Software Reuse and Modernization of Legacy Systems

Parastoo Mohagheghi
26 September 2012
Contents

• Types and examples of software reuse
• Definitions related to software reuse
• Legacy systems and the modernization problem
• Migration of legacy systems to service oriented architecture
Sources

- PhD thesis: The impact of Software Reuse and Incremental Development on the Quality of Large Systems. Parastoo Mohagheghi, NTNU, 2004
- SMART report by SEI. http://www.sei.cmu.edu/library/abstracts/reports/08tn008.cfm; Sections 1-4
SOFTWARE REUSE TAKES MANY FORMS
Ad-hoc code reuse

- Copy and use “as-is” or modify the source code, using libraries.
- Simple, no planning beforehand, all developers do it.
  - Can increase coupling in an application, unpredictable effects.
  - Savings in productivity are limited.
  - Are there any copyright issues?
Reuse of objects or classes

- Inheritance reuse
  - Fundamental in object-orientation
  + Taking advantage of developed behavior, decrease development time.
  - Fragile class hierarchy that is difficult to maintain.
Reuse of components

- Refers to reuse of self-sufficient, encapsulated components that hide the source code. Functionality is accessed through their interfaces.
  - Smaller components such as Java Beans or ActiveX components
  - Larger components developed in-house or external such as Commercial-Off-The-Self (COTS) or Open Source Software (OSS)

+ Designed with reuse in mind.
+ Larger components save your time significantly.

- Reliance on vendor.
- Difficult to understand or debug.
Component-Based Development

• CBD is an approach to the old problem of handling the complexity of a system by decomposing it. Already in 1972, David Parnas wrote about the benefits of decomposing a system into modules.

• **Components** are units of independent development and acquisition. A component conforms to and provides the physical realization of a set of interfaces.

• Components adhere to a **component model** or **component framework** that enables composition of components. Examples: EJB or .Net, or domain-specific frameworks.

• **Composition** is the term used for components, instead of integration.
Framework reuse

• Collections of classes or components that implement the basic functionality for a technical problem or a business domain, and some customizable functionality

+ Frameworks provide complex functionality that takes years to develop and provide significant reuse of design and code.

- Complex to reuse.
- Must accept framework requirements.
Pattern reuse

- Not reusing the code but the idea of how to solve a problem.
- Many types of patterns: structural, behavioral, creational etc.

+ High level of reuse that is independent of programming languages or frameworks.

- You still have to write the code and implement the pattern.

Template method: Creates a skeleton but allows classes to override specific Parts; source Wikipedia
Product families

- A collection of products that are based on the same technology— for instance a collection of TVs based on the same software architecture.

- Large-scale reuse, planned, common architecture.

- Must agree on a reference architecture, and invest time on planning reuse.
<table>
<thead>
<tr>
<th>Reuse Category</th>
<th>Example(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architected</td>
<td>Service domains, Domain components, Internal open source</td>
</tr>
<tr>
<td>Pattern</td>
<td>Architecture patterns, Design patterns, Analysis patterns</td>
</tr>
<tr>
<td>Framework</td>
<td>External open source, Technical frameworks, Business frameworks</td>
</tr>
<tr>
<td>Artifact</td>
<td>COTS application, Legacy application, Domain model</td>
</tr>
<tr>
<td>Module</td>
<td>User interface components, Technical components, Web services</td>
</tr>
<tr>
<td>Template</td>
<td>Use case template, Project plan template, Document template</td>
</tr>
<tr>
<td>Code</td>
<td>Class libraries, Functional libraries, Cut &amp; paste</td>
</tr>
</tbody>
</table>

Copyright 2005 Scott W. Ambler
Recently: Reuse at run-time

- **Mashup web applications**: applications that use services of several on-line applications
  - Provides quick solutions
  - Underlying services may not be available.
  - Quality depends on many factors such as response time of services, security, and Service Level Agreements.

Service engineering

- Service-Oriented Architecture is an architectural pattern.
- Developing application by composing services.
- More on that later.
How to define software reuse?

• Doug McIlroy first introduced the idea of systematic reuse as the planned development and widespread use of software components in 1968.

• Many software organizations around the world have reported successful reuse programs such as IBM, Hewlett-Packard, Hitachi and many others.

• Morisio et al. define reuse as:

**Software reuse** is the **systematic** practice of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality and business performance.
Some definitions

- **Proactive reuse**: Reuse is planned. Reusable components are identified and developed for reuse.

- **Reactive / extractive reuse**: develop a first product and identify what is reusable.

- **Component engineering** is developing single components.

- **Application engineering** is developing applications based on reusable and new components.
Reuse “as-is” or modify

• Generally in any reuse, assets may be reused:
  – **Verbatim** which means reusing an asset “as-is” in a black-box style;
  – **Modified** in a white-box style to make an asset reusable for a new target.
Developing for and with reuse


- Developing components so that they become reusable is called developing **for reuse**.
  - Domain analysis to understand the needs

- Developing systems of reusable components is called development **with reuse**.
  - Application engineering to build individual products
Why reuse software?

- Improved productivity and shorten time to market
- Improved quality with reliable reusable software
  - Reduced maintenance effort
- (Reduced cost)

Good programmers know what to write.
Great ones know what to rewrite (and reuse).
Eric Steven Raymond, The cathedral and the bazar, 1999
Why not reuse software?

• Reuse is not free!
  – One must invest in search, understanding, integration and sometimes modification.

• Less control
  – Product roadmap of COTS out of control
  – Reliance on the quality of reused software

• Difficult to analyze defects
  – Lack of documentation
  – Is the problem in the new code or the reused code?
  – What if you don’t have access to the source code of reused components?

• Vendor lock-in problem
Software reuse is not risk-free!
CASE STUDY: Evolution from a Single Product to a Product line
Ericsson: The GPRS system (General Packet Radio Service)

The case is described in detail in my PhD thesis.
The initial software architecture of SGSN nodes

Middleware (MW) + subsystems:
MW handles broking, resource management, transaction handling and other middleware functionality on the top of WPP.
Subsystems handle incoming and outgoing signals and traffic.
The need for evolution

- Standardization of UMTS (Universal Mobile Telecommunications System) for 3G systems (GPRS is still 2G).

- Ericsson decided to develop the new SGSN using the same platform and components used for SGSN in the GSM market.

- The method to initiate software reuse between these two products were:
  - Identify commonalities between the two systems; domain engineering.
  - Analyze the existing solution for SGSN in the GSM market to identify reusable parts.
  - Develop an architecture that has the potential to be reused and be evolvable for the two systems.
The evolved software architecture is a layered one

- **Application-specific Layer**
- **SGSN-G and SGSN-W**
- **Business-specific Layer**
- **Common Services Layer**
  - including the application framework
- **System platform**: OS, ORB, Interfaces, run time machines etc.
Benefits of software reuse

- Size of the release is almost 470 KLOC, where 240 KLOC is modified or new code (MKLOC= Modified KLOC). 61% of the code is from the reused components -> improved productivity.
- Quality of the reused components is much better; they have significantly fewer trouble reports.

<table>
<thead>
<tr>
<th>Component</th>
<th>#TRs all</th>
<th>No. of Reused Comp.</th>
<th>%TRs Reused</th>
<th>No. of Non-reused Comp.</th>
<th>%TRs reused</th>
<th>Non-reused</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsystems</td>
<td>1519</td>
<td>9</td>
<td>44%</td>
<td>3</td>
<td>56%</td>
<td></td>
</tr>
<tr>
<td>Blocks</td>
<td>1063</td>
<td>29</td>
<td>41%</td>
<td>20</td>
<td>59%</td>
<td></td>
</tr>
</tbody>
</table>
MODERNIZATION OF LEGECY SYSTEMS
What is a legacy system?

• Computer systems or applications that are still in use although the technology / programming language / platform are outdated.

• Almost every company has it! It is functional and may be part of core business.

• Can be problematic:
  – Compatibility with other software
  – Oldness of technology; maintenance cost increases with time because of out-of-date technologies
  – Knowledge is lost / people have left the company / updated without updating the documentation
What to do with legacy systems in long term?

- **Replacement**: Developing a system from scratch or acquiring a new system that replaces the legacy system.
  - The cost is sometimes too high.
  - Requires extensive testing of the new system.
  - Suited when modernization cannot be applied.

- **Modernization**: more extensive changes than usual maintenance, but conserving portions of the existing system.
Modernization strategies

• **White-box modernisation**
  - Requires an understanding of the internal parts of a system, and involves re-structuring, re-architecting, modelling and creating abstractions of the system, and re-implementing the system or parts of it.
  - Tools for reverse engineering, creating models from code and semi-automatic restructuring of code are available.

• **Black-box modernisation**
  - Concerned with the input/output, i.e. the interfaces, of the legacy system, and 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.
Example: UI modernization

Screen scraping of UI

Source: A Survey of Legacy System Modernization Approaches
Example: DB modernization

- Translation of data (DB gateways) to connect to a modern DB.
- A database gateway is a specific type of software gateway that translates between two or more data access protocols.

Open Database Connectivity (ODBC) is Microsoft’s interface.
Java Database Connectivity (JDBC) is an industry standard defined by Sun.
Example: DB replication

- Copying and maintaining database objects in multiple databases that make up a distributed database system.
  - Changes applied at one site are captured and stored locally before being forwarded and applied to the centralized repository.
  - Replication provides users with fast, local access to shared data and greater availability to applications because alternative data access options exist.
  - Even if one site becomes unavailable, users can continue to query, or even update, data at other locations.
Example: Component wrapping

Function 2 can be accessed via an adapter or service broker. Function 1 is reimplemented in a Bean.
Services Oriented Architecture

- **SOA** is a way of designing systems composed of services that are invoked in a standard way.

- SOA is an architectural style—it is neither a system architecture nor a complete system.

- Typically WSDL (Web Service Description Language) for describing the web services and SOAP (Simple Object Access Protocol) as communication protocol.
Elements of SOA

• At a high level, a service-oriented system is composed of:
  
  – **Services**: reusable components that represent business or mission tasks, such as customer lookup or account lookup.
  
  – **Service Consumers**: These are clients for the functionality provided by the services.
  
  – **SOA Infrastructure**: The infrastructure connects service consumers to services. It usually implements a loosely coupled, synchronous or asynchronous, message-based communication model, but other mechanisms are possible.
    
    • Example: Enterprise Service Bus (ESB) to support web service environments
Example
Benefits of SOA

- **Loosely coupled** systems; a service may be replaced by another one.
- **Reuse** and **composability**: Services may be reused and combined in new ways to develop applications.
- **Interoperability**: services can talk together.
- One service can replace many desktop applications.
- Web services provide **easy access**.
- One of the most attractive promises of an SOA environment is that it enables **reuse of legacy systems** by wrapping them as services.
Legacy system challenges in SOA

- **Separating UI from functionality**
  - If user interface code is tightly coupled with business or mission function code, there will be a large amount of rework to separate out what is purely functional, given that services should be user-interface agnostic.

- **Asynchronous nature of SOA**
  - May be in conflict with synchronous nature of a legacy system.

- **Underlying components**
  - An organization might run into licensing issues with underlying commercial products where functionality is now exposed to a greater number of consumers, potentially outside the organization.
SOA challenges

• What shall be exposed as a service?
• What is the right granularity for the service? How generic should it be?
• Will the services scale to the size of the user community? How will performance be affected?
• What security measures need to be taken, given the nature of the user community?
• What are the quality requirements (response time, performance, number of users, etc.)?
The SMART process for migration


- SMART stands for Service Migration and Reuse Technique.

- The method helps organizations to make initial decisions about the feasibility of reusing legacy components as services within an SOA environment.
The SMART process

1. Establish Migration Context
2. Migration Feasible?
   - No
   - Yes
3. Define Candidate Services
4. Describe Existing Capability
5. Describe Target SOA Environment
6. Analyze the Gap
7. Develop Migration Strategy
Establish the context

• Understand the business and technical context for migration.
  – Information is gathered about the rationale, goals, and expectations for migration to an SOA environment, the technical and business drivers, programmatic constraints such as budget and schedule, and any previous related efforts or analyses.

• Identify stakeholders.
  – Information is gathered to identify who
    • (1) is driving and paying for the effort,
    • (2) knows about the legacy system and the target SOA environment (and what they know),
    • (3) creates the demand or need for potential services.

• Understand the legacy system and target SOA environment at a high level.
• Identify a set of candidate services for migration.
Feasibility decision point

- Determine if the legacy system is a good candidate for migration to services. Potential determinations are:
  - There is enough migration potential to continue.
  - The migration has potential but requires additional information to make an informed decision.
  - The migration is not feasible:
    - There are no identifiable consumers for the services.
    - Functionality in the legacy system does not have potential for use by multiple consumers.
    - Adequate input for the candidate services would require the construction of very complex applications.
    - There appears to be incompatibility between the legacy system and the target SOA environment. No functionality in the legacy system of a stateless nature.
Define candidate services

- Select a small number of services (usually 3 to 4) from the initial list of candidate services that were identified as part of the Establish Context activity.
  - Good candidate services are ones that perform concrete functions, have clear inputs and outputs, and can be reused across a variety of potential applications.
  - These candidate services are now specified more completely to include a definition of service inputs and outputs, and quality of service (QoS) requirements.
  - The Service Table artifact is created during this activity and updated.
Describe existing capability

• Gather information about the legacy system components that contain the functionality meeting the needs of the services.
  
  – descriptive data about legacy components—name, function, size, language, operating platform, age
  
  – architecture views
  
  – design paradigms
  
  – system quality
  
  – change history
  
  – user satisfaction
  
  – existing problems
Describe target SOA environment

- Gather information about the target SOA environment for the selected services including:
  - major components of the SOA environment
  - impact of specific technologies and standards used in the environment
  - guidelines for service implementation
  - state of target environment
  - interaction patterns between services and the environment
  - QoS expectations and execution environment for services
Analyze the gap

• Provides preliminary estimates of the effort, risk, and cost to convert the candidate legacy components into services, given the candidate service requirements and target SOA characteristics.
Develop strategy

• Develop a migration strategy that may include:
  – feasibility, risk, and options for proceeding with the migration effort
  – identification of a pilot project to migrate a simple service (or set of services) that has high visibility and low risk, especially if the organization is new to SOA.
  – order in which to create additional services
  – guidelines for identification and creation of services
  – specific migration paths to follow. For example, an approach may be to wrap the existing legacy code initially and rewrite the components in a different language in the future
  – needs for additional information or training
Don’t over-plan!
Case: DI Systems in Norway

- DI Systemer (DISYS) is a Norwegian software vendor with about 50 employees and a turnover of EUR 7 Mill in 2010.
- Key software products and services are CRM, accounting, payroll, invoicing, web portals and Application Service Provider.
- DISYS is participating in the EU project REMICS (Reuse and Migration of legacy systems into Interoperable Cloud Services; http://www.remics.eu) with the goal of modernizing their system.
- The project started in 2010 and will run for three years. Coordinator is SINTEF ICT in Oslo and there are 10 partners from 7 countries in the project.

Source: Software Engineering Challenges for Migration to the Service Cloud Paradigm
A view of the system
The existing solution

• The product portfolio is developed with different tools and languages and has evolved during decades.
  – The following tools/languages are being used in the software development: COBOL, Delphi, C# .NET, ASP .NET, and UML for modeling.

• The software is consumed by the DISYS customers in different runtime environments: desktop standalone installations, and traditional Client/Server solutions in a Local Area Network (LAN). Some users also host the DISYS software in virtual machines executed in DISYS ASP centre or in their own data centre.

• There are practically no shared resources, i.e. there is one software installation per DISYS customer. Each customer has several users, typically in the range from 10 to 50.
Problems with the existing system

- The report consumers do not have direct access to the system to initiate reports or change them according to their needs.
- DISYS software developers must develop and maintain reporting software in different software projects and programming tools.
- New and amended reports and report layouts must be programmed and installed at the user’s runtime environment as a part of a general software update.
- The sales department must keep track of several software products.
- There is no simple way to integrate reporting with data from 3rd party data providers.
Target solution: Software as a Service in the Cloud

- Obtain one reporting program solution common for all our deployment platforms.
  - Distribute new reports to users without having to compile and deploy new programs to every single customer installation

- Web based reporting: there should be no need to install anything else than a web browser to retrieve reports.

- Save costs by reducing the need for support during installation and use.

- Meet scalability issues by requesting more hardware from the service provider when needed.
Steps in the modernization process

• We have identified legacy software components that could be modernized and migrated to the service cloud.

• The migrated software architecture will be a combination of a modernized database in Microsoft SQL, Web-services and Asp .NET components.

• A data mapping from numerous source data providers to a model representation at the report data storage in the migrated system will be defined.
  – Contrary to the COBOL data structures in the legacy data storage of the DISYS programs, the third party data provider data storage is a MS SQL database.

• The performance, load-balancing and security of the migrated solution should be tested.
Conclusions

- Software reuse is a **key method** for saving cost and improving quality. Reuse take many forms.
- There are many challenges; **technical as well as organizational**
  - Understanding legacy software, understanding business goals with reuse, managing quality requirements (scalability, security etc.), defining a reuse or migration strategy, costs and risks
- **Technology support** is important:
  - Tools: Reverse engineering, modeling, code analysis, etc.
  - Proven architectural models
  - Methods
- Also **management support** is important:
  - Decision-taking, funding, resources