Software Evolution And Maintenance

CHAPTER 5

Maintenance Management Problems

- Problems of software Maintenance
- Software Reuse
- Legacy Systems

When you have told anyone you have left him a legacy the only decent thing to do is to die at once.
—Samuel Butler

Before software can be reusable it first has to be usable.
—Ralph Johnson

Problems Of Software Maintenance

- A number of authors have identified maintenance-related issues that affect maintenance resource management and processes and their specific tools/techniques.
  - These issues can be categorized based on two viewpoints

1. **External**: perceptions of customers (i.e., users and stakeholders)
2. **Internal**: perceptions of software maintenance engineers and managers

Issues In Software Maintenance

These issues can be categorized based on two viewpoints

1. ***External viewpoints:*** From the customers' point of view, the main issues reported are *high cost*, *slow delivery of services*, and *fuzziness of prioritization* [internal information systems and information technology (IS/IT) priorities versus users' priorities]
2. ***Internal viewpoints:*** From the maintainers' point of view, key maintenance issues include
   - Software that has been poorly designed and programmed
   - A major lack of software documentation.
   - The biggest problem in software maintenance is not technical but rather its management.

External Viewpoints

1. **External viewpoints:** perceptions of customers (i.e., users and stakeholders)

- Most problems that are associated with software maintenance can be traced to deficiencies of the software development process.
- There are several technical and managerial problems encountered while maintaining software

i. Cost

- Various research studies proposed that software maintenance consumes 60% to 80% of cost in whole SDLC.
- Maintenance costs are mainly due to enhancements, rather than corrections .
- High cost => Inadequate communication by maintenance managers about the type of maintenance work.
- Maintenance managers
  - group enhancement and corrective work together in their management reports, statistics, and budgets, which warps costs in both maintenance budgets and reports.

External Viewpoints

ii. Slow delivery of services

- misunderstanding of the fundamental difference between the electronic/computer hardware maintenance and software maintenance domains.
- For instance, a software user often does not have a clear idea of what constitutes a software maintenance activity,
  - for example, how the repair of a broken printer or computer component may simply involve a modular-component replacement activity.
- He or she also might note that electronic equipment design, being more mature than software design, is based on a thorough "modularization" of components, which does not create side-effects during maintenance activities.

Internal Viewpoints

2. **Internal viewpoints:** From the maintainers' point of view

i. Impact Analysis

- One of the most important challenges in software maintenance is to find out the effects of a proposed modification on the rest of the system.
- Impact analysis is the action of assessing the probable effects of a change with the plan of reducing sudden side effects.
- The task involves assessing the correctness of a projected modification and evaluating the risks related with its completion, plus the estimates of the effects on properties, energy and development.

Internal Viewpoints

ii. Corrective Changes

- It is hard to find the correct place to do the changes.
- It can be difficult to recognize the code base.
- If the preliminary design is reduced a minute change might insist architecture changes that take a lot of time.
- If there has been a complete workaround of one problem then the next are even harder to crack.
- Design errors are tough to repair because it takes a lot of time and understanding of the entire code base and are linked to risks.

Internal Viewpoints

iii. Adaptive Changes

- Adaptive changes are frequently not easy due to deficiency of information about what the software is being modified to.
- The diverse facts of the new technology to adjust to be difficult to take hold of.
- Impact analysis and discovering interfaces to the new things are difficult problems due to unbalanced preliminary design are a matter of concern.

Internal Viewpoints

iv. Program Comprehension

- Another key issue is program comprehension which involves that
  - Extensive amount of time should be expended by maintenance engineers to read and understand the code,
  - The relevant documentation to have a better perspective on its logic, purpose and structure to maintain a part of software and to enhance the quality of software

Internal Viewpoints

- An immature software development process creates a number of problems in the final product delivered to the customer and to the maintenance organization
- **Software age** also has an impact on maintenance cost when the software was built using older techniques that did not take in account
  - modern architectural concepts
  - with complex structures
  - non standardized code
  - poor documentation, leading to further difficulty in maintenance work.
- Software maintenance problems can be grouped into three categories:
  1. *Problems of alignment with the organization's objectives,*
  2. *Process problems, and*
  3. *Technical problems*

Internal Viewpoints

- Another survey of participants at successive software maintenance conferences presents a list of 19 key maintenance problems, ranked by importance, as perceived by software maintenance engineers.
- The majority of the problems reported by this survey can be categorized in the maintenance process and management categories (items 3, 7, 8, 9, 10, 13, 14, 15, 16, 18, and 19. (In the table 5.1)
- A second group of problems have been identified as originating from the software development process itself (items 4,6, 11, and 12.)

Internal Viewpoints

- When the development quality is poor, the software is still transferred to the maintenance organization with a backlog of changes and problems that should have been addressed by the developers and which are hard to handle with a small number of individuals.
- Other maintenance problems stemming from the same source are:
  - Poor traceability to the **processes** and **products** that created the software
  - Changes rarely documented
  - Difficulty of change management and monitoring
  - Ripple effects of software changes

Internal Viewpoints: Problems Ranked By Software Maintainers

Rank	Maintenance problem
1	Managing changing priorities (M)
2	Inadequate testing techniques (T)
3	Difficulty in measuring performance (M)
4	Absent or incomplete software documentation (M)
5	Adapting to rapid changes in user organisations (M)
6	A large backlog of requests for change (M)
7	Difficulty in measuring/demonstrating the maintenance team's contribution (M)
8	Low morale due to lack of recognition and respect for maintenance engineer (M)
9	Not many professionals in the domain, especially experienced ones (M)
10	Little methodology, few standards, procedures and tools specific to maintenance (T)
11	Source code in existing software complex and unstructured (T)
12	Integration, overlap and incompatibility of existing systems (T)
13	Little training available to maintenance engineers (M)
14	No strategic plans for maintenance (M)
15	Difficulty in understanding and meeting user expectations (M)
16	Lack of understanding and support from IS/IT managers (M)
17	Maintenance software runs on obsolete systems and technologies (T)
18	Little will or support for reengineering existing software (M)
19	Loss of expertise when a maintenance engineer leaves the team or company
	(M): Management Problem (T): Technical Problem

Table 5.1

Software Reuse

- To use previously developed software, rather than 'reinventing the wheel' by writing all the code from scratch. This is the concept of **software reuse**
- Productivity can be increased by software reuse because less time and effort is required to specify, design, implement and test the new system.
- The quality of the new product tends to be higher, primarily because the reused components will have already been through cycles of rigorous testing.
- Reuse results in a more reliable, more robust and higher quality product.
- The telling observation has been made that 60-85% of programs used to build systems already exist and can be standardized and reused.

Software Reuse

- **Data** - factual information such as measurements used as a basis for calculation, discussion, or reasoning.
- **Personnel** - the individuals involved in a software project.
- **Product** - a concrete documentation or artifact created during a software project.
- **Program** - code components, at the source and object code level, such as modules, packages, procedures, functions, routines, etc. Also commercial packages such as spreadsheets and databases.
- **Reuse** - the reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development or maintenance of that other system.

Software Reuse

- The knowledge that can be reused comes from three main sources: the process, the personnel and the product.

- **Process:** Process reuse may be the application of a given methodology to different problems.
- The application of methodologies such as formal methods, or object-oriented design in the development of different products is an example of process reuse.
- Another example of reusing a process is the use of a cost estimation model - such as Boehm's COCOMO II model - when estimating the cost of a maintenance project.

- **Personnel**
- The reuse of personnel implies reusing the knowledge that they acquire in a previous similar problem or application area. This expertise is known as domain knowledge.
- An example of this is 'lesson learned‘ knowledge - the knowledge acquired through meeting and addressing a particular situation in practice.

- **Product**
- Product reuse involves using artefacts that were the products arising from similar projects. Examples include the reuse of data, designs and programs.

Software Reuse

- There are four types of reusable artifacts derived from Product reuse as follows:

- **Data reuse:** It involves a standardization of data formats.

- Reusable data enables data sharing between applications and also promotes widespread program reusability.

- Reusable data plays an important role in database systems. For different applications to share the data held in these databases, the data needs to be in a format that facilitates data transfer between applications.

- An example of data that needs to be reusable is patient data in medical information systems, primarily because different individuals are often involved in the care of a patient - the family doctor, the hospital consultant, the pathology laboratory,

- **Architectural reuse:** This means developing: (i) a set of generic design styles about the logical structure of software; and (ii) a set of functional elements and reuse those elements in new systems.

- The architectural design is an abstract or diagrammatic representation of the main components of the software system, usually a collection of modules.

Software Reuse

- **Design reuse:** This deals with the reuse of abstract design. A selected abstract design is custom implemented to meet the application requirements.

➢ The design of a system can be represented in many ways, for example using context diagrams, data flow diagrams, entity-relationship diagrams, state transition diagrams and object models.

➢ The redeployment of pre-existing designs during the development of similar products can increase productivity and improve product quality.

- **Program reuse:** This means reusing executable code.

➢ Program reuse is the reuse of code components that can be integrated into a software system with little or no prior adaptation; for example, libraries, high-level languages and problem-oriented languages.

➢ For example, one may reuse a pattern-matching system, that was developed as part of a text processing tool, in a database management system.

Software Reuse

- The reusability property of a software asset indicates the degree to which the asset can be reused in another project.
- For a software component to be reusable, it needs to exhibit the following properties that directly encourage its use in similar situations

1. Environmental independence: This means that a component performs a task, and it makes minimal interactions with other components.
2. High cohesion: A component is said to have high cohesion, if its subsystems cooperate with each other to achieve a single objective.
3. Low coupling: If a component or function performs one specific task and has few or no dependencies, it will be much more reusable than if it performs multiple tasks, has many side effects, and needs other components.
4. Adaptability: being easily changed to run in a new environment.

Software Reuse…

5. Understandability: If a program component is easily comprehended, programmers can quickly make decisions about its reuse potential.
6. Reliability: It is the ability of a software system to consistently perform its intended function without degradation or failure.
7. Portability: It is the usability of the same software in different environment. Therefore, components that are portable are excellent candidates for reuse.

Benefits Of Reuse

- The most obvious advantage of reuse is economic benefit.
- Other tangible benefits of reuse are as follows:

1. Increased reliability: Reusable components tend to reveal more failures because of the extra efforts put in their design and their extensive usage.
2. Reduced process risk: a working, presumed fault-free component is being reused, thereby reducing much uncertainty.
3. Increase productivity: productive because their activities, namely, specification, design, implementation, and testing, consume less time and effort.
4. Compliance with standards: Reusing code in a software system being developed can improve its compliance with standards, thereby raising its quality
5. Accelerated development: Software development time can be reduced by reusing assets.
6. Improved maintainability: Reusable components generally possess some desired characteristics: modularity, low coupling & high cohesion, & programming style.
7. Less maintenance effort and time: Reusable components are easy to understand and change during a software modification.

Reuse Models

- Development of assets with the potential to be reused requires additional capital investment.
- The organization can select one or more reuse models that best meet their business objectives, engineering realities, and management styles.
- Reuse models are classified as:
    - **Proactive**
    - **Reactive**
    - **Extractive**

Reuse Models

Proactive approaches

- In proactive approaches to developing reusable components, the system is designed and implemented for all conceivable variations; this includes design of reusable assets.
- A proactive approach is to product lines what the Waterfall model is to conventional software.
- The term product line development, which is also known as domain engineering, refers to a "***development-for-reuse***" process to create reusable software assets (RSA)
- This approach might be adopted by organizations that can accurately estimate the long-term requirements for their product line.
- This approach faces an investment risk if the future product requirements are not aligned with the projected requirements.

Reuse Models

Reactive approaches

- In this approach, while developing products, reusable assets are developed if a reuse opportunity arises.
- This approach works, if
  i. it is difficult to perform long-term predictions of requirements for product variations.
  ii. an organization needs to maintain an aggressive production schedule with not much resources to develop reusable assets. The cost to develop assets can be amortized over several products.
- However, in the absence of a common, solid product architecture in a domain, continuous reengineering of products can render this approach more expensive.

Reuse Models

Extractive approaches

- The extractive approaches fall in between the proactive approaches and the reactive ones.
- To make a domain engineering's initial baseline, an extractive approach reuses some operational software products.
- Therefore, this approach applies to organizations that have accumulated both artifacts and experiences in a domain, and want to rapidly move from traditional to domain engineering.

Factors Influencing Reuse

- Reuse factors are the practices that can be applied to increase the reuse of artifacts.
- Frakes and Gandel identified four major factors for systematic software reuse: ***managerial, legal, economic and technical***.

Managerial

Systematic reuse requires upper management support, because:

i. it may need years of investment before it pays off.

ii. it involves changes in organization funding and management structure that can only be implemented with executive management support.

Legal

- This factor is linked with cultural, social, and political factors, and it presents very difficult problems.
- Potential problems include proprietary and copyright issues, liabilities and responsibilities of reusable software, and contractual requirements involving reuse.

Factors Influencing Reuse

Economic

- Software reuse will succeed only if it provides economic benefits.
- A study by Favaro found that some artifacts need to be reused more than 13 times to recoup the extra cost of developing reusable components.

Technical

- This factor has received much attention from the researchers actively engaged in library development, object-oriented development paradigm, and domain engineering.
- A reuse library stores reusable assets and provides an interface to search the repository.
- One can collect library assets in a number of ways: (i) reengineer the existing system components; (ii) design and build new assets; and (iii) purchase assets from other sources.

Success Factors Of Reuse

The following steps aid organizations in running a successful reuse program:

- Develop software with the product line approach.
- Develop software architectures to standardize data formats & product interfaces.
- Develop generic software architectures for product lines.
- Incorporate off-the-shelf components.
- Perform domain modeling of reusable components.
- Follow a software reuse methodology and measurement process.
- Ensure that management understands reuse issues at technical and non- technical levels.
- Support reuse by means of tools and methods.
- Support reuse by placing reuse advocates in senior management.
- Practice reusing requirements and design in addition to reusing code.

Legacy Systems

- Brodie and Stonebraker used the following definition to define legacy systems:
  - any information system that *significantly resists modification* and *evolution* to meet new and constantly changing business requirements.
- Bennett used the following definition to define legacy systems:
  - Legacy software systems are *large software systems* that *we don't know how to cope with* but that are *vital* to our organization.
- Supporting the definitions are a set of acceptable features of a legacy system:
  - large with millions of lines of code.
  - geriatric, often more than 10 years old.
  - written in obsolete programming languages.
  - lack of consistent documentation.
  - poor management of data, often based on flat-file structures.
  - degraded structure following years of modifications.
  - very difficult, if not impossible, to expand.
  - runs on old processor.

Legacy Systems

- A legacy system is a piece of software that: you have *inherited*, and is *valuable* to you.
- Legacy software is old software that is still useful.

Typical **problems** with legacy systems:

- Original developers *not available*
- *Outdated* development methods used
- Extensive patches and *modifications* have been made
- *Missing* or outdated documentation

  ⇒ *so, further evolution and development may be prohibitively expensive*

Legacy Systems

- There are several categories of solutions for legacy information system (LIS).
- These solutions generally fall into six categories as follows:
  - ***Freeze***: The organization decides to not carry out further work on an LIS.
  - ***Outsource***: An organization may decide that supporting legacy (or any) software is not its core business. The organization may outsource it to a specialist organization offering this service.
  - ***Carry on maintenance***. Despite all the problems associated with supporting a software system, the organization decides to carry on maintenance for another period.
  - ***Discard and redevelop***: The organization throws all the software away and redevelops the application once again from scratch, using a new hardware platform and modern architecture tools and database.
  - ***Wrap:*** It is a black-box-based modernization technique: surround the LIS with a new layer of software to hide the complexity of the existing interfaces, applications, and data, with the new interfaces. Wrapping gives existing components a modern and improved appearance.
  - ***Migrate:*** a legacy system is ported to a modern platform, while retaining most of the system's functionality and introducing minimal disturbance in the business environment.

Legacy Systems

- In 1988, Dietrich et al. first introduced the concept of a "wrapper" at IBM.
- ***Wrapping*** means encapsulating the legacy component with a new software layer that provides a new interface and hides the complexity of the old component.
- The encapsulation layer can communicate with the legacy component through sockets, remote procedure calls (RPCs), or predefined application program interfaces (API).
- The wrapped component is viewed similar to a remote server; it provides some service required by a client that does not know the implementation details of the server.
- By means of a *message passing mechanism*, a wrapper connects to the clients.
- On the input front, the wrapper accepts requests, restructures them, and invokes the target object with the restructured arguments.
- On the output front, the wrapper captures outputs from the wrapped entity, restructures the outputs, and pushes them to the requesting entity.
- However, this technique does not solve the problems with legacy systems.

Types of Wrapper

- Orfali et al. classified wrappers into four categories:
  - Database wrappers.
  - System service wrappers.
  - Application wrappers.
  - Function wrappers.

1. Database wrappers:

- Database wrappers can be further classified into forward wrappers (f-wrappers) and backward wrappers (b-wrappers)
- The forward wrappers-approach, depicted in Figure 5.1, shows the process of adding a new component to a legacy system.
- Therefore, by means of translation service involving both legacy data and queries, the wrapper integrates the new component with the legacy system.
- The backward wrappers-approach has been depicted in Figure 5.2
- In this approach, data are migrated first then new components are developed that use the new database; the legacy components access the new data via wrappers.

Types of Wrapper

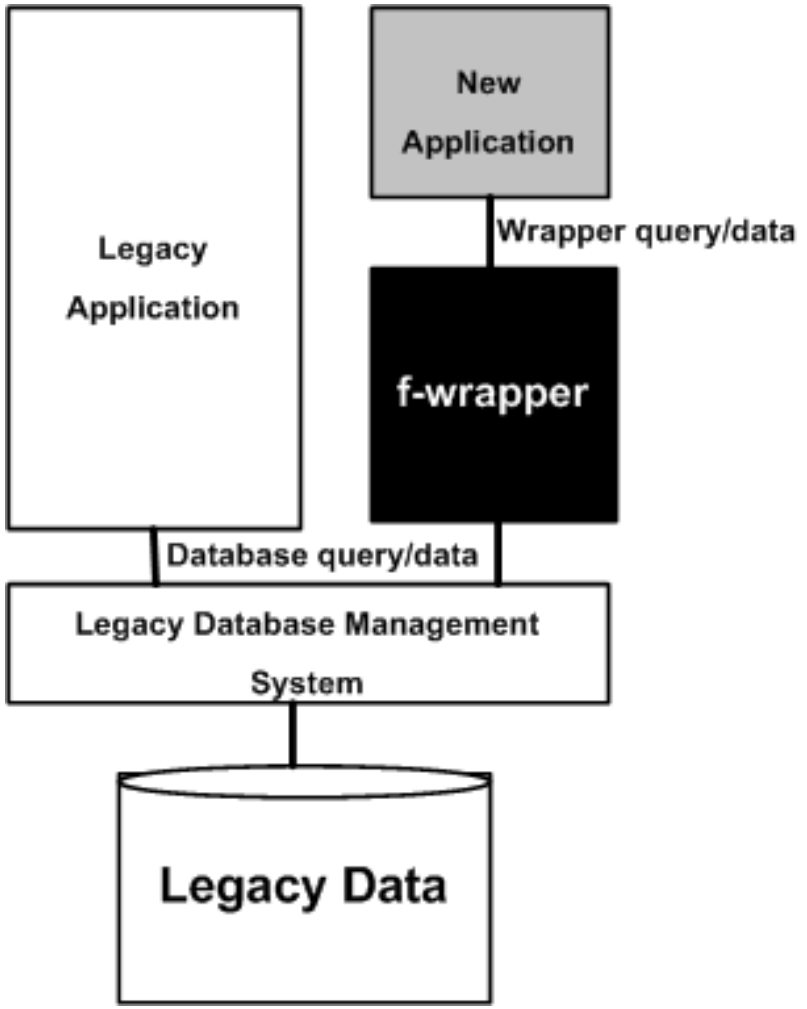


Figure 5.1 Forward wrapper

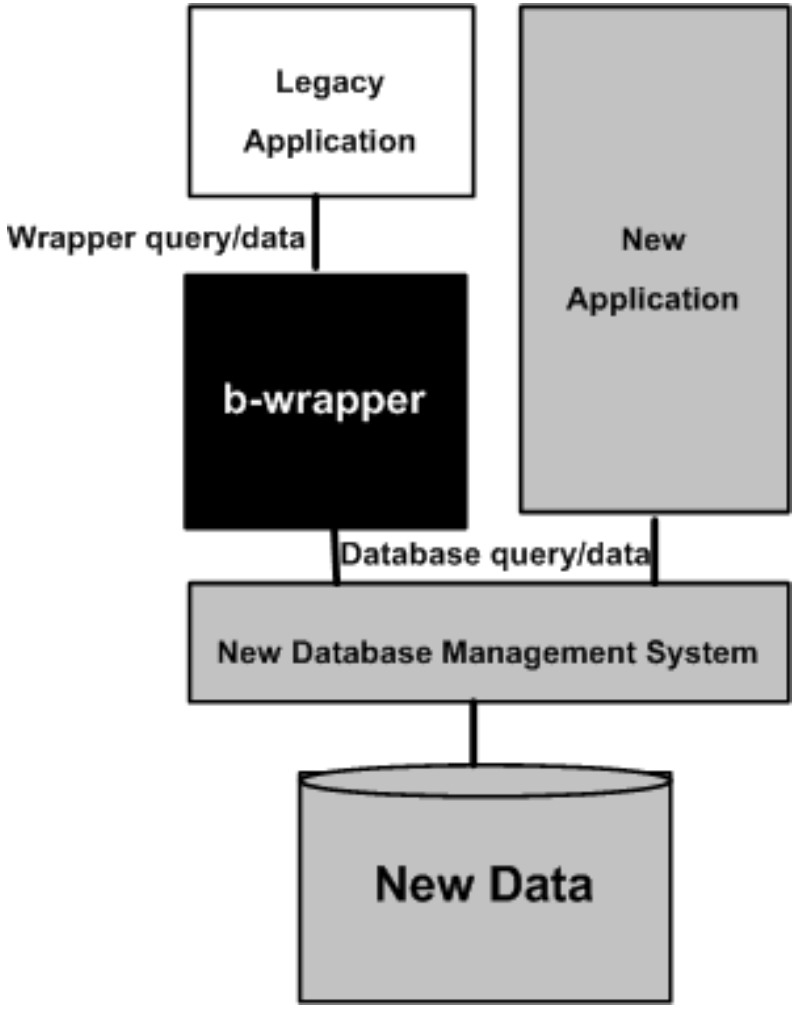


Figure 5.2 Backward wrapper

Types of Wrapper

2. System service wrappers:

- This kind of wrappers support customized access to commonly used system services, namely, routing, sorting, and printing.
- A client program may access those services without knowing their interfaces.

3. Application wrappers:

- This kind of wrappers encapsulate online transactions or batch processes.
- These wrappers enable new clients to include legacy components as objects and invoke those objects to produce reports or update files.

4. Function wrappers:

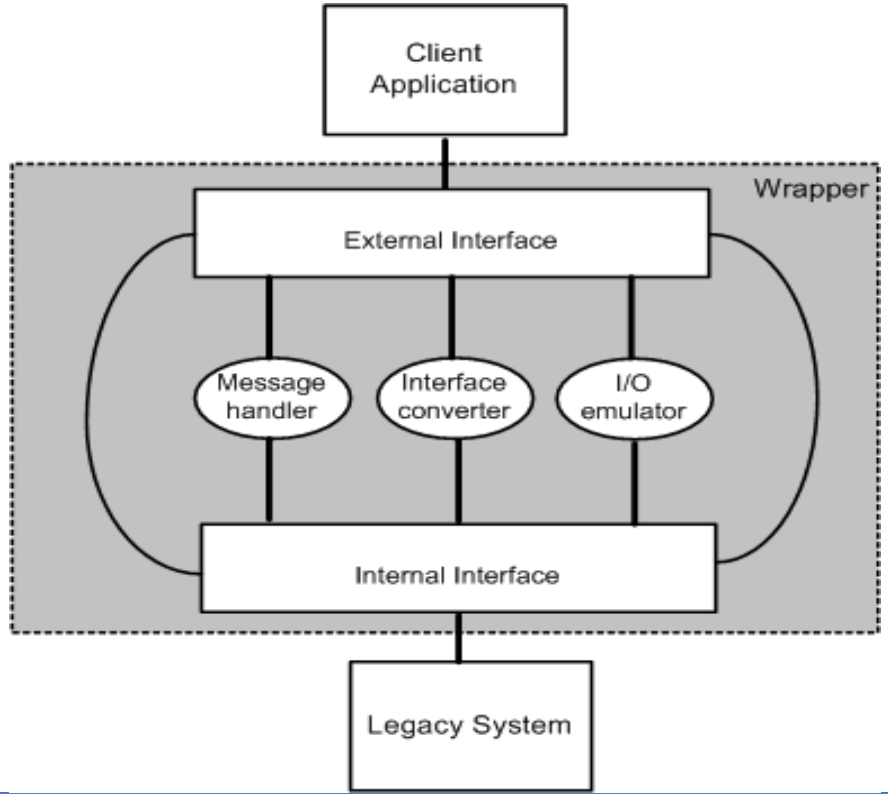
- This kind of wrappers provide an interface to call functions in a wrapped entity.
- In this mechanism, only certain parts of a program – and not the full program – are invoked from the client applications.
- Therefore, limited access is provided by function wrappers.

Constructing A Wrapper

- A legacy system is wrapped in three steps as follows:
  - A wrapper is constructed.
  - The target program is adapted.
  - The interactions between the target program and the wrapper are verified.
- A wrapper, is a program which receives input messages from the client program, transforms the inputs into an internal representation, and invokes the target system with the newly formatted messages.
- The wrapper intercepts the outputs produced by the target system, transforms them into a format understood by the client, and transfers them to the client.

Constructing A Wrapper

- Conceptually, a wrapper comprises two interfaces and three event driven modules as shown in Figure 5.4.
- The two interfaces are an internal interface and an external interface, and the three modules are message handler, interface converter, and I/O-emulator.



Constructing A Wrapper

External interface:

- The interface of the wrapper that is accessed by the clients is called the external interface.
- The wrapper and its client generally communicate by passing messages.
- Messages normally comprise a header and a body.
- The header of a message contains control information:
  - (i) identity of the sender.
  - (ii) a time stamp.
  - (iii) the transaction code.
  - (iv) target program type.
  - (v) the identity of the transaction, program, procedure, or module to be invoked.
- The message body contains the input arguments. Similarly, the message body contains the output values in an output message.
- The values contained in a message are normally ASCII strings separated by a delimiter, say, a back slash ("\").

Constructing A Wrapper

Internal interface:

- A wrapper's internal interface is the interface visible to the server.
- The internal interface is dependent upon the language and type of the wrapped entity.
- For example, if the wrapped entity is a job, the internal interface of the wrapper is a job control procedure, and the job control procedure is interpreted to execute the job.
- On the other hand, if the encapsulated entity is a program, procedure, transaction, or a module, the internal interface is a list of parameters in the language of the encapsulated software.
- Therefore, the internal interface is extracted from the source code of the encapsulated software.
- The parameters of the internal interface are specified in terms of the target language, thereby necessitating a translation from the input ASCII strings to the target types.

Constructing A Wrapper

Message handler:

- The message handler buffers input and output messages, because requests from the client may occasionally arrive at a faster rate than they can be consumed.
- Similarly, outputs from the wrapped program may be produced at a faster rate than they can be sent to the client.

Interface converter:

- This entity converts the internal interface into the external interface and vice-versa.

I/O-emulator:

- I/O-emulator intercepts the inputs to and outputs from the wrapped entity.
- Upon intercepting an input, the emulator fills the input buffer with the values of the parameters of the external interface.
- On the other hand, when it intercepts an output value, the emulator copies the contents of the output buffer into the parameter space of the external interface.

Screen Scraping

- Screen scraping is a common form of wrapping in which modern, say, *graphical, interfaces* replace *text-based interfaces*.
- The new interface can be a GUI (graphical user interface) or even a web-based HTML (hypertext markup language) client.
- Tools, such as Verastream from Attachmate, automatically generate the new screens.
- Screen scraping simply provides a straightforward way to continue to use a legacy system via a graphical user interface.
- Screen scraping is a short term solution to a larger problem.
- Many serious issues are not addressed by simply mounting a GUI on a legacy system. For example, screen scraping:

  (i) does not evolve to support new functions.

  (ii) incurs high maintenance cost.

  (iii) ignores the problem of overloading.

  Overloading is simply defined as the ability of one function to perform different tasks

Migration

- **Migration** of LIS is the best alternative, when wrapping is unsuitable and redevelopment is not acceptable due to substantial risk.
- However, it is a very complex process typically lasting five to ten years.
- It offers better system understanding, easier maintenance, reduced cost, and more flexibility to meet future business requirements.
- Migration involves changes, often including restructuring the system, enhancing the functionality, or modifying the attributes.
- It retains the basic functionality of the existing system.
- We will discuss general guidelines for migrating a legacy system to a new target system.
- A brief overview of the migration steps is explained next.

Migration

- LIS migration involves creation of a modern database from the legacy database and adaptation of the application program components accordingly.
- A database has two main components: schema of the database and data stored in the database.
- Therefore, in general, migration comprises three main steps:

  (i) conversion of the existing schema to a target schema;

  (ii) conversion of data; and

  (iii) conversion of program.

i. Schema conversion:

- Schema conversion means translating the legacy database schema into an equivalent database structure expressed in the new technology.
- The transformation of source schema to a target schema is made up of two processes.
- The first one is called DBRE and it aims to recover the conceptual schema that express the semantics of the source data structure.

Migration

- The second process is straightforward and it derives the target physical schema from this conceptual schema:

  (i) an equivalent logical schema is obtained from the conceptual schema by means of transformations; and

  (ii) a physical schema is obtained from the logical schema by means of transformations.

ii. Data Conversion:

- Data conversion means movement of the data instances from the legacy database to the target database.
- Data conversion requires three steps: **extract, transform, and load** (ETL).
- First, **extract data** from the legacy store. Second, transform the extracted data so that their structures match the format.
- In addition, perform **data cleaning** (a.k.a. scrubbing or cleansing) to fix or discard data that do not fit the target database.
- Finally, **load** the transformed data in the target database.

Migration

iii. Program Conversion:

- In the context of LIS migration, program conversion means modifying a program to access the migrated database instead of the legacy data.
- The conversion process leaves the functionalities of the program unchanged.
- Program conversion depends upon the rules that are used in transforming the legacy schema into the target schema.

Testing and Functionality:

- It is important to ensure that the outputs of the target system are consistent with those of the LIS.
- Therefore, new functionality is not to be introduced into the target system in a migration project.
- If both the LIS and the target system have the same functionality, it is easier to verify their outputs for compliance.
- However, to justify the project expense and the risk, in practice, migration projects often add new functionalities.

Migration

Cut over or Roll over:

- The event of cutting over to the new system from the old one is required to cause minimal disruption to the business process.
- There are three kinds of transition strategies.

Cut-and-Run: The simplest transition strategy is to switch off the legacy system and turn on the new system.

Phased Interoperability: To reduce risks, cut over is gradually performed in incremental steps.

Parallel Operation: The target system and the LIS operate at the same time. Once the new system is considered to be reliable, the LIS is taken off service.

MIGRATION METHODS

Reading Assignment(Page:202-217)

- Seven approaches to migration have been explained.
- No single approach can be applied to all kinds of legacy systems, because they vary in their scale, complexity, and risks of failure while migrating.
- The seven approaches are as follows:
  I. Cold turkey
  II. Database first
  III. Database last
  IV. Composite database
  V. Chicken little
  VI. Butterfly
  VII. Iterative

Reference

- ✓ Alain April, Alain Abran (2008), Software Maintenance Management Evaluation and Continuous Improvement.
- ✓ Pierre Bourque, École de technologie supérieure(2014), Guide to the Software Engineering Body of Knowledge (SWEBOK) Version 3.0, A Project of the IEEE Computer Society.
- ✓ Penny Grub, Armstrong A Takang, Software Maintenance Concepts and Practice, 2nd edition