This occurs via standards. In this respect, Abstract Syntax Tree Metamodel (ASTM) and Knowledge Discovery Metamodel (KDM) are two inevitable ADM standards devoted to modernization.

7.1. Reverse and forward engineering are indivisible components of modernization

This chapter is a description of a neutral model-driven modernization approach. “Neutral” above all means that there is no reference to the way (“how”) the proposed approach is implemented in a CASE tool. In addition, we also show that this approach is not a method in the sense that it is not a ready-to-use set of well-established principles and recognized best practices. As already written, ADM is only a software modernization framework. The instantiation of this framework gives rise to an operational method in Chapter 8 of this book.

Within this chapter, we in particular focused on the reverse engineering activity. A postulate is that a rich expression of the legacy application in the form of models is a guarantee for generating the (modernized) companion application.

Concretely, as already written, there is a pivot UML model of the application that is intended to be injected in the forward engineering subprocess of modernization. The core of a neutral modernization method is then the computation of this pivot model from the source code. Being COBOL or something else, we might imagine the representation of the complete legacy application in UML with a focus on business logic; the application is concomitantly expunged from any technical/technological details and features.

Contrary to programming languages, UML was not invented to specifically depict execution flows. From this hypothesis, we might believe in Picasso paintings when looking at the representation of the complete legacy application in UML. As shown in further detail in Chapter 8, this global UML model is segmented into several submodels with formal intelligible relationships and subsequent navigation. Submodels are in particular supported by UML *Class Diagrams* for entities and business objects (the latter are small functionality pieces). Services and their dynamics in the SOA sense are depicted by means of UML *Use Case Diagrams* and *Activity Diagrams*. There is intelligent management of the overall model with added traceability links from the legacy intermediate models (reverse) to the produced intermediate models (forward).

In practice, upstream phases of modernization do not, as shown in Figure 7.1, depend upon UML, but ASTM and KDM. ASTM and KDM are normalized formalisms to initially depict any legacy material in the form of models. Accordingly, obtaining a pivot UML model as a single input of forward engineering, is the transformation of ASTM models into KDM models and then the transformation of the latter ones into the UML ones.

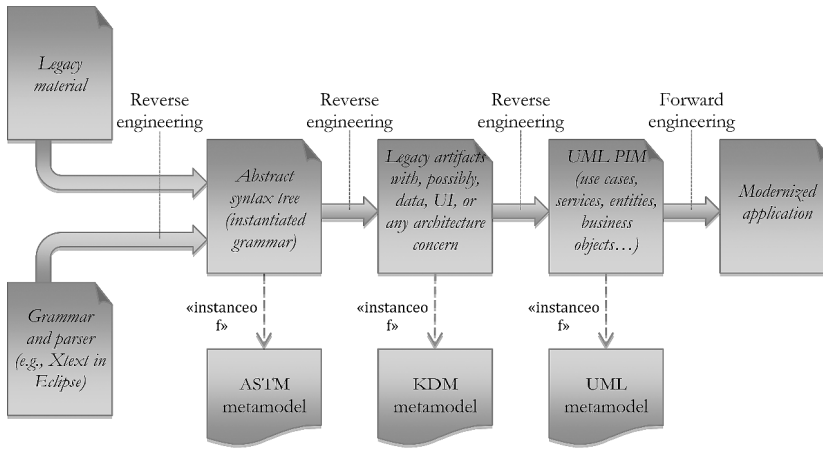


Figure 7.1. *Model-driven modernization as a well codified, neutral and discrete model transformation process*

7.2. Architecture-Driven Modernization (ADM)

ASTM and KDM were invented under the auspices of the ADM initiative at the OMG. One key motivation is the interoperability of reverse engineering tools. As raised before, the problem of narrow-minded modernization processes in tools is the absence of visibility on, and accessibility to, the extracted legacy material. ADM considers the necessity of making this material explicit and immediately treatable by third-party tools. Like UML, as simple XML Document Type Definitions (DTD), ASTM and KDM give us the full opportunity to understand and process models. Again, models are neutrality factors that aim at removing any adherence to any technology.

Ulrich in [ULR 14] especially recalls the three key activities behind software modernization:

– "Assessment: Analysis and exposure of system and business artifacts, architectures, data and process flows, system structure and behavior";

– "Stabilization and standardization: Tasks that structure, rationalize, realign, modularize and otherwise refactor existing systems";

– "Transformation: Extraction of data definitions, data and business rules, along with the reuse of existing system artifacts in the redesign of target architectures".

Thus model-driven modernization is globally the permanent possibility of modifying model content at any stage of the modernization process to put into practice the three core ADM activities above. What is missing in these points is the prevalent possibility of implementing new functional requirements. Although the result of modernization is *a minima* an iso-functional (modernized) application, adding new functionalities beyond extant ones calls for a specific approach. The ability to cope with model content is thus a good means for enhancing requirements. In this context, again, the overall UML pivot model plays the central role at the end of reverse and start of forward.

In the ADM spirit, software modernization is viewed as an upside down MDA-like approach. More precisely, while initial ASTM and KDM models are most likely PSMs, the pivot UML model is the much-desired PIM. *Unraveling the legacy platform features from the ASTM/KDM models to obtain the UML model is therefore the core job of reverse engineering as a first modernization subprocess* (Figure 7.2).

As for MDA, in the forward engineering spirit, it is in contrast the weaving of a PIM and a PDM to obtain a PSM. In practice, from an architectural viewpoint, several inter-related PSMs exist. Most of the time, upstream PSMs are conceptual descriptions of architectures with component (provided/required) interfaces. Downstream PSMs are the same with component implementations coming from a more or less complete generation/representation of code statements. PSMs are

linked to each other in transformation chains. Architectural models may also coexist with deployment models enriched with configuration data for deployment.

In fact, ADM is nothing other than a modernization framework without any instruction manual: there is no elaborate method and particularly no accurate process in the sense that the steps and micro-steps to fully clean up the legacy application to reach the PIM level, are never defined and described.

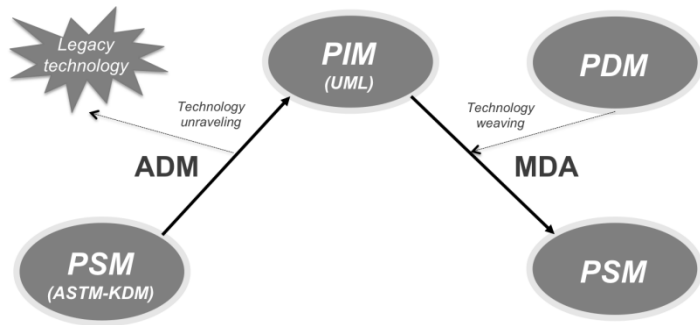


Figure 7.2. *ADM as upside down MDA*

A probably unanticipated consequence of ADM is a better take-up of MDD. MDD penetration in industry has effectively been slowed down by the laborious learning and poor mastering of model fabrication. Instead of building models from scratch, ADM is viewed as a more meaningful way of (re)-developing software with models since the latter are produced in an assisted (even automated) way. However, the expected model is again the UML pivot model, which is by definition based on the UML agreed and thus shared formalism. Nobody really wants the heritage of ASTM and/or KDM models as primers for application redesign. In other words, ASTM and KDM benefit from being hidden at modernization time to lower complexity; people cannot deal with too many modeling languages, UML is enough.

The great challenge about the definition an ADM-compliant modernization method is therefore having a well-formalized reverse engineering workflow with UML only (see Chapters 8 and 9).

In this line of reasoning, Figures 7.1 and 7.2 hide the fact that there are many intermediate ASTM and/or KDM models throughout the reverse engineering chain before creating the "PIM (UML)" oval in Figure 7.3. Theoretically, all intermediate models can be exchanged in ADM-compliant model-driven modernization tools. This is true in terms of format interoperability; this only results from the fact that ASTM and KDM are normalized and have *de facto* implementations in Eclipse Modeling Framework (EMF). Nonetheless, the deep sense of these models, their transformation purpose as well, are only known by each tool since versions of the ASTM/KDM model at mid-term points of the modernization process (see examples in next sections) are "user-defined" in the ADM philosophy. In Figure 7.3, the global model transformation chain is modularized through compact well-isolated transformation blocks, each named $t_{i_{m2m}}$. The legacy technology is especially vanishing in a progressive way when we go up to the "PIM (UML)" oval.

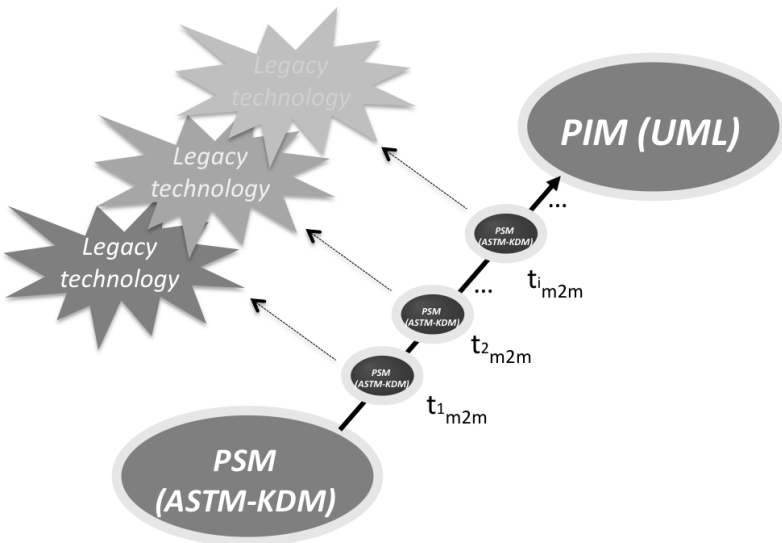


Figure 7.3. *ADM is a discrete set of model-to-model transformations*

In this context, the choice of UML has a pivot language between reverse and forward is not at all a recommendation of ADM.

However, this choice is extremely relevant and critical because UML is really exchangeable by its broad presence in numerous tools. This is not the case of ASTM and KDM at this time even if the ADM task force would like to increase take-up and large-scale use of both ASTM and KDM.

7.3. ASTM and KDM at a glance

From Wikipedia, "Knowledge Discovery Metamodel defines an ontology for the software assets and their relationships for the purpose of performing knowledge discovery of existing code". In Figure 7.4, we also point out that only the *Infrastructure* and *Program Elements* layers of KDM are concerned with code while the *Resource* and *Abstractions* layers address, among others, software architecture issues. Moreover, as stated below, *Abstractions* goes over reverse engineering by dealing with forward engineering.

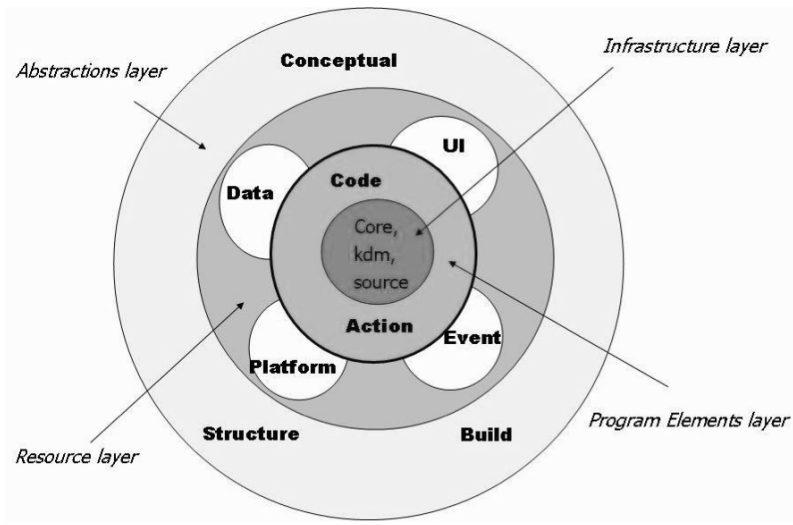


Figure 7.4. *Overview of KDM structuring from OMG documentation*

Even though KDM plays the central role in the representation of software artifacts, it is complemented by ASTM as follows (from OMG ASTM specification): "The Abstract Syntax Tree

Metamodeling (ASTM) and the Knowledge Discovery Metamodeling (KDM) are two complementary modeling specifications developed by the OMG Architecture Driven Modernization Task Force. Their relationship can be clearly understood by recognizing that the KDM establishes a specification for abstract semantic graph models, while the ASTM establishes a specification for abstract syntax tree models. Thus, in contrast to other software representation standards, such as the Knowledge Discovery Metamodel or the Unified Modeling Language, the ASTM supports a direct 1-to-1 mapping of all code-level software language statements into low-level software models". It is also noted that: "ASTM is one of the sources of information for the KDM".

From experience, the articulation between the two is not as clear as claimed by the prior OMG text. Since KDM also operates at the code level (*Infrastructure* and *Program Elements* layers in Figure 7.4), a native overlapping exists between the two.

In practice, ASTM models are the first populated models at legacy code parsing time. Parsing requires a perfect knowledge of the legacy language grammar. However, old languages like BASIC variants (QBASIC, Visual BASIC, etc.), COBOL variants (ANSI, IDS, MINI, ACCU, etc.), RPG from IBM, BAL from BULL, (eclectic) fourth-generation languages, etc. may reasonably be viewed as "scary". For example, such a trivial *IF-THEN-ELSE* control statement may have a weird instantiation in a program[1]:

```
REM Some test expression:
10 IF … GOTO 40
REM 2-digit precision:
20 LET precision = 2
30 GOTO 50
REM 1-digit precision:
40 LET precision = 1
REM 'Currency' subroutine call:
50 GOSUB Currency
```

1 This is QBASIC in which lines are numbered. *GOTO* statements rely on this numbering that avoids textual labels as in COBOL for example.

The only way to catch this code as a model is based on ASTM and a grammar management tool like *Xtext* in the Eclipse IDE.

Syntactically, the code above is an imbroglio of *GOTO* statements finally leading to a call to currency conversion facilities (see approaching COBOL code in section 2.2.3 in Chapter 2). From a semantic viewpoint, there is an evident possibility of modeling this code in an algorithmic formalism that does not depend upon any programming language. The move from ASTM models to KDM models is then a more or less automatic reinterpretation procedure from syntax to semantics.

The legacy programming language grammar is either classical and therefore matches GASTM (Generic ASTM) or it is more or less wobbly. In the latter case, Specialized ASTM (SASTM) is required. While GASTM is an existing metamodel with metatypes like *JumpStatement*, SASTM has to be built by software (reverse) engineers as an extension (mainly through inheritance) of the existing GASTM. As an illustration, the ASTM OMG documentation offers a SASTM metamodel for SQL. The specificity of SQL constructs (primary key, foreign key, constraint, etc.) is typically not covered by GASTM.

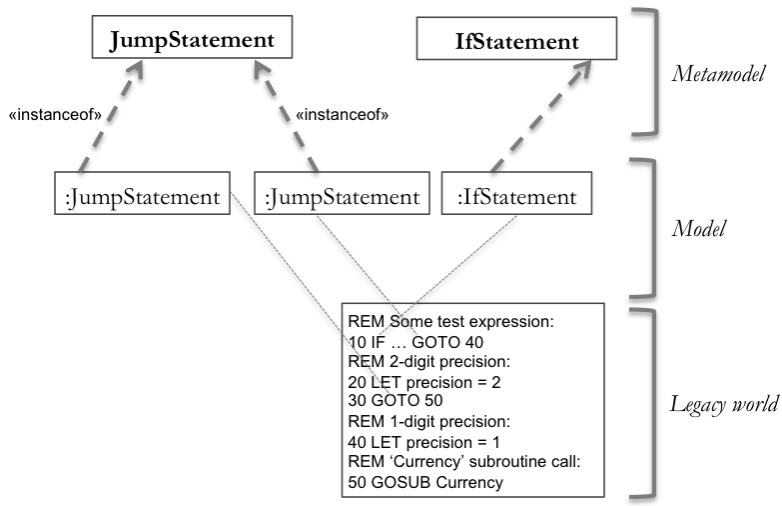


Figure 7.5. *GASTM sample*

Returning to the QBASIC code above, each *GOTO* statement (located at lines 10 and 30) is an instance of *JumpStatement* (Figure 7.5). As for the *IF ...*, it may simply be viewed as an instance of *IfStatement*. Both *JumpStatement* and *IfStatement* are members of GASTM. For concision and thus better comprehension, we do not depict (from the source code) the induced relationships (if any) both at the metamodel and model levels. In other words, the QBASIC code above might lead us to instantiate many ASTM metatypes. We will come back to this issue with KDM.

So, there are no simple means to directly populate KDM models from the legacy material while KDM was initially intended to play this role. As a proof, KDM owns the *Goto* metatype. We may thus build a KDM model similar to that in Figure 7.5 by simply replacing *JumpStatement* objects by *Goto* objects at the model level. However, this hypothetical KDM model has no added value. In fact, ASTM has been released after KDM to solve residual problems. This explains today's articulation between ASTM and KDM, which is as follows:

– The very first capture of the legacy material in the form of models with KDM supposes the development of homemade code parsing tools, which would surely differ from one legacy technology to another. Instead, an ASTM model is just an abstract syntax tree grounded on a common grammar (GASTM) with, possibly, (a lot of or a few) non-common extensions (SASTM). The availability of a COTS grammar management tool like *Xtext* in Eclipse is a robust approach to initiate a modernization process. The transformation of what is extracted by *Xtext* toward ASTM is in particular easy and straightforward.

– ASTM models, by obeying a standard, aim at being interchanged between tools. The absence of ASTM would increase the use of tailored mechanisms with poor interoperability. Despite the existence of *de facto* hands-on "standards", say the *Xtext* tool, ASTM is a better source of openness.

– The ASTM to KDM mapping is not normalized, i.e. it is provider-defined. The creativity of providers of modernization

solutions then relies on their implementation of the move (model transformation) from ASTM to KDM. It is also important to observe that this move is not, in terms of actions and outputs, frozen once and for all. In other words, the intelligence of modernization methods may in particular be measured through their ability to produce rich and coherent KDM models. As for their understandability, it may be low or high, depending upon the sought objective. Put simply, the technical processing of KDM models by any ADM-compliant tool does not imply the possibility of detecting and interpreting the semantics behind model elements and their relationships. To attenuate this, ASTM models are more easily and straightforwardly comprehensible due to the access to their underlying grammar.

– Contrary to ASTM, KDM offers metatypes to deal with, not only code, but data, user interface (UI) or any architecture concern. The advantage and expected role of KDM (compared to ASTM) is the connection of the legacy code vision with any useful orthogonal information on the legacy application. Typically, we must be able with KDM to trace the fact that a legacy element is an indirect instance of *AbstractUIElement* (e.g. an instance of *Screen*) belonging to the KDM *UI* package. Indeed, in the KDM metamodel, *AbstractUIElement* holds an association (*0..** cardinality) with *ActionElement* that belongs to the *Program Elements Layer* package. This approach automatically limits the role and scope of ASTM to the sole expression code-centric legacy artifacts, at modernization startup especially.

7.4. Variations on ASTM

A legacy programming language with *GOTO* (BASIC, COBOL, FORTRAN, C, etc.) cannot be understood and treated as any other that is not equipped with hard-wired jumps. In a more complicated way, we may consider some Smalltalk code as legacy source code. Since test expressions are instances of the *Boolean* Smalltalk type, we may represent a given test ("…" below) by means of the *Boolean* metatype of ASTM (a subtype of *PrimitiveType*).

```
“Some test expression:”
…
“1-digit precision:”
ifTrue: [precision := 1.]
“2-digit precision:”
ifFalse: [precision := 2.]
“‘Currency’ instance method call:”
Currency convert: … and: precision.
```

Going on with Smalltalk, an *IF-THEN-ELSE* control statement is conceptually considered as an instance of the *Message* class of Smalltalk whose receiver is the said test (so, an instance of the *Boolean* Smalltalk type). As a result, in the Smalltalk code above, if really is an instance of the *Message* class. To match the Smalltalk grammar, a *Message* metatype may then be added to an in-house SASTM, leading to the GASTM + SASTM model in Figure 7.6. *Boolean* in Figure 7.6 is that of GASTM. Again, no relationships appear while the Smalltalk source code underlies relationships between *Message* and *Boolean* at the metamodel level, *:Message* and *:Boolean* at the model level as well.

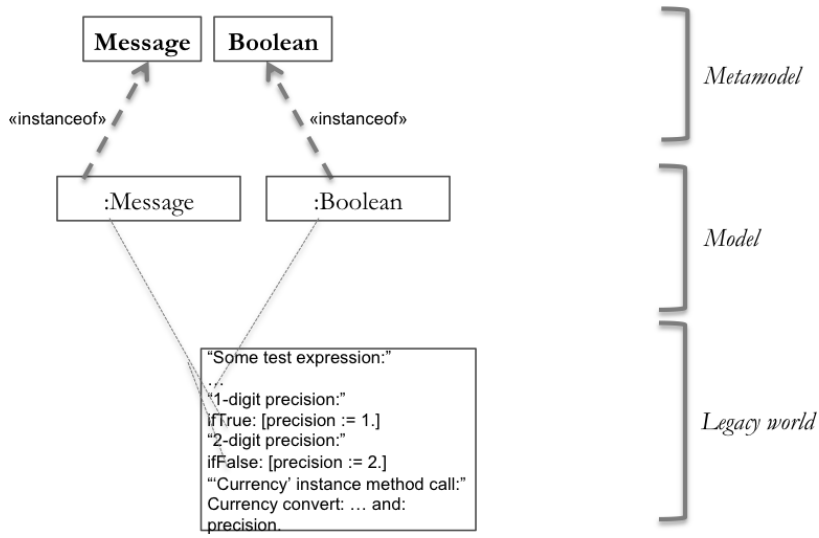


Figure 7.6. *GASTM + SASTM sample*

7.5. From ASTM to KDM

When putting all together, we observe that the COBOL code sample in section 2.3 of Chapter 2, the QBASIC code sample and the Smalltalk code sample are nothing other than an *IF-THEN-ELSE* algorithmic occurrence. Only the routing to the *Currency* conversion functionalities appears, in COBOL, as a temporary program exit (CICS delegation) while QBASIC and Smalltalk perform a local call to, respectively, a subroutine and a class operation.

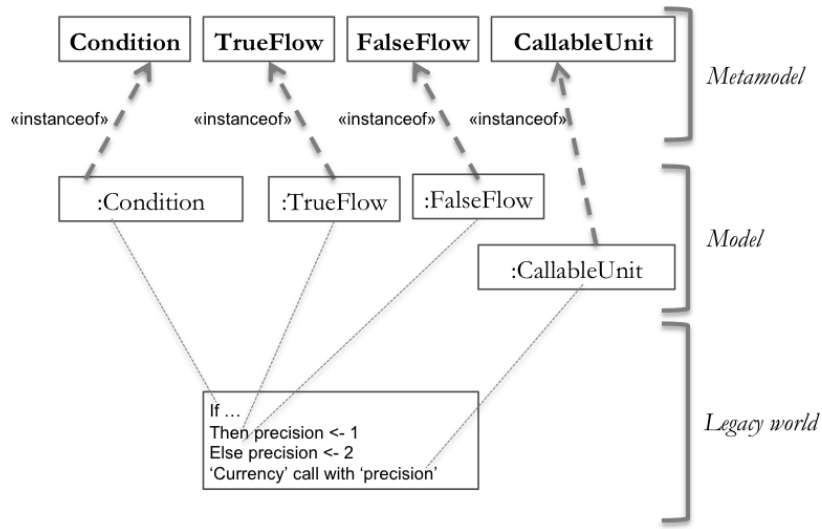


Figure 7.7. *KDM sample*

As a result, Figure 7.7 is a KDM model that removes the unnecessary language details. In other words, this model is a trustworthy representation of the business logic engraved both in the COBOL, QBASIC and Smalltalk code samples.

This KDM is nonetheless incomplete in terms of discovered semantics. The Condition object named… in the code must itself be developed to make the business rule explicit. Furthermore, the use of currency conversion facilities is a "system call" in COBOL (link to CICS middleware platform) while it is local in QBASIC and

Smalltalk. This call respects formal encapsulation principles in Smalltalk while it is grounded on the global visibility of the "precision" variable in QBASIC. All of this information must be incorporated into the reverse engineering chain of models, to maintain as much knowledge as possible before computing the UML pivot model to be used as input of forward engineering.

We experience here the reason why the ADM model transformation process is made up of many micro-steps. The model transformation program suite may in particular benefit from being generic. The best is its expression as a transformation from GASTM to KDM. If SASTM metatypes exist as subtypes of GASTM metatypes, transformation based on the latter also touches the former through polymorphism.

Anyway, the transformation program which moves from GASTM + SASTM models to KDM models can be complex. This encourages us to sparingly build in-house SASTM. Unfortunately, COBOL often calls for specialized metatypes as we did with *Message* for Smalltalk. As an illustration, we may look at this kind of assignment in COBOL (assign to the program variable *c*, found in structure *d*, the value of the program variable *a*, found in structure *b*):

```
MOVE a OF b TO c OF d.
```

Modernizing it leads us to primarily setup a homemade *MOVE* metatype in a SASTM instead of using *Assign* in GASTM. All this is taking place before mapping this code piece to an instance of the *Assign* metatype in KDM. The two existing *Assign* metatypes in both GASTM and KDM cannot capture the deep semantics of this uncommon assignment even if it is well known by COBOL specialists.

7.6. Variations on KDM

KDM is not prescriptive in terms of usage. We mean, the same code sample and, more broadly, the same legacy system may lead to

different models, something, in any event, embarrassing for an industrial (and thus robust and scalable) usage of KDM.

In this spirit, KDM has a modular organization in packages. Each package has, in terms of possible utilization, a more or less high importance depending on the legacy technology, the domain and status of the legacy application, the complexity and volume of the legacy material and so on. Chapter 6 points out some details about the *Infrastructure* and *Program Elements* layers of KDM (Figure 6.9). The *Runtime Resources Layer* and *Conceptual Layer* of KDM in Figure 7.4 offer other packages: *Platform Package*, *Data Package*, *UI Package* and *Event Package* for the former layer and *Structure Package*, *Conceptual Package* and *Build Package* for the latter layer.

As mentioned earlier, reverse and forward engineering are two inseparable pillars of modernization. As a consequence, KDM has been designed so that its *Conceptual Layer* is mostly devoted to forward engineering. More precisely, while the *Structure Package* proposes notions for modeling legacy system architectures, the *Conceptual Package* and *Build Package* turn to the modernized vision of the legacy system.

There are two good reasons to exclude the *Conceptual Layer* of KDM from a suited model-driven modernization method:

1) Modeling architectures of legacy systems might have value when these have to be partly reflected in renewed systems. In practice, this assertion is almost always false, in COBOL specially. We mean, the form of legacy architectures has poor interest because, most of the time, we want to restructure all aspects in a service-based fashion. Do not forget that, for instance, KDM may serve in Java-to-Java cases when, possibility, architectures can go through modernization processes with few changes. These cases are sufficiently “simple”, even meaningless, to eliminate them from this book’s study intention. Modeling architectures is informative and may thus gain insights into legacy system inner workings (e.g. CISC “system call” in COBOL). Nonetheless, we show in section 2.2.3 of Chapter 2 that the detailed analysis and interpretation from the code reveal knowledge on architectural issues, and, more interestingly, the (justified and

displaced) intricacy of these with business logic. Again, our focus on SOA as target architectural paradigm calls for a thorough inventory, which often throws away legacy architectures.

2) The *Conceptual Package* and *Build Package* are direct competitors of UML. As self-contained modernization language, KDM is complete. However, the today's spread of UML encourages and strives us to use it to the detriment of these two KDM packages. Models by their very deep nature favor interoperability. The expression of the modernized system in UML instead of KDM allows a greater independence and thus competition: forward engineering may be performed by more competitor tools, these in particular that do not have reverse engineering facilities.

From this observation, an ADM modernization approach is first and foremost a kind of "KDM decantation" to first identify and then put into practice the appropriate KDM packages, as, for some of them, optional helpers. This remark may frighten readers about the investigation and investment on KDM before running any modernization process. This demonstrates again and again the need for a well-codified fluid method to avoid such oversized efforts.

Figure 7.8 shows another angle for priming modernization activities. In enterprise applications, data are spread out between "presentation" (UI), "persistence" (files, databases) and "service", i.e. computation at large by means of working data in memory for presentation/persistence intermediation. This concise approach allows the definition of the way code macro- and micro-pieces have to be unraveled toward the very first KDM models of an ADM-compliant reverse engineering chain.

From this hypothesis, Figure 7.8 is a metaphor about the way the presentation/persistence/service circle may be enlarged to make the business logic emerge.

In this global logic, from experience, the *Runtime Resources Layer* of KDM with *Platform Package*, *Data Package*, *UI Package* and *Event Package*, is only useful for legacy applications with "good" existing structuring. For example, the KDM *Event Package* is a state

machine-oriented formalism that may be efficient for capturing the interaction of UI components, provided that the *UI Package* serves as a description support of the legacy structuring of these components. This situation is typically the counterexample of COBOL with green character-based inputs/outputs.

In COBOL software modernization, the *Runtime Resources Layer* of KDM is the source of informative and fairly contemplative models in the sense that they store interesting information on the legacy system. However, information in this model has a secondary role compared to the models coming from ASTM and the *Infrastructure* and *Program Elements* layers of KDM, i.e. those devoted to code modeling, analysis, interpretation with inevitable external intervention (legacy people) and final transformation toward UML.

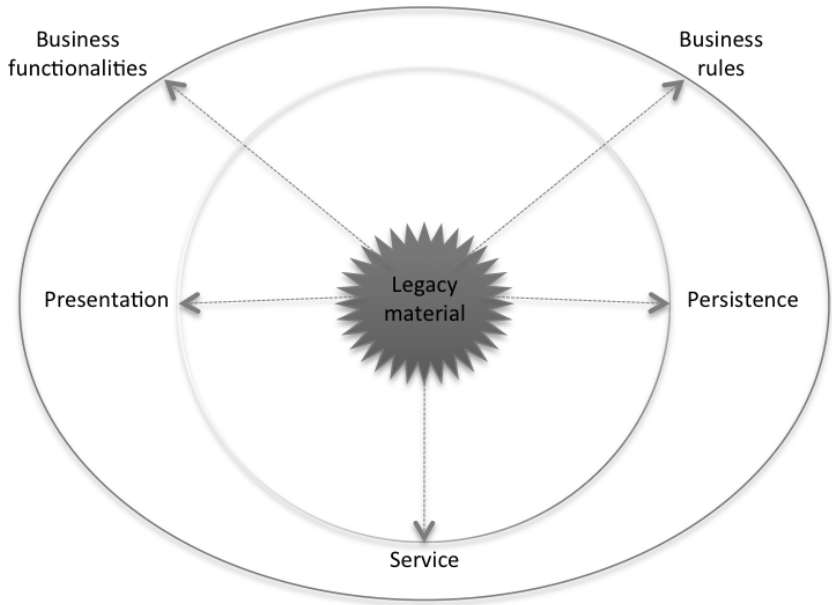


Figure 7.8. *Modernization primarily occurs by separating presentation, persistence and service "corners" in the COBOL "ocean of details"*

7.7. Automation

It is improbable to manually deal with ASTM and KDM models. Thousands, even millions of lines of code are concerned with modernization and thus lead to equivalent sizable models with numerous elements. Beyond this, KDM models aim at containing not only program elements, but also data, UI, architecture, etc. software artifacts.

In this context, ASTM and KDM are provided as open-source "running" metamodels, in Eclipse especially (EMF/Ecore), a sign of confidence and an assurance of portability and sustainability. Moreover, beyond packaged commercial products around ASTM/KDM, environments like MoDisco (www.eclipse.org/MoDisco) offer rich functions to manipulate ASTM/KDM in a more friendly way than the basic support of EMF/Ecore. From experience, MoDisco is only reserved for highly skilled software (re)-engineers. The manipulation of such an environment is both rough and tough. The open nature of this product leads to a lot of adaptation/enhancement before any intensive repetitive usage at a large-scale industrial degree, for COBOL in particular.

As will be shown in Chapter 8, MDD reverse engineering cannot be fully automated. Legacy people's intervention is therefore a key complement of automation in a well-defined modernization method and an associated tool. This supposes a lot of assistance to, for instance, build in-house SASTM. Transformation programs from ASTM to KDM and to KDM to UML cannot also be fixed once and for all due to too many fluctuations and versatilities of legacy technologies within and around COBOL. Chapter 8 not only shows this method but its high customization degree to cope with the heterogeneity of COBOL legacy systems.

7.8. Conclusions

Model-driven software modernization is a set of principles, which derives from the ADM initiative and task force at the OMG. Even though ADM puts forward the ASTM and KDM standards as modernization-specific modeling languages (DSMLs), there is no

prescribed method to carry out ADM end-to-end modernization processes.

As shown in the forthcoming chapter, ADM covers forward engineering in an anecdotal way. This encourages us to consider an ambitious and realistic method that is based on UML as the articulation axis between reverse and forward, instead of KDM. Any modernization method cannot exist outside tools, given the huge volumes of legacy information: code, data, configuration information, etc. To that extent, there is a clear prevalence of UML and a UML-like profile to include execution capabilities in the models that actually have the capability of terminating the reverse engineering phase of modernization: full application generation.

8

Software Modernization Method and Tool

Architecture-driven modernization (ADM) is a stimulating framework for performing model-driven software modernization. However, the availability of an *industrial method* to put ADM into practice is the centerpiece of any common business-oriented language (COBOL) software modernization project.

This chapter first describes such a method in relation to a professional computer-aided software engineering (CASE) tool called BLU AGE. The chapter does not enter into accurate technical details to show ADM at work (the purpose of Chapter 9). Nonetheless, it lists the necessary elements to have well-formed unified modeling language (UML) models such as pivot models between reverse and forward engineering.

Secondly, this chapter discusses such a method in a project management logic. This method in essence makes a peculiar utilization of the knowledge discovery metamodel (KDM) (including the abstract syntax tree metamodel (ASTM)) and UML in an ADM and MDA-compliant style. In effect, the industrial nature of this method imposes scalability because legacy applications are never toy cases. Volumes (code sources, data, etc.) play a crucial role in the sense that model-driven development (MDD) methods and tools may

fail because of them (see remark on MoDisco in Chapter 7). A methodical modernization approach thus calls for:

– reproduction (similar cases are processed in the same manner);

– systematization and phasing of modernization actions in a well-established workflow,

– automation also covering legacy personnel's involvement, management and assistance (key modernization actors and roles, models shared across teams, wizards in tooling for building model templates and populating models, etc.);

– tailoring capabilities when having original, even borderline, cases.

Honestly Speaking, COBOL software modernization is rather a matter of "software archeology" than "noble" software engineering.

8.1. BLU AGE overview

BLU AGE is a software tool suite based on Eclipse and more precisely its Eclipse modeling framework (EMF) constituent. The main purpose of BLU AGE is to provide industrial tooling to drive large-scale modernization projects. These projects are managed in an industrial manner to optimize productivity. Productivity is measured in a number of lines of code (LoCs) modernized per day and per stakeholder (also known as "consultant"). Of course, productivity depends on the type of legacy technology (COBOL or COBOL-like languages, fourth-generation languages such as PowerBuilder, NatStar, etc.). For COBOL-like languages, a consultant can reach more than 1,000 LoCs per day using BLU AGE. This includes the concomitant achievement of numerous project management tasks like testing the transformed LoCs.

The BLU AGE suite is composed of family of three self-contained products:

– BRM (BLU AGE reverse modeling): From a legacy code, it is used to generate a model to be injected in the forward engineering component of BLU AGE or any code generator. BRM provides a set of sub-tools to extract the business logic from the legacy code and to

transform it into a UML model. This model is in essence independent of the target technology;

– BFE (BLU AGE forward engineering): From a UML model, it is used to generate a modernized application by choosing a target technology: EJB, Spring, .NET, cloud platforms, etc. The model used in the transformation is independent of the target technology; it only represents the business logic. With BFE, users can select the transformations to apply in order to obtain an application conforming to the desired target architecture. In modernization situations, the model to be used for these transformations is extracted from BRM;

– BDM (BLU AGE database modernization): It is used to modernize databases in a modernization project. Within legacy systems, databases mostly exist through flat files. Instead of having relational structures, data are often badly organized in these files through possible hierarchies. With BDM, users are able to first modernize data schemes by defining set-based relationships to definitely compensate the absence of rationale data organization. Next, by introducing modern data types (to put aside character-oriented COBOL data types), BDM produces migration scripts to migrate data from legacy files to relational databases.

The proposed approach is based on model transformations and, more generally, the MDA "way of life". What does it mean? If we consider everything as a model (including legacy and modern code), then we are able to apply transformations in order to compute new models until we finalize the UML one: that abstracting the modern code, including application artifacts such as Web pages, configuration files, etc. Transformations are just assisted actions by means of BLU AGE editors. In exceptional circumstances, transformations may be programmed using Java in EMF.

BLU AGE implements the MDA approach to perform automated transformations from a model to move to the final code: "platform-independent model (PIM) -> platform-specific model (PSM) -> code". BLU AGE extends this approach for the reverse part to compute a model from the legacy code: "Code -> PSM -> PIM". This is described in a conceptual manner in Figures 7.1 and 7.2.

In BLU AGE, the PIM as a UML model is always compliant to the UML metamodel. The PSM in the forward direction is a model compliant to the BLU AGE metamodel (a UML subset) based on EMF (Ecore format). The PSM in the reverse direction is a model compliant to the KDM metamodel. Figure 8.1 is an overview of this principle.

Transformations are always automated even if users may have to provide transformation information. In fact, users have in particular to select what is to be transformed and how it is from a small set of choices. Such choices are strongly guided by the tool and its incorporated method.

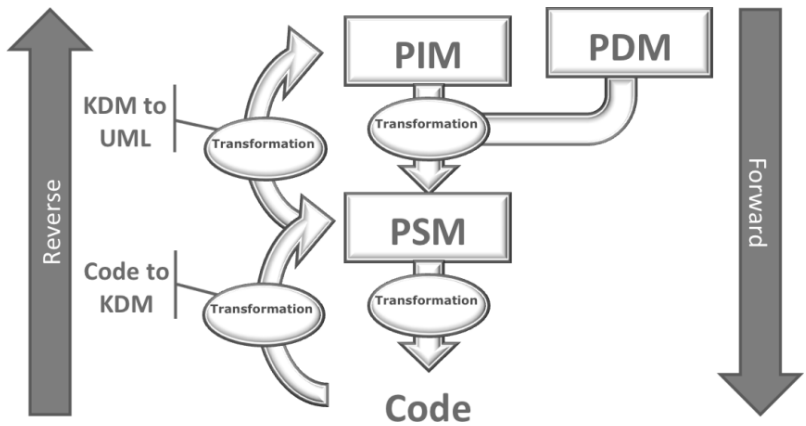


Figure 8.1. *Model-driven modernization with BLU AGE*

8.2. The toolbox

BLU AGE is a CASE tool based on a non-intrusive and easy-to-learn technology. All deliverables coming from BLU AGE can be maintained with or without BLU AGE. The code generated from UML models is not dependent upon any BLU AGE runtime or any third-party software library. Using MDA in general and BLU AGE in particular makes users free from technical complexity, i.e. they do not require on in-depth knowledge of enterprise middleware platforms

such as EJB, Spring, .NET, etc. Roughly speaking, users do not have to be experts in target technologies; they only focus on models.

The tooling is associated with a method in case of modernization. The method aims at assisting the production of the necessary UML models from which the modern code is later generated. Nonetheless, the UML models may be created from scratch outside any modernization concern. BLU AGE helps us to generate an application by reversing any legacy code. Extracted from the legacy code, the business logic has to be first cleansed from legacy technical (obsolete) details and next properly expressed in UML models.

In parallel, legacy *user interfaces* (UIs), COBOL "screens" in general, are reshaped and stored as Web pages. In fact, these are HTML mockups (UI prototypes). The Web pages are designed (with assistance) from the legacy COBOL "green screens" in case of modernization. They can also be designed from scratch. UML model elements are referenced as HTML elements in mockups and vice versa: mockups widget names are manipulated in UML models to formalize the application's interaction. Typically, BLU AGE uses *UML activity diagrams* to express the kinematics of Web pages, i.e. how we move from one page to another, what action is launched when entering or exiting an activity once the application's mouse is clicked, etc.

More generally, in the case of modernization, there is a seamless workflow ("reverse" then "forward") with the following outputs:

– the production of an initial model including a modernized representation of the legacy data;

– the production of HTML mockups from legacy inputs/outputs;

– the production of possible information (comments) to better understand the application to be modernized.

Later on, the modeling of the legacy services and presentation layer is ruled by the following supports:

– an automatic process based on code pattern definition and "transmodeling" (also see sections 8.2.2.4. and 8.4.2.2);

– a finalization process using any UML-compliant modeling tool to adapt the reversed code to UML models obeying to strict rules. This is a coercive format above called “BLU AGE metamodel” (also known as “BLU AGE metalanguage”).

8.2.1. *BLU AGE format required for forward engineering*

The BLU AGE PIM model is a fully compliant UML model with limitations. These limitations are healthy in order to obtain a simpler language than the overall UML. The code generator (including a syntactical checker) works from this UML subset. The underlying format is composed of:

– entities, business objects and value objects defined as UML classes in class diagrams. For example, the BLU AGE metalanguage imposes that business objects own elementary operations (data accesses and simple computations mainly), while entities only have fields. Business objects inherit from entities as exemplified in Figure 8.2 (top left-hand side);

– services defined as UML interfaces. Services hold operations whose type can be:

- CRUD (*create, read, update, delete*) operations to handle data in a basic manner from and toward databases. Implementation is later determined according to the chosen persistence framework (e.g., Hibernate, Java Persistence API (JPA), etc.),

- processes as behaviors of operations from which a consultant is able to design complex business functionalities containing conditions, iterations and calls to other operations. Detailed behavior is defined within an UML activity diagram. When we reverse a business functionality from a legacy piece of code, we have a corresponding “process activity diagram” (see an example in Figure 8.2, top right-hand side),

- Web service call to call an exposed Web service. The resulting implementation is later generated according to the Web service provider and inherent properties (simple object access protocol

(SOAP), RESTful, etc.). Basically, consultants have to set up the Web service's URL in the model,

-specific service call to reuse an existing code or application programming interface (API) in the model. The library containing the implementation is automatically inserted as a reference resource of the BFE project. The UML model thus contains the reference to what UML operations map to, i.e. model pieces or pre-packaged software components;

– screen activity diagrams (Figure 8.2, bottom right-hand side) that represent for each screen (or "page" in a Web application) all the available actions/events (for example, hover the mouse over a given button) and the functionality to be possibly executed on the server side when this action is triggered;

– HTML mockups that represent the graphical layout of screens. A screen activity diagram has one and only one mockup. Mockups are bound to the UML model by means of a set of markers. During the generation process, mockups are synchronized with the overall UML model to let the possibility of producing a modernized application with modernized inputs/outputs;

– in case of modernization of batch programs, job and steps activity diagrams are also defined. A job establishes a sequence for a set of steps expressed in activity diagrams.

Figure 8.2 shows the case of a COBOL record creation. This is the managed interaction between diverse UML elements. In other words, all diagrams and all pieces inside each diagram are managed in a consistent manner. BLU AGE requires specific links between elements according to their nature (entity, business object, service, etc.) but also their place and role in a given diagram, being instances of the "Class", "Activity", "Interface", "Operation", etc. UML meta-types.

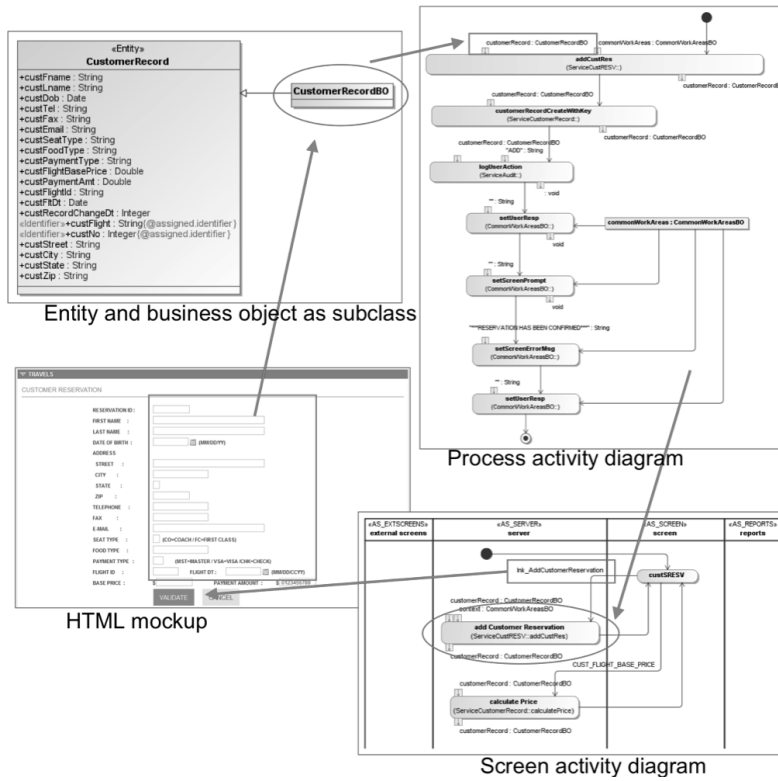


Figure 8.2. *BLU AGE PIM diagrams and their inter-relation*

8.2.2. *Reverse tooling*

In a modernization project, the tooling manages all model pieces and their interdependence in a consistent manner so that we may:

– ensure the completeness of the modernized application (not to miss a business functionality, for instance, in the legacy code);

– enable the progress tracking of the modernization actions, to manage the project as a whole;

– automate the processes as much as possible to improve the productivity;

– compose a team with people that may be non-experts in the legacy technology in general and the legacy architecture in particular.

BLU AGE proposes a set of components out of the box based on the Eclipse workbench to meet these needs.

8.2.2.1. *Views and perspectives*

BLU AGE proposes a set of views and perspectives to read and to understand the legacy code using an annotation editor. Once posed, annotations are source of navigations. For example, Figure 8.3 shows how to access the content of a COBOL *Perform* code block (also known as "paragraph call"), provided that this one has been previously annotated. Smart navigation also allows us to access data definitions from data occurrences, etc.

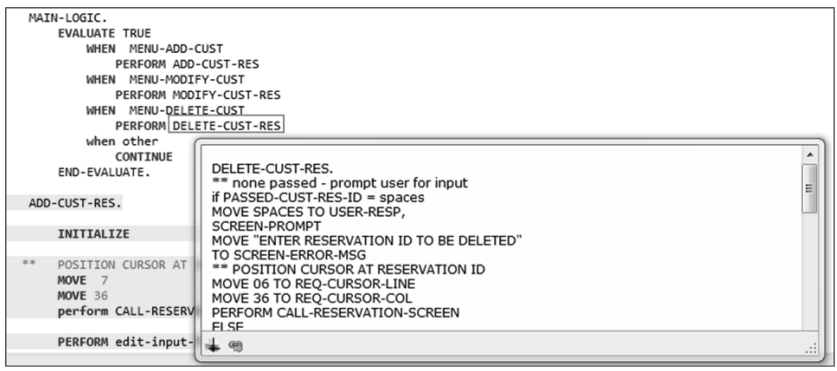


Figure 8.3. *Annotation editor*

Another useful view is the function view. When a user selects a piece of code, she/he can quickly see which paragraph is calling it, what the paragraphs inside the current paragraph are called, what the data items handled in this paragraph are including the direction (updated value or provided value). A sample of "function view" appears in Figure 8.4.

One has to insist on the fact that the legacy code management is backed up by the underlying KDM representation. From the first ASTM and the next KDM, the legacy code only "exists" as a PSM model. For

transparency and useless complexity (ASTM and KDM models are not really intelligible), users cannot visualize such PSM models. Instead, they cope with the legacy code in a more or less intuitive manner.

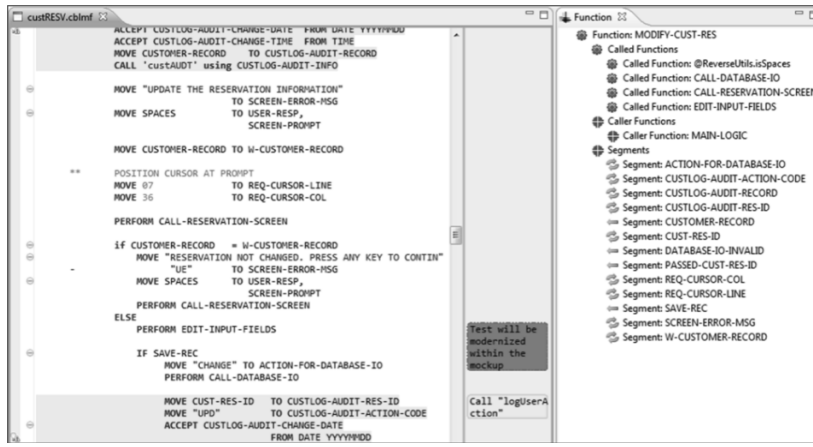


Figure 8.4. *Function view*

The segment view (Figure 8.5) proposes a way to have a representation of data items at a glance. In COBOL, many data items are defined within a group having a tree view (Figure 8.5, right-hand side). Users have to select a data item from the code and the segment view is concomitantly displayed from its selection context.

8.2.2.2. *UI extraction*

When modernizing an online transaction processing (OLTP) COBOL program contains screens, a modernized behavior for the application's screens is required. BLU AGE proposes a feature to transform automatically "green screens" to HTML pages (Figure 8.6). The new screens still contain references to the legacy application using HTML tag properties. These new HTML pages are used as input artifacts in BFE. Extracted HTML mockups need to be graphically adapted to customers' needs. This is facilitated by the fact that screen extraction does not forget any old element even if some are destined to fade. Modernization cannot occur without an HTML designer in charge of completing and tuning the modern content in the resulting Web pages.

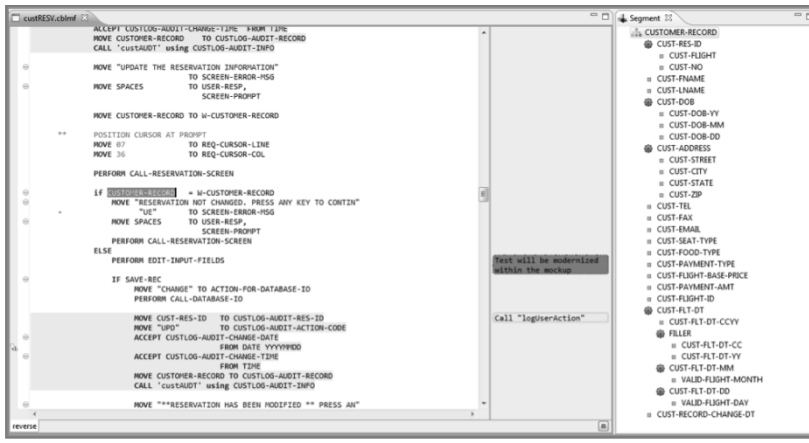


Figure 8.5. *Segment view*

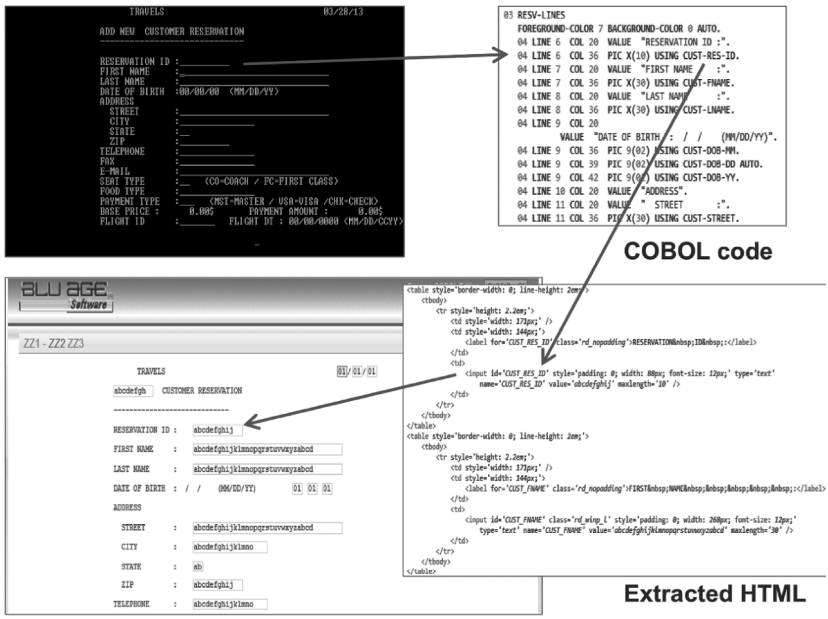


Figure 8.6. *HTML mockup extraction*

8.2.2.3. *Annotations*

The annotation mechanism is a feature used to mark the legacy code with a color code (also see section 8.4.2.1). Annotations in Figure 8.7 have a meaning and can be used to provide guidelines during reverse engineering. For example, we may annotate the code to check what part is intended to be kept, skipped, transmodeled or to establish that a well-defined part of the code has been modernized to something already present in the "modern world". In the end, all the code should be annotated even if many parts are intended, due to their obsolescence or meaningless status, to be dropped.

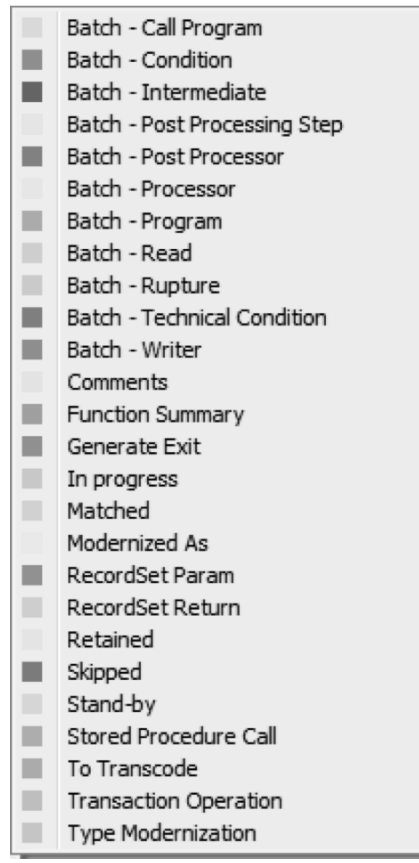


Figure 8.7. *BLU AGE annotations*

8.2.2.4. *Pattern selection and application*

Patterns are connected with the legacy code for code comprehension and massive processing when scalability issues are rising. Patterns are pieces of code that are repetitive code blocks at various places despite some well-established differences are tolerable from one block to another, both belonging to the same pattern. Once formalized, applying a pattern leads to automatically generated other annotations. Figure 8.8 shows in three steps: 1) pattern identification and 2) pattern design that mainly consists of separating variable and invariable parts (2). 3) Once done, legacy code blocks are assigned to the pattern as variants: same shape/structure with variations.

Pattern discovery and matching is highly iterative in BLU AGE to progressively understand the inner workings of the legacy code. Code comprehension occurs on an exponential scale as soon as many patterns are detected and many inferences are carried out. This leads to numerous annotations that drastically reduce the remaining code to be modernized.

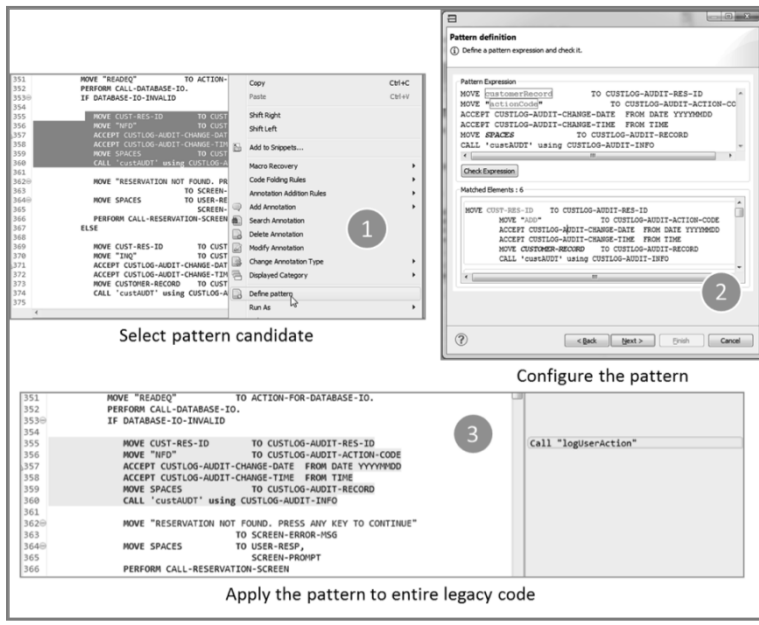


Figure 8.8. *Pattern discovery, tailoring and application*

8.2.2.5. *Data item extraction*

In COBOL, data items are most of the time modernized as UML classes. BLU AGE provides a standard feature to select a data item from the legacy code and transform it into a class by applying automated transformations to set its name (according to naming conventions in the "modern world") and its type. Users may be invited to define their own transformations in case of tricky cases.

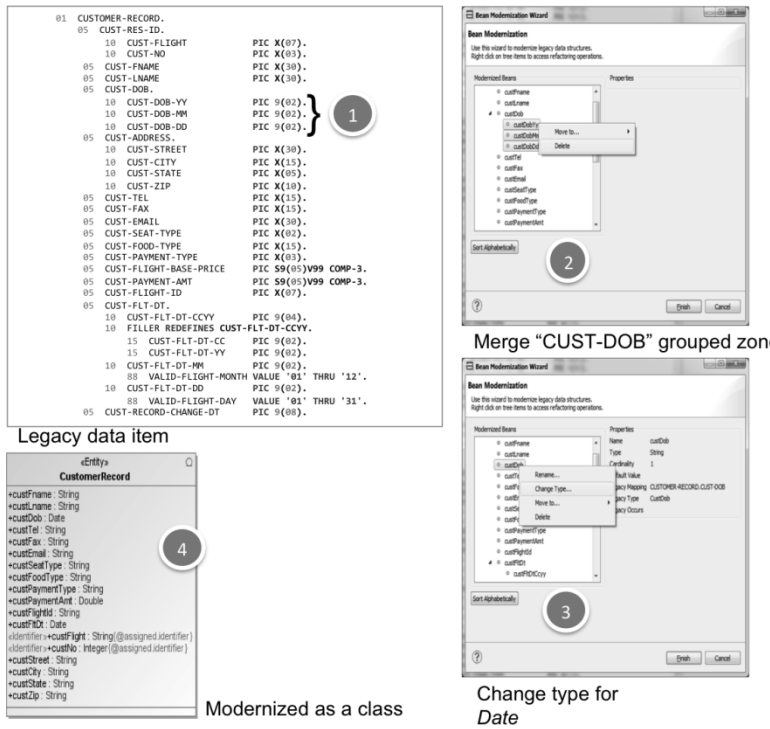


Figure 8.9. *Data item modernization*

Data item extraction often relies on the characterization of the shape of grouped zones within the overall data hierarchy (step 1 in Figure 8.9) and to apply targeted transformations (steps 2 and 3) like moving a character suite (*PIC X(...)* item or suite of items) to, for instance, a *Date* abstract data type in the modern technology ("custDob" attribute of "CustomerRecord" is of type *Date* in

Figure 8.9, step 4). All applied transformations to data items are stored in order to keep a mapping between model elements (classes and attributes) and legacy data items. At any time, we know all legacy data items that were previously transformed into a class or, in scarcer cases, to an attribute only.

8.2.2.6. *Transmodeling as business logic (rules and functionalities) extraction*

Transmodeling is the core feature of BLU AGE. It amounts to selecting a piece of code to move it to a UML element. This piece of code is transformed into an activity diagram representing the selected functionality. If the selected piece has annotations, these annotations are later used to provide instructions to the transformation engine ("skipped" or "modernized as"). Transmodeling is concerned with approximately 80% of consultants' modernization actions.

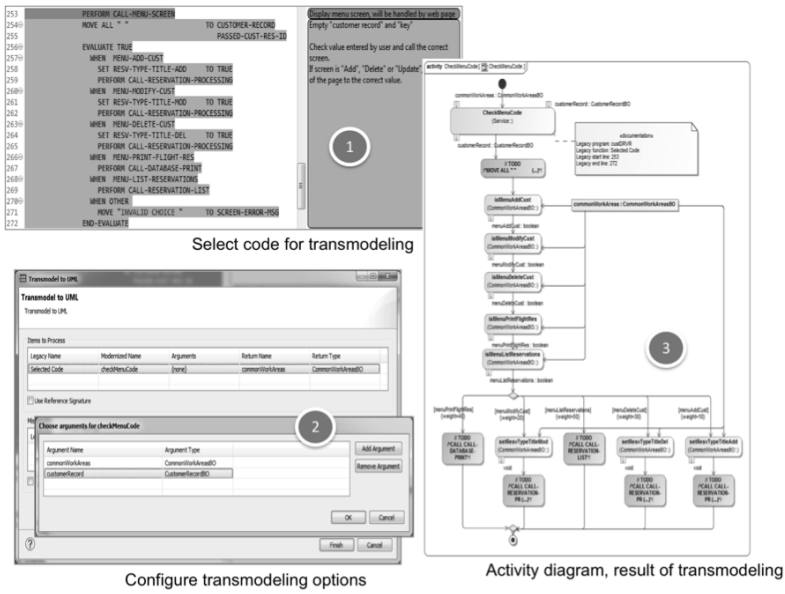


Figure 8.10. *Transmodeling*

Figure 8.10 illustrates the transmodeling approach. Code is previously annotated or punctually selected (stage 1) for

transformation as a UML activity diagram (stage 3). Stage 2 is the modernization stakeholders' intervention. Although model transformations are pre-programmed, they require argument values to customize them through BLU AGE facilities.

8.3. BLU AGE as an ADM- and MDA-compliant tool

Let us consider the modernization of an OLTP COBOL application with numerous "green screens" toward an Enterprise JavaBeans™ (EJB) (say, version 3.×) Java application with, for instance, the JavaServer faces (JSF) presentation framework and the JPA persistence framework (Figure 8.11, right-hand side).

The first step (Figure 8.11) leads to transforming the legacy code into a PSM model: KDM model. With BLU AGE, this transformation is tailorable starting from archetype transformations in a knowledge base. These interpret the legacy code based on a grammar. Within this step, BLU AGE extracts business entities and generates HTML mockups from the legacy "green screens".

The second step (Figure 8.11) is the transformation of the PSM model (KDM format) to a PIM model in UML format. These transformations are hardwired with manual intervention so that the business logic is properly separated from the (obsolete) legacy platform matter: obsolete transaction management marks, persistence techniques, error-handling mechanisms, etc., have to mutate or disappear. In effect, all "prehistoric" matter in the legacy software is intended to be handled by special (new) frameworks (presentation framework like JSF, persistence framework like JPA, transaction management framework like JTA, etc.) available in the retained modern technology. In these transformations, BLU AGE assists people by means of pattern discovering; this is a way to automatically extract similar code. During all these transformation phases, the legacy code (managed via KDM models) is annotated in relation with progression milestones. This keeps a history on what has been already done, and consequently, what is left to be done.

In the third step (Figure 8.11), BLU AGE uses the PIM model to fully generate the final application, depending on the target technology. This step is fully automated. BLU AGE generates the PSM and the modern code "behind the scenes". Next, this code is compiled and packaged. In this step, the final application may be deployed and tested against the legacy application. Evolutions about the modern application come up after this step when possible discrepancies are observed.

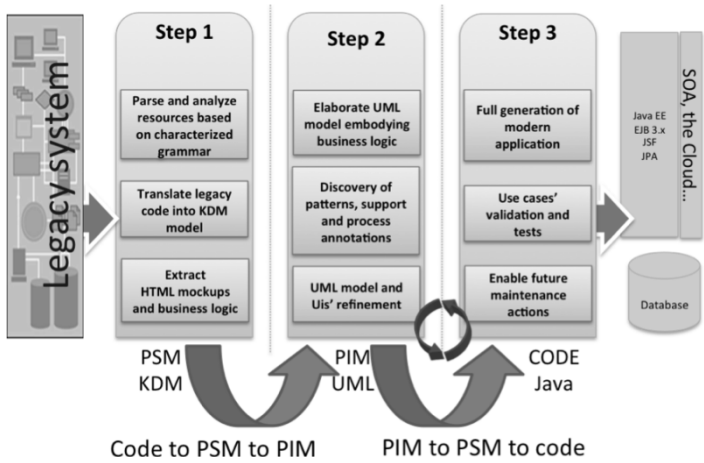


Figure 8.11. *Modernization approach in three key steps*

BLU AGE is above all a model transformation engine. BLU AGE takes models as inputs, applies transformations and produces new models. These new models can be used as input models for other transformations. The BLU AGE engine works with *BLU AGE shared plug-ins* (BSPs) and a knowledge base. BSPs (also known as PDMs in section 6.6.1 in Chapter 6) describe "model to model" and "model to text" transformations, while the knowledge base mostly stores "text to model" transformations. In fact, because everything is a model (including the code), we have at our disposal an innovative means to describe the modeled elements by using metamodels. So, if we are able to describe transformations at the metamodel level, then we apply these at the model level. Figure 8.12 describes this process.

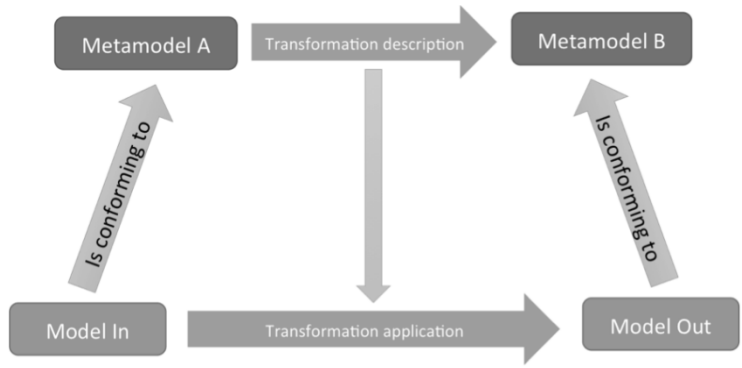


Figure 8.12. *Model transformation based on metamodeling*

Returning to the OLTP COBOL-to-EJB 3.× case, Figure 8.13 shows the way BLU AGE transforms the COBOL code (legacy language) to a PSM model expressed in KDM during the first step of the reverse process. This occurs using a language grammar (initially that of COBOL) because a model is written in a language defined by a metalanguage. In the MDA spirit, the metalanguage is the metamodel. So, if some code is written according to a metalanguage, then transforming it into a model can be performed by formalizing the metalanguage and by applying transformations, which are translations from one language to another.

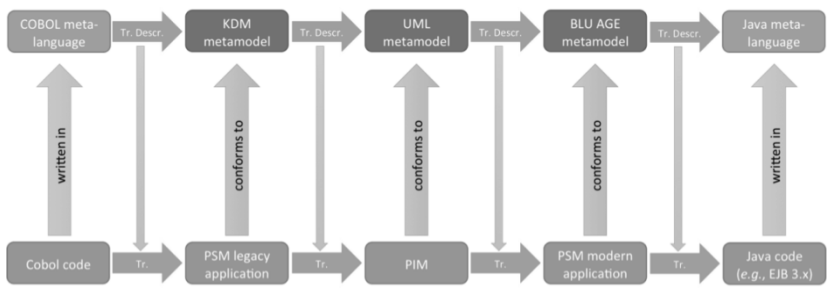


Figure 8.13. *Full transformation workflow from a legacy application to a modern application*

8.4. Modernization workflow

In practice, modernizing a large-scale legacy application is not only a matter of model production. Having a coherent and efficient method is mandatory. Modernization projects from 100,000 LoCs to almost 30,000,000 LoCs generate huge models whose management is highly complex. Models vary in size, nature, role, etc. A method is required to introduce some discretization (a process with steps) to support this management. Such a process is decomposed into three coarse-grain phases with well-defined building blocks:

– The *Initialization* phase where the story begins (details in Figure 8.14). Here, we set up the basic elements of the modernization project, we mine the legacy code and we establish how automation aims at running in relation with measurable gains of productivity. It is, indeed, important to control advances to respect the project's budget, to be able to inform customers about these advances.

– *Realization*, which is the longer phase. It is sequenced by iterations of four to six weeks. This phase consists of building the forward UML model (to be used to generate the final modern application). Activities in this phase are in essence highly assisted by means of BLU AGE.

– *Validation and deployment*. *Realization* already includes some partial validation in iterations, but *validation and deployment* aims at showing concrete results to customers. Namely, at the end of this phase, the modern application is deployed on the chosen production environment. So, deployment, including data migration and change management, is business-critical. A deployment plan must be created and carefully followed up in order to tame risks.

8.4.1. *Initialization*

8.4.1.1. *Explore artifacts*

This is the first activity when starting a modernization project. All the legacy artifacts including at least the legacy code are gathered. Elements such as documentation, database schemes, files, dictionaries, data sets for testing, etc., are, most of the time, scattered. Exploring

these elements allows us to answer to crucial questions: is the set of artifacts "complete" in order to start the modernization under conditions acceptable for success? What additional information do we need? Etc.

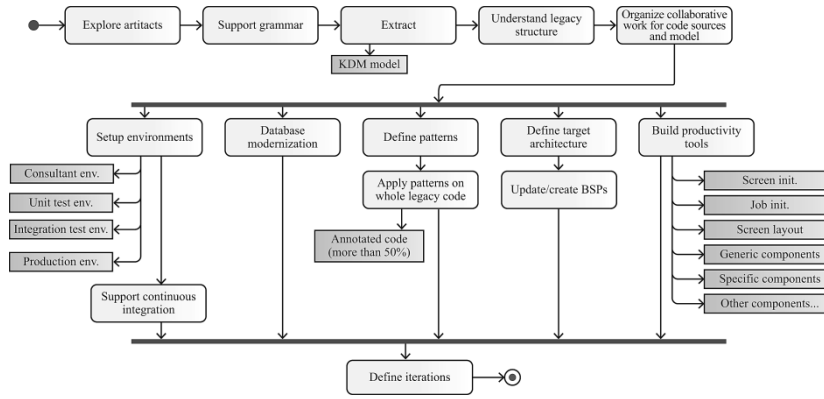


Figure 8.14. *Initialization phase with sub-activities*

8.4.1.2. *Support grammar*

As suggested before, transforming the legacy code into a PSM model (from ASTM to KDM) amounts to characterizing a grammar. Often, this leads to the adjustment of an existing grammar by deeply scanning the different shapes of the legacy code. This adjustment is important because it defines the way to translate the legacy code into a model. The COBOL grammar provided with BLU AGE supports all COBOL standards, but COBOL has a lot of variants. When facing a particular COBOL dialect, an adaptation of the grammar leads to reviewing the native metalanguage/metamodel using generic ASTM (GASTM)/specialized ASTM (SASTM) (see section 7.3 in Chapter 7).

To extend or deal with grammars in general (even creating a new one), BLU AGE provides a third-party set of functionalities called BLU AGE Factory. Only software architects with high expertise may use this feature.

8.4.1.3. *Extract*

This activity takes the legacy code as an input, applies the necessary transformations defined from the grammar and generates the PSM model in KDM format. This task is fully automated in terms of users' guidance and assistance. The resulting KDM PSM model is used by BLU AGE to generate the UML PIM model, but it is mainly used to hold/maintain code annotations (to figure out modernization at large, including progresses: amounts of already processed code as models). There is a particular high-end support for navigations, easy and straightforward accesses to legacy and modernize artifacts, synthesized views, etc., concerning the code organization and the resulting computed models. More generally, the synchronization between the legacy code on the one side and the PSM and PIM models on the other side is total.

Sometimes, the application to be modernized is not easily and straightforwardly readable. This corresponds to macro-languages like COBOL Pacbase (also see section 2.2.1 in Chapter 2) from which the COBOL code is generated and not manually written by developers. In these particular cases, BLU AGE is also able to process the native legacy inputs to generate a PSM KDM model that is somehow closer to a "human-readable form" making it easy to handle by consultants.

The main outputs of the *Extract* activity are:

– legacy PSM in KDM format,

– generated UI prototypes in HTML format (mockups) when the legacy application has screens and related constitutive elements.

The extraction process is run once. It can be automated, scheduled and executed in background (see the idea of "continuous integration" below) if the amount of legacy code is important or this code has to be transformed into a human-readable form that calls for extra processing.

8.4.1.4. *Understand legacy structure*

When completed, the extracted matter allows us to read and understand the legacy code structure. Basically, we dig into the code

using (hidden underlying) KDM models to make apparent how the business logic is engraved in this code:

– How are program files structured?

– Where is the business logic present?

– Where is the kick-off code?

– Is there some underlying framework used from the legacy middleware (transaction management, persistence, logging, error management, etc.)

– Do we need legacy technology experts?

– Are there screens? How are they composed (screen layout)?

8.4.1.5. *Organize collaborative work for code sources and model*

A project never involves only one or two individuals. Consequently, in this activity, the collaborative work environment has to be endowed with appropriate collaborative supports:

– code source repository (SVN, CVS, GIT, etc.);

– model repository (team work server devoted to model concurrent readings, writings, etc.).

In this collaborative environment, the extracted matter is stored in a way which is consistently shareable between modernization stakeholders. Model and code source repositories relate to each other to have effective gateways and thus support an effective collaborative work.

8.4.1.6. *Set up environments, support continuous integration*

Consultants need a working environment to run both BLU AGE and the modernized application. Round-trip engineering requires gateways between the two in order to carry out tests and, from positive or negative validations, apply corrective actions at reverse and forward engineering time. The overall working environment is configured as follows:

– Consultant environment: It is a machine with BLU AGE tooling and a local testing environment (application server, database, etc.).

– Testing environment: It especially includes a unit functional testing environment to validate self-contained functionalities, an integration testing environment to validate the complete application, ensuring non-regression in particular.

– Tester environment: Whereas testers build automatic tests to be principally executed in the spirit of continuous integration.

– Production environment: It has to be used for very final validations in true end users' daily-business contexts.

All of these environments must be documented and the documentation must be shared using tools like *Redmine* or *Sharepoint*, for example.

Continuous integration is a key part of any modernization project. It consists of providing an automatic testing environment to be used on scheduled time. Basically, at each moment of an integration, the model is taken from the model repository, generation is launched, generated application is deployed on a testing environment, automated tests are executed and reports are produced. This process is useful to eliminate regressions.

Continuous integration is a robotized perspective of modernization. All fastidious repetitive tasks that follow creative modernization tasks must in particular be assigned to "jobs", as tasks performed during the night, for instance.

8.4.1.7. *Database modernization*

The purpose of *Database modernization* is to transform a data organization and to migrate data from a renewed data schema. For an average COBOL legacy application, data are commonly stored in a file operating system. In such a legacy context, the concept of the relationship between data is not formally supported compared to what is offered by a relational database. So, *database modernization* is almost the restructuring of three COBOL general-purpose data types that are:

– numeric items consisting of digits 0–9;

– alphabetic items consisting of the A to Z (a to z) letters and the space (blank) character;

– alphanumeric items consisting of digits, alphabets as well as special characters.

Using BDM, the initial "odd schema" based on these data types is recomposed in order to first establish a migration script and next convert/migrate the legacy data to a relational database. This action also results in the generation of entities (i.e. an entity/relationship model as a UML class diagram).

This process is iterative because transforming a large data set is never simple; actions have to be organized regarding priorities about, for example, the criticality of first-class data compared to others. Another example is the prioritarization of data used by the first reversed functions.

8.4.1.8. *Define patterns, apply patterns on whole legacy code*

This activity is a fruitful activity for modernization in the sense that we have to find out patterns of code. Patterns of code are a great support for code comprehension, synthesis and refactoring. A pattern is a piece that is repetitively present in code sources even if some belonging elements vary from one piece to another. In programming languages, patterns come from copy–paste actions of developers.

To discover patterns, a representative sample of legacy code is necessary. Once found, the pattern is characterized with invariable and variable parts. Pattern matching is then applied on the overall code. Pattern selection and application is in particular a set of means for establishing:

– usual implementations, i.e. technical features (e.g. inevitable code sequences) such as:

- logging,

- audit,

- security (authentication, authorization),

- user session and/or context (open, closed, etc.),

- file system access (or data element access),

- description of data items with their deep nature;

– technical;

– persisted;

– screen-centric;

– etc:

- error messages, error management in general,

- navigation (menu, command line, hyperlink, etc.);

– specific functional/technical features:

- specific actions (command line navigation, communication with other systems, etc.),

- where the business logic is effectively present, is there a recurring structuring for this logic?

- Events.

When pattern occurrences are found, annotation actions on the marked code are available. These annotations are used to provide transformation guidelines to be used at transmodeling time. The most common annotation actions to be applied are:

– "skipped": this is technical code that is differently implemented in the modern application by using up-to-date-frameworks (JSF, JPA, etc.);

– calling an existing action already implemented somewhere else as an already known UML element (operation). In this case, transmodeling generates a call to this element (e.g., requesting a logging or database resource through a devoted call).

When patterns are identified, we may apply them on the entire code. From experience, more than 50% of the entire code is matched to patterns and annotated accordingly. During the project's whole life,

other patterns may be identified requiring new code parsing. Patterns are stored in a specific project within Eclipse. It means that they can be reused for other projects. More generally, patterns are the central part of the knowledge base used in any modernization project.

8.4.1.9. *Define target architecture, update/create BSPs*

Modern applications use "framework-oriented code". This means that application behaviors for persistence, presentation, transaction management, security, etc., requirements rely on technical services offered by the target technology. Choosing frameworks is a tricky task because compatibility (i.e. frameworks' interoperability) issues are numerous. In this activity, the target technology is defined, namely:

– the general technology: Java, .NET, cloud platforms, interactive (Web) technology versus technology for batch processing, etc;

– application and database servers;

– frameworks: presentation (JSF, Struts, Spring Web MVC, etc.,), SOA (EJB, Spring, etc.) even non-SOA, persistence (Hibernate, JPA, etc.), security [e.g. *Java authentication and authorization service (*JAAS)], communication [e.g. *Java message service* (JMS)], etc.

This activity is also the place to stress screen layout. How do users navigate through the renewed application? Is there a need for "menu" sections in pages? How should a screen be decomposed? What are the reusable UI components? How do users navigate through data grids? How should the data items in data grids be updated/deleted? Etc.

Off-the-shelf BLU AGE proposes a large set of BSPs used to handle standard frameworks. In using BLU AGE Factory, updating or creating BSPs is a critical aspect of application evolution. In fact, applications may be regenerated from new BSPs corresponding to other or versioned technologies and/or individual frameworks.

8.4.1.10. *Build productivity tools*

This activity makes a lot of sense for large modernization projects. Its main objective is to set up specific tools to be used by consultants

who help them to gain productivity. For example, defining and implementing/integrating tools to:

– initialize a new screen, a new job, etc. Typically, session variables in Web applications require initializations that may occur at the first time a home page is displayed;

– manage screen layout (a Web design tool as an integrated element of the modernization framework);

– build generic (or not) components to be used in many places of the modern application. A specific component to be translated is, for instance, a syntactical analyzer of command line inputs: “Add”, “add”, “adding”, etc., are different inputs that must lead to the execution of the same “Add a new reservation” service in an airline reservation system (also see the case study in Chapter 9).

All these elements have to be documented and “on duty”; the team of consultants has to be trained in mastering them.

8.4.1.11. *Define iterations*

BLU AGE projects made up of iterations, etc. An iteration is between four and six weeks; it must:

– have a scope (technical and/or functional), something to be shown to project clients and users;

– be validated at its end by users, at least by executing tests in relation to satisfactory (measurable) scores.

The content and the order of iterations are to be defined with enough time to manage any crisis and, consequently, to deal with risk management.

A good iteration would be a set of functions decomposed into sub-functions. It can represent a set of screens that may be tested in unit functional tests or in integration tests. It is important to prepare iteration content with something that can be assessed by the project clients and/or users.

8.4.2. *Realization*

Realization as detailed in Figure 8.15 is the longer phase. Projects' iterations have been defined so that the modernization tasks are now assigned to team's members with a strict delivery planning. In most cases, a task is the modernization of a screen including entering/exiting actions and internal behaviors. A task is more rarely the modernization of a well-delimited step in the context of a batch program.

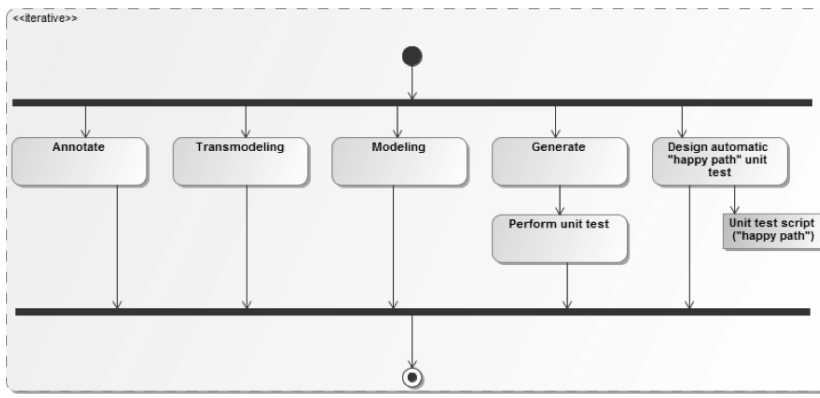


Figure 8.15. *Realization activities*

8.4.2.1. *Annotate*

One key principle behind BLU AGE is the possibility of annotating the legacy code. In fact, the PSM is inductively annotated. Indeed, there is a perfect bijection between the legacy code and its representation in KDM. This mapping is maintained at all times by BLU AGE. Annotations are shown on the legacy code as shown in Figures 8.16 and 8.17. Posing an annotation often leads to a comment on the legacy code (Figure 8.16). More importantly, annotating also results in transmodeling guidelines (Figure 8.17), for example:

– do not transmodel ("skipped" predefined annotation);

– this piece of code is already transmodeled in "this" operation;

– transmodeling this piece of code is required to model a call operation to "this" existing operation.

Annotations may be provided “by hand” while a consultant works on a legacy part. They can also result from transmodeling or are automatically generated when running the pattern matching algorithm.

BLU AGE also proposes statistics about the annotated code. This is helpful to follow the course of the modernization project. Annotations are also shared between team members using a source code controller system (CVS, SVN, GIT, etc.). So, everyone is able to determine if a specific piece of code has already been transmodeled and what is its modern shape.

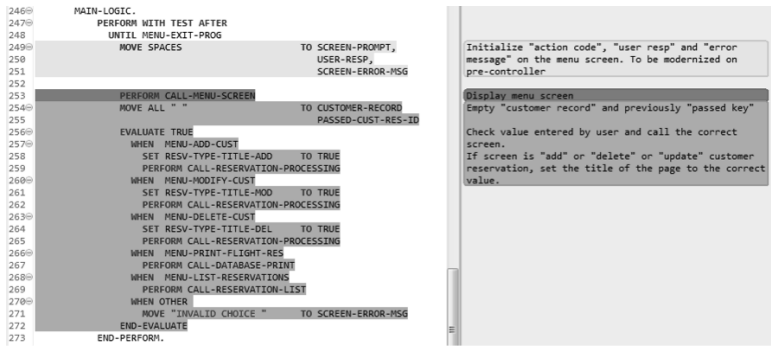


Figure 8.16. *Annotate phase, example of annotation used for documentation*

Figure 8.17. *Annotate phase, example of annotations used as transmodeling guidelines*

8.4.2.2. *Transmodeling*

Transmodeling is a suitable principle in the BLU AGE modernization method. Once selected, a sequence of COBOL statements (which always has an underlying representation as a KDM model piece) from the legacy application may be immediately transformed into an UML activity diagram. This dynamical UML diagram is later used to generate a specific operation in the target architecture. In practice, transmodeling takes into account existing annotations as follows: if a piece of code has been annotated by "Modernized As" <X>, the transmodeling processor generates in UML a call operation action. In the UML model created by BLU AGE, this amounts to an instance of the *CallOperationAction* UML meta-type with a link to the "X" model element (the *operation* meta-navigation of *CallOperationAction* is used).

Another relevant transmodeling approach is the generation of "business objects" from COBOL data items (Figure 8.18). BLU AGE provides a way to generate a UML class associated at transmodeling time with a chosen data item. This class contains attributes; a type is defined from an automatic mapping with a default choice that can be overridden.

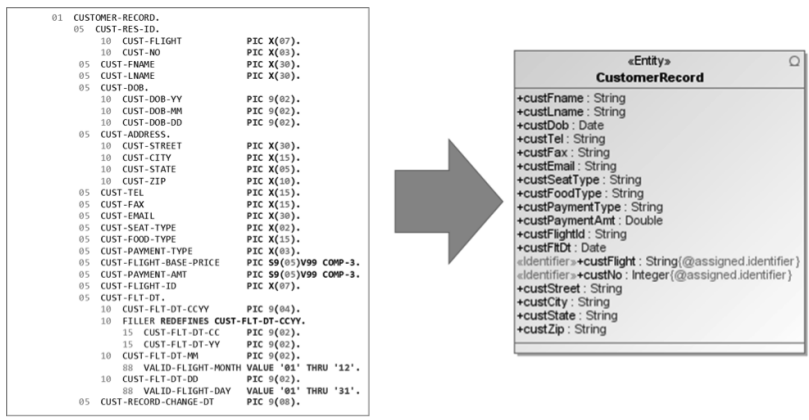


Figure 8.18. *Transmodeling a COBOL data item as a UML Class*

Transmodeling systematically leads to comments containing information (e.g., LoC numbers) on the original legacy code. This information is later used as comments on the generated code.

8.4.2.3. *Modeling*

"Modeling" here means the manual intervention on the generated UML model as result of modernization. Most of the model is generated automatically, but consultants sometimes have to amend the generated model, even create model parts from scratch.

8.4.2.4. *Generate and perform unit test*

A best practice is to generate and, sequentially, perform tests on what has been generated. Namely, when a part is modeled, for instance, when a fully fledged functionality has been transmodeled as a UML activity diagram, it is opportunistic to produce, deploy and run the equivalent application piece as a business service. Testing this particular functionality within its new execution environment allows checking that everything is working as expected. Consultants conduct many generations per day with the possibility of carrying out numerous elementary tests.

8.4.2.5. *Design automatic "happy path" unit test*

Testing people aims at building automated tests with powerful devoted tools like *Selenium*, for instance. Test building is concerned with some functionality or some sub-functionality that is testable, i.e. a self-contained piece. It may be a complete screen or only a screen action when possible.

The main advantage of this practice is to elaborate tests to be run at the stages when continuous integration occurs. These are unit tests for "normal" data and "normal" behaviors (also known as "happy path" tests). This excludes borderline data and cases, which are subject to sophisticated control in the application, e.g. the disruption of a transaction because of a server failure. By their very deep nature, "happy path" tests are easier to design compared to more customized tests (see the prior section) whose execution cannot really be

automated in the context of continuous integration, namely some unit tests may only exist for one day for a specific purpose.

In common practice, every night in general, “happy path” (unit) tests may be executed to make sure that there is no regression about the already modernized functionalities. When a regression is found, people may quickly react to apply any necessary correction on the modernized matter.

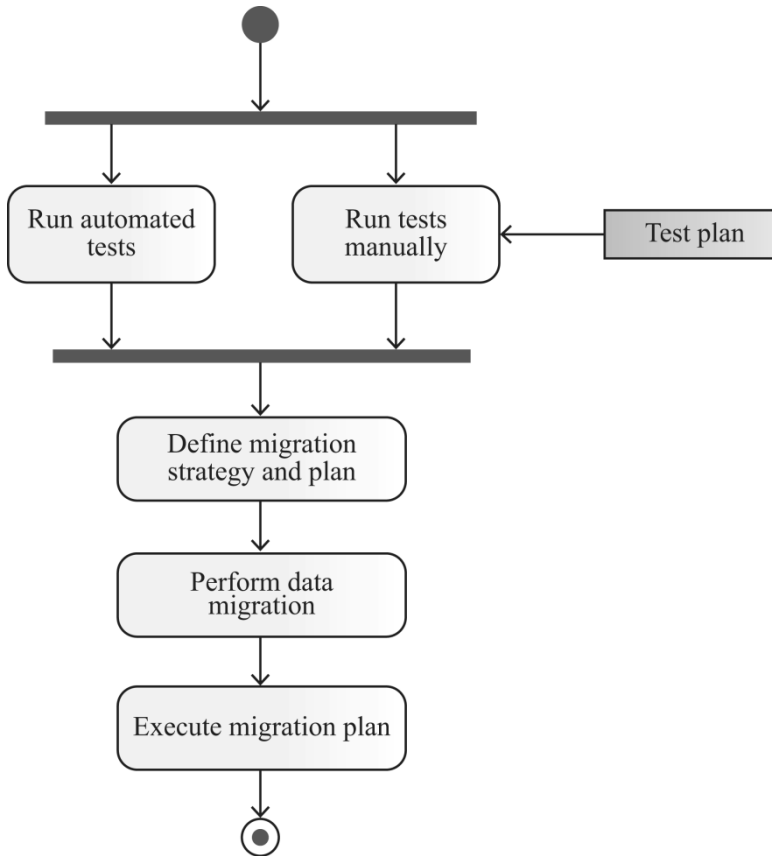


Figure 8.19. *Validation and deployment*

8.4.3. *Validation and deployment*

Validation and deployment is the last phase of a modernization project. This phase is concerned with the validation with end users; they check the real completeness and effective functioning of the modernized application against conditions that are representative of daily business.

8.4.3.1. *Run automated tests*

Most tests have been automated during the previous phase. These can be executed by a robot on the entire renewed application; the latter is deployed *on a production-like platform* to meet daily business requirements. All the tests must succeed. This is usually a kind of formality because if we are able to enter in the *validation and deployment* phase, then it means that unit and integration testing already succeed for the testing environment that often slightly differs from the production platform.

8.4.3.2. *Run tests manually*

Most of the time, modernization actors benefit from numerous pre-built tests in the test plan. While most tests may be run automatically, users may ask for running some tests on they own for tricky behavior checking. Being automated (in the scope of continuous integration) or not, some tests have to be run manually for gaining extra information of the applications' robustness.

8.4.3.3. *Define migration strategy and plan*

The *Define migration strategy and plan* phase is the preparation of all the actions to be performed to have the definitive application in real execution conditions; it serves both the business and end users. Because of the big number and importance of details, a strategy and a plan are necessary in relation to an accurate schedule to stop modernization and restart business to a more innovative application. Namely, concerns must include rollover actions in case of difficulties, even failures. Thinking deeply about such a plan is important for identifying all details linked to all migration

sub-processes. Milestones, people with special responsibility including leader(s), contingency plans, etc., are parts of the overall migration strategy and plan.

8.4.3.4. *Perform data migration*

The redesigned application is probably now running as expected, but, in general, data sets used for testing are only representative samples. It is then time to execute the data migration script on the entire legacy data. This script has been prepared during the *initialization* phase. It is then later possible to deploy the modernized application on its production environment with the "true" business data. Enhanced tests can be required to check the application beyond data samples. For example, volumes may show performance problems at this step.

8.4.3.5. *Execute migration plan*

This is the very final activity of a modernization project. It consists of running the plan for some days until customers have entire satisfaction.

8.5. Conclusions

Reality in COBOL software modernization calls for tools and methods beyond concepts (MDD, etc.) and standards (ADM, MDA, etc.). Industrial expectations are above all productivity. Another key preoccupation is: how much COBOL matter may be moved to newer shapes including drastic simplifications? This question arises because modern platforms offer many predefined services at the technical level (logging, persistence, presentation, error management, etc.) or at the business level when predefined Web services are to be reused in the modern code. In this context, modernization stakeholders crucially need information about a project's advances and testing/validation against the behavior of the old system. BLU AGE offers an end-to-end method that in particular smartly supports inline deployments and executions on the contemporary execution environment.

As a global response to all of these expectations and concerns, BLU AGE has been intensively used in a lot of business-critical projects including Obamacare implementation in the USA, e-government applications in France, accounting in Norway, tourism management in Spain, etc. The next chapter aims to show BLU AGE in action to have further insights on COBOL software modernization.

9

Case Study

This chapter is a pedagogical and practical illustration of the ideas, principles and solution elements discussed in Chapter 8. This chapter is self-contained. It describes a step-by-step modernization process to redesign a legacy Common Business-Oriented Language (COBOL) application to finally have a modern Java Web application.

Numerous decisions are taken in a modernization process. In this chapter, the creation of *BLU Application Generator* (BLU AGE) Shared Plugins (BSPs), i.e. *Platform Description Models* (PDMs), to have the possibility of generating applications toward various (sometimes heterogeneous) platforms, is not addressed. Thus, the first set of decisions is the choice of the target platform among the available BLU AGE BSPs. In this chapter, Java is used along with:

– user Interface (UI): Spring Web Model-View-Controller (MVC) as the presentation framework. To that extent, Figure 9.1 shows at application generation time, the choice of Spring Web MVC along with possible configurations;

– service layer: Spring;

– persistence layer: Hibernate;

– database server: PostgreSQL.

This chapter also explains the reverse engineering process for screens and the steps involved in creating a Unified Modeling

Language™ (UML) model piece-by-piece. Pieces are intended to be connected with each other in a go-with-the-flow manner. They are indeed assembled in order to satisfy the constraints imposed by BFE and its integrated metalanguage. A shown in this chapter, the organization of model pieces in UML packages is strict along with the native UML metatypes and metarelationships to tie them together.

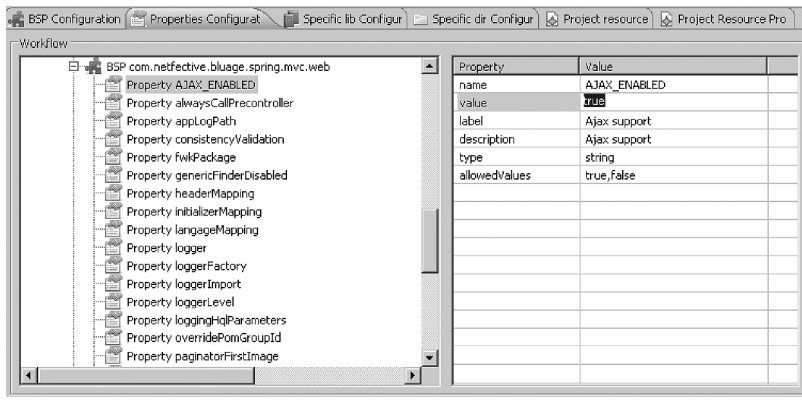


Figure 9.1. *Enabling AJAX in Spring Web MVC at forward engineering configuration time*

9.1. Case study presentation

The application used in this chapter is an airline reservation system. It is composed of the following screens:

– *Menu* (Figure 9.2): manages the reservation system as a whole. Users select the menu option number and are redirected to the requested screen;

– *Add a new reservation* (Figure 9.3): this screen is a standard creation form. The payment amount is calculated taking into account the flight date and base price;

– *Modify an existing reservation* (Figure 9.4): users type the reservation ID in the first field so that the page displays the record for edition;

– *Delete an existing reservation* (Figure 9.5): in the same way, users type the reservation ID to be deleted; the page then displays the record (it is not editable). Confirmation or invalidation of deletion is required;

– *Print flight reservation* (Figure 9.6): users select a flight ID; the application then prints a report (Figure 9.7) as a text file containing all the reservations for the selected flight;

– *List all reservations* (Figure 9.8): the screen displays all of the existing reservations in a data grid-based screen area. Users are able to apply an action on the left column. Actions are "U" or "D" for *Update* or *Delete* respectively. According to the selected action, the "Modify an existing reservation" (Figure 9.4) or "Delete an existing reservation" (Figure 9.5) screen is displayed.

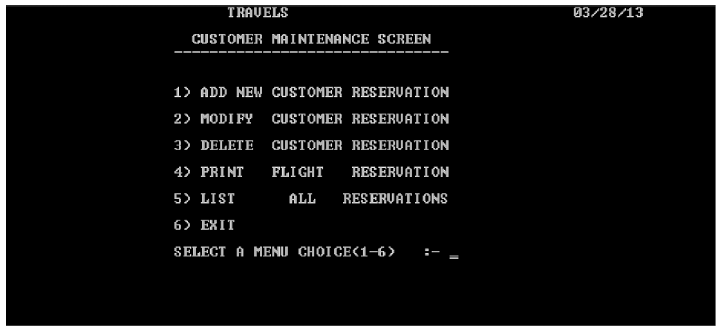


Figure 9.2. *"Menu" screen*

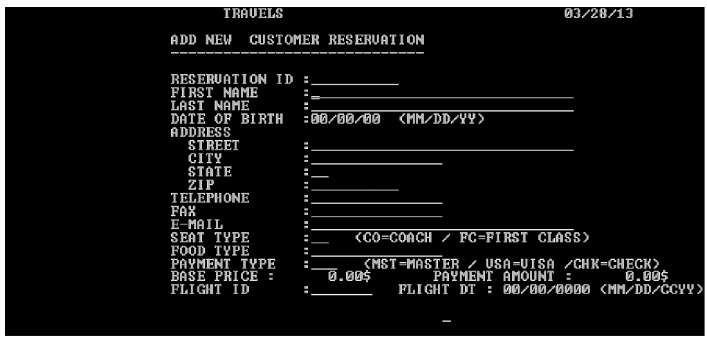


Figure 9.3. *"Add a new reservation" screen*

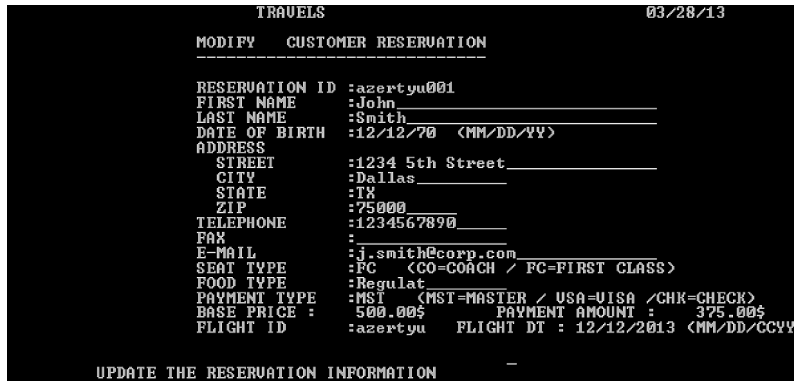


Figure 9.4. *"Modify an existing reservation" screen*

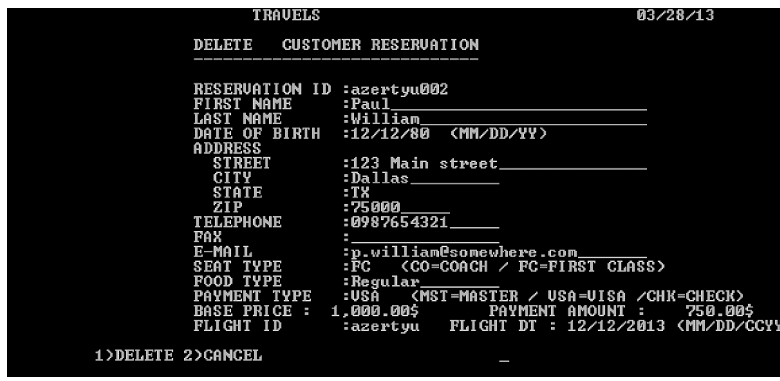


Figure 9.5. *"Delete an existing reservation" screen*

Figure 9.6. *"Print flight reservation" screen*

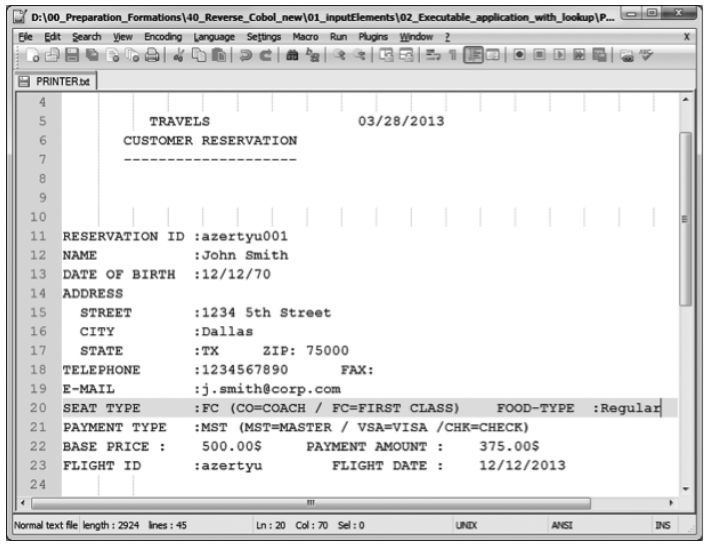


Figure 9.7. *Report of reservations for a given flight ID*

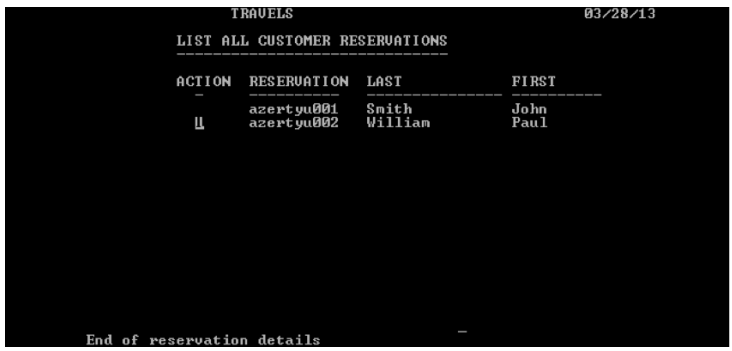


Figure 9.8. *"List all reservations" screen*

9.2. Legacy modernization in action

The BLU AGE legacy modernization process imposes the following steps:

1) creating a modernization project in which the creation of static HTML is operated;

2) mapping data items of the "legacy world" to UML classes in the "modern world";

3) annotating legacy code;

4) pattern identification and matching;

5) transmodeling;

6) generating and testing the new application.

Each step is described by means of a representative sample from the airline reservation system.

9.2.1. *Creating modernization project*

In BLU AGE, a modernization project is a "COBOL Reverse Project", say *AirLineReservationReverse* (Figure 9.9), and a "Forward Project" (see section 9.2.5). The latter is essentially the UML model used for application generation once the reverse phase is terminated. The former mainly possesses COBOL programs and static HTML files. HTML mockups aim at evolving at modeling time both from the material found in the COBOL screens and, later, the extracted application's behavior (screen chaining, business functionalities, etc.).

The legacy material is integrated in the *AirLineReservationReverse* project as, for instance, a ZIP file (Figure 9.10). For the sake of simplicity, the airline reservation system is considered as being in the ANSI COBOL format, which, in BLU AGE, does not call for Abstract Syntax Tree Metamodel (ASTM)/Knowledge Discovery Metamodel (KDM) adjustments.

9.2.2. *Better dealing with the legacy material*

In BLU AGE, activating the "Generate BLU AGE Mockup" option (Figure 9.11) translates the old screens into HTML. The resulting (global) folder structure is shown in Figure 9.12. The "pages" folder contains the HTML mockups that have to be later completed, in relation with the UML models, from discovered dynamical features.

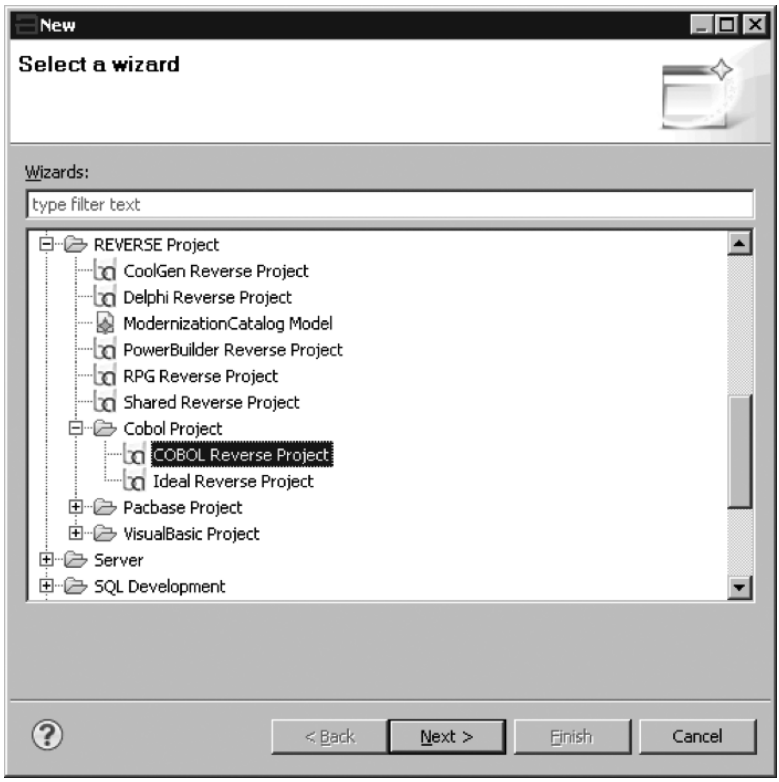


Figure 9.9. *Enacting "COBOL Reverse Project" option*

Data items are defined in the *.cpy* files (a.k.a. "copybooks") and, in a general-purpose COBOL approach, included into programs at compilation time.

The AirLineReservationReverse project is such that:

– "custDRVR" is the main program called.

– "custDBIO" and "custAUDT" are functions used to manage persistence (they contain a COBOL *FILE SECTION*).

– The "custSxxxx" program family is the screen definitions.

– The "custxxxx" program family includes the business functions.

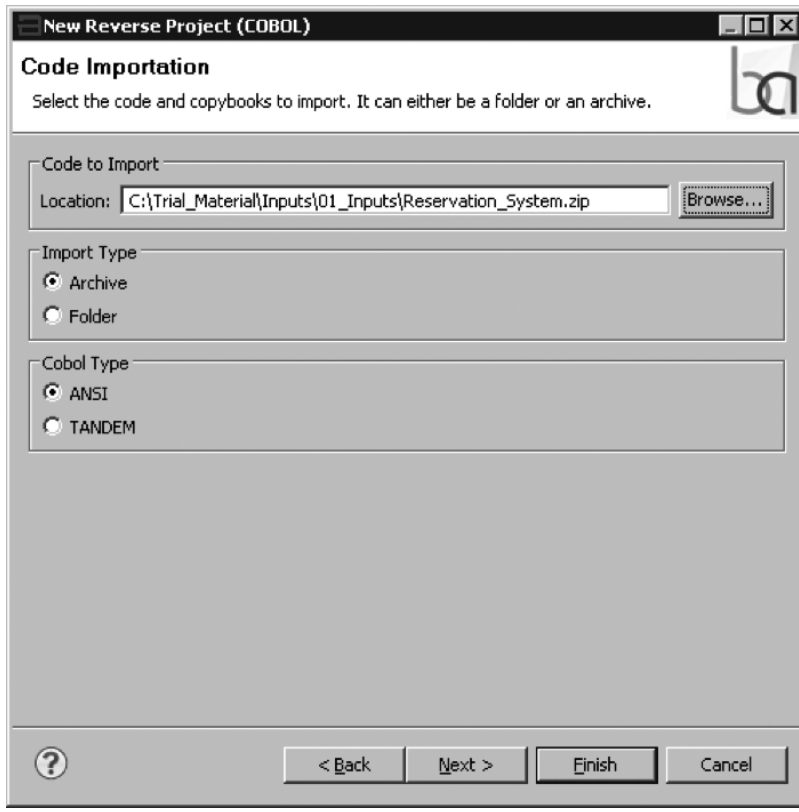


Figure 9.10. *Path to COBOL programs to be modernized*

Each program (e.g. "custAUDT" in Figure 9.13) has an ordinary structure in terms of divisions. Furthermore, as shown in Figure 9.14, the *DATA DIVISION* of a COBOL program also might have a *LINKAGE SECTION*, *WORKING-STORAGE SECTION* or *FILE SECTION* (for data storage-oriented programs). Any data division includes the data items that are originally defined in copybooks.

The initial processing of the legacy material enables the discovering of all the relevant data items and, in which program(s) they are used, their role(s) as member of the *LINKAGE SECTION*, *WORKING-STORAGE SECTION* or *FILE SECTION*. In Figure 9.15,

columns represent data items, rows represent COBOL programs and mnemonics represent data item roles.

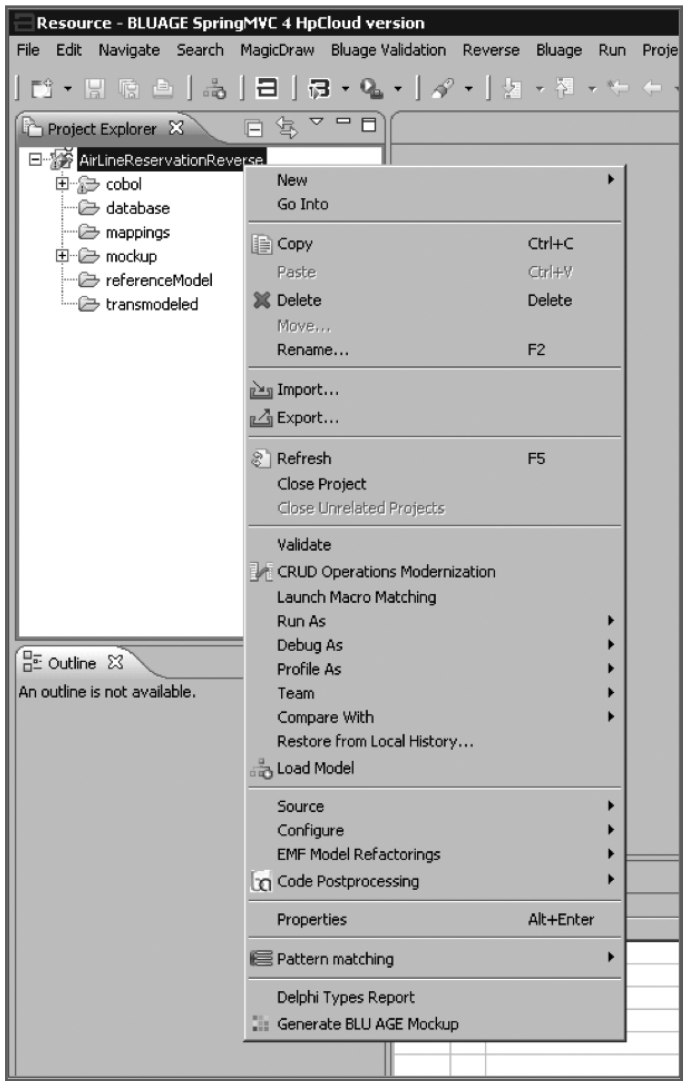


Figure 9.11. *"Generate BLU AGE Mockup" option*

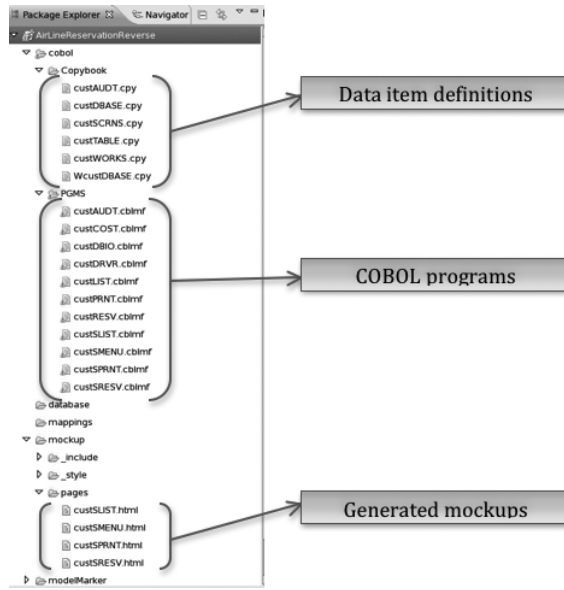


Figure 9.12. *Programs, data items and HTML pages*

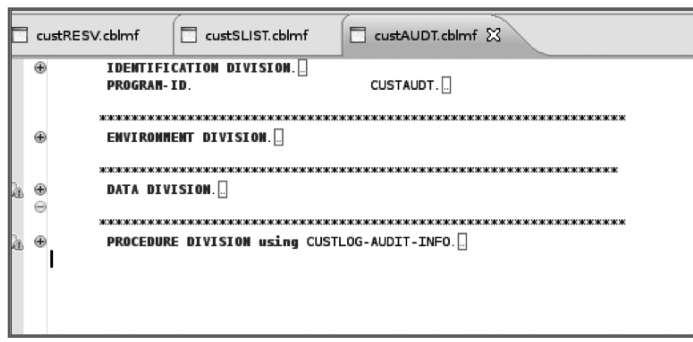


Figure 9.13. *Four divisions of a COBOL program*

The table in Figure 9.15 assists us in deciding which data items need to be modernized and which do not. Some data items are defined for technical purposes and can be replaced by functions made available in the modern architecture. For example, data grid-based presentation can be implemented reusing common Web “data grid UI

components" instead of "list components" as often observed in COBOL legacy applications.

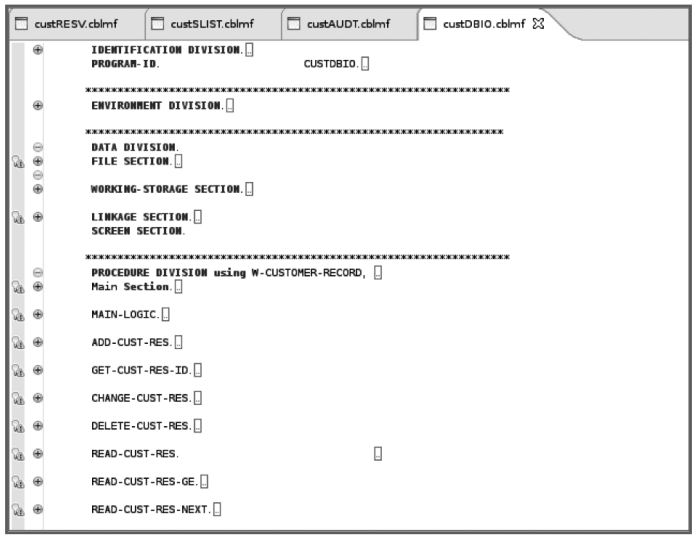


Figure 9.14. *Possible sections of a COBOL program*

	CUSTLOG-AUDIT-INFO	CUSTOMER-RECORD	W-CUSTOMER-RECORD	CUSTOMER-TABLE	COMMON-WORK-AREAS	COMMON-WORK-AREAS-SCREENS
custAUDT	LNK					
custCOST		LNK			LNK	
custDBIO		FILE	LNK	LNK	LNK	
custDRVR	WS	WS		WS	WS	
custLIST		LNK	WS	LNK	LNK	
custPRNT		LNK		LNK	LNK	
custRESV	WS	LNK	WS	LNK	LNK	
custSLIST		LNK		LNK	LNK	WS
custSMENU					LNK	WS
custSPRNT					LNK	WS
custSRESV		LNK	WS	LNK	LNK	WS

LNK Linkage section
WS Working Storage section
FILE File section

Figure 9.15. *Programs, data items used in these programs, sections to which they belong*

As a result, Figure 9.16 gives a list of all the data items, their purpose and whether (or not) they have to be modernized.

Data item	Description	Status
CUSTOMER-RECORD	Database record entry	Modernized as persisted element
W-CUSTOMER-RECORD	Technical data item used in some COBOL code blocks (check whether or not the current "record" has been modified)	Not modernized
CUSTOMER-TABLE	It contains table representation for the list page. It is also used to store keys to update or delete the selected "record". In the modernized application, a collection of "CustomerRecord" objects is intended to be used	Not modernized
COMMON-WORK-AREAS	Storage of the legacy application execution context: error message on previous action, last selected menu choice, available database access. It also enumerates the meaning of many specific values for data items	Partially modernized
COMMON-WORK-AREAS-SCREENS	Pure technical COBOL data item used to manage "foreign keys", cursor positions, etc.	Not modernized
CUSTLOG-AUDIT-INFO	Passed data item to log user actions	Partially modernized
CUSTOMER-LOG-RECORD (found in "custAUDT")	It represents the table to store user actions	Modernized as persisted element

Figure 9.16. *Data items that need to be modernized (or not)*

9.2.3. *Strategy for modernizing screens*

The screen layout may be simply reshaped as follows:

– in the legacy system, navigation between screens is based on "screen code" from the "Menu screen". This principle is kept along with the access to other screens by means command lines inputs;

– for enhanced navigation, direct hyperlinks are also introduced in the home page, tabs in all pages as well (“Menu”, “Add”, etc., tabs on the top right corner of Figure 9.56);

– the application renewed screens are all divided into three parts as follows:

- *header*, in which the navigation tab is setup;

- main body, in which the content extracted from legacy is intended to be displayed;

- *footer*, it contains “error messages” and “users’ dialogs” in general;

- the screen containing the data grid (“custLIST”) is modernized using standard data grid functionality common in Web technologies;

- “first”, “previous”, “next”, “last”… shortcuts are realized using icons at the bottom of the grid;

- actions associated with lines are placed on the right (“edit”, “delete”, “select”, etc.);

- clicking on “edit” allows users to edit the fields on the same page (a feature not available in the legacy application).

Otherwise, precontrollers are used to initialize pages. Namely, the “Add a new reservation”, “Modify an existing reservation” and “Delete an existing reservation” functionalities rely on the same legacy screen-managing program: “custRESV”. This program deals with several contextual variables that are also necessary in the modernized application. Legacy initializations must then be ported to the modern application.

9.2.4. *Strategy for modernizing data items*

– “COMMON-WORK-AREAS” data item containing the “application context” is modernized as a transient object; an instance is intended to be passed with each action. The name of this instance is “context”. It is a common Web session variable;

– "CUSTLOG-AUDIT-INFO" data item (used to pass values to be logged) is modernized as a transient object;

– "CUSTOMER-RECORD" data item is modernized as an entity;

– "CUSTOMER-LOG-RECORD" data item is also modernized as an entity.

9.2.5. *Creating forward project*

In BLU AGE, the creation of a "Forward Project", say *AirLineReservationForward* (Figure 9.17) is the first required action. The *mockup* directory of the "Forward Project" has to be populated (drag-and-drop action is enough) from that of the "COBOL Reverse Project".

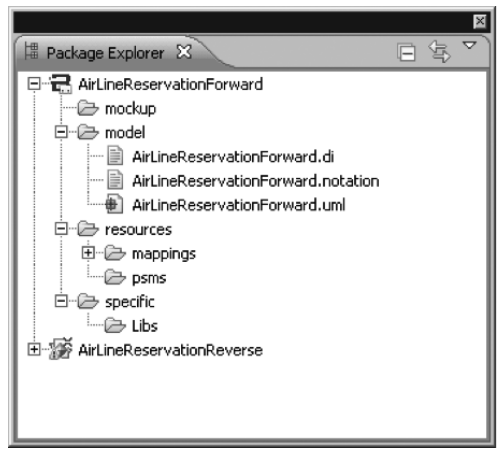


Figure 9.17. *Creating "Forward Project"*

The second required action is the creation of a "Model Project" in a modeling tool such as Papyrus, MagicDraw, Modelio, etc. Considering all modeling tools in MDD, BLU AGE is recognized as tool-agnostic provided that the chosen tool supports UML in a fully compliant style. For example, Figure 9.18 shows the initial structure of a BLU AGE UML model in MagicDraw.

We insist on the fact that BLU AGE deals with a subset of the UML language. In other words, BLU AGE is a set of UML profiles. BLU AGE UML models thus obey to a very strict format including the initial hierarchy of UML packages in Figure 9.18. There is no great difficulty to "speak" the BLU AGE metalanguage. The UML format imposed by BLU AGE is automatically managed by the tool (including its direct creation with empty zones) so that it is permanently ensured that no digression occurs.

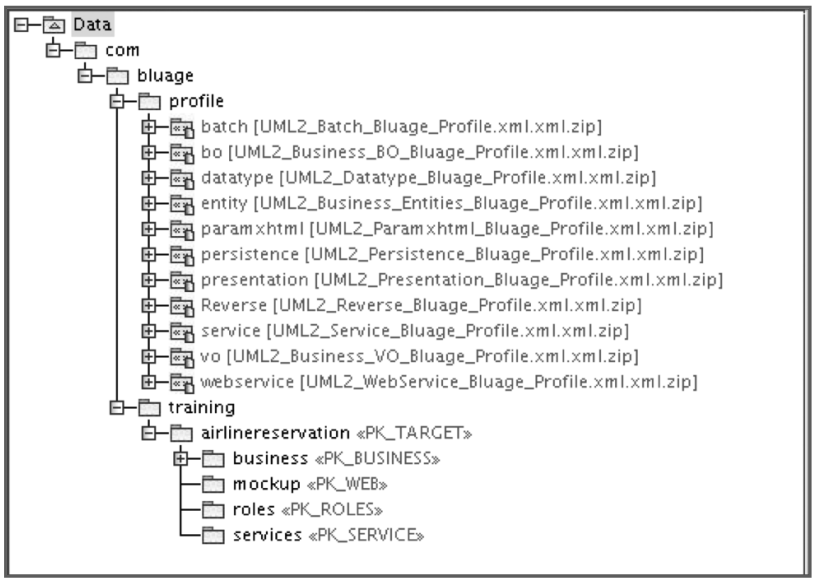


Figure 9.18. *Initial (template) BLU AGE model*

In BLU AGE, stereotypes are used to identify the property of any model element. As an illustration, *<<PK_TARGET>>* is a stereotype assigned to the UML package embodying the overall air reservation system. In conjunction with an applied stereotype, contextual properties may also be assigned to a stereotyped model element due to "tagged values" (as sketched in Figure 6.8).

– *<<PK_TARGET>>* is the root of the directory structure;

– <<*PK_BUSINESS*>> owns all the extracted entities and their respective business objects;

– <<*PK_WEB*>> is used to define the presentation layer (activity diagram related to each HTML mockup);

– <<*PK_SERVICE*>> is used to define services (operations and their respective activity diagrams as internal behaviors);

– <<*PK_ROLES*>> is used to define user roles and use cases (UML *Use Case Diagrams* are used for that).

As expected, screens transformed into pages in the "Forward Project" (Figure 9.19, right-hand side) concomitantly exist as UML activity diagrams (Figure 9.19, left-hand side) in the <<*PK_WEB*>> package of the "Model Project". This strong attachment allows BLU AGE to modernize screens from the other model pieces: the application's business logic that is itself connected with the data management tiers.

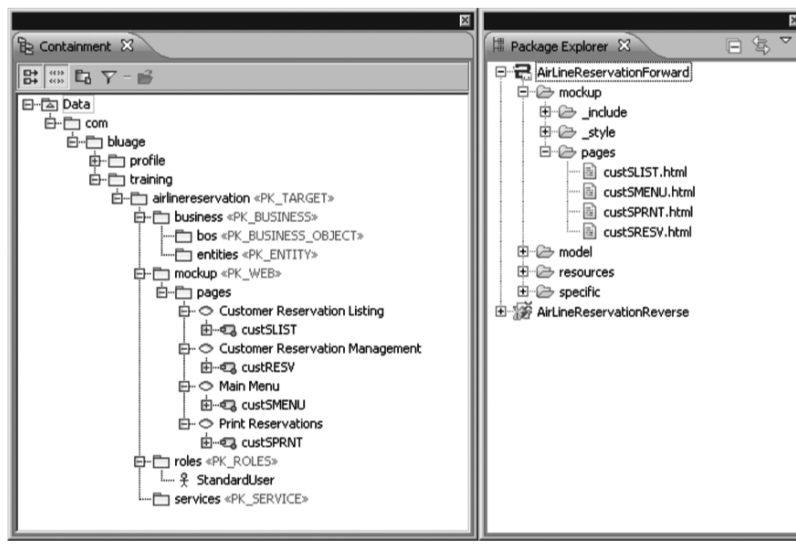


Figure 9.19. *Model and HTML mockups for a forward project*

9.2.6. *Entity extraction*

In the list of candidate data items for modernization, "CUSTOMER-RECORD" must become an entity, i.e. a UML class (named "CustomerRecord", for instance) to be located in the *<<PK_BUSINESS>>* package. Figure 9.20 shows the result of modernization: "CustomerRecord" as new entity and "CustomerRecordBO" as its direct subclass with elementary data access operations.

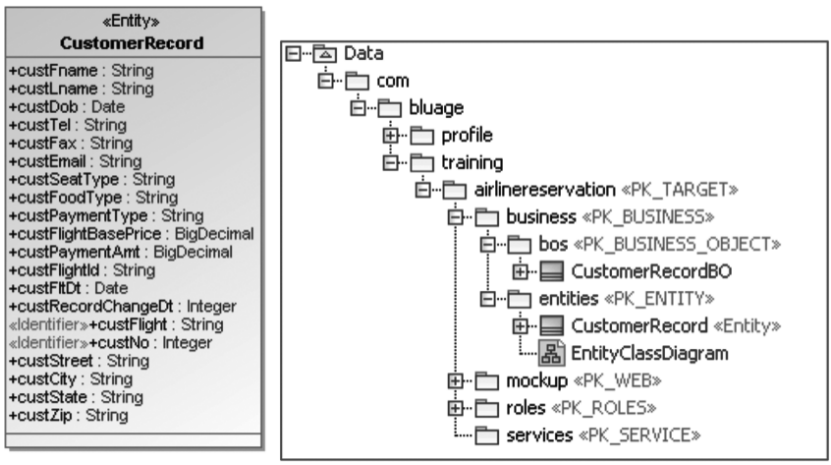


Figure 9.20. *"CustomerRecord" and "CustomerRecordBO" as modernized entities*

Legacy fields are subject to a default treatment or specific actions from users. For example, there is an initial composite primary key ("CUST-RES-ID = CUST-FLIGHT + CUST-NO") defined in the *INPUT-OUTPUT SECTION* of the "custDBIO" program. Besides, data item types may change, groups of data items may be flattened, e.g. "CUST-ADDRESS" in Figure 9.21 has disappeared in Figure 9.20 for the benefit of its subfields ("custStreet", etc.). This is the same for "CUST-RES-ID". Beyond, its two subfields have been marked with the *<<Identifier>>* stereotype to abstract the COBOL composite primary key.

Applying this principle of all of the data items in Figure 9.16 leads to the UML class diagram in Figure 9.22. Note that "COMMON-WORK-AREAS" from the "custRESV" program is marked as *TransientObject* as advised in section 9.2.4. This is the same for "CUSTLOG-AUDIT-INFO", which does not aim at persisting.

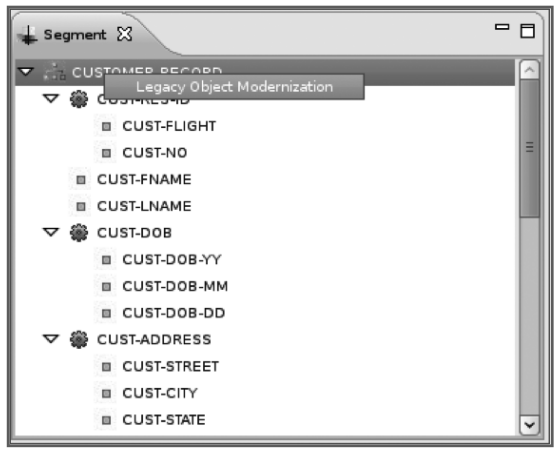


Figure 9.21. *Data item modernization using BLU AGE segment view*

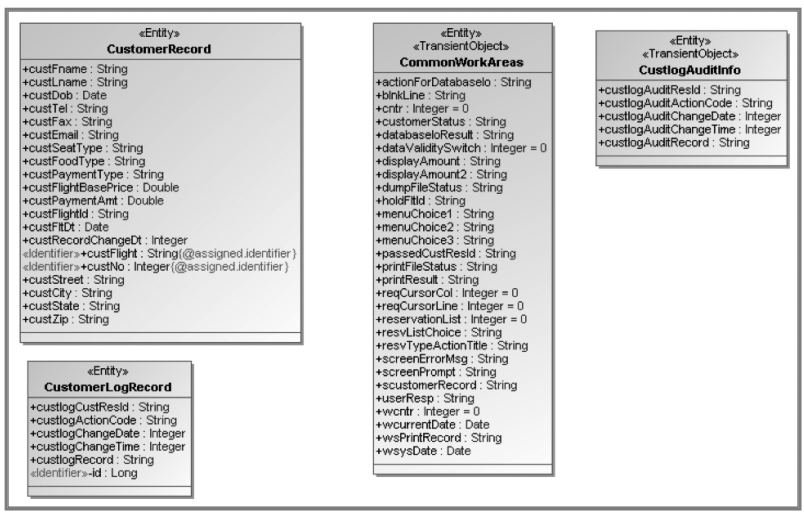


Figure 9.22. *Extracted entities*

9.2.7. *From screens to pages and UI components*

In the legacy application, navigation between screens is grounded on the "Menu" screen (Figure 9.2). Old pages have a "Travels" header and display a current date (see, for instance, Figure 9.3). In the modernized application, reusing Web UI components is put into practice.

Typically, new pages are designed with header and footer components. These components are useful to:

– define global navigation (user logout, tabs suite ("Menu", "Add", etc., tabs on the top right corner of Figure 9.56) to directly access to other pages, etc.);

– store and manage user session by means of the *CommonWorkAreas* singleton object named "context" (this object stores the context and should be placed as member of the Web session);

– display the current date as done in the legacy application;

– display error messages at the bottom of pages.

Web UI components are inserted in the application's pages as done in Figure 9.23. In BLU AGE, the behavior of each page is also modeled as an activity diagram. So, the application's interaction inside pages (from pages to pages as well) is specified by means of activity diagrams using UI components' ID like "header" in Figure 9.23.

G-MARKER is a third-party tool of the BLU AGE suite to concomitantly design Web pages and their assigned model pieces (activity diagrams) so that consistency reigns.

9.3. Annotations

Annotations apply to COBOL programs (elements with the *.cblmf* suffix in Figure 9.12). Opportunistically, the first posed annotations

are code documentation (Figure 9.24) so that the forthcoming code transformations may benefit from a better trace.

```
...
<body>
   <form method = "get" name="my_form" action = "" id="my_form" >
      <!-- ghost -->
      <div id="rd_ghost" >
         <small>
            _ css deactivated </small>
      </div>
      <div id="rd_jghost" >
         <small>
            _ javascript deactivated </small>
      </div>
      <!-- end ghost -->
      <!-- shape -->
      <div id="rd_shape" >
         <!-- brand -->

          <component id="header" name="header" data-role="BluageComponent">
             <attribute name="context" data-instancename="context" data-tablefield="CommonWorkAreas"/>
          </component>

         <!-- end brand -->
         <!-- content -->
         <div id="rd_content" >
...
```

Figure 9.23. *Header component included in an HTML mockup*

```
227
228       Main Section.
229           CALL "CSPID" USING PID.
230           DISPLAY "PID = " PID.
231           ACCEPT W-SYS-DATE                FROM DATE YYYYMMDD.      Get system date to be displayed
232           MOVE W-SYS-YY                    TO  CURR-YY.
233           MOVE W-SYS-MM                    TO  CURR-MM.
234           MOVE W-SYS-DD                    TO  CURR-DD.
235
236           MOVE "CREATE"                    TO ACTION-FOR-DATABASE-IO.
```

Figure 9.24. *Annotated code with comment*

Many annotations are for documentation and readability. Only three annotations have a great impact on code transformation:

– "Batch" is an annotation prefix used for batch programs only (see also Figure 8.7);

– "Skipped" and "Modernized As" are control annotations for transmodeling. The latter requires additional parameter values to indicate the precise envisaged modernization.

9.4. Pattern definition

In BLU AGE, a pattern is identified (Figure 9.25) and later characterized (Figure 9.26) to leverage transmodeling (i.e. the conversion of legacy code pieces to UML model elements) in an efficient way: code pieces obeying to the same pattern are transmodeled in the same way to address scalability issues of modernization. "The same way" means that the same (duplicated) code suite may be refactored as a call to an operation, i.e. an instance of the UML *CallOperationAction* metatype.

9.4.1. *Pattern for simple statements*

Let us consider the "CALL "C$PID" USING PID" pattern in the "custDRVR" program (Figure 9.24). This code is not intended to be ported to the target because it is a pure technical reference to the old platform. To skip it, we endow a pattern with "Skipped" (Figures 9.25 and 9.26).

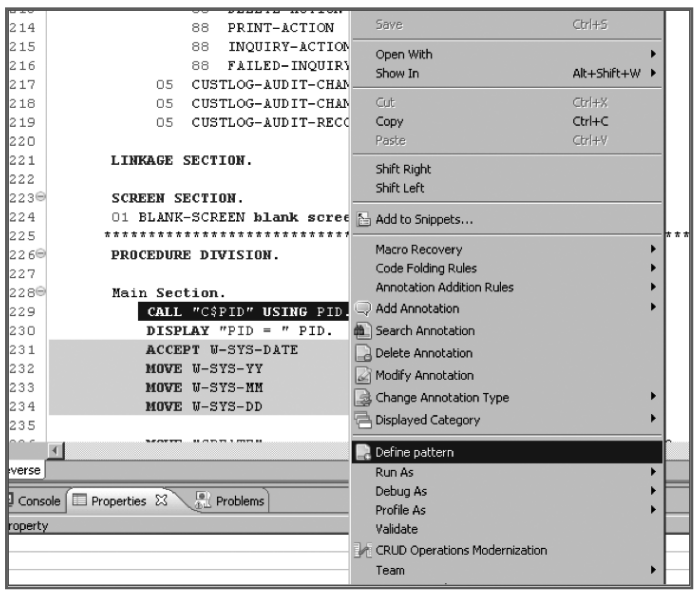


Figure 9.25. *Defining patterns*

Once created, the identified pattern is inserted in the matching algorithm; it may be executed for all of the remaining programs of the legacy application. This leads to generated annotations as that automatically posed in Figure 9.27.

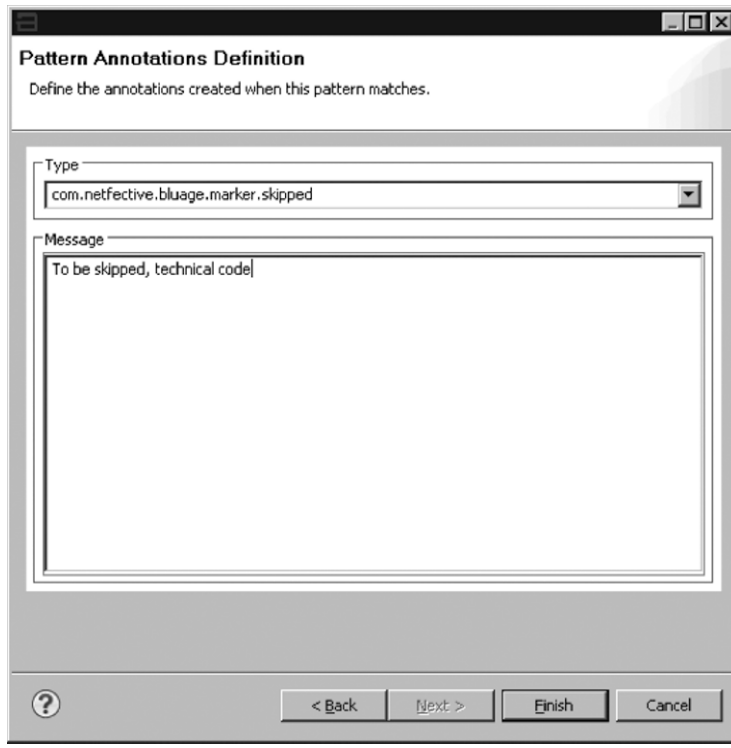


Figure 9.26. *Pattern-type characterization*

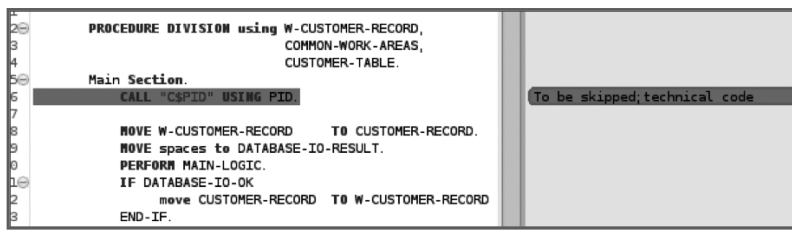


Figure 9.27. *Matched patterns*

9.4.2. *Patterns for operation calls*

Patterns can also be used to control an operation call. Typically, the "custRESV" program includes the following code: "INITIALIZE CUSTOMER-RECORD" (Figure 9.29). Since "INITIALIZE" is a predefined COBOL statement, BLU AGE is able to automatically detect the semantics of this statement, i.e. filling in the "CUSTOMER-RECORD" data item with default standard values, e.g. zero for numerics.

Therefore, this code deals with "CUSTOMER-RECORD" so that the creation of a new instance of the "CustomerRecord" class is required in the modern application. In BLU AGE, this corresponds to an operation returning an instance of "CustomerRecord". More precisely, it is an instance of "CustomerRecordBO" as direct subclass of "CustomerRecord"; polymorphism applies. Practically, this first leads to manually add to the UML model, a new UML *Interface* object[1] named "ServiceInitialize" in the package stereotyped *<<PK_SERVICE>>* (Figure 9.28). Next, this service is, by means of devoted wizards, automatically equipped with a contained element (see hierarchy in Figure 9.28): this is an operation named "initCustomerRecord" (a.k.a. "ServiceInitialize.initCustomerRecord") with a return parameter whose type is "CustomerRecordBO". The activity diagram (Figure 9.28, right-hand side) shows the resulting UML model piece. This activity diagram as member of "ServiceInitialize" is also named "initCustomerRecord". Formally, the overall diagram is interpreted by BFE as the behavior of "ServiceInitialize.initCustomerRecord".

By convention, an activity diagram and its first executed action (i.e. an instance of the UML *CallOperationAction metatype*) must share the same name. This activity diagram is finally linked to the "ServiceInitialize.initCustomerRecord" operation as described in section 9.5.

1 The BLU AGE metalanguage embodies "services" as UML *Interface* objects, *Interface* being a predefined UML metatype.

Once characterized, this pattern is matched to many other code pieces in other programs leading to generated annotations as shown in Figure 9.29. All matched legacy pieces are definitely replaceable by a routing to the abstract (i.e. technology-free) behavior in Figure 9.28 (right-hand side).

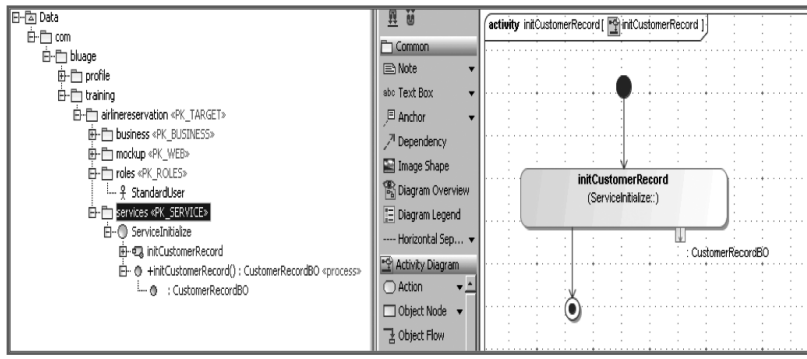


Figure 9.28. *Created service and contained operation*

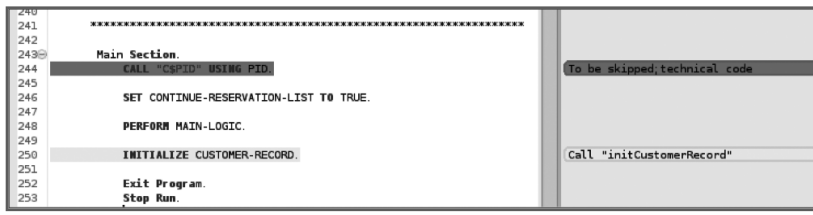


Figure 9.29. *Matched pattern for an operation call*

9.4.3. *Patterns for operation calls with arguments*

Patterns are also used where a part of the legacy code is fixed while the other is variable (“CUST-RES-ID” and “NFD” in Figure 9.30). Moving such a code piece to the UML model is similar to what is described in prior section (the creation of a UML *Interface* object named “ServiceAudit” with a contained operation name

"logUserAction" in Figure 9.31). In addition, COBOL variables must be bound to UML variables in activity diagrams. Typically, "CUST-RES-ID" is bound to the newly introduced "CustomerRecord" UML variable while "NFD" is bound to "ActionCode". Both variables appear as input variables of the "logUserAction" activity in Figure 9.31.

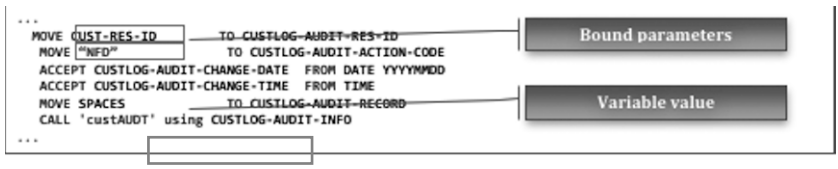


Figure 9.30. *Operation call with arguments*

As for "SPACES", it is not a COBOL variable (it is a COBOL constant). Nonetheless, we should have the possibility of replacing it by something else in another code piece matching the pattern under construction. BLU AGE thus invites us to set "SPACES" as something "variable" in the designed pattern (Figure 9.32).

The pattern matching algorithm is such that code pieces matching the pattern may have something different from "SPACES". This is the case of the code piece in Figure 9.33 that is modernized as the activity diagram in Figure 9.31.

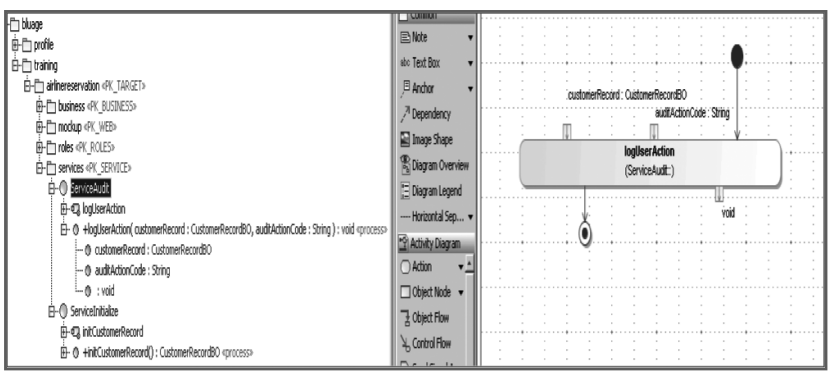


Figure 9.31. *UML model piece for operation call with arguments*

9.4. Database exchange modernization

Data persistence-related statements are part of the "custDBIO" program. This program has functions, which are used by other programs to create, retrieve, update or delete data from the persistence storage (flat files). These functions have to be modernized using Create, Read, Update, Delete (CRUD) operations. Since Hibernate has been set up as the persistence framework for this modernization project, BLU AGE allows the expression of CRUD operations in Hibernate Query Language (HQL).

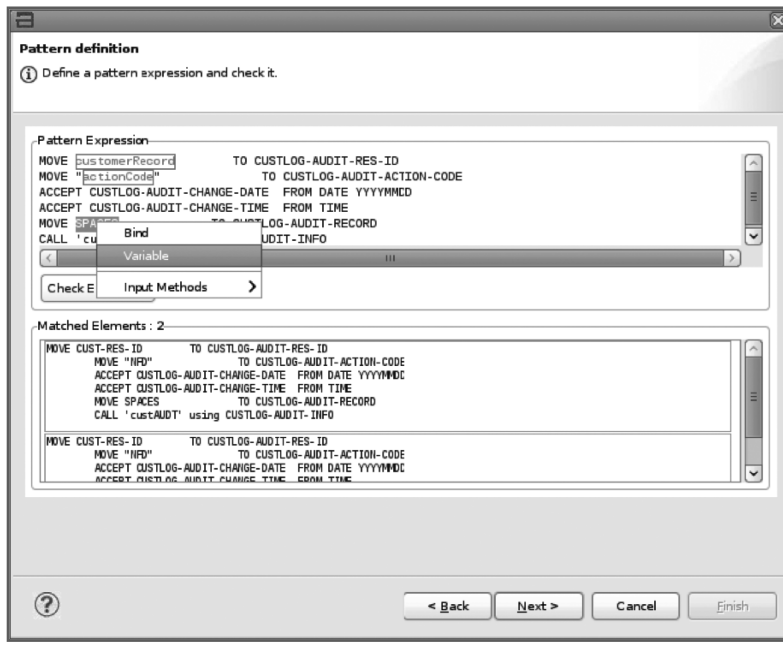


Figure 9.32. *Adding local variable to a pattern*

Practically, the "custDBIO" program owns a "MAIN-LOGIC" section (Figure 9.34). The surrounding box in the figure is the persistence storage creation and does not need to be ported to the modern target because Hibernate, as persistence framework, supports such a creation in a transparent way.

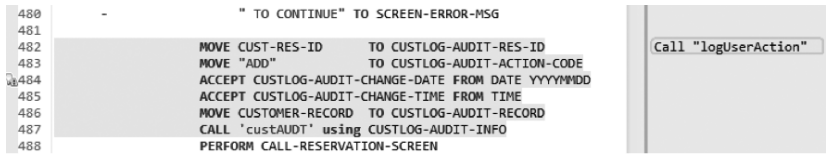


Figure 9.33. *Matched pattern for an operation call with arguments*

```
266
267       MAIN-LOGIC.
268
269           IF  DATABASE-CREATE
270               OPEN I-O CUSTOMER
271               IF  NOT-PRESENT
272                   MOVE "CREATED"
273                       TO   DATABASE-IO-RESULT
274                   OPEN output CUSTOMER
275               END-IF
276               CLOSE CUSTOMER
277           END-IF.
278
279           IF  DATABASE-ADD
280               PERFORM ADD-CUST-RES
281           END-IF.
282
283           IF  DATABASE-CHANGE
284               PERFORM CHANGE-CUST-RES
285           END-IF.
286
287           IF  DATABASE-DELETE
288               PERFORM DELETE-CUST-RES
289           END-IF.
290
291           IF  DATABASE-READ
292               PERFORM READ-CUST-RES
293           END-IF.
294
295           IF  DATABASE-READ-EQUAL
296               PERFORM READ-CUST-RES
297           END-IF.
298
299           IF  DATABASE-READ-GREATER-EQUAL
300               PERFORM READ-CUST-RES-GE
301           END-IF.
302
303           IF  DATABASE-READ-NEXT
304               PERFORM READ-CUST-RES-NEXT
305           END-IF.
306
307           IF  DATABASE-CLOSE
308               CLOSE CUSTOMER
309           end-IF.
```

Figure 9.34. *Database exchange program*

In the same line of reasoning, the "READ-CUST-RES-GE" and "READ-CUST-RES-NEXT" *PERFORM* occurrences (final two entries in Figure 9.35) can be omitted ("Skipped" annotation). These are used to fetch the next or previous customer record, something automatically handled by BLU AGE at the model level and transmitted to Hibernate at the implementation level.

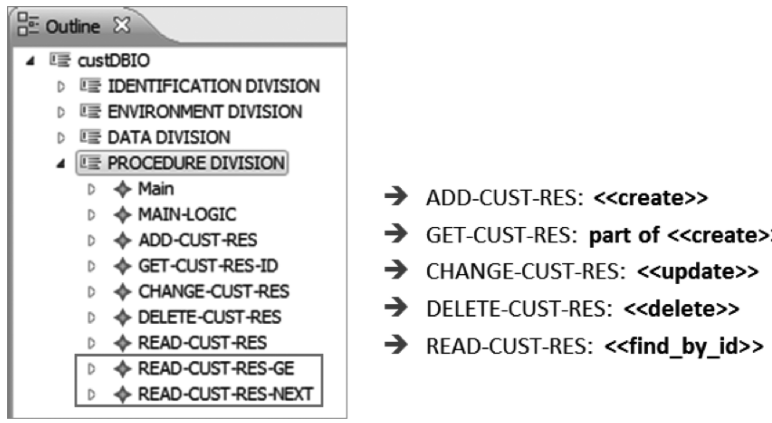


Figure 9.35. *Database exchange actions to be modernized*

Figure 9.35 shows the original functions to be modernized and their action semantics from a CRUD perspective. For example, the "ADD-CUST-RES" block is marked with the <<*create*>> stereotype. In this context, "ADD-CUST-RES" is later annotated as shown in Figure 9.36, provided that the availability of the "customerRecordCreateWithKey" UML operation.

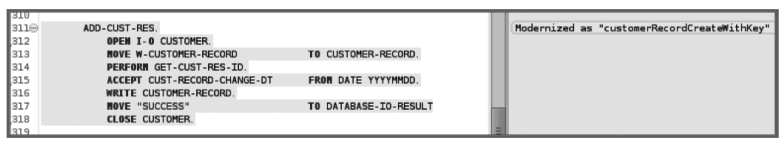


Figure 9.36. *Annotation for "ADD-CUST-RES"*

In fact, several elementary UML operations have to be previously designed as member of, again, a UML *Interface* object (a service in BLU AGE) here named "ServiceCustomerRecord" (Figure 9.37). For example, the "customerRecordGetKey" operation (second operation in Figure 9.37) is designed and stereotyped with <<*hql_operation*>> so that we may assign to it the following HQL query: "SELECT max(custNo) from CustomerRecordBO WHERE custFlight = custFlightId".

Once defined, these core operations contribute to modeling the behavior of "customerRecordCreateWithKey" as represented in the activity diagram in Figure 9.38.

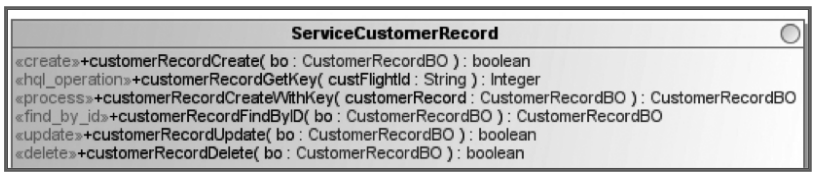


Figure 9.37. *"ServiceCustomerRecord" as UML Interface object (BLU AGE service)*

9.5. Transmodeling

In BLU AGE, the transformation of tricky code portions into UML model pieces may require the manual design of the latter. For other (more easily identifiable) portions, pattern matching is a significant helper to assign very similar (intelligible) portions to the same model piece, dealing (or nor) with variable parts (see above). Pattern matching avoids, as much as possible, manual intervention. In this scope, pattern matching works in high conjunction with automatic transmodeling.

In general, the main principle behind transmodeling is the direct generation of a model piece (an activity diagram in most cases) from a code sequence or scattered code blocks (the case of "initMenuPage" below). Transmodeling suppresses the manual design of the

equivalent model piece except, sometimes, its completion (see Figure 9.42). Because of the very deep peculiarity of the code (UI, persistence, operating system, ordinary computation, etc.), transmodeling is subject to varied strategies.

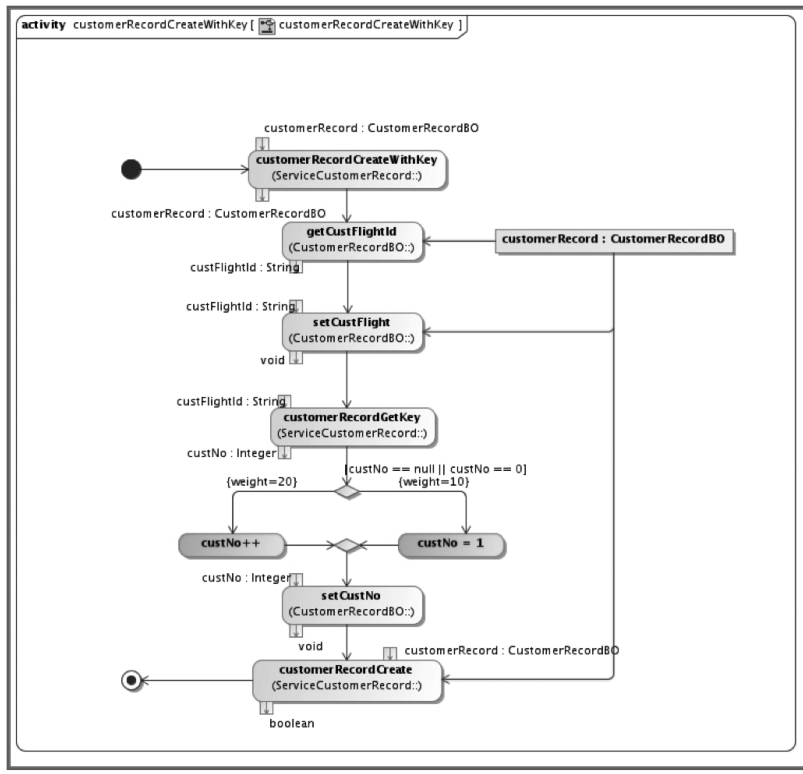


Figure 9.38. *Nominal behavior to create a customer record with key*

As an illustration ("custDRVR" program), Figure 9.39 shows two annotated sections (top of figure) that must be retained (marked "Retained") with a specific strategy while another (bottom of figure) is devoted to another kind of transmodeling.

The first two retained code portions (line 231 and lines 249–251) are parts of the same business logic. They are modernized as part of an activity diagram, which is described in Figure 9.40, left-hand side. This behavior is used within a precontroller (Figure 9.40, right-hand side).

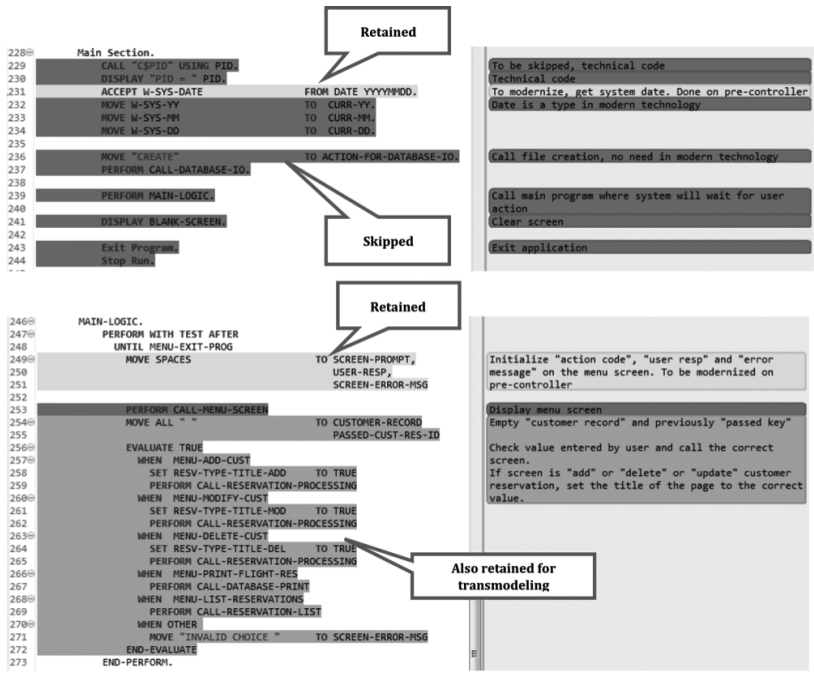


Figure 9.39. *Annotated code with different modernization strategies*

As already mentioned, a precontroller must realize initializations before the application's home page is opened. Initialized elements are stored in the context of a "CommonWorkAreas" transient object. Remember that the "CommonWorkAreas" UML class in Figure 9.22 has only one instance named "context" in the renewed application. Typically, this instance owns the "screenErrorMsg" attribute (see also Figure 9.22) to embody the "SCREEN-ERROR-MSG" data item in

Figure 9.39 (line 251). Put simply, the behavior of a precontroller is depicted as an activity diagram including, among others, the "SetScreenErrorMsg" activity (Figure 9.40, left-hand side) that initializes "screenErrorMsg". In UML, this component is factorized, once and for all, to be run at application's start-up and, more generally, when we go back to the application's home page.

Again, the linking of the three activity diagrams in Figure 9.40 relies on the use of the *CallBehaviorAction* and *CallOperationAction* UML metatypes as follows: the activity diagram number 3 is the modern vision/design of the old "custSMENU" screen ("custSMENU.cblmf" program file in Figure 9.12). The "custSMENU.html" file in Figure 9.12 is in particular the transcription of this legacy artifact as an HTML mockup. This screen has to be initialized before it is displayed. A precontroller action has been added to this diagram (3-a). It is modeled by a *CallBehaviorAction* UML object. The called behavior is defined by the "initMenuPage" activity (activity diagram number 2, right-hand side). This activity also contains the "initMenuScreen" *CallOperationAction* object (2-b) representing the operation to execute. The associated operation is modeled by another activity (activity diagram number 1).

In other words, in Figure 9.40 (top right-hand side) one may observe the compacted formalization of the precontroller. "initMenuScreen" is an activity, which is member of the "ServiceInitialize" UML *Interface*. "initMenuScreen" is also an operation whose process is specified in Figure 9.40, left-hand side. An instance of *CallOperationAction* creates the bridge from the "initMenuScreen" activity to the "initMenuScreen" operation. By convention, the first activity in the activity diagram representing the operation has the same name that the modeled operation. In other words, the first activity in Figure 9.40, top on left-hand side, is named "initMenuScreen".

The two arrows in Figure 9.40 give the flow of utilization, i.e. where processes as activity diagrams are reused in other processes. As a result, the third activity diagram in Figure 9.40, bottom of right-hand side, only tells us how the navigation to "custSMenu" occurs via "initMenuPage". Following the same line of reasoning, the

"initMenuPage" activity is linked to the "initMenuPage" operation whose behavior is the activity diagram in Figure 9.40, top of right-hand side.

Contrary to line 231 and lines 249–251, the code portion delimited by lines 254–272 is selected so that the transmodeling facility of BLU AGE is used in another manner. Several manipulations within transmodeling result in having the "checkMenuCode" activity diagram in Figure 9.41. Simply speaking, this is the process of moving to the "good" screen ("Add a new reservation", etc.) according to the user's choice.

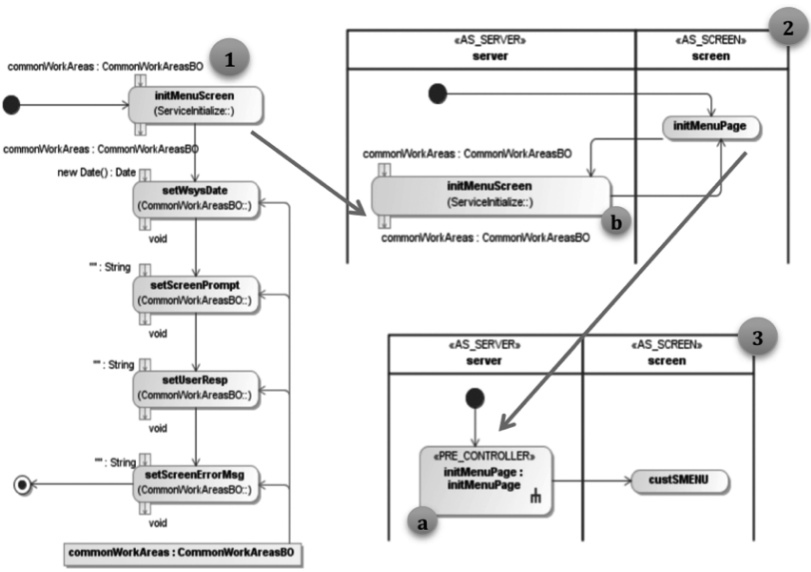


Figure 9.40. *UML activity diagrams showing precontroller behavior*

Activities named "// TODO …" are templates; they are subject to completion. For example, the upper template activity in Figure 9.41 is expanded as a subprocess. In Figure 9.42, the first introduced activity of this subprocess is "initCustomerRecord" (see also section 9.4.2). Again, using *CallOperationAction*, this corresponds to calling the predefined "initCustomerRecord" operation.

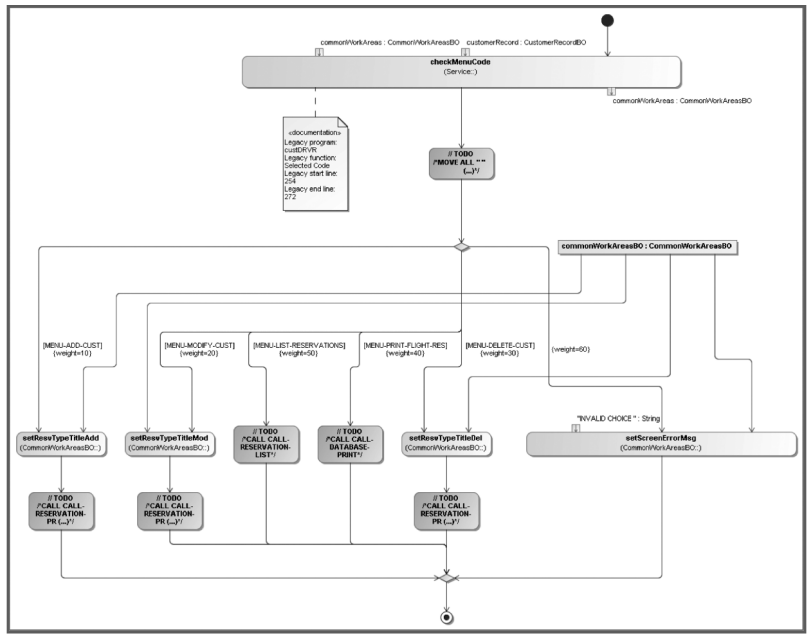


Figure 9.41. *UML activity diagram as a result of transmodeling lines 254–272*

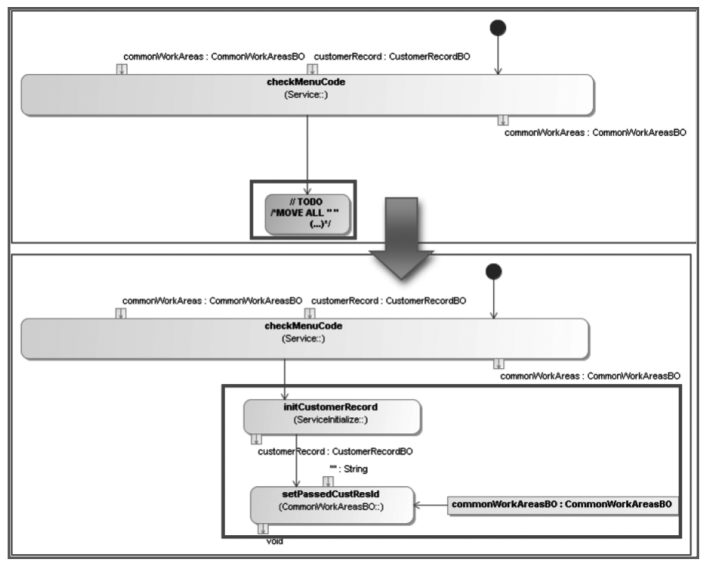


Figure 9.42. *Replacing "TODO activities" with appropriate operation calls*

Incrementally, the UML model, as a result of "reverse modeling", is a set of services extracted from the legacy code (Figure 9.43). The careful design of services is important. Indeed, transmodeling automatically names services and creates content, which later requires renaming. For example, "ServiceMenu" is introduced as the name for the service encompassing the "checkMenuCode" operation and associated activity diagram (operation's detailed behavior).

Service elements are common callable model elements. As an illustration, Figure 9.44 shows how this new service is called within the "InitMenuPage" activity diagram in order to enhance the activity diagram in Figure 9.40, bottom of right-hand side.

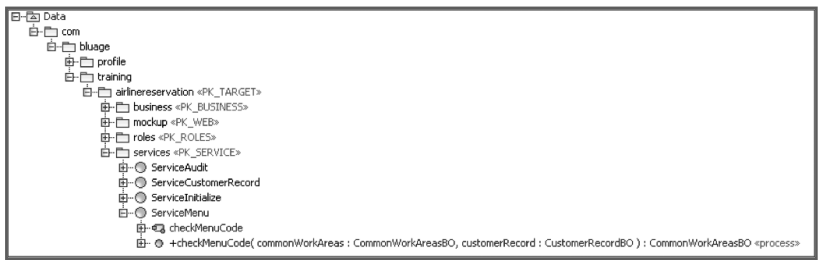


Figure 9.43. *Tree structure for "ServiceMenu" and "checkMenuCode" as contained element (activity diagram and associated operation)*

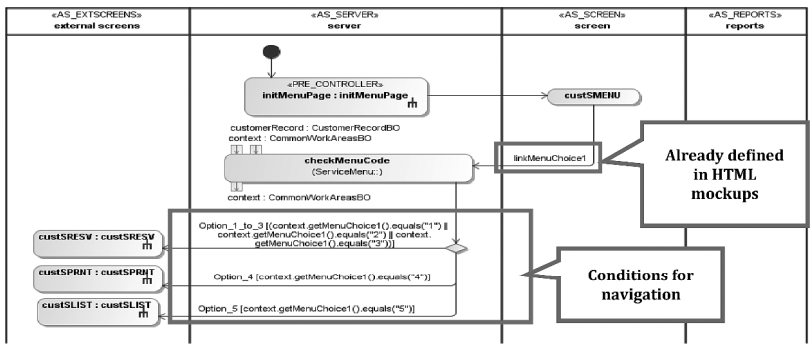


Figure 9.44. *Processes' enhancement*

Figure 9.44 also shows how variables in HTML mockups are manipulated in models. This occurs with a synchronization (consistency checking) that is ensured by the G-MARKER third-party tool.

9.6. Transmodeling complex functionalities

The "Add a new reservation" function is a representative key function of the legacy application. The current behavioral logic behind it is as follows:

– if the user chooses "1", "2" or "3", the "custRESV" program is called from the "custDRVR" program;

– the "MAIN-LOGIC" paragraph in "custRESV" (Figure 9.45, line 274) checks the type of action to be performed. "ADD-CUST-RES" embodies the choice of "Add a new reservation". The business logic behind "ADD-CUST-RES" is then as follows:

- initializing a new customer record ("CUSTOMER-RECORD", line 288);

- positioning the cursor (lines 291–292);

- displaying the screen (line 293);

- calling a code block named "EDIT-INPUT-FIELDS" to manage users' input text (line 295);

- if "SAVE-REC" is active (lines 298–330) then:

- set the action code (e.g. "MOVE "ADD "…") and call the "CALL-DATABASE-IO" paragraph (lines 298–300);

- log user's actions (lines 302–307);

- clean up the "USER-RESP" and "SCREEN-PROMPT" screen zones (lines 309–310);

- display the result message and redisplay the screen (lines 311–313);

- clean up "USER-RESP" again (line 315);

- display a message to be printed or exit; according to user's choice, go to the "Print flight reservation" screen (lines 317–329);

- reinitialize "CUSTOMER-RECORD" (line 330).

The above code analysis enables annotating of the COBOL "custRESV" program as done in Figure 9.45.

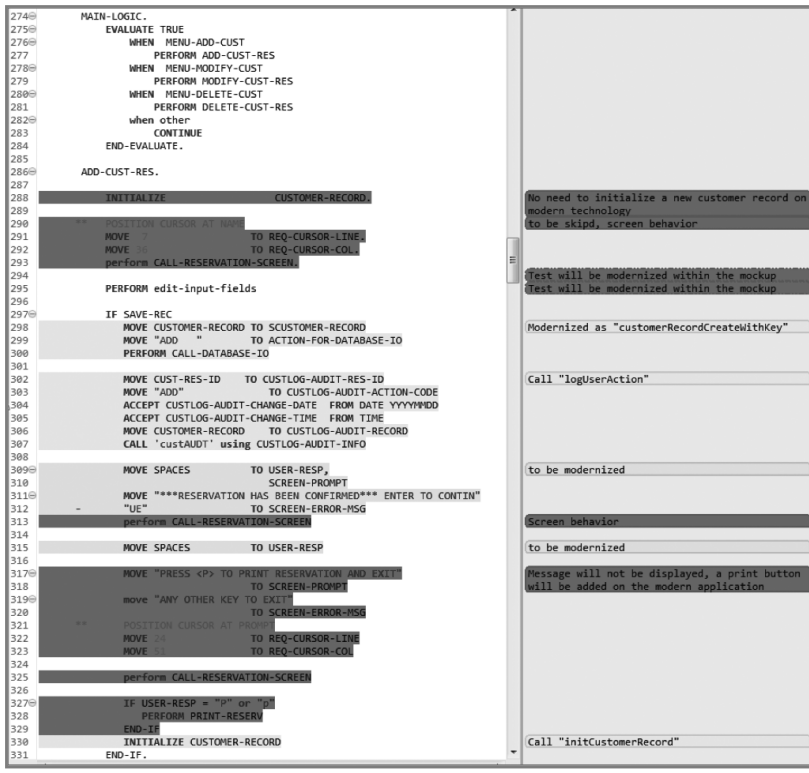


Figure 9.45. *Annotated code for the "custRESV" program*

In line 295 (Figure 9.45), the call to the "EDIT-INPUT-FIELDS" paragraph in particular allows the validation of the values entered by the user. In the modern application, data validation is grounded on integrated validators. They are already parts of HTML mockups; accordingly, line 295 is not modernized (Figure 9.46).

"EDIT-INPUT-FIELDS" also checks for the customer payment method. To that extent, it may call the "custCOST.cblmf" program file to calculate the costs of booking a flight. More precisely, "PERFORM CALL-CALCULATE-PRICE" in Figure 9.46 (line 748) embodies the call to "custCOST" for calculating this.

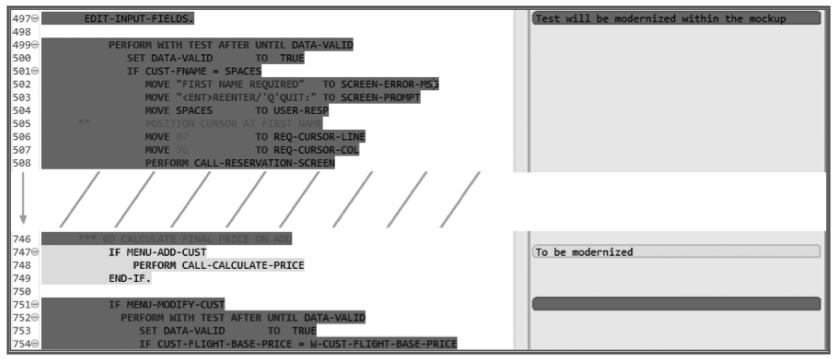


Figure 9.46. *"EDIT-INPUT-FIELDS" paragraph source in "custRESV.cblmf"*

9.6.1. *Transmodeling the "custCost" program*

The steps involved for modernizing "custCOST" cover the transformation of the "WS-COST-FIELDS" data item into to a UML class named "WsCostField" (Figure 9.47). The way to do so is explained in section 9.2.6.

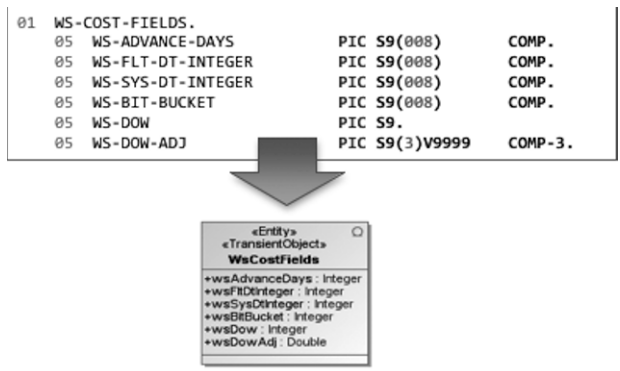
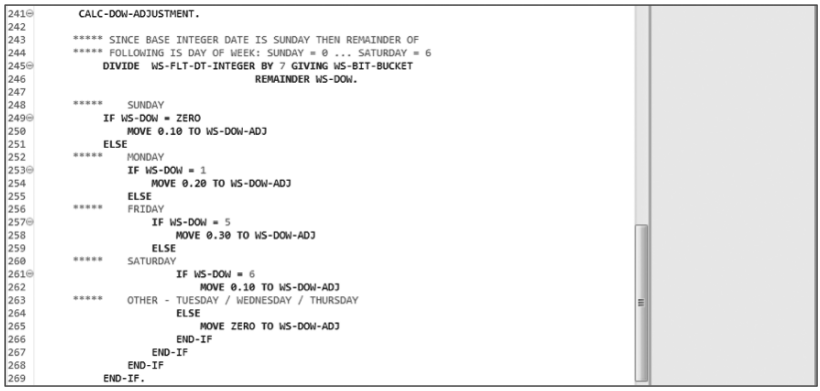


Figure 9.47. *"WS-COST-FIELDS" modernization*

Another step is the modernization of the "CALC-DOW-ADJUSTMENT" paragraph (lines 241–269 in Figure 9.48).



```
241       CALC-DOW-ADJUSTMENT.
242
243       ***** SINCE BASE INTEGER DATE IS SUNDAY THEN REMAINDER OF
244       ***** FOLLOWING IS DAY OF WEEK: SUNDAY = 0 ... SATURDAY = 6
245            DIVIDE  WS-FLT-DT-INTEGER BY 7 GIVING WS-BIT-BUCKET
246                                           REMAINDER WS-DOW.
247
248       *****    SUNDAY
249            IF WS-DOW = ZERO
250               MOVE 0.10 TO WS-DOW-ADJ
251            ELSE
252       *****    MONDAY
253               IF WS-DOW = 1
254                  MOVE 0.20 TO WS-DOW-ADJ
255               ELSE
256       *****    FRIDAY
257                  IF WS-DOW = 5
258                     MOVE 0.30 TO WS-DOW-ADJ
259                  ELSE
260       *****    SATURDAY
261                     IF WS-DOW = 6
262                        MOVE 0.10 TO WS-DOW-ADJ
263       *****    OTHER - TUESDAY / WEDNESDAY / THURSDAY
264                     ELSE
265                        MOVE ZERO TO WS-DOW-ADJ
266                     END-IF
267                  END-IF
268               END-IF
269            END-IF.
```

Figure 9.48. *"CALC-DOW-ADJUSTMENT" paragraph*

The interesting point in BLU AGE is its capacity to cope with algorithmic constructs such as *IF-THEN-ELSE* and others. Remember that in section 7.3 in Chapter 7, such simple control statements in programming languages may possibly raise strong difficulties in terms of modeling. When setting up ASTM in BLU AGE, the very peculiar grammar of the legacy language is fully understand so that BLU AGE does not stumble over, for instance, the transformation of "CALC-DOW-ADJUSTMENT" into a model.

Transmodeling "CALC-DOW-ADJUSTMENT" is shown in Figure 9.49 and Figure 9.50. Again, "ServiceUtils" is the name assigned by the BLU AGE engineer to the generated service from "CALC-DOW-ADJUSTMENT". "WsCostField" ("WsCostFieldBO" as direct subclass) objects are input and/or output of activities to compute the costs of booking a flight.

Going on with transmodeling leads to the modernization of the "MAIN-LOGIC" paragraph in "custCOST" (Figure 9.51, lines

177–239). In Figure 9.52, the modernized "MAIN-LOGIC" paragraph is assigned to "ServiceCustomerRecord" both with an operation and an activity diagram describing the behavior of this operation. Both have the "calculatePrice" name.

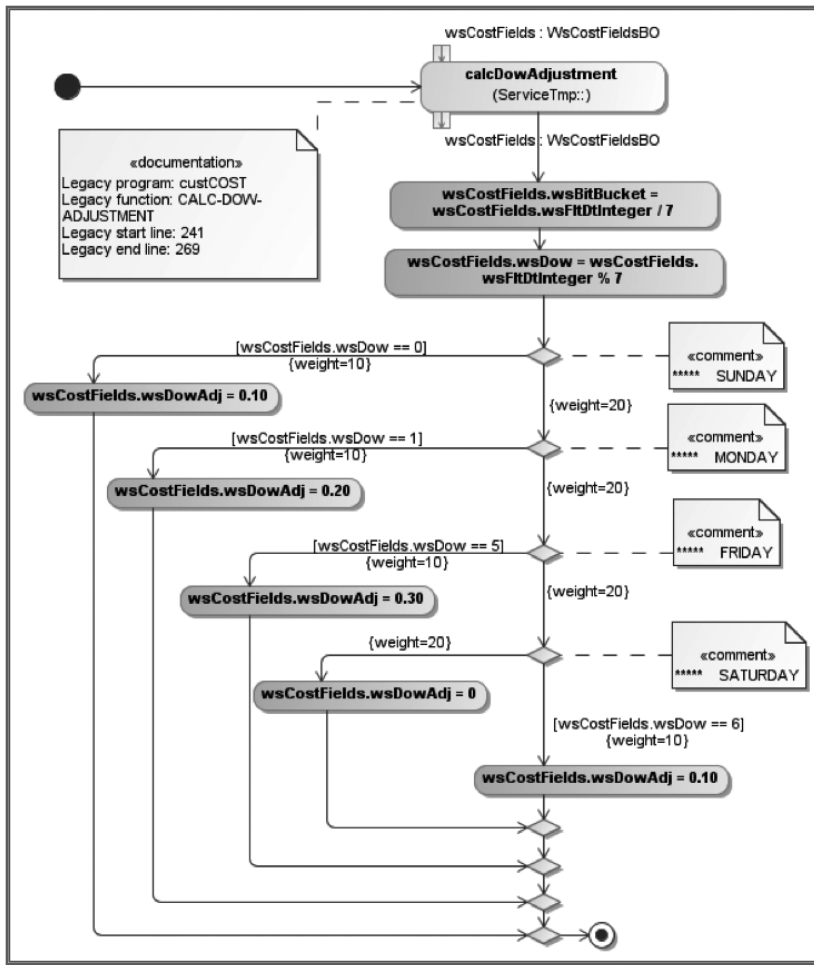


Figure 9.49. *Transmodeled "CALC-DOW-ADJUSTMENT" paragraph*

Figure 9.50. *Tree structure for "ServiceUtils" and "calcDowAdjustment" as contained element (activity diagram and associated operation)*

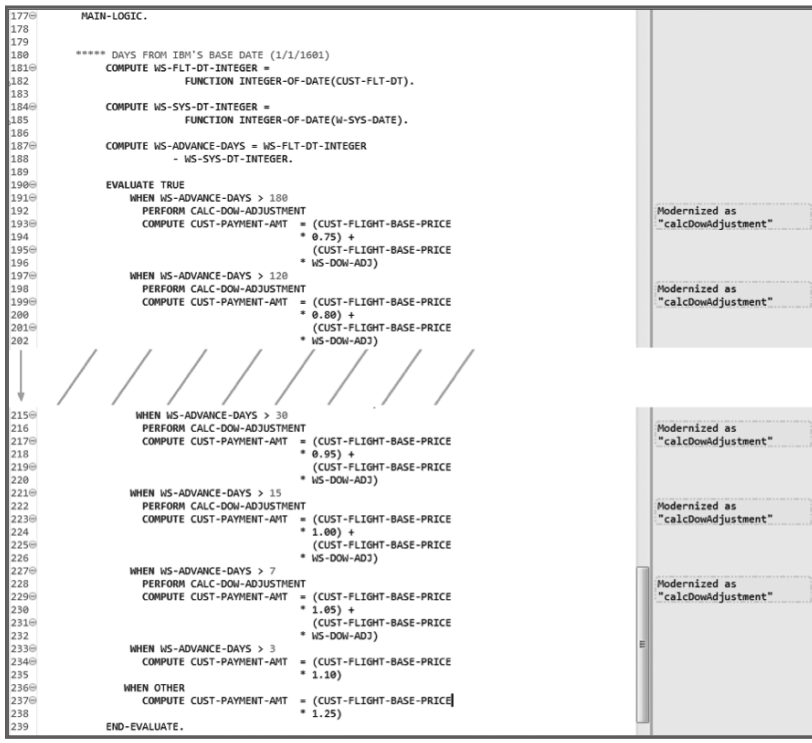


Figure 9.51. "*MAIN-LOGIC" code to be modernized as "calculatePrice" operation*

Figure 9.53 shows the final result after having substituted activities named "// TODO …" for calls to concrete operations. For instance, "getCustFltDt" is an operation of a database access object (a.k.a.

"business object" with "BO" suffix) offered by a BLU AGE. Accordingly, "customerRecord" is injected as parameter for this operation to retrieve the good flight date of a given customer.

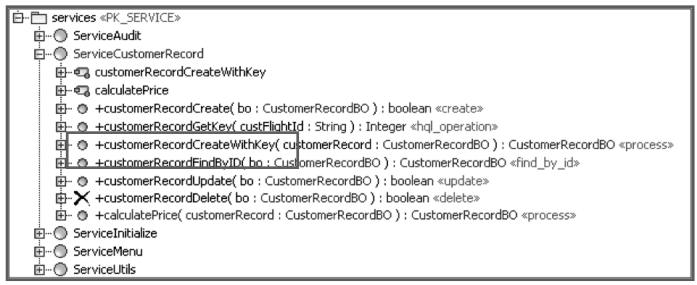


Figure 9.52. *Tree structure for "ServiceCustomerRecord" and "calculatePrice" as contained element (activity diagram and associated operation)*

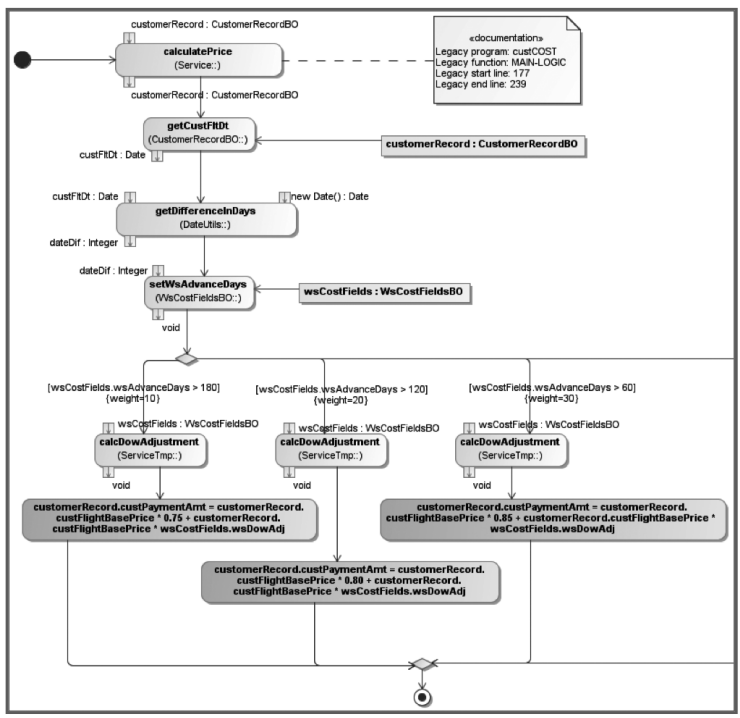


Figure 9.53. *Changes operated for "MAIN-LOGIC" as transmodeled paragraph*

9.6.2. *Modernizing "Add a new reservation"*

Step-by-step, macrooperations are constructed from microoperations to have the possibility of modernizing the overall "custRESV.cblmf" program file. When all functionalities required within "ADD-CUST-RES" (Figure 9.45, from line 286 to line 331) are modernized, "ADD-CUST-RES" can itself be transmodeled as shown in Figure 9.54.

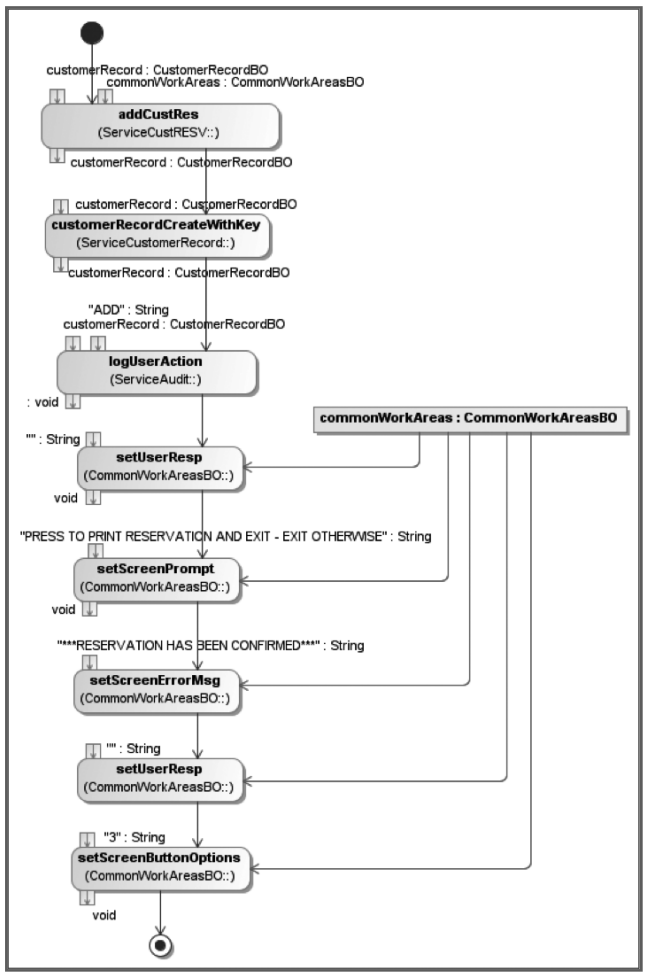


Figure 9.54. *"ADD-CUST-RES" behavior*

The “Add a new reservation” process in Figure 9.54 must then be linked to the “custSRESV” activity diagram (Figure 9.55) depicting the behavior of the devoted page for booking reservations.

The very final business service that may, for instance, be exposed as a Web service is named “ServiceCustRESV” in the UML model tree structure. Such an approach allows the smart progressive transformation of the legacy application as a Service-Oriented Architecture (SOA)-like modern application.

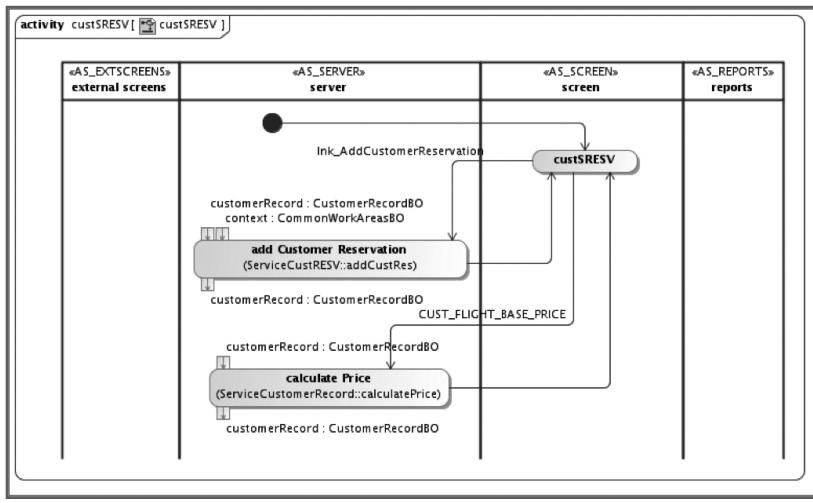


Figure 9.55. *Fully modernized “custSRESV” program with UI interaction*

9.7. Application generation and testing

Four key folders are created from scratch as the result of running the application generation process:

– *AirLineReservationForward-entities*;

– *AirLineReservationForward-service*;

– *AirLineReservationForward-tools*;

– *AirLineReservationForward-web*.

These folders include all of the necessary directories and files (source, configuration (*JavaServer Faces* (JSF), etc.)) embodying the modernized application. These files are very common files (Java, XML, JSF, etc.) and can be manipulated outside BLU AGE in any Integrated Development Environment (IDE). Moreover, maintenance may occur from this point when people want to leave modeling (however, this is not advisable). BLU AGE also includes a set of facilities for setting up database servers, applications servers like Apache Tomcat (it is used here to deploy the subject application based on Spring) or Java EE-compliant servers: Apache TomEE, GlassFish, JBoss, etc.

Partly or wholly testing the application later occurs through Web pages. For instance, entering "1" (the original behavior in the COBOL "Menu" screen) within the application's home page (Figure 9.56) leads to the "Add a new reservation" page (Figure 9.57). Entering customer and flight information plus a value for "BASE PRICE" leads to the calculation of "PAYMENT AMOUNT". This relies on having "AJAX_ENABLED" set to true in Figure 9.1.

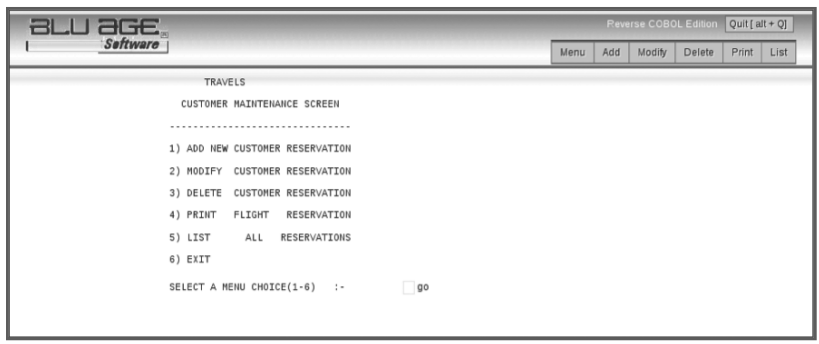


Figure 9.56. *Modernized application (home page)*

9.8. Conclusions

The difficulty of COBOL software modernization is the move from theory (promoted via Model-Driven Development (MDD) and

Architecture-Driven Modernization (ADM)) to practice, facing up large-scale industrial case studies. This chapter tries from a representative case study (*On Line Transaction Processing* (OLTP) COBOL application) to show that modernization strongly depends upon a relevant assistance. Things to do are represented by a plethora of non-homogenous concerns; consistency checking of modernization actions is then a critical task that makes no sense outside a professional tool.

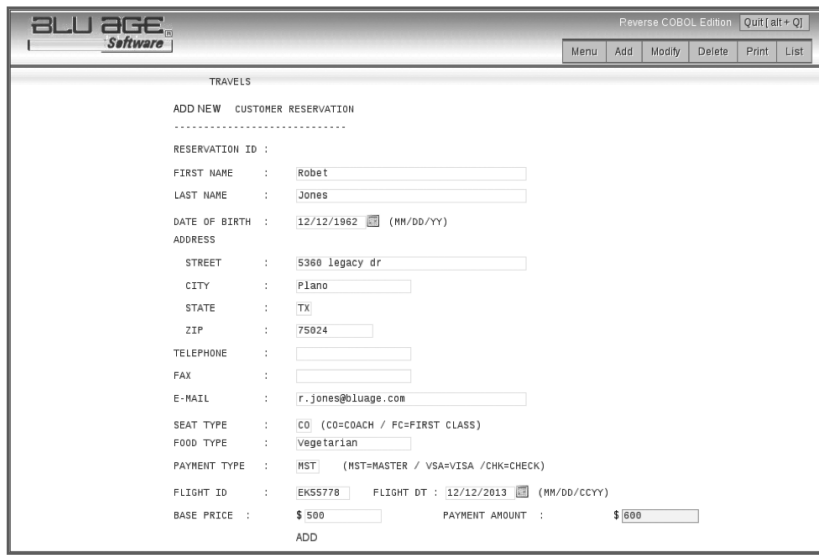


Figure 9.57. *"Add a new reservation" page*

Success thus resides in agility with a true ability to go backward in case of problems. In this context, models play the role of a malleable matter. The great point is the fact that the reversed application in the form of models may move, immediately (or later because of technology evolution), to any platform. The BLU AGE method with integrated BSPs effectively applies the Model-Driven Architecture® (MDA) weaving principle: Platform-Independent Models (PIMs) are

woven with PDMs (i.e. BSPs) to produce Platform-Specific Models (PSMs) linked to the target platform toward the final code.

In BLU AGE, all software artifacts become models, but people may always run them as executable models for tests, even final controls (deployment, test and round trip engineering when possible problems arise) against, in particular, the production environment.

Bibliography

[BAR 84] BARNES J.G.P., *Programming in Ada*, 2nd edition, Addison-Wesley, 1984.

[BAS 00] BASS L., BUHMAN C., COMELLA-DORDA S., *et al*., Volume I: Market Assessment of Component-Based Software Engineering, CMU/SEI-2001-TN-007 Technical Note, Carnegie Mellon University, May 2000.

[BAT 14] BATLAJERY B.V., KHADKA R., SAEIDI A.M., *et al.*, Industrial Perception of Legacy Software System and their Modernization, Utrecht University technical report UU-CS-2014-004, February 2014.

[CHI 90] CHIKOFSKY E., CROSS J., "Reverse engineering and design recovery: a taxonomy", *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[COM 00] COMELLA-DORDA S., WALLNAU K., SEACORD R.C., *et al.*, A Survey of Legacy System Modernization Approaches, Software Engineering Institute Technical Note, April 2000.

[ERO 94] EROSA A., HENDREN, L., "Taming control flow: a structured approach to eliminating Goto statements", *Proceedings of ICCL'94, Toulouse, France*, pp. 229–240, May 16–19, 1994.

[FAV 11] FAVRO J., LAWRENCE PFLEEGER S., "Software as a business", *IEEE Software*, July/August 2011.

[GAL 08] GALLO M., KRUPKA B., Innovation in the Travel Industry: Understanding the Golden Segments for Driving Growth under Uncertainty, Daemon Quest Global Research Center, report, 2008.

[LIE 80] LIENTZ B.P., SWANSON E.B., *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, Addison-Wesley, 1980.

[MCF 12] MCFEDRIES P., *Cloud Computing: Beyond the Hype*, HP Technology Series, 2012.

[NAS 08] NASCIO, Digital States at Risk! Modernizing Legacy Systems, survey, December 2008.

[OMG 99] OMG, COBOL Language Mapping Specification, June 1999.

[OMG 00] OMG, Currency Specification, version 1.0, June 2000.

[ORA 08] ORACLE, Unlocking the Mainframe: Modernizing Legacy Systems to a Service-Oriented Architecture, white paper, June 2008.

[POP 11] POPP K.M., "Software industry business models", *IEEE Software*, July/August 2011.

[RIE 01] RIEHLE D., FRALEIGH S., BUCKA-LASSEN D., *et al.*, "The architecture of a UML virtual machine", *Proceedings of of OOPSLA'01*, ACM Press, pp. 327–341, 2001.

[SEA 02] SEACORD R.C., "Modernizing legacy systems", *Software Engineering Institute News*, December 2002.

[SEL 13] SELIP S., *Healthcare Membership System Modernization – COBOL Pacbase to JEE: High in Quality and Right on Target*, BLU AGE® Corporation, February 2013.

[STE 08] STEINBERG D., BUDINSKY F., PATERNOSTRO M., *et al.*, *EMF – Eclipse Modeling Framework*, 2nd ed., Addison-Wesley, 2008.

[SZY 02] SZYPERSKI C., GRUNTZ D., MURER S., *Component Software – Beyond Object-Oriented Programming*, 2nd ed., Addison-Wesley, 2002.

[ULR 10] ULRICH W.M., NEWCOMB P.H., *Information System Transformations: Architecture Driven Modernization Case Studies*, Morgan Kaufmann, 2010.

[ULR 14] ULRICH W.M., "A status on OMG architecture-driven modernization task force", *Model-Driven Evolution of Legacy Systems workshop in EDOC'14*, 2014.

[VIG 13] VIGNOLA C., Batch Applications for the Java Platform, Version 1.0, Final Release, April 2013.

Index

H, I

J, K, L

M

O, P

Q, R

S

T, U

V, W, X

Other titles from

in

Computer Engineering

2014

BOULANGER Jean-Louis
Formal Methods Applied to Industrial Complex Systems

BOULANGER Jean-Louis
Formal Methods Applied to Complex Systems: Implementation of the B Method

GARDI Frédéric, BENOIST Thierry, DARLAY Julien, ESTELLON Bertrand, MEGEL Romain
Mathematical Programming Solver based on Local Search

KRICHEN Saoussen, CHAOUACHI Jouhaina
Graph-related Optimization and Decision Support Systems

LARRIEU Nicolas, VARET Antoine
Rapid Prototyping of Software for Avionics Systems: Model-oriented Approaches forComplex Systems Certification

OUSSALAH Mourad Chabane
Software Architecture 1

OUSSALAH Mourad Chabane
Software Architecture 2

QUESNEL Flavien
Scheduling of Large-scale Virtualized Infrastructures: Toward Cooperative Management

RIGO Michel
Formal Languages, Automata and Numeration Systems 1: Introduction to Combinatorics on Words
Formal Languages, Automata and Numeration Systems 2: Applications to Recognizability and Decidability

SAINT-DIZIER Patrick
Musical Rhetoric: Foundations and Annotation Schemes

TOUATI Sid, DE DINECHIN Benoit
Advanced Backend Optimization

2013

ANDRÉ Etienne, SOULAT Romain
The Inverse Method: Parametric Verification of Real-time Embedded Systems

BOULANGER Jean-Louis
Safety Management for Software-based Equipment

DELAHAYE Daniel, PUECHMOREL Stéphane
Modeling and Optimization of Air Traffic

FRANCOPOULO Gil
LMF — Lexical Markup Framework

GHÉDIRA Khaled
Constraint Satisfaction Problems

ROCHANGE Christine, UHRIG Sascha, SAINRAT Pascal
Time-Predictable Architectures

WAHBI Mohamed
Algorithms and Ordering Heuristics for Distributed Constraint Satisfaction Problems

ZELM Martin *et al.*
Enterprise Interoperability

2012

ARBOLEDA Hugo, ROYER Jean-Claude
Model-Driven and Software Product Line Engineering

BLANCHET Gérard, DUPOUY Bertrand
Computer Architecture

BOULANGER Jean-Louis
Industrial Use of Formal Methods: Formal Verification

BOULANGER Jean-Louis
Formal Method: Industrial Use from Model to the Code

CALVARY Gaëlle, DELOT Thierry, SEDES Florence, TIGLI Jean-Yves
Computer Science and Ambient Intelligence

MAHOUT Vincent
Assembly Language Programming: ARM Cortex-M3 2.0: Organization, Innovation and Territory

MARLET Renaud
Program Specialization

SOTO Maria, SEVAUX Marc, ROSSI André, LAURENT Johann
Memory Allocation Problems in Embedded Systems: Optimization Methods

2011

BICHOT Charles-Edmond, SIARRY Patrick
Graph Partitioning

BOULANGER Jean-Louis
Static Analysis of Software: The Abstract Interpretation

CAFERRA Ricardo
Logic for Computer Science and Artificial Intelligence

HOMES Bernard
Fundamentals of Software Testing

KORDON Fabrice, HADDAD Serge, PAUTET Laurent, PETRUCCI Laure
Distributed Systems: Design and Algorithms

KORDON Fabrice, HADDAD Serge, PAUTET Laurent, PETRUCCI Laure
Models and Analysis in Distributed Systems

LORCA Xavier
Tree-based Graph Partitioning Constraint

TRUCHET Charlotte, ASSAYAG Gerard
Constraint Programming in Music

VICAT-BLANC PRIMET Pascale *et al.*
Computing Networks: From Cluster to Cloud Computing

2010

AUDIBERT Pierre
Mathematics for Informatics and Computer Science

BABAU Jean-Philippe *et al.*
Model Driven Engineering for Distributed Real-Time Embedded Systems 2009

BOULANGER Jean-Louis
Safety of Computer Architectures

MONMARCHE Nicolas *et al.*
Artificial Ants

PANETTO Hervé, BOUDJLIDA Nacer
Interoperability for Enterprise Software and Applications 2010

PASCHOS Vangelis Th
Combinatorial Optimization – 3-volume series
Concepts of Combinatorial Optimization – Volume 1
Problems and New Approaches – Volume 2
Applications of Combinatorial Optimization – Volume 3

SIGAUD Olivier *et al*.
Markov Decision Processes in Artificial Intelligence

SOLNON Christine
Ant Colony Optimization and Constraint Programming

AUBRUN Christophe, SIMON Daniel, SONG Ye-Qiong *et al.*
Co-design Approaches for Dependable Networked Control Systems

2009

FOURNIER Jean-Claude
Graph Theory and Applications

GUEDON Jeanpierre
The Mojette Transform / Theory and Applications

JARD Claude, ROUX Olivier
Communicating Embedded Systems / Software and Design

LECOUTRE Christophe
Constraint Networks / Targeting Simplicity for Techniques and Algorithms

2008

BANÂTRE Michel, MARRÓN Pedro José, OLLERO Hannibal, WOLITZ Adam
Cooperating Embedded Systems and Wireless Sensor Networks

MERZ Stephan, NAVET Nicolas
Modeling and Verification of Real-time Systems

PASCHOS Vangelis Th
Combinatorial Optimization and Theoretical Computer Science: Interfaces and Perspectives

WALDNER Jean-Baptiste
Nanocomputers and Swarm Intelligence

2007

BENHAMOU Frédéric, JUSSIEN Narendra, O'SULLIVAN Barry
Trends in Constraint Programming

JUSSIEN Narendra
A to Z of Sudoku

2006

BABAU Jean-Philippe *et al.*
From MDD Concepts to Experiments and Illustrations – DRES 2006

HABRIAS Henri, FRAPPIER Marc
Software Specification Methods

MURAT Cecile, PASCHOS Vangelis Th
Probabilistic Combinatorial Optimization on Graphs

PANETTO Hervé, BOUDJLIDA Nacer
Interoperability for Enterprise Software and Applications 2006 / IFAC-IFIP I-ESA'2006

2005

GÉRARD Sébastien *et al.*
Model Driven Engineering for Distributed Real Time Embedded Systems

PANETTO Hervé
Interoperability of Enterprise Software and Applications 2005