GN3 Software Architecture Strategy Best Practice Guide 4.0

Last updated:	07-09-2012
Activity:	SA4
Dissemination Level:	PU (Public)
Document Code:	GN3-09-185
Authors:	Branko Marovic (AMRES), Marcin Wrzos (PSNC), Marek Lewandowski (PSNC), Andrzej Bobak (PSNC), Antoine Delvaux (DANTE), Waldemar Zurowski (DANTE), Tihana Žuljevic (CARNET), Candido Rodriguez (RedIRIS), Spass Kostov (BREN), Ognjen Blagojevic (AMRES), Rade Martinovic (AMRES), Szymon Kupinski (PSNC), Anand Patil (DANTE), Paweł Kędziora (PSNC)

Table of Contents

Table of Figures

Table of Tables

1 Introduction

The software architecture and methodology used greatly influence the way the development is performed, productivity, quality of resulting product, and its flexibility and ease of maintenance. Having the actual goals, timeframe and milestones of software development defined, as well as process to achieve them in place, are all crucial elements of a successful software project. This document, targeting software project managers, software architects and developers, gives some best practices related to strategic issue of achieving a consistent software architecture and viable project life cycle.

Traditional approaches to software development are not suitable for designing flexible and distributed software systems. To find an appropriate solution for developing and designing such systems a shift in paradigm is necessary. This affects the structure of the software, what the key software assets are, and the way the development process is conducted. Technological, social and market forces have changed the software development process and environment during the last decade, leading to continuously growing adoption of agile software development methodologies and service-oriented architecture (SOA).

Many concepts and trends in software engineering overlap. Besides SOA, these include:

- Model-driven architecture (MDA).
- Usage of workflows.
- Visual programming and use of graphical editors for laying out components and wiring.
- Implementation through configuration and integration.

Which tools and approaches should be used depends on the needs and constraints of the software project, and the expertise and experience of the development team. Another two factors that greatly influence approach and tooling are respect of short deadlines which can be fostered by relying on readily available components, and tools that accelerate development versus long term control over all key aspects and components of the system.

Furthermore, multilevel structuring of software through services, components, modules, packages, libraries, classes, objects, methods and data requires a clear decomposition at each of the mentioned levels, as well as clear specification of their responsibilities and interfaces. Interfaces become crucial elements of software, while actual implementations, although necessary, become an increasingly changeable commodity.

2 Software Development Process

A software development process (also known as the software life cycle), is a set of defined steps in the development of a software product. There are several models for such steps, each describing approaches to a variety of tasks or activities that take place during the process. Repeatable, predictable processes are meant to improve productivity and quality. [SDP]

The most important software development process models are the following:

- **Waterfall processes**

  In this model the process is divided into multiple steps. After each step is finished, the process proceeds to the next step. As is typical with most waterfall models, there are five stages in the software development process:

  - Requirements:

    The behaviour of the software is defined, answering the question of "What are we going to build?" This step is critical to avoid unforeseen costs and deliver the right software.

  - Design:

    The design of the software is specified. The design plans include the software's interface (specifically the user interface) and its architecture (providing an abstract representation of the system). This involves ensuring that the software will meet the product requirements, and addressing how the software system will interface with other software products, underlying hardware and host operating systems.

  - Implementation:

    Software engineers program the code for the project, and the software is tested.

  - Release:

    Once code has been successfully tested, it is approved for release.

  - Maintenance:

    This stage covers the actions that have to be done if new problems are discovered or new requirements arise. This can take longer than the initial development of the software.

- **Iterative development**

  A cyclic software development process developed in response to the weaknesses of the waterfall model. It starts with an initial planning and ends with deployment with the cyclic interaction in between.

- **The Capability Maturity Model (CMM)** [CMM]

  CMM is a methodology used to structure, evaluate and improve an organisation's software development processes. Capability Maturity Model Integration (CMMI) [CMMI] evolved from CMM. Its goal is to help organisations improve performance, meet their immediate business objectives or avoid risks.

  It can also be used to compare maturity of projects or organisations, based on five levels of maturity: initial, repeatable, defined, managed and optimizing. At the initial level, processes are chaotic, poorly controlled, undocumented, and reactive. Once a development reaches a repeatable stage processes are project specific, but sufficiently documented to allow repetition of same steps and maintenance. At the defined level, existing processes are standardised and consistently followed across the organisation. These processes are proactive, and subject to some improvements. Once the managed level is reached, processes are quantitatively controlled and managed using predefined metrics. At the top level (optimizing), process management is focused on ongoing process optimisation and qualitative improvements.

2.1 Agile Software Development

Agile software development was developed to react against "heavyweight" methods, which use a heavily regulated, regimented, micro-managed the waterfall model. Initially, agile methods were called "lightweight methods". Agile software development methods are based on the principles defined in the Agile Manifesto [AGILE]. Some of the principles behind the Agile Manifesto are:

- Customer satisfaction through continuous delivery of useful software.
- Working software is the principal measure of progress and is delivered frequently.
- Even late changes in requirements are welcome.
- Close cooperation between business people and developers.
- Face-to-face conversation is the best form of communication, although distributed groups are welcome.
- Projects are built around motivated individuals.
- Continuous attention to technical excellence and good design.
- Simplicity.
- Self-organising teams.

Some of the agile software development methods are:

- Agile Modelling.
- Agile Unified Process (AUP).
- Agile Data Method.
- Dynamic Systems Development Method (DSDM).
- Essential Unified Process (EssUP).
- Extreme Programming (XP).
- Feature Driven Development (FDD).

- Getting Real.
- Open Unified Process (OpenUP).
- Scrum.
- Lean software development.

Agile methods break tasks into small increments with minimal planning, and do not directly involve long-term planning. Iterations are short time frames that typically last from one to four weeks. Each of those iterations are worked on by a small team and carried out throughout the software development cycle, including planning, requirements analysis, design, coding, unit testing and acceptance testing. The goal is to have an available release (with minimal bugs) at the end of each iteration. Multiple iterations become a product release.

It is emphasised that teams should have face-to-face communication about written documents if the team is in the same location. Distributed groups should maintain daily communications through video conference, voice, email, etc.

Most agile teams work in single small rooms, which improves the communication between people there (with a typical size of 5-9 people). Larger development efforts may be delivered by multiple teams working toward a common goal or different parts of an effort. All agile teams contain a customer representative who acts on behalf of stakeholders and makes a personal commitment to being available for developers to answer mid-iteration problem-domain questions.

During the daily face-to-face communication, team members, the customer representative and any interested stakeholders report to each other what they did the previous day, what they intend to do on the day and what their roadblocks are. This prevents problems from being hidden.

Agile software development has the working software as the primary measure of progress. Thanks to this and the daily face-to-face meeting, it produces less written documentation than other methods.

However, there are a lot of voices against these agile methods. Agile methods are placed at the opposite side of “plan-driven” or “disciplined” methods. This distinction is not right, as it implies that agile methods are “unplanned”. However, they do rely on “adaptive” processes.

Agile practitioners believe many of the criticisms to be misunderstandings of agile development. Criticisms of agile development include that it:

- Is often used as a means to get money from customers through lack of defining a deliverable.
- Lacks structure and necessary documentation.
- Lacks something if there aren’t any senior-level developers.
- Incorporates insufficient software design.
- Requires meetings at frequent intervals, which can be expensive to customers.
- Can lead to more difficult contractual negotiations.
- Can increase the risk of scope creep due to the lack of detailed requirements documentation.
- Is impossible to estimate the work effort needed because at the beginning of the project no one knows the entire scope/requirements. [AGILECRIT]

These criticisms are concerned that agile development is quite similar to “cowboy coding”, which is the absence of a defined method, since it has frequent re-evaluation of plans, emphasis on face-to-face communication, and sparse use of documents. This actually is not true, as agile teams do follow defined processes.

An interesting related concept is Lean software development, which mixes Agile principles with key principles of the Toyota Production System (TPS). TPS is based on “Continuous Improvement and Respect for People”, which are applied to software development and combined with Agile principles form the principles of Lean software development, as follows [LEAN]:

- Eliminate waste.

  Everything not adding value to the customer is considered to be waste, including unnecessary code or functionality, delays in software development process and bureaucracy.

- Amplify learning.

  Instead of adding more documentation or detailed planning, different ideas could be tried by writing code and building, facilitating learning in the process.

- Decide as late as possible.

  Leave all options open as long as possible.

- Deliver as fast as possible.

  Make deliveries often and quickly as soon as a requirement is met.

- Empower the team.

  Managers listen to developers how the job should be done.

- Build in integrity.

  Understand and solve problem at the same time, not sequentially.

- See the whole.

  Have well-defined relationships between different contributors to produce a system with smoothly interacting components.

Respect for people and continuous improvement, and a “challenge everything” and “embrace change” mindset are the pillars of Lean. If a Lean adoption program ignores the importance of these, fostering unquestioning Lean adoption, then the essential understanding and conditions for sustainable success with Lean will be missing.

2.2 Implementation of Agile Development

2.2.1 OpenUP

The Open Unified Process (OpenUP) is a set of best practices for software development that has been created based on agile methodologies. OpenUP is driven by four, core principles listed and compared to the Agile Manifesto in Table 2.1.

OpenUP	Agile Manifesto
Collaborate to align interests and share understanding.	Individuals and interactions over process and tools.
Balance competing priorities to maximise stakeholder value	Customer collaboration over contract negotiation.
Focus on architecture early to minimise risks and organise development.	Working software over comprehensive documentation.
Evolve to continuously obtain feedback and improve.	Responding to change over following a plan.

Table 2.1: OpenUP and Agile Manifesto comparison

OpenUP structures the project lifecycle into four phases (see Figure 2.1):

- Inception.
- Elaboration.
- Construction.
- Transition.

Team members in an OpenUP project contribute their work in very small increments, which typically represent the outcome of a few days of work. The application evolves one increment at a time, and progress is effectively seen every few days. Such approach increases work visibility, trust and teamwork.

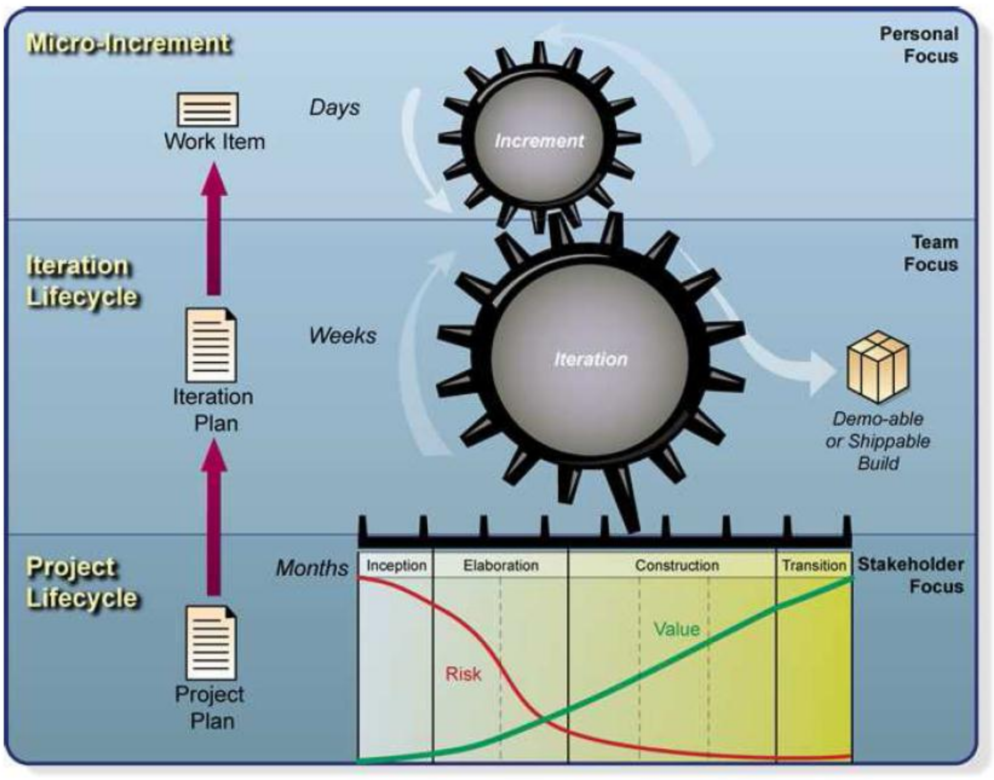

Figure 2.1: OpenUP life cycle [OPENUPLIFE]

OpenUP can be successfully applied to projects that are characterised by the following attributes:

- At the beginning of the development requirements (functional and technical) are not completely defined.
- There are frequent releases (e.g. monthly) containing relatively small changes and new features.
- Users (clients) test the software and provide rapid and comprehensive feedback (users' feedback effects in frequent requirements changes).
- Budget and deadline of the project can be changed (because of changes of the requirements).
- The entire development process is software-oriented (and not documentation-oriented).
- A representative of users (clients) can help developers whenever they need help.

For further details on OpenUP, see [OPENUP].

2.2.2 OpenUP, ITIL and SA4 Best Practices

The Information Technology Infrastructure Library's (ITIL) "ITIL Lifecycle Suite," [1] is a set of concepts and practices for Information Technology Services Management (ITSM). It is an operational management framework designed in response to business needs. ITIL provides a common language for describing IT concepts and activities related to service delivery and it bridges the gaps between developers, service providers, and management through its definition of a set of common practices and processes that are relevant to a service life cycle, IT service chain, organisational functions, and supporting components, procedures, and artifacts. ITIL is popular with IT professionals and organisations, due to its individual approach, strong training certification model, and associated market value for both companies and individuals.

. Since it applies not only to software development, but also to other technical (not necessarily IT) projects, ITIL defines management and development procedures on a very general level. As a result, its procedures cannot be used directly in GN3 software projects. However, such procedures may usefully inform GN3 best practice. ITIL has been designed to relate not only to regular daily activities (such as problem management, release management) but also to executive duties (e.g. strategy). The usage of approach methodologies described in this document is presented in Figure 2.2, below.

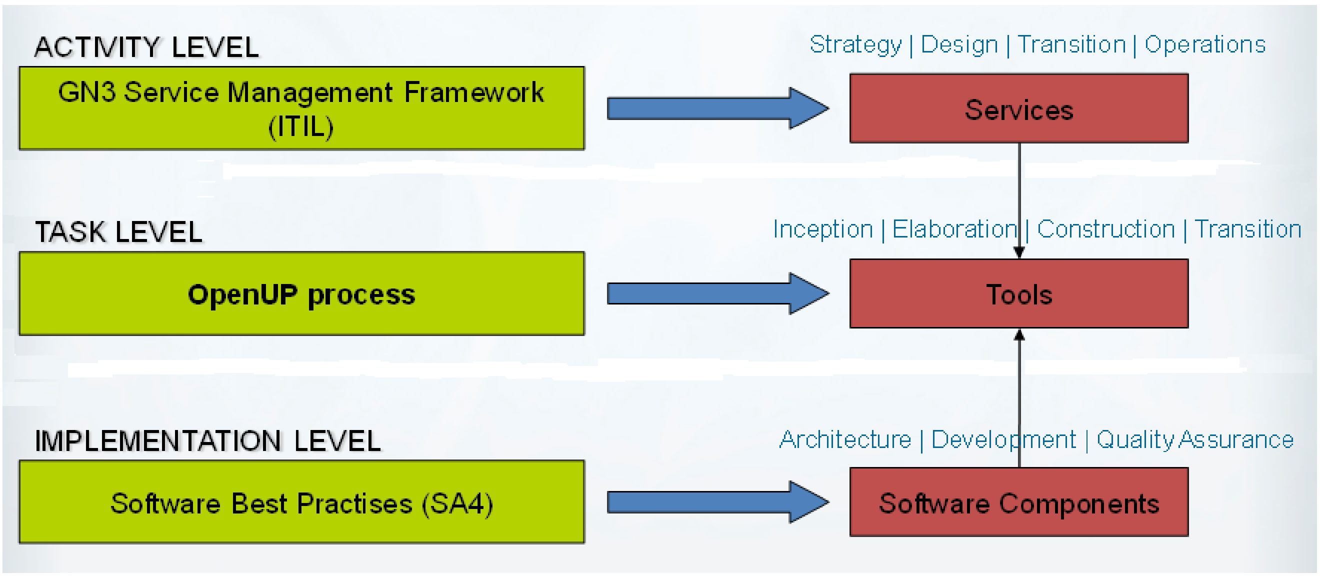
ACTIVITY LEVEL
GN3 Service Management Framework (ITIL)
Strategy | Design | Transition | Operations
Services
TASK LEVEL
OpenUP process
Inception | Elaboration | Construction | Transition
Tools
IMPLEMENTATION LEVEL
Software Best Practises (SA4)
Architecture | Development | Quality Assurance
Software Components

Figure 2.2: Best practices and conventions

2.2.3 Usage of Software Tools

To foster agility in software development projects, the management of development teams and bureaucracy should be kept to a minimum. Agile teams are at their best if they can self-organise, build their own processes following their intuition gained from past experience, and improve best practice after each project iteration.

[1] The Stationery Office, 2011 (ISBN: 9780113313235); see also http://www.itil-officialsite.com/

However, self-management within Agile development relies on up-to-date lists of work items in day to day operations. Priorities need to be given to work items based on customer requirements, development scheduling, problem severity or other specific metrics derived from collected information from products owners and developers. Tasks can then be assigned to individual developers or pairs (XP) as the project progresses. Work progress and information needs to be passed along to other developers or testers quickly and efficiently, without being slowed down by overly defined forms and complex document templates.

Such self-organisation can greatly benefit from versatile project development tools such as Trac [TRAC], Redmine [REDM], Endeavour [ENDEV] or the Atlassian tool suite [ATL]. Tools that do not enforce a specific methodology but provide the developers, team leaders and product owners with means to achieve their daily goals in a Getting Things Done ([GTD]) approach.

Using these tools, the team can list feature requests, bug reports, use cases, work items and other tasks easily and quickly. These items can then be organised according to various criteria (e.g. problem severity, customer priority, work effort) and assigned to the appropriate persons. From those organised lists, project plans can be sketched, risks evaluated and release progress followed in an easy, non-intrusive way suitable for use in daily work.

2.2.4 Project and Release Management

Although agile methodologies allow fast and flexible software development, their improper application can decrease the quality and maintainability of the product. Frequent changes in requirements or architecture can be reflected in unused or abandoned source code and in illegible and overly complicated project structure. The approach of a close deadline or lack of incentive may tempt developers to provide features similar to already implemented ones by copying and pasting, and superficially adapting existing code. This results in code that is difficult to maintain. The growing mess further complicates maintenance and extensions and also decreases motivation. Furthermore, convoluted code only encourages developers to add more clutter. Therefore, agile development requires code to be kept as tidy as possible at all times. This is only possible if the code is frequently refactored to fully reflect actual reality in terms of scope, design and intentions. Unused and untested code should be removed from the source code repository, so it cannot be used (accidentally or deliberately) by developers who are not aware of all weak points of the exploited materials. To avoid cluttered and abandoned code, its structure and content should be reviewed regularly. Every inspection should ensure that duplications, inconsistencies and inadequate programmatic structures, names or files are eliminated. Although significant refactoring may be especially scheduled to keep the current production system running and to minimise duality between developed and maintained code, it is important to keep the code and code related documentation neat at all times. Preserving code integrity and cleanness greatly simplifies its maintenance and greatly reduces the need for especially scheduled refactoring.

As most developers work on multiple projects, they have to divide their time between different activities. Development teams are often dispersed. Although this approach is convenient, both for management (effective employment policy) and developers (avoiding routine), it must be applied carefully and steps need to be taken to increase the level of coordination. High-quality software can only be achieved if clear goals are defined, priorities agreed and if the people involved have a sufficient percentage of their daily allocated time to spend. An agile approach assumes that iterative working versions of software are released on a regular, frequent (e.g. once a month) basis and that rapid feedback is provided by users. User feedback is the most important factor in preparing the development plan for the next release. Thus all issues raised by users (e.g. performance, functionality, improvements) need to be addressed and scheduled to be solved in one of following iterations. If user issues are not taken into account (these do not have to be immediately implemented, sometimes a comment or clarification is sufficient), the basic concept of agile development is not followed.

Agile methodologies are welcomed by developers, because of their lightness and lack of formality. However, lack of formality does not mean lack of rules. To achieve success and deliver high-quality software on time and schedule, project managers, in cooperation with the development team and users, must follow methodology guidelines. An agile development life cycle is useless without strengthened internal planning and achievement monitoring. Selectivity in following guidelines may cause poor quality and serious delays in a project.

2.3 Test-Driven Development

Software testing is a process of checking whether source code written by software developers meets the requirements defined by the client. Testing is usually carried out automatically, with tools for automated test execution (e.g. during software compilation) or manually, by following defined scenarios (executing parts of code) against a defined set of input data. Manual testing is discouraged, due to developers' and tester's tendency to pay less attention to and devote less effort on testing code that was already successfully tested in previous iterations of the software development process.

There are basically two approaches regarding creating tests in software development: Test-First Development (TFD) and Test-Late Development (TLD).

- Test-Late Development allows tests to be designed after the source code is written. Usually tests are run before shipping the solution to the client. The major drawback of TLD is when a failing test is found, it requires modifying existing source code, which may also be used elsewhere in the software, causing those units to work improperly.

- Test First Development enforces designing the tests before any code is developed. Test-Driven Development (TDD) is the most popular TFD methodology. It focuses on creating exhaustive unit tests (see the *Unit Testing* section of the *Quality Assurance Best Practice Guide*) based on user scenarios. The definition of user requirements and functionalities operating them should be carried out according to the following steps:

    - Describe the relevant scenarios of software usage. Scenarios should be defined by (or at least in cooperation with) future users of the software.
    - Define functionalities needed to be implemented in order to realise each scenario.
    - Provide a set of examples of expected software behaviour (input, expected output, expected change of system objects' state, expected behaviour of the system).

The software development life cycle steps in a TDD approach are:

- ○ Design tests that describe the desired behaviour of the relevant part of software (method, class, etc.).
- ○ Run the tests to make sure the code can be executed and is not failing.
- ○ Write the source code, implementing the expected behaviour and satisfying the tests.
- ○ Run all tests. If any test fails change the source code and re-run all tests.
- ○ Refactor the code, remove duplicated fragments, change design of all unclear fragments of the software, re-run all tests.
- ○ Repeat until all desired behaviour is implemented.

For further information about TDD, see [TDD].

The writing of software tests should begin with gathering knowledge about the software’s domain and expected behaviour. The knowledge-gathering process should be performed in very close cooperation with future users of the software (domain experts). Domain experts should describe the actions they want to perform and expected results (user stories and scenarios). Developers should also design tests based on provided user stories. The last step of the process is the implementation of the tests.

There is a set of rules that should be followed regarding the design and implementation of unit tests:

- One test should cover one action in the application. It is best to design multiple tests against one action in order to exhaustively test the behaviour of the application.

- The test’s structure should be easy to understand and self-explanatory. It is best to use a “given-when-then” scheme, e.g. “Given a list of products, when a product’s best before date is due, it should then be flagged as being ‘on sale.’”

- Tests should not be dependent on other modules of the software. Each test should be provided with its own data set that will allow each test to run separately. Note that results of tests should not influence other tests or be used as data sources or inputs for other tests.

- Data values should vary with regard to the expected range of the input values. It is also advised to test application against data values that are out of expected ranges and against incomplete or corrupt data.

- If structure or logic of the software changes due to a change in requirements, the source code of all failing tests should be adjusted (which sometimes means rewriting) and tests shall be run again.

TDD expects tests to cover as close to 100% of lines of code as possible. Although this approach may be considered controversial, it is advised to cover all public methods with unit tests.

2.3.1 Test Doubles

It is often difficult to provide elements required by the test to run properly because some parts of the system are not yet implemented, are inaccessible, or their state is difficult to manage via tests. In such cases one can use:

- Dummies – objects passed around but not actually used, e.g. in order to fill list of parameters. Examples: null values, empty implementations of required interfaces, etc.
- Stubs – minimal implementations of interfaces or base classes, e.g. methods returning hardcoded values.
- Fakes – working shortcut implementations not suitable for production environment, e.g. in-memory databases.
- Spies – similar to stubs. Spies record which members were invoked so the test can verify that members were invoked as expected.
- Mocks – objects simulating system objects' behaviour, created by a dedicated framework and configured to provide return values or expect particular invocation parameters or sequences.

Dummies, fakes, stubs and spies are usually implemented by the developer. Mocks are usually created with a dedicated framework, e.g. Mockito [Mockito]. List of popular mock objects frameworks is available at [MockObj].

Further materials regarding the use of mock objects in unit testing may be found at [MocksTDD].

2.3.2 Shortcomings of TDD

A high percentage of passing unit tests may bring a false sense of security, resulting in reduced focus on integration and functional tests.

Unit tests are usually created by the developer, who also writes the source code. This may result in blind spots, both in designing the tests and implementing the source code, which may then produce a misinterpretation of the requirements and the writing of incorrectly functioning code.

Badly written tests (so-called fragile tests) may, over time, become a substantial maintenance overhead. This weakness of TDD can be solved by applying pair programming approach or code reviews. More information on these techniques can be found in the *Internal Code Reviews* section, in the *Quality Assurance Best Practice Guide* (see [QA]).

2.4 Bringing Agility into Deployment and Operations

The development of the GN3 test infrastructure and growing availability of virtualised resources, emphasises the need to modify the software engineering process in order to allow agility beyond the development phase.

Enabling self-service delivery of computing resources, automation of test procedures, and ease of configuration replication enables the frequent deployment and configuration of complex multi-tier systems. This enables the process to reach beyond continuous integration by performing rapid and frequent migration from tests to production infrastructure, and avoids surprises such as misconceptions or omissions between delivery, deployment, and operations. Archiving such a seamless transition between stages of software process is related to a DevOps approach to testing and acceptance procedures, configuration management automation, and modification of overall process.

2.4.1 DevOps

While agile development was gaining wide acceptance in the software development community, it became apparent that a similar approach would also benefit the software-engineering process related to deployment and operational procedures. This resulted in establishment of a DevOps concept and movement to develop principles, methods and practices that would reduce the 'distance' between development and operations teams [DevOps]. At the core of DevOps thinking is the realisation that software development (Dev) and operations (Ops) teams need to communicate and collaborate to enable organisations to efficiently transform developed software into running services. Furthermore, deployment, operations and maintenance concerns must be considered when developing any reasonably complex IT system.

Both goals can be achieved by short release cycles and rapid deployment, so that each newly developed software feature is deployed as soon as it is ready, thus producing rapid feedback to the developers on the ability for system administrators to deploy, configure and operate the upgraded software. Assessing, streamlining, optimising and repeating the transition between development, internal testing, and certification, pre-production deployment, and user-acceptance testing shortens the required time and minimises novel challenges, ensuring that the system or new feature is ready to go to production when needed.

DevOps is particularly suited to delivering "software-as-a-service" over the Internet, and in organisations where development, integration and operation are handled simultaneously. Therefore, this approach is extremely well-matched with ongoing GÉANT related developments, services, and roles of involved organisations.

DevOps is closely related to agile, continuous integration, configuration management, virtualisation, and cloud computing. It emphasises the importance of rapid, automated and repeatable creation or allocation and configuration of execution environment, as well as defined and measureable, agile processes for operations. In order to break the barriers between development and operations, speed up delivery, and enhance the supplied solutions, DevOps employs some practices that are well suited with use of virtualised or cloud infrastructure:

- Self-service paradigm in allocating of computing resources.
- Use of standard machine images from a common store for automated installation.
- (Near) zero or automated configuration of applications or components.
- Loose coupling of architectural elements.
- Scaling through replication, which is the simplest way to address increased load.
- Use of database sharding [Sharding] to improve performance.
- Availability hardening through fault injection and design of solutions, providing either recovery or containment of adverse effects.

A summary of some closely related concepts follows:

- Continuous Integration

  Continuous integration (CI) provides an important step in implementing agile software development processes. Automated building and testing of software gives rapid feedback to developers about problems.

- Continuous Deployment

  It is quite possible for software that builds correctly and passes integration tests to fail in the production environment. Continuous deployment fills the gap by taking continuous integration a step further with live, staged deployment to allow user acceptance testing (UAT) of the developed software, and enforce streamlining of all steps and procedures necessary to establish the production version of the system. Continuous integration (CI) frameworks, such as Hudson or Jenkins, allow external tests, which permits this level of continuous deployment.

- Automated Configuration Management

  DevOps extends the ideas of continuous deployment by combining them with best practices of system administration through automated configuration management and monitoring. Configuration management toolsets, such as Puppet, Chef and Quattor, permit the configuration descriptions to be handled the same way as code (version-controlled and managed).

- Cloud Computing

  Cloud services are commonly categorised as providing an infrastructure, a platform or a user-facing software application as a service over the network (Infrastructure as a Service (Iaas), Platform as a Service (PaaS), and Software as a Service (SaaS), respectively) [Cloud Services]. One aspect of what makes something a “cloud” service is that it must have an API that is accessible over the network. This allows automation and configuration management to be extended from application systems to the infrastructure or platform on which the service runs.

Specialisation between development and operations is often desirable, but development and operations teams can often have conflicting goals, as developers are pushed to rapidly deliver new features, while operators are pushed to keep the systems stable and reliable. Developers are often seen as feature-driven, freewheeling, collaborative, self-organising, and organic, while operators, with their policy-driven and command-and-control attitude focused on uptime and compliance. This difference leads to isolation between project groups, and may greatly affect efficiency and overall service quality. DevOps aims to bring these sides together with the common understanding that cooperating will bring the greatest benefits to the organisation delivering the service.

Using virtualised infrastructure or cloud services for development and operations, often in an environment not fully under the team’s control also requires the teams to think through the configuration and integration of components more thoroughly, and at an earlier stage, as it is no longer good enough to get the system working on a developer's laptop or on a hand-crafted server configuration.

An illustration of relationship between software engineering tools used and deployment and operations is given in Figure 2.3, together with two examples of possible migration paths towards DevOps. It shows the evolution from fixed infrastructure to virtual cloud infrastructure (on the horizontal axis) and from “traditional” software development and deployment to continuous delivery (on the vertical axis). Moving towards clouds yields the benefits of elastic resource provisioning, while moving towards continuous integration permits frequent automated deployment.

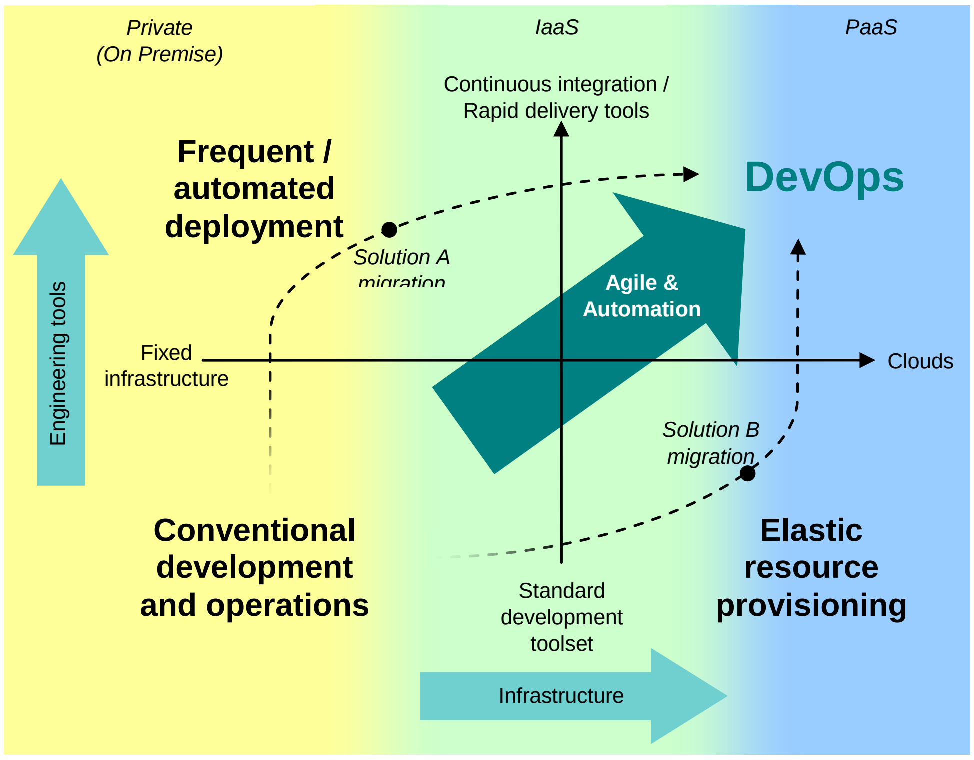

Figure 2.3: Migration from conventional developments and operations to DevOps

DevOps brings in other specialities such as Quality Assurance, Security, and Deployment to work together in close cooperation. The core aim is to ensure efficient, reliable, flexible processes from development to operations, resulting in production services that are more reliable, maintainable, upgradable and when necessary, easier to troubleshoot.

2.4.2 ITIL and DevOps

The ITIL procedures, tasks and checklists can be implemented in different organisational and infrastructural settings and aligned with various software development methodologies. ITIL encompasses elements relevant to technology service delivery, and is not solely focused on design of technology solutions. However, some of the ITIL elements with agile and DevOps techniques are clearly complementary:

- ITIL provides guidelines for sustaining the IT service-availability and supporting the agreed level of service over time. This is done by focusing on reliability, maintainability, serviceability, resilience, and security, during Service Design. Furthermore, flexibility and scalability of infrastructure provisioning in cloud-based services streamlines a significant part of ITIL capacity management.

- ITIL Service Transition relates to the delivery of services into operational use, so its change-, asset-, configuration-, release-, and deployment-management processes deal with changes. Providing an agile and cost-effective path for this may greatly improve the software rollout, distribution and installation, as well as management of user expectations.

- Finally, ITIL Service Operation points towards best practices for delivery of service levels, which also includes event and incident management, problem monitoring and solving, and fulfilment of user requests, while balancing between reliability and cost.

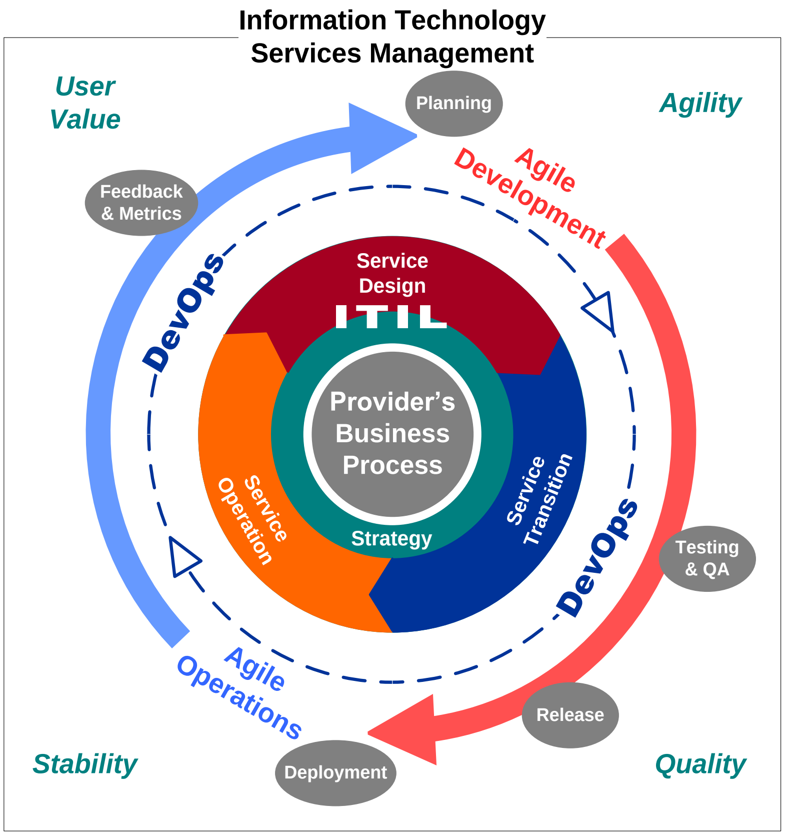

Figure 2.4: Agile, DevOps, and ITIL (Adapted from [DevOps_1])

Embracing agile methodologies requires giving Service Design priority (Figure 2.4). On the other hand, agile development and operations strongly emphasise transitional elements of ITIL. The change management and control of risks become agile, as they are done with automated testing and deployment in short cycles. This also requires adequate communication between developers, administrators, and service/product support teams.

ITIL is a framework offering some best practices that are at a level above the agile development. However, implementations of ITIL can be rather heavyweight, and are often coupled with waterfall and stage-gate development and deployment approaches, which presents a challenge for aligning and binding the agile and DevOps with ITIL processes and functions. Using continuous delivery events, records, and tools to support ITIL change and configuration management should facilitate amalgamation of ITIL and DevOps, allowing for change traceability and full integration automation. ITIL can be and is used to provide the common language of service development in conjunction with a complementary, agile approach to these tasks.

2.4.3 Extending Testing and Delivery

In order to improve agility of testing and deployment process, the tools that replace manual performing of tests and deployment should be applied whenever possible, and will help ensure that the procedure is repeatable, automated, and that it produces traceable artifacts. Such an approach enables continuous delivery, allows for an extremely short release cycle and warrants that the performed steps may be reproduced. This method may greatly affect the release cycle schedule and procedure described in the Release and Deployment *Management* section of the *Software Developer Best Practice Guide* [SDG].

In order to ensure automation of testing and QA, the software is deployed to a series of environments where the tests are run [a blog at Cloudosity: Production and Non-Production environments, http://www.cloudosity.com/2010/08/19/production-and-non-production-environments/]:

- Development (DEV)

  Build and some basic functional and unit tests are done in DEV environment. There may be several instances of this environment, as it is heavily used by developers.

- System Integration Testing (SIT)

  Integration tests and some additional functional tests are performed in a SIT environment used by Continuous Integration tools. It is also used by developers, but its usage is triggered by commits or nightly builds.

- User Acceptance Testing (UAT)

  User Acceptance Testing is done in a UAT environment. Several changes of a Release Candidate (RC) may be combined and made available to testers and internal users in order to verify that the changes function as intended. Alternatively, this testing may be performed on individual feature branches.

- Staging

  In highly critical missions, this testing may be followed by deployment to an optional staging environment that almost fully emulates the production one. All changes that are to be included into a planned release are packaged and deployed here. A variation of this approach is deployment of a release candidate on the not-critical production instance serving selected representative users willing to tolerate possible usability problems.

- PROD (production)

  Final deployment of operational product is done into PROD environment.

- Stress and Volume Testing (SVT)

  There may also be an asynchronously operated SVT environment, which is fully separated from the common DEV – SIT – UAT – Staging – PROD path, but may be quite very close to Staging or PROD environment.

All these environments can be virtualised and managed through a set of tools accessible by developers. Not all of listed environments must exist simultaneously, but developers and operators must be able to easily activate or recreate them.

By stepping up from the infrastructure level to the software platform on which services run (including databases, web servers, messaging systems, etc.) through use of cloud technology, DevOps approach can ensure that the platform is consistent for all stages of development, QA, and operations.

3 Software Design

A well-thought-out software architecture is crucial. It is characterised by an ability to add new related features without major effort, and an ability to sustain changes.

Before the start of coding, the software architecture should be outlined, key scenarios selected and use cases covered. Clear decomposition, allocation of responsibility, definition of functionality and elimination of repetitions are key elements of good design that dramatically affect later development.

It is useful to work with several users and customers from the very beginning. For example, try to identify and consult at least two users participating in the same use case but with slightly different views, or have at least two clients for an API. They will provide different perspectives and sets of requirements and usages within the same core paradigm. This makes easier to prioritise requirements and define key features. Missing such feedback may affect the overall design and result in failing to spot some key elements of design, which could backfire later. At the same time, getting feedback from system users helps finding an appropriate level of generalisation while avoiding unnecessary complexity. With them, problems in design, implementation, and usability will be detected early, which is crucial in an agile development philosophy.

It is essential that developers have a relatively clear target, and an idea of what should be achieved in the current development cycle. Basic design requirements must be stable during a development cycle, and no significant requirements should be added during a cycle.

During design phase and the coding that follows it, potentially useful features that are not critical should be avoided. The design specification should cover everything that is not controversial at the time of starting the coding, while solving any remaining problems can be left for a later phase. The presence of dilemmas during design and implementation often means that the final outcome can allow, or may indeed require multiple answers or solutions. Therefore, presence of such dilemmas is a clear indication that the design should be made more flexible in order to be able to accommodate all possible options.

3.1 Design of Software Interfaces

The primary element of software design related to the goals and approaches described in this guide is a separation of responsibilities and a specification of clear, yet reusable, interfaces:

- APIs and other software interfaces have a significant impact on software flexibility and developer and user vision of the system or the services provided through these interfaces. Since they describe what the system provides, they define users' expectations and the ways they interact with it, and determine what features must be implemented. Interfaces are the element of design that should be both comprehensible (to facilitate appropriate usage) and resilient to evolution (to minimise dependencies). An interface should be as specific as possible in terms of the features it provides and the target domain it represents.

- The software interface design should lead developers, at both client and service side, as much as possible. For example, instead of having a sequence of actions prescribed in an attached document only, it is more effective to enforce such a sequence by using factory methods that create objects required in calls of subsequent methods. Each interface should be restrained to a single level of abstraction.

- The interface should hide the details of the underlying implementation. It is usually easier to change the underlying implementation of an interface than the interface itself, since such a change affects all parties that use it. For example, an object model based on XML-related data transfer objects for method parameters and results should be avoided, as this may imply heavy memory consumption and make it very difficult or even impossible to change the actual technology behind the interface. Likewise, inheritance of exposed classes should be avoided in favour of the composition and implementation of interfaces. Flexibility is achieved by using simple data objects or simple abstract interfaces for arguments and returned values. The actual objects should be provided through the interface or from an object factory.

- It is often useful to introduce an interface to encapsulate and generalise even a single-known usage scenario or existing implementation. In such cases, when absence of other users or alternative implementations makes it difficult to come up with a generalised interface design, using software patterns and related existing interface specifications may help in defining the content of the API. Using facades and adaptors hides uncertainties and specifics of mutable or questionable implementations. It also reduces coupling between the service implementations and used lower-level components.

- There is a risk that the dependency on current execution platforms, low-level components or operational assumptions may limit further evolution of software, or require significant adaptation to a new or modified underlying platform or service. One should therefore minimise software dependencies by building pluggable units. This avoids hardwiring and dependency on mutable assumptions and components. Use a standard-based or neutral interface instead, with thin wrappers (adapters) separating the abstract concept from the actual implementation of a system.

- Before an interface is defined, it is important to check whether an appropriate or applicable interface specification already exists. If there is already a suitable standard or public interface, use it. Such specifications often exist as standards defined by international organisations, industry initiatives, or are specified though the Java Community Process (JCP) as Java Specification Requests (JSRs), or by the Internet Engineering Task Force (IETF) as Request for Comments (RFCs).

- Even if an already existing related interface is not completely suitable, consider whether it could be extended or a subset could be used as a template for writing a specific interface specification. Such usage of a popular API allows developers to easily familiarise themselves with a new API and transfer already existing knowledge.

- Generic interfaces that allow passing various numbers of parameters through their boundaries should be used whenever the purpose or concrete scenario of interface application is not clear in advance. Such soft and permissive interfaces pass parameters as key-value pairs, often allowing delivery of almost any type of content. Examples of such contents are software configuration, HTTP request parameters, session attributes, or user context within an application. However, whenever there are only a few parameters to be transferred or parameters can be adequately represented by objects, it is better to use more rigid interfaces, as they allow stronger compile and run-time control.

- It is good to keep high-level interfaces stateless wherever possible. This makes the design more adaptable to usage with Service-Oriented Architecture (SOA). While the design of interfaces at the object oriented programming level should rely on its state and context to minimise the number of arguments and the need to maintain the conversation context, SOA methods encapsulate complex business processes involving many objects, so they can rely on the arguments to specify all necessary details. However, if a stateful interface significantly simplifies the usage for both parties and reduces resource use (e.g. communication duration or costs), it should be adopted. In such a case, the lifespan of the elements of the stateful context should be specifiable through the interface in a way that clearly defines their scope and lifespan. Also, the interface specification should state the outcome if the sequence of contract is not regularly followed or concluded, and the corresponding functionality has to be implemented.

- Using the Enterprise Service Bus (ESB) architecture enables a more adaptable design of distributed system that comprises collaborating services. The bus paradigm provides the additional advantage of freeing service developers from having to deal with common aspects such as security, service discovery and message management, allowing them to concentrate on the direct implementation of business processes.

3.2 Software Decomposition

There is a strong tendency for software development to be done through the development of components, and customisation and integration of existing solutions. This is caused by:

- The presence of legacy applications.
- A growing market of software components.
- Availability of open source libraries and modules.
- Existing large component products, which cover all business segments.

These factors push for large-scale componentisation of the software. Such trends are also evident in the organisation and structure of modern software design. While object-oriented programming (OOP) embraced and shed another view on the concepts of structured programming, Service-Oriented Architecture (SOA) was similarly built upon OOP, but introduced new rules appropriate for orchestration of distributed and increasingly independent services. Each of these software decomposition approaches introduces a new complementary set of concepts, guidelines and rules for laying out software to address the issues of the corresponding decomposition level.

3.2.1 Service-Oriented Architecture

Service-Oriented Architecture uses services as fundamental elements for developing applications to make it possible to quickly adapt to external or internal changes. The goals of SOA are therefore to reduce friction, enhance visibility and to allow the software, developers, and customers to thrive on change instead of suffering from it.

Under the SOA approach, modular, accessible, self-describing, implementation-independent, interoperable, and reusable components are published as services which can be remotely invoked and consumed by other applications or combined with other services. SOA promotes the idea of using loosely coupled services, with each representing a block of business functionality. These blocks can be combined to support a business process, providing an approach that supports their discovery and use:

- Services are often designed to ignore the context in which they are used.
- Value can be created by combining modular services.
- Non-proprietary service interfaces.
- Implementation can be changed in ways that do not break all the service consumers.
- The service consumer expresses ‘intent’.
- Service providers define ‘offers’.
- There is need for mediation that is handled by SOA.
- Typically, model-oriented business logic is wrapped by (web) services.

When designing software architecture, numerous challenges can arise, especially when shaping architecture according to service-orientation design principles, such as:

- ‘Service Statelessness’ – By minimising the amount of its state management responsibilities, the availability of a service is increased. This is directly associated with the service’s ability to be effectively scaled in response to high-reuse requirements.

- The principle of ‘Service Loose Coupling’ promotes the independent design and evolution of a service’s logic and implementation while still guaranteeing baseline interoperability with consumers that have come to rely on the service’s capabilities. There are numerous types of coupling involved in the design of a service, each of which can impact the content and granularity of its contract. Achieving the appropriate level of coupling requires that practical considerations be balanced against various service design preferences.” [ERL]

At the business level, services can be seen as entities encompassing vertically packaged elements that closely integrate with related business processes and several layers of software. They are often provided by software systems or suites and their modules identified through popular three-letter acronyms, such as:

- ERP (Enterprise Resource Planning).
- CRM (Customer Relationship Management).
- SCM (Supply Chain Management).
- DMS (Document Management System).
- CMS (Content Management System).
- GIS (Geographical Information System).
- EAI (Enterprise Application Integration).
- BPM (Business Process Management).

Although most of these products are not applicable within the scope of GN3 software development and usage, the componentisation proven by the success of these systems demonstrates that modular purchasing, integration and integration of software systems is feasible.

When dealing with common tasks performed within software systems, software architects should define an approach used to implement the following responsibilities, for which many development frameworks provide standard application layers, containers or components:

- Access to information.
- Transactions.
- Business rules.
- Identity management.
- Interaction with the user.
- Workflow.
- Management policies.
- Manage documents and other content.
- Reporting.
- Process monitoring.
- Spatial data.

For further information, see [SOA].

3.2.2 Software Reuse

In theory, reuse is a pretty straight-forward idea: to make a piece of software that is useful for more than just a single purpose. However, reuse is neither easy nor automatic. It may increase the complexity, cost, effort, and time to build software, at least in the first iteration. Only later does software reuse provide benefits. It can also be awkward and challenging to build solutions that incorporate software programs developed by other teams.

Reusability has therefore not always been a design characteristic organisations have chosen to pursue for their internal solutions. Problems that occur when trying to reuse elements of previous development are:

- Programming language artifacts are at very low level and can only be effectively reused by a programmer who has intimate knowledge of their usage and implementation.
- There is no easy way to discover what reusable assets exist.
- Code at that level is not network enabled, meaning the code cannot simply be called across machine boundaries, nor it is transparently reusable from different programming languages.

To facilitate software reuse, you should follow these recommendations:

- The code should be written and organised in such a way that it can be reused. Therefore, planning, designing and coding for reuse are required.
- Existing components created in other related projects that provide some common functionality should be used whenever appropriate, not only once. Since this may require refactoring of shared components and of the systems that already use them, it is important to design and implement them with reusability in mind from the very beginning.
- Control reuse of sensitive or external components. If possible, use adapters to extract and separate useful or harmful parts or properties of a system.
- Assess software components before acquisition.
- The discovery of reusable code is another important issue, so its appropriate promotion and description in software repositories is an important and often neglected aspect of software reusability.

A service has reuse potential if it:

- Provides capabilities that are not specific to any one business process.
- Is useful for the automation of more than one business process.

Service reusability is a core principle that represents fundamental design characteristics key to achieving many strategic goals associated with SOA. Service-orientation provides principles that prepare a service for reuse from the very beginning. To accomplish service reusability, existing services must be discoverable by architects of new projects. Best practice is to provide a UDDI registry with a service repository to enable people to quickly find existing services and related information, such as SLAs, service owners, availability, and so on.

3.2.3 Model-Driven Architecture (MDA)

Model-Driven Architecture (MDA) has been proposed for developing of complex distributed systems. MDA separates the modelling task of the implementation details, without losing the integration of the model and the development in a target platform. The core paradigm of MDA is model transformation. With MDA, system construction consists of a sequence of models and transformations between these models. The model-driven approach starts with a platform-independent model (PIM), which is subsequently transformed into a platform-specific model (PSM). The PSM is then transformed to code.

Within GN3 context, using MDA approaches and tools should be considered in particular for the creation of rapid prototypes, data-centric applications with large and non-specific user interfaces, and skeleton code. However, if applications or their parts need to implement an optimised user interface to cover heavily utilised use cases or realise guided or thoroughly controlled workflows, it is better to stick to more conventional frameworks that allow more freedom in design of user interface and application flow control.

For further information, see [SOA] [MDA].

3.2.4 Integrating Complex Architectures with ESB

One of the key components of SOA architecture is the Enterprise Service Bus (ESB). The operation of an ESB is comparable to those of a physical bus that carries bits between devices in a computer. This means that in an architecture that uses an ESB, all communications are handled via the ESB, which acts as a broker between applications.

An abstract view of the ESB is as follows:

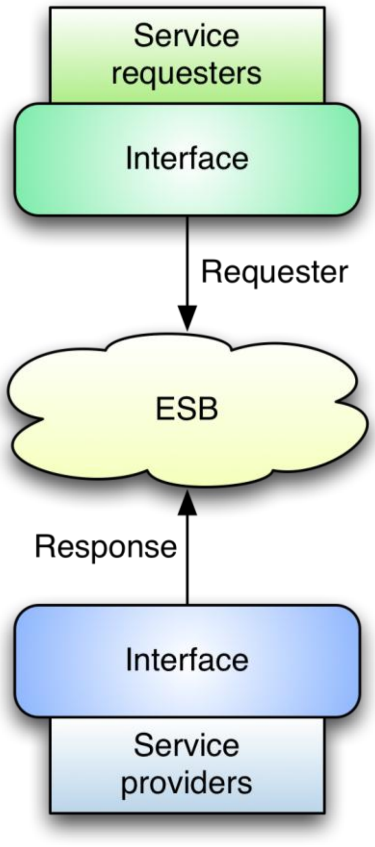

Figure 3.1: ESB representation

An ESB offers a set of infrastructure capabilities for handling the communications between service requesters and service providers. In particular, the ESB is able to communicate with the requester using his/her preferred interface (for instance webservice, FTP and others), whilst allowing the provider to use a different interface. This means that the set of elementary functions have been isolated and can be provided by different entities: the services. All those services communicate with each other using well-defined protocols based on XML message exchange using standardised message exchange patterns.

One of the ESB responsibilities is to translate the information received by a service requester to the format the service provider expects. The ESB is also responsible for routing the request to reach the right service and for assigning the necessary priorities among requests. This process of handling requests and prioritising them is also called 'orchestration'.

In more sophisticated architectures, the ESB also performs monitoring and statistical functions, for instance, providing information on how often a particular service is requested, by whom and in how many of the requests issues occurred.

The main benefits of an ESB can be summarised as follows:

- An ESB allows for loosely coupled services; the benefit of this is that new services/functionalities can be more easily added to an existing infrastructure.
- Simplification and standardisation of the interfaces used by the service requesters and the service providers.
- Services can also be requested by other systems, because of the adoption of the ESB.

The bus paradigm provides the additional advantage of freeing service developers from having to deal with common aspects such as security, service discovery and message management, allowing them to concentrate on the direct implementation of business processes. Most (if not all) current frameworks are oriented towards single enterprise deployment that, though it can be extremely complex, relies on a central top authority.

Within the GN3 context, the GEMBus initiative intends to introduce the advantages of ESBs into an open collaborative environment, taking a step further into federated infrastructures and supporting the definition of a multi-domain ESB, a “bus of buses”. The actual realisation of GEMBus as a SOA multi-domain middleware, able to support the deployment and composition of services spanning different management domains, calls for applying the federation mechanisms that play a key role in the collaborative environment of current academic networking. Federation preserves management independence for each party as long as they obey the (minimum) set of common policies and technological mechanisms that define their interoperation. Metadata constitutes the backbone of such federations, as it describes the components provided by each party and their characteristics in what relates to the federated interfaces.

3.3 Practical Concerns in Software Architecture

Before a fully functional product is developed that becomes an operational solution, the following four additional development phases take place:

- Proof of Concept.
- Prototype.
- Pilot.
- Production.

The **Proof of Concept** phase is supposed to answer the general question about the feasibility of solution to be delivered. Other issues (like scope of the final product) can also be addressed here. If this phase results in a positive decision, a **Prototype** stage can be launched. In this phase general issues about the architecture, communication interfaces and user interfaces should be addressed. On the basis of the prototype, necessary decisions about architecture, functionality and layout need to be made. These decisions are reflected in the development that precedes a **Pilot** phase. Generally speaking, the pilot phase is supposed to provide production functionality and quality, but on a smaller scale. During this phase feedback from users should be collected. The **Production** phase provides all the required functionality, delivered in the most effective and reliable manner.

In particular, different development approaches and technical solutions are acceptable. It is important to provide the most accurate solution for particular phases (e.g. the most basic and simplest technologies for the Proof of Concept and Prototype phases and the most effective solutions for the Pilot and Production phases). The following sections provide guidelines that can be applied when selecting appropriate tools during particular development stages.

3.3.1 Choosing a Database

Each information system that aims to process complex data must be equipped with an independent data layer. Independence means that that the data layer can be accessed and managed without the necessity of referring to the system itself. A correctly designed, developed and managed data layer is necessary for the security and efficiency of the whole system. Generally speaking, the data layer is a database. There are several types of databases: relational, hierarchical, object, stream and flat file [TopInf].

3.3.1.1 *SQL Databases*

Relational database

Relational database management systems (RDBMS) became an industrial standard in 1970s and 1980s. The idea of relational database assumes that data stored in a database is organised in tables and relations between them. A relational database is located on a separate server (database server) so if an application crashes, it does not (or should not) affect the data itself. RDBMS is based on ACID principles (Atomicity, Consistency, Isolation, Durability):

- Atomicity – requires that databases modifications (within transaction) follow an “all or nothing” rule. Thus each transaction is said to be ‘atomic’.

- Consistency – requires that each transaction takes the database from one consistent state to the other, which means that only valid data is written to a database.

- Isolation – requires that a transaction cannot access data that is currently modified by another transaction.

- Durability – requires that data provided by each committed (successfully finished) transaction is able to recover against any kind of system failure, both hardware and software.

If these principles are not followed, the relational database is not RDBMS.

Relational databases can be accessed using Structured Query Language (SQL). SQL is a standardised solution that (in general) allows the querying of SQL-based databases independently of the database engine vendor. As a consequence, migration between databases provided by different vendors is possible without any (hardly any in real life) changes to the source code of the application.

In-memory database

Although the aforementioned principles determine the strength of the relational databases, they also entail a number of issues that make the development of applications taking advantages of RDBMSs more complex (e.g. transactions, rollbacks, etc.). Thus, in order to simplify daily development and testing, lightweight engines can be used. Lightweight products (e.g. HSQLDB) allow the creation of in-memory tables and relations that significantly reduce time overhead that results from using disk-based databases. In-memory databases provide atomicity, consistency and isolation. Because of lack of durability these databases are not complete RDBMSs, and thus cannot be applied in an operational environment.

3.3.1.2 *Non-SQL Databases*

Object Databases

Object databases were introduced to meet the requirements of applications developed in an object-oriented way. Although object-oriented programming became a standard approach, object databases are still a niche field. This situation is unlikely to change, due to the popularity of object-relational mapping frameworks (see *Object-Relational Mapping* on page 29). Flexibility provided by these frameworks and advantages of RDBMSs resulted in a lack of popularity of object databases.

Object databases are very fast. As far as relational databases are concerned, a lot of time is spent on join operations. Objects store many-to-many and one-to-many relations and take advantage of pointers, which are linked to other objects, so that relations are established in this way. Although the concept of objects and pointers results in a significantly decreased level of interoperability and a lack of support for standard analytic tools, object-oriented databases can be applied in an operational environment, if needed.

XML Databases

XML database is a persistence layer that permits data to be stored in XML format. An XML database can then be queried with XQuery and XPath. XML databases became popular with increasingly common usage of XML for data transport. In this case using an XML database can be efficient, since numerous conversions from XML to relations can be avoided. However, where efficiency (in terms of read / write time cost) is concerned, XML databases should be avoided. Relational databases provide advanced indexing mechanisms that make read / write operations more efficient.

When deciding to use an XML database, it is important to bear in mind that many popular XML database systems (e.g. eXist) do not follow ACID principles. As a consequence, it cannot be applied in an operational environment.

File Databases

File databases (or “flat file database”) exist in a single file (either textual or binary). Data is stored in the file in rows and columns, but without relationships or links between particular records. Because read / write operations to a database are in fact read / write operations of a file, the possibility of concurrency in database access is strictly limited.

Flat file databases are very common in standalone applications, where concurrent access is impossible and data volume is small. Standalone application using flat file as a database is very fast (of course dependently on data volume) and fully portable. In case of textual files, manual modifications are possible.

File databases can only be applied in production systems if particular records do not refer to each other (no links and relations), the data volume is small and concurrent access to data is impossible.

3.3.1.3 *SQL Versus Non-SQL Databases*

Although both SQL and non-SQL databases have been on the market for many years, only SQL databases have become an industry standard. Even if ACID principles are not required, relational databases provide flexibility and generality that make it possible to access and process data in ways that are beyond initially assumed use cases. Non SQL databases can only be used when the data to be stored is completely unstructured or constantly changes its form. In all other cases, a traditional, SQL-based solution should be applied. Relational databases proved their usability, security and maintainability. In the long term, a properly configured, tuned and managed SQL database is nearly always a better option than a non SQL solution [SQL].

3.3.2 Object-Relational Mapping

Object-relational mapping (ORM) is a technique of converting data between the object-oriented and relational world. Although the majority of applications are developed using an object-oriented paradigm, the most popular databases rely on a relational approach. ORM is a layer between the business logic and database that translates business objects (with their collections and mutual dependencies) to a relational, database world. Although using object-relational mapping is not necessary, it usually reduces the amount of source code that needs to be written. If ORM is not used, developers need to take advantage of stored procedures and appropriate drivers (in case of Java it would be JDBC) and the Data Transfer Object (DTO) pattern. Using DTO (although it is considered to be an anti-pattern) and stored procedures can be more effective (as far as time cost is concerned) because the developer can take advantages of dedicated (database vendor oriented) mechanisms (e.g. specific sequence of join operations). On the other hand, such a solution assumes that the database provider will never change. Thus, maintainability and portability of the whole system is significantly limited. Object-relational mapping is very convenient, since it reduces the amount of code that needs to be written and the number of steps that need to be taken during testing: changes applied to tables and columns are automatically propagated to the database, data is cleaned, transactions and sessions are coherent. Furthermore, the separation of concerns imposed by frameworks, simplifies test design, since the responsibilities and aspects that are implemented differently in production and testing environments are provided by the framework, not the code that is being tested.

Use of object-relational mapping is not required but recommended where operational environments are concerned. Usually the maintainability and portability of a system is more important than increased efficiency related to usage of vendor-specific mechanisms. More detailed information about this topic can be found in the section *Data Access Optimisation* in the *Quality Assurance Best Practice Guide* [QA].

3.3.3 Communication Protocols and Data Serialisation

There are many layers in the software stack that can be used to implement communication. The lower and more basic the level of API used, the more controllable and efficient the result. However, creation and maintenance of protocol documentation, maintainability of related code and interoperability are crucial aspects of services developed within GN3. Currently, there are several approaches in common use:

- Generalised or language/platform-dependant remote procedure call (RPC) solutions, like XML-RPC or Java RMI.
- SOAP-based web services.
- REST web services.
- Newly popularised binary protocols, like Protocol Buffers or Thrift.

XML is a popular background technology used in the majority of these approaches. It is a de facto industry standard for exchanging data between applications and the key element of the most common flavours of web services. There is a plethora of tools and libraries supporting XML. Most developers are familiar with XML, and it can be quite easily debugged and read by humans, with assistance of various validation and visualisation tools. The use of XML and HTTP produces significant communication and processing overhead. The communications overhead imposed by XML can be addressed using compression, while latency can be greatly reduced using modern StAX parsers and limiting the usage of security and other additional layers in the software stack.

SOAP-based web services, also known as WS-* web services or (Big) Web Services, also use XML and are defined by using WSDL. They are standard in enterprise applications and offer readily available solutions to implement many additional features, like addressing, security or transactions. The growth of their popularity in the last decade has greatly reduced the presence of earlier RPC approaches. And while RPC solutions provide only the method invocation paradigm, Web Services also provide the means to transfer message or structured XML content.

REST (REpresentational State Transfer) was invented in response to the complexity and overhead of SOAP-based web services. RESTful services are based on the paradigm of stateful resources that are identified through URIs and accessed by a set of standard operations, like HTTP GET, POST, PUT and DELETE, and represented using standard web metadata and formats [Rest]. The returned content can be encoded using SOAP, XML (often flavoured as POX – Plain Old XML), JSON, or even HTML and other multimedia types. Since REST is architecture, not protocol or standard, there are many interpretations and implementations of REST. It does not even provide a standard way to specify the protocol or automate code or interface generation, but there are some frameworks providing implementation of RESTful services, like JAX-RS. The REST approach should be considered whenever some identifiable resources should be accessed by clients.

It should be borne in mind that properly used XML is quite self-documenting. SOAP web services are even more verbose in that regard, while this is not the case with REST. As concluded in [Rest] "to use RESTful services for tactical, ad hoc integration over the Web and to prefer WS-* Web services in professional enterprise application integration scenarios with a longer lifespan and advanced QoS requirements."

To fill the gap not covered by SOAP services and REST, new protocols and corresponding serialisation frameworks were invented. Google Protocol Buffers, Apache (Facebook) Thrift, Apache (Cisco) Etch, and Hadoop Avro [ProBuf] are based on message paradigm, although some also provide RPC. They tend to use binary formats featuring protocol versioning and even provide additional size optimisation via variable-width integer encoding.

The idea behind these new approaches is that the developer should not have to worry about the actual protocol and formatting. In comparison with XML, their usage is developer-friendly and more directly bound to the programming languages. All of them use similar message or interface description languages (IDL) to describe messages in a protocol and language-neutral way, but also to preserve mapping to the basic data types and object and field constructs used in programming languages. Even in that regard, there are some limitations. For example, Protocol Buffers do not use plain data objects, as they rely on builders and programmatic methods for binding to protocol implementation [ProBufJava], which also results in lack of support for inheritance between data objects [ProBufRMS].

The developers are increasingly interested in adopting these new messaging and serialisation solutions in new developments, or even migrate existing developments that are based on XML or SOAP web services.

- New protocols are simpler due to lack of history and as they intentionally omit some features – so their implementations have a smaller memory footprint.
- They are efficient in terms of performance [ProBufPerf] and message size [ProBufMes].
- They vary in the number of supported programming languages; but most all of them support Java, C++, and Python [ProBufComp].

All these protocols can transfer contents over HTTP and other network protocols and can be even used to deliver content for RESTful services. However, they intentionally avoid relying on HTTP by default to allow performance improvement and use of more efficient communication channels. The message size is typically reduced to 10–50% of equivalent XML, but, for small messages, this does not result in a proportional change in performance or message throughput [ProBufPerf], despite the fact that as much as 100-times improvements are claimed [ProBufDev]. From available comparisons, it is only worth to migrate from XML if the XML message size is above ~30-40KB and if communication is clearly identified as a major bottleneck.

For new developments, these frameworks should be considered when the medium for internal communication between two highly integrated components needs to be built, and where there is no perspective for interaction with external entities over the same data flow.

All these alternative solutions pose many risks. All of them are at an early development stage, so their usage in scenarios they were not developed for may have limitations or cause serious issues. Only one or two of them will gain wider community acceptance, while the majority will be abandoned in a couple of years. Therefore, their usage is only justified at this point if latency and throughput are the main concerns. However, in these scenarios it should also be borne in mind that even language-bound solutions like native serialisation and RMI can be quite efficient. When it ultimately comes to protocol and message debugging, it is certainly far easier with XML.

Finally, whatever solution is used to implement serialisation and communication, it should be implemented so that it is used only within the edge classes dedicated to file storage and interfacing with external systems. The same recommendation applies to the configuration related code. This way, one solution could be easily replaced with an alternative without disturbing the rest of the system.

3.3.4 Data Binding and XML Data Binding

Data binding associates two data representations and performs conversion between them or maintenance in sync. Simple data objects are increasingly used by various frameworks and libraries to represent domain-specific information held in a non-object container. For example, object-relational mapping frameworks provide mappings between relational databases and objects. Binding of JavaBeans [JBAPI] with the UI controls is frequently the feature of GUI and web frameworks. These trends also emphasise the need to transfer and convert information between different domains and representations in a way which decouples data models and isolates them from mapping rules.

There are even object-to-object mapping tools offering various levels of flexibility and performance [JBPerf]. Examples of such tools are Apache Commons BeanUtils [BeanUtils] (specifically copyProperties() methods of BeanUtils, BeanUtilsBean and PropertyUtils classes), Dozer [Dozer] or Transmorph [Transmorph], which is more focused on mapping between diverse data types, like beans, arrays and maps. Most of them use Java reflection or JavaBeans introspection APIs. Since they cause significant cost in performance if used during each individual copying, reflection and introspection should be avoided in performance-critical code, where sufficient decoupling can be achieved by hand-written or automatically generated object-to-object conversion code.

For XML, there are many ways to transform content into objects and vice versa. The serialisation of Java objects into XML is called marshalling, while building memory objects or object graphs from an XML representation is known as unmarshalling. One of simplest approaches to this conversion is to use readily available java.beans.XMLEncoder and java.beans.XMLDecoder classes. However, this becomes much more complex if precisely specified XML format has to be dealt with, if performance becomes an issue or if the object model and XML start to evolve in their own directions.

These problems inspired development of many tools for binding between XML and data objects. Which one is most suitable depends on current and expected future needs. Some tools process XML schema in order to create data objects, other start from objects. Therefore, the first to determine is what to start from and whether both reading and writing of XML is needed. Heaving both within the same API may add unnecessary complexity to the API.

The majority of tools can use Java reflection to match objects and their attributes with XML. However, using reflection on the fly is expensive, especially if it is used during each marshalling or unmarshalling. Therefore, it is better to use an explicit mapping mechanism for specifying mapping (binding) rules, if available. Even if the implicit conventions are satisfactory, mapping rules can greatly improve performance. These rules can be specified in dedicated XML, programmatically (through configuration or population of conversion classes) or even in Java annotations embedded in data classes. The problem with annotations is that they pollute data objects and allow defining of binding for only one XML format. All this is of concern for keeping data classes independent from associated XML schemas.

The general recommendation is to choose a binding tool which supports reflection, but can use other means to define mappings and conversions. Some tools also generate or require object which are derived from a specific class or are not real JavaBeans. Some can be attached to standard XML parsers, and some have parsers of their own. Some can stream objects while reading XML, while others always create the whole hierarchy. Finally, it needs to be considered whether XML comments, entity references and order of siblings must be preserved by binding tools. The comparison of some XML binding tools is listed in Table 3.1.

	JAXB	Castor	JiBX	XMLBeans	Digester	Betwixt	Javolution	XStream
Start from XML schema	X	X	X	X				
Start from Beans	X	X	X		X	X	X	X
Read XML (unmarshalling)	X	X	X	X	X		X	X
Write XML- (marshalling)	X	X	X	X		X	X	X
Reflection based mapping	X	X			X	X	X	X
File (XML) based mapping	X	X	X		X	X		
Mapping management API	X	X			X	X	X	X
Java annotations mapping	X				X			X
Content filtering				X	X			

Table 3.1: Features of XML data binding tools

Further information about these tools:

- JAXB (Java Architecture for XML Binding) [JAXB] is a standard Java EE API. It uses XML schema to create data objects similar to JavaBeans, but also provides options to specify the names of generated classes and properties or supply existing implementation classes to be used by the binding. It includes a tool to generate a schema from a set of annotated classes. It allows the unmapped XML content to be preserved. JAXB implementations, like Metro, MOXy, and JaxMe, provide additional extensions.

- Castor [Castor] supports both generative and dynamic versions. It can start from Java objects or from XML schema and allows mapping rules to be defined. Castor is a more general data-binding framework, which besides moving data between XML and Java programming objects, also provides Object-Relational Mapping. It is designed to allow easy migration from JAXB.

- JiBX [JiBX] is a high-performance binding tool, which can start from both Java and XML schema. It works by inserting binding definitions into original bytecode, and, like JAXB, it can create schema from beans.

- XMLBeans [XMLBeans] creates hierarchies of strongly typed objects, based on the schema of original XML. The produced classes are not editable. Like JAXB, it provides complete access to the underlying XML. Additionally, it allows selection (filtering) of objects to be extracted from XML.

- Digester [Digester] does not create classes from the schema, but uses classes provided by the developer. The developer can limit what is to be extracted from XML. This only works in one direction, creating object hierarchies from XML.

- Betwixt [Betwixt] is a simple tool that complements Digester by providing marshalling. Its XML reading capability is provided by Digester.

- Javolution [Javolution] as a framework for real-time applications that also includes compact binding tool that starts from Java classes.

- XStream [XStream] starts from Java.

JAXB is richest in features and by default provided with Java. However, for specific usage it may be more complex to use than other tools. For example, if only domain specific objects need to be populated from XML, Digester is the simplest solution. If your domain model is well represented by used XML and your XML schema is stable, you can opt for a tool like XMLBeans, which does not allow elaborated mapping. And if your XML and object model diverge, or mapping capabilities of your XML binding tool become insufficient, you can resort to simple object-to-object binding described at the beginning of this section.

3.4 XML Design

XML is a set of rules for encoding structured data into electronic textual documents. It is designed in a way that allows simple generation and usage by computers, computing-capable devices, and over the Internet, and is still interpretable by humans. There are many generic tools and frameworks that facilitate usage of XML in applications and ease information encoding, communication, and retrieval.

Although XML can be applied in diverse scenarios by using a variety of tools programming interfaces, there are a few general recommendations that should be observed by software developers, particularly while defining XML-based languages:

- Defining an XML markup is a substantial naming and structuring process, and therefore must be conducted very carefully (please refer to the general naming guidelines described in the GN3 *Software Developer Guide* [SDG]).

- All types of XML documents used should be described using an appropriate XML schema. The preferred XML specification language format is XSD (XML Schema Definition) 1.0, or later. It is good practice to include schema version and revision tags into schema filenames (for example myschema_1_2.xsd).

- Data models and document hierarchies evolve, and it is possible that there will be more then only one use of the schema. Often an XML-based format becomes more permanent than the software for which it was initially created, and several systems end up mapping information from some XML markup to their internal structures and data models. All these concerns should be kept in mind while designing an XML schema, meaning that design of an XML schema should never be taken lightly, since they may be used for a long time.

- Preferably use elements to structure information, not attributes. Although it may be tempting to use attributes because they become readily available in SAX processing, this results in inextensible code that is hard to modify. Elements-based markup is much more extensible in terms of additional structuring the data and change of cardinality. Documents built upon such markup are also more readable for humans. Also, stripping out the XML markup from a file will remove structure and attributes, but the basic information will still be there; this is not the case if attributes are used.

- It is better to publish an XML schema before publishing actual data to external users, so that other participants in exchange could prepare and implement the part they are responsible for.

- The XML-based markup should never be optimised for size. Never try to use abbreviated names, reduce whitespaces, use attributes for content, or combine multiple values in a single attribute or element. All reasonably verbose XML should be preserved. Where optimisation is needed, XML documents can be compressed (e.g. to files or streams in gzip format), or alternatives, like binary formats and ASN.1, can be used.

- When it comes to parsing XML, stream-oriented APIs like SAX and StAX are preferred in all usages where the XML content can be handled as it comes, or the size of document may be of concern in terms of memory usage.

- XSLT should be used only for relatively simple XML, but never for usages like creation of interactive HTML-based user interface, or transformation or formatting of textual content within elements or attribute values. With such usages, the documents containing XSLT become difficult to read and maintain.

For further information on XML, see [XML].

3.5 Object-Oriented Design

The object-orientated design (OOD) approach provides a means of organising solution logic into classes that essentially act as containers for definitions of related behaviours and properties. Object-oriented approaches support and formalise the notion of decomposition and recomposition. Programming logic is separated into classes which provide methods that can subsequently be combined into aggregates.

Much of popular service orientated architecture (SOA) owes its existence to the concepts, principles, and patterns that originated from object-orientated design. SOA inherits all of the primary OOD goals but increases their scope and adds others.

Object-orientation design comprises a rich set of fundamental and supplemental design principles that structure and organise object logic within and across classes. Those principles [PRINCIPLES] are:

- Encapsulation.
- Inheritance.
- Generalisation and Specialisation.
- Abstraction.
- Polymorphism.
- Open-Closed Principle (OCP).
- Don’t Repeat Yourself (DRY).
- Single Responsibility Principle (SRP).
- Delegation.
- Association.
- Composition.
- Aggregation.

The two key elements of object design are localisation of responsibilities and precise establishment of interfaces. The following recommendations related to OOD should be considered during all phases of software development:

- Insist on responsibility allocation and repetition avoidance.

- Think more about interfaces than about implementation.

- Programming towards interfaces allows wider applicability of the code and plugging of various implementations of the same functionality. For example:

```
Animal animal = getAnimal(Dog.class);

animal.makeNoise();
```

is better than:

```
Dog animal = new Dog();

animal.bark();
```

This is also a means to separate responsibilities – different interfaces (and classes that implement them) should be responsible for the mutually independent functionality.

- Inheritance and protected fields and methods are means of implementation. They are not an essential part of design.

- Classes bound by inheritance should be related in both nature and implementation, not only because they share a few common attributes or because the subclass needs functionality implemented in the parent class. Instead, inheritance of interfaces, composition (has-a relationship) and delegation (that is structural and behavioural patterns) should be used to obtain the required business logic or data structures and to break long class inheritance trees. Excessive use of inheritance blurs responsibilities and makes the code hard to read and understand. It also buries some aspects in parent classes and mandates their presence in subclasses. This advance mandating is sometimes desired, but it makes it more difficult to accommodate unforeseen changes in behaviour, implementation or informational content.

- At the highest application level, individual responsibilities that should not be directly connected with the implementation of the primary functionalities of the system are logging, security, and caching.

- Java package names should be meaningful and comply with the general decomposition of the project. They should be selected to minimise the conflict in the package namespace and give a clear idea of what is in them and where to put something.

- Packages should separate functional responsibilities or components and group classes with a strong relationship. Package hierarchies should not be too deep. It is enough to have one or two levels within the project.

- Some packages may be used in multiple projects, and they should be named according to that. Generic names, such as `*.lib` and `*.util` are suitable for packages for general purposes, which may contain several mutually unrelated classes. Classes from these packages can be used from a lot of places within the system, and should be carefully designed and implemented.

- Implementations of methods should be short. They should represent short and understandable functional units that implement some functionality of an object, using some functionalities or lower-level objects.

- A reader should be able to understand what method works on the basis of its declaration, and how it works after a few minutes.

- Consider using a `switch/case` structure or classes with the methods implemented in the case branches instead of `if/else if`. The construction of switch/case imposes the use of constant values of simple types and `enum` or `static final` constants instead of testing strings or objects.

- However, `if/else` if and `switch/case` structures can often be avoided by using object decomposition and alternative method implantations. This reduces the possibility of error when new branches are added and replaces the checking of conditions with the execution of dynamically selected methods. The application of `switch/case`, structures and localises conditional logic very well, while the advantage of object decomposition is method size reduction.

- Paired activities, such as `Stream.open()` and `close()` or opening and closing connections should always work in the same method or at least class, so the life cycle of objects can be localised and is clearly visible. Closing should not be done in the output branches of code that uses the open resource.

- With the input parameters, a minimum required interface should be set that meets the expectation of the argument. This makes the method easier to use with concrete objects that are available as potential arguments. For example, `Map` is better than `HashMap`; `Collection` is better than `Set`, `SortedSet` or `TreeSet`. The number of input parameters should be minimised.

- Regarding the returned value, it is better to give the most concrete interface that meets expectations on the result, but to leave precise specification of returned types to the implementation. This reduces dependency on implementation while allowing usage of all features provided by the returned interface. In a specification of a method, declaring a returned value as `Set` is probably better than `Collection`, and is certainly far better than `AbstractSet`, `HashSet`, `LinkedHashSet` or `TreeSet`.

- If the method should be forwarded to a large number of parameters, it is better to pack these parameters in the appropriate objects. Otherwise, the method will be difficult to use, as the user will not know what each parameter means. This is especially difficult if several consecutive parameters are of the same type. The names of parameters should describe what they represent, not the type they presented.

- The use of `instanceof` or `getClass()` comparisons should be avoided, unless they simplify the implementation based on classes that cannot be changed, and there is no need for the introduction of Delegation or the Visitor pattern.

- As a rule, use EJB annotations instead of XML to describe EJB classes. This improves readability and encourages locality. This also applies to named queries, etc., unless the tool used imposes another approach.

- For some well-known mistakes (anti-patterns) related to object-oriented software design that should be avoided, see [ANTIPATTERN].

3.6 Using Design Patterns

A design pattern identifies a common problem and provides a recommended reusable solution. Design patterns provide additional intelligence that can enrich a design framework with a collection of proven solutions to common problems. It essentially documents the solution in a generic template format, so that it can be repeatedly applied. Knowledge of design patterns not only arms one with an understanding of the potential problems designs may be subjected to, it provides answers to how these problems are best dealt with.

Design patterns are born out of experience. Pioneers in any field had to undergo cycles of trial and error and by learning from what did not work, approaches that finally did achieve their goals were developed. When a problem and its corresponding solution were identified as sufficiently common, the basis of a design pattern was formed.

Design principle and design best practice both propose a means of accomplishing something based on past experience or industry-wide acceptance.

When it comes to building solutions, a design principle represents a highly recommended guideline for shaping solution logic in a certain way and with certain goals in mind. Design standards are (usually mandatory) design conventions customised to consistently pre-determine solution design characteristics in support of organisational goals and optimised for specific enterprise environments.

As with design principles, the application of design standards results in the creation of specific design characteristics. As with design patterns, design standards foster and refine these characteristics to avoid potential problems and to strengthen the overall solution design.

There are several reasons for using design patterns:

- **They have been proven**. Patterns reflect the experience, knowledge and insights of developers who have successfully used these patterns in their own work.
- **They are reusable**. Patterns provide a ready-made solution that can be adapted to different problems as necessary.
- **They are expressive**. Patterns provide a common vocabulary of solutions that can express large solutions succinctly.

Design patterns should be known and used. It is good to name by used patterns (classes by specifying patterns at the end of the name, methods by pattern primitives). However, design patterns should be used within reason to avoid over-design.

According to different sources, there are 20 to 50 design patterns in Java. These patterns are grouped into:

- Creational

  How an object can be created. This often involves isolating the details of object creation, so that your code does not depend on what types of objects there are and therefore does not have to be changed when a new type of object is added. Some creational patterns are Factory Method, Abstract Factory Method, Builder Pattern, Prototype Pattern, and Singleton Pattern.

- Structural

  Designing objects to satisfy particular project constraints. Objects are inter-connected in a way that ensures that changes in the system do not require changes to those connections. Some structural patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy.

- Behavioural

  Objects that handle particular types of actions within a program. This includes processes that you want to perform, such as interpreting a language, fulfilling a request, moving through a sequence (as in an iterator), or implementing an algorithm. Some behavioural patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template, and Visitor.

There are a few commonly used formats (or ’forms’) used to describe patterns. The two most popular formats are called GoF Form and Canonical Form. These two pattern formats share many common characteristics. The main difference is that the canonical format places a more explicit emphasis on forces and the resolution; whereas the GoF format breaks up the solution into more sections and uses more formal design diagrams along the sample code.

The developer should consider whether to introduce public constructors for the class and how to manage them, because it is often desirable to clearly separate the responsibilities for creation of objects, management of their life cycle, re-use (pooling), and selection of specific types or parameters of instantiated objects. In that regard, factory methods and classes can significantly improve the design.

Respect the MVC pattern and minimise dependency propagation of the View layer in the Model-Control – for example by transferring HTTP* objects in Stateless Session Bean methods that represent a business facade. You should observe that one MVC set in web-based Java applications is:

- M: EJB.
- V: JSP or Facelet and JSF components.
- C: Control Servlet (C) or JSF backing bean.

Another MVC exists within the EJB layer:

- M: Entity beans.
- V: Public interfaces of entity beans and the possible Seam annotations [SEAM].
- C: Business methods of stateless session bean used to control access and transactions.

Note that some frameworks (like Seam) blur this separation between MVC elements to make the development more efficient. However, as long as the approaches recommended by the used application development framework are followed, this is not a problem, since the very design of modern software frameworks embeds and enforces many well thought-out and proven patterns.

The general design of a distributed system should base the interaction through the interfaces of business methods on service principles, especially if the service provider may be a remote entity. This means that the service facade should be implemented with high level granularity methods in which the client of the service sees simple objects that are not depending directly on the remote system. In other words, a business action through the interface to (potentially) remote objects should not be implemented through a large number of individual actions on individual objects. In the case of EJB, service facade is presented through a stateless session bean. Business methods implemented in this way can be either used directly in the implementation of the web View layer or exposed as web services.

3.7 User Interface Design

A quality user interface is the result of application of eight key principles [SHNEIDERMAN].

1. Strive for consistency.

   Consistent sequences of actions should be required in similar situations.

   ○ Identical terminology should be used in prompts, menus, and help screens.

   ○ Consistent commands should be employed throughout.

2. Allow frequent users to use shortcuts.

   As the frequency of use increases, so does the user’s desire to reduce the number of interactions and to increase the pace of interaction. Abbreviations, function keys, hidden commands, and macro facilities are very helpful to an expert user.

3. Offer informative feedback.

   For every operator action, there should be some system feedback. For frequent and minor actions, the response can be modest, while for infrequent and major actions, the response should be more substantial.

4. Design dialogs to yield closure.

   Action sequences should be organised into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans and options from their minds, and an indication that the way is clear to prepare for the next group of actions.

5. Offer simple error handling.

   As much as possible, design the system so the user cannot make a serious error.

   If an error is made, the system should be able to detect the error and offer simple, comprehensible mechanisms for handling the error.

6. Permit easy reversal of actions.

   This reassures the user that errors can be undone, and thus encourages them to explore of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.

7. Support internal locus of control.

   Experienced operators like to feel in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.

8. Reduce short-term memory load.

   The limitation of human information processing in short-term memory requires that displays be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.

The following details further elaborate on some of these rules.

Consistency with user expectations, and coherent interpretation of user behaviour should have the highest priority. Users expect to see consistent layout and visual and behavioural patterns (for example, composite GUI controls repeated throughout the application). Shortcuts and actions should maintain their meanings across the system. Visual elements, their layout and groupings should be used consistently. However, this should not lead to total uniformity and difficulty in distinguishing different visual objects. Make objects that act differently look different. Also, while pursuing consistency and trying to simplify their own work, developers often forget or ignore another key requirement: to look at the user's productivity rather than at the developer’s or the computer’s.

Form submission buttons, controls and buttons for creation of new or deletion of existing object should be placed in similar locations on analogous forms. The layout of the elements should go from the left to right (for languages with left-to-right writing) and from the top towards bottom according to decreasing importance, or chronologically, if the interaction flow is sequential (as in wizards). This applies to navigation menus, input fields, buttons, and overall page/window layout. If the displayed items are laid out in columns, each column should be sortable. If the input items of a form are laid out in columns, the entry (tab) order should be per column, with inputs and corresponding labels grouped together.

If several objects of the same type are displayed within one table or selection list, you should provide the most adequate (alphabetic, temporal, numerical or hierarchical) sorting. It is even better to allow the user to change sorting criteria and order.

Try to minimise the need for scrolling for minimal supported windows or display size. If you want to display a larger amount of data that may require scrolling, but also want to preserve the contextual part of the page displayed, use paging or limit the size of the scrollable area in which the data is shown. This can be achieved in HTML by using the `<iframe>` element or some component for scrollable tabular display.

Labels before a column of inputs fields should be aligned, as should the input fields. The colon character should be consistently used at the end of the labels, or consistently eliminated altogether (current industry standard for web applications is to use the colon character). Mandatory fields should be marked with the asterisk after the colon and one blank space (e.g. "Name: *"), or using other easily recognisable notion. Labels describing individual checkboxes and radio buttons should be placed after the corresponding input, not in a separate cell of layout table. In HTML, they should be associated by using the `<label [for="..."]>` element, but also using a labelled group or corresponding text in the column dedicated to the labels to describe the whole group of controls.

Another common recommendation is to right-align a group of action buttons that affect a whole form below the form’s content. However, if actions are related to a single-line input field, the button should be placed immediately after the field, at the same height. On the other hand, if buttons or links are related to an entire set of vertically spaced logical forms (groups) that are visually separated (into separate visual frames or table), the action buttons should be placed below all forms, with vertical spacing no smaller than spacing between forms. In this case, buttons should be left-aligned if the related forms are not right-aligned. If a single field is enhanced with an additional button, this grouping should be consistently used.

Examples of such composite visual components include access to search windows, access to calendars, clearing contents and resetting time to "now". Since such buttons need to be compact and visually compelling, it is useful to render them using an icon instead of text, which should remain present as alternative representation.

Standard and frequently used actions and statuses should be consistently placed and have recognisable visual clues (icons) along the button or link text. Typical actions that may be visually improved are:

- Save.
- Cancel.
- New/Create.
- Update/Edit.
- Delete.
- Add.
- Remove.

If a button or link initiates one these operations, it is better to provide an additional description in a tooltip, than a verbose description around or within the button or link. Use tooltips whenever an additional explanation is needed that should not be too intrusive. However, if an action is critical, irreversible or destructive, it is better to introduce an additional page or dialog requiring additional confirmation, than to rely on a tooltip explanation.

Applications should attempt to anticipate what the users want and need. Do not expect users to search for information or evoke necessary tools. All information and tools needed for each step in a process should be pushed to the user. Although the interaction should be anticipative and guided, some flexibility must exist for users to feel in charge. A way out, undo, and shortcuts to key parts of the system should therefore always be provided. This feeling of control is also emphasised by providing concise, up-to-date, accessible, and non-intrusive information.

Regarding error handling, design the form so that users cannot make a serious error. For example, prefer menu selection to form fill-in and do not allow alphabetic characters in numeric entry fields. If a user makes an error, instructions should be available to explain the error and offer simple, constructive, and specific instructions for recovery. Segment long forms and send sections separately, so that the user is not penalised by having to fill in the form again – but make sure to inform the user that multiple sections are coming up.

In a user interface, the titles of pages, windows, tabs and other grouping elements, as well as buttons, toolbar and menu items should use Title case, unless the application already consistently uses Sentence case for some of these items. Standalone links, labels before inputs, values in selection lists, column titles and descriptions should use Sentence case. If text is localised to another language, apply the capitalisation conventions of that language, as many languages do not use title capitalisation.

Terminology used within the addressed domain, names used to describe possible user actions, and the tone of referring to the user, should be consistent. All actions with equivalent outcome should be named consistently. For example, Submit, OK, Update, Commit, Accept, Cancel, Revert, and Reject, as well as Search and Find all have different meaning and should not be used interchangeably. Also, New and Delete should be used together, as should Add and Remove. The tone of communication with the user should be neutral and respectful. Using simple jet respectful imperatives in instructions to the user is fine, as it makes the statements shorter and reflects users' control over the system. Styles that are too personal or evasive (e.g. third person, passive, or infinitive) produce clutter and should be avoided.

Accessibility matters. Approximately 10% of male and 0.4% of female users are colour blind, and older users may want to be able to switch to a larger text size. Making applications more accessible improves their structure and the experience provided to all. For some simple recommendations that may greatly improve the usability of web-based applications for people with sensory or motor impairments, see [ACCESSIBILITY].

The related HTML elements include `<label>`, `<fieldset>` and `<legend>` tags, and `accesskey`, `alt` and `tabindex` attributes. For details on these tags and advice, see [WEBDESIGN].

For the programmatic implementation of web-based user interfaces, some clear technical recommendations are:

- Use separate CSS files to adjust the appearance of the application and to separate the look from form functionality for pages rendered using HTML. While doing this, use a limited set of styles and avoid modifying the CSS styles within the pages. For an illustration of what can be accomplished with CSS, see [ZEN].

- Do not rely on web browser's Back button. The application must support back navigation. However, if the used implementation supports this, the Back button should also work.

- Separate the programmatic logic as much as possible from the presentation layer. For example, minimise Java code by migrating it into the control servlet, use the JSTL library, and create custom JSP tags or encapsulate logic in methods that can be concisely invoked.

- If the title of a page is dynamically created, you should construct it beforehand and then insert it into `<title>` and `<h1>` or its visual equivalent.

- If you are using a JSF (JavaServer Faces) based presentation framework with Facelets, you can localise the presentation and improve the visual consistency by applying page templates.

- Never use a single-thread model for servlets. Information associated with an ongoing interaction or session should never be placed within a servlet's instance variable. An object stored within a servlet's static variable, context (`ServletContext`) and session (`HttpSession`) scope must be thread-safe.

- Always use relative URL names within a single web application.

- Perform server-side redirection (`RequestDispatcher`) whenever possible. Client-side redirection should only be used if the client should become aware of the new URL. The servlet code snippet is `response.sendRedirect(res.encodeRedirectURL(…))`.

- If HTML or another markup (XHTML, XML, WML) is dynamically created, you must check all content and attribute values that are created dynamically (for example, from a database or previous inputs) must be checked for presence of Less Than symbols (<), Greater Than symbols (>), ampersands (&), and quotation marks ("), and ensure they are properly quoted. Dynamically created JavaScript also requires apostrophes (') to be quoted. This problem does not exist in frameworks that prohibit direct markup generation and instead provide this functionality through value assignment to placeholders.

- It is more user-friendly to perform most of the input validation at the client side by using JavaScript, AJAX or JSF. However, implementation of server-side validation is always required.

For further details, see [USABILITY].

4 IPR and Licensing

4.1 GÉANT Intellectual Property Rights Policy

The GN3 Intellectual Property Rights (IPR) Policy has been agreed by the GÉANT Consortium, and is now available to all project participants [IPRPOL]. Additionally, an IPR Coordinator has been appointed to the project. The IPR Coordinator will act as the official point of contact for all IPR issues in GN3.

IPR coordination has built on the IPR audit and licence analysis work to develop a full IPR policy, dealing mainly with copyright and software licensing issues, but also addressing the other aspects of Intellectual Property (IP) policy. The policy itself, which was agreed by the NREN Policy Committee, was based on a similar policy developed in the USA by Internet2. It consists of a broad set of guidelines, supplemented by standard licences and a set of frequently asked questions. In addition to the policy document, a simple, introductory “Guide to IPR” has been produced.

The main features of the IP policy are:

1. Permissive Open Licensing, based on FreeBSD licence.

2. Red–Green lists for importation of IP, whereby participants are, subject to registration obligations, free to import green list IP but need to seek IP advice from the IPR co-ordinator if an item is on the red list.

3. Model licence agreements.

All IPR-related documents are available from the IPR coordination area of the GÉANT Intranet [GÉANT IPP].

4.2 IPR and Software Development

Software developers are encouraged to familiarise themselves with the GÉANT IPR Policy. All information related to IPR and licensing is available from the IPR coordination area of the GÉANT Intranet [GÉANT IPP]. If you have any further questions, please contact the IPR Coordinator.

4.2.1 Inward Licensing of Software

Imported software can form an important component of any software development. But there are important considerations before third-party software can be imported into a GÉANT system. Activity Leaders (AL) are free to authorise the importation of software to be incorporated in a development, subject to the following considerations:

- Consult the IPR coordinator before using pre-existing software owned or controlled by them or their organisation.
- Ensure that the activity has the necessary right to use any software/library being imported.
- If the licence type for the software appears on the Green list of permissible licences, and the licence conforms to the licence type (for example it is a genuine, 4-clause FreeBSD licence), the AL can authorise the importing of the software.
- For all other (non-Green) licences, AL's need to discuss the proposed import with the IPR co-ordinator.
- Use standard GÉANT inward licence (see Appendix C of the *GÉANT Intellectual Property Policy Document* [GÉANT IPP] if no licence is available for the imported software/library, in consultation with the IPR coordinator.
- Inform IPR coordinator of any and all software being imported, and send a copy of the appropriate licence.

4.2.2 Outward Licensing of Software

The Project has defined a preferred licence for outward licensing of GÉANT Software, which must be used in all GÉANT software deliverables. If you have a reason to believe that the GÉANT software outward licence is not appropriate for your needs, contact the IPR co-ordinator:

- Include the standard disclaimer in all software deliverables.
- Include the standard copyright statement in all software deliverables.
- Include standard GÉANT outward licence in all software deliverables (see Appendix A [GÉANT IPP]).
- Satisfy obligations imposed by imported IP (e.g. include Apache licence).
- Report any and all published software to the IPR coordinator for the IP register.

The licence should be placed at the top of the binary and source distribution archives and source repository, on a separate page on the website that is easily accessible from the home page. A separate NOTICE file should also be included that contains the licences of components.

The license file is important because it allows automated scripts and users to understand licensing consequences of using software. Such consequences are related to business-friendliness, hidden costs, redistribution, re-use, modification, and other aspects that licenses typically cover.

If standard licenses like GPL or ASL are included in a software build, use the latest version, and carefully read the instructions what should be changed in a licence (e.g. company and year range information). If a copyright notice forms part of the license, note that year span should not go into the future. The ending year is the last year of development, not the end of the copyright duration.

The license file should contain only the license of the main software. If third-party software or separate software components are used, their licensing should be added to the separate NOTICE file together with the corresponding information about their authors.

Check licenses' terms and conditions about writing licensing info into the distributed files (particularly the source). Maintaining them could be tedious (although a bit easier with automated scripts), so do not add licences if not necessary. Note, however, that licensing obligations **must** be followed if terms and conditions imposed by inward licences require an addition.

Also, do not forget to mention and reference the license in the software GUI, documentation, distribution, source code, or on the web, as well as all used components and their licenses, as required in corresponding licenses.

Examples may be found on the following websites:

- http://openjpa.apache.org/license.html
- http://netbeans.org/about/legal/product-licences.html
- https://intranet.geant.net/sites/Services/SA2/T5/AMPS/Pages/ExternalLibrariesListandLicencesUsedinAMPS.aspx

4.3 SA4 T3 Tool for Checking Component Licences

The software deployment and release process exists to ensure that the released software is compliant with GN3 IPR Policy. In fact, it also involves the monitoring of the licences of the third-party dependencies and transitive dependencies used by the GN3 software products. In order to handle such a task, support in the form of software tools is required, in-line with the software development life cycle. The Artifactory software tool [Artifactory] was selected due to its many advantages. Besides playing a role of an artefact repository, it allows the integration with the continuous integration server (Jenkins) and project build system (Maven) used Project-wide in GN3. Extended with a set of plugins (Pro Power Pack), Artifactory enables the creation of a repository of licences used throughout the monitored developments, and also allows licence checks of every software build performed by a continuous integration server and deployed into Artifactory.

Currently, GN3 Artifactory instance [ArtifactoryGN3] hosts Maven repositories used in GÉANT project, keeping software artefacts developed or used during the project. It also proxies and is a cache of a number of official Maven repositories, including a central Maven repository. The GN3 Artifactory service, available through the web interface, is situated to be a part of SA4 Task 3 SW development infrastructure.

4.3.1 Managing Licences

One of the add-ons of Artifactory is the repository of licences used throughout the monitored software projects (builds). Artifactory comes with the repository preloaded with a number of open source licences approved by Open Source Initiative (OSI) [OSI]. The repository allows adding new licences, including those defined by GN3 IPR, or modify the existing ones. Each licence entry's metadata includes licence key (short identifying string), long name, URL, notes and regular expression describing the pattern of the licence. It is also possible to flag licences as approved or unapproved from the perspective of GN3 IPR policy, which corresponds to their classification as green or red in GN3 IPR documentation. The licences repository is to be maintained in a centralised and consistent manner, and is therefore, not to be updated by individual software projects or developers. The licences kept in this repository are further used during licence checks for each software build.

4.3.2 Examining Licences in Software Builds

Artifactory can monitor licences used in software builds deployed by continuous integration server as a result of a post-build job. Each software build is viewable in the Artifactory build browser. The build description provides:

- General build information (including URL of Jenkins job results).
- The list of published modules.
- Environment information.
- Release history.
- Licence information.

The licence tab, as a part of the build browser, provides information on all third-party licence dependencies, and transitive dependencies of a given software build. Dependencies (towards other artifacts) are grouped by licence type, including a short summary, which aggregates the numbers of approved/unapproved and not-found licences. The summary of licence checks also has filtering features, narrowing the selection of artifacts to scopes such as test, compile, runtime or provided.

As Artifactory matches build artifacts with licences defined in the licence repository, the licence tab also shows whether the used licence is approved or unapproved. By default, Artifactory carries out the licence discovery by scanning POM files (`<license>` element) of build artifacts in search for licence text and then, by using regular expressions, recognises what licence to assign to a particular artifact.

Through license tab view, the IPR coordination team (admin group) is able to manually change licence assignment of a given artifact. Once the licence is assigned to an artifact, this change is visible across all software builds which use a given dependency. GN3 members have permissions to read-only the licence repository content and license tab view of particular builds to be aware of the actual licenses used by a given project.

Artifactory also offers an email notification summarising the licence violations, to a defined group of users, as described in the following section. This feature allows receipt of the licence check results after each software-build deployment to Artifactory. Requesting the email notification can be done through the GN3 software coordinators of a given project. The notification summarises build licences in the following categories:

- Unapproved –- the found licence is declared as ‘not approved’ in the Artifactory licence repository.
- Unknown – licence information was found, but is not related to any licence managed in Artifactory.
- Not found – no licence information could be found for the artifact.
- Neutral – the licence found is unapproved, however, another approved licence was found for this artifact.
- Approved – the licence is declared as approved in the Artifactory licence repository.

Integration of Jenkins and Artifactory enables the creation of a robust build environment, where software licencing information can be continuously traced and validated. This setup can provide valuable information, both to software developers and IPR coordination, supporting a successful delivery of complete software distribution.

4.3.3 Integration of Jenkins CI with Artifactory – cNIS Case Study

Artifactory is situated as a tool to support the IPR coordination team and software developers. The integration of Jenkins CI and Artifactory, carried out by using cNIS project as a case study, helped to define the following actions/practices related to the integration itself and the usage of Artifactory by software developers:

- Use Jenkins Continuous Integration Service to automatically publish software builds into Artifactory. The steps of integrating Jenkins and Artifactory are described in SA4-T3 intranet pages [SA4IPRSup]
- To request adding a project to Artifactory, the software coordinator of a given project is supposed to make a request through SWD Service Desk, using Help Desk type issue.

- Artifactory can send email notifications providing a summary of licences used by your project after each build. To be notified by Artifactory, contact proper software coordinator to request it for you.
- When adding/defining a new dependency in a project, verify if the POM file of a given dependency also describes the licence (<license> element). If not, signal it to your software coordinator.
- If the dependency does not have a licence defined, group such cases and notify it to the software coordinator, who passes it further to the IPR coordination team at ipr-coordinator@geant.net.
- Browse Artifactory build repository and notify licences of what dependencies/artifacts shall be added to the repository. Such notifications should then be sent to ipr-coordinator@geant.net by the proper software coordinator.
- Respond to problems notified by automatic emails sent by Artifactory after each build, consulting the IPR coordination team regarding any unapproved and not-found licences.

References

[ACCESS]	http://msdn.microsoft.com/en-us/library/aa291312%28VS.71%29.aspx
[AGILE]	http://agilemanifesto.org/
[AGILECRIT]	http://en.wikipedia.org/wiki/Agile_software_development
[ANTIPATTERN]	http://en.wikipedia.org/wiki/Anti-pattern#Object-oriented_design_anti-patterns
[APACHETMPL]	http://www.apache.org/licenses/LICENSE-2.0.html
[ARTIFACTORY]	http://www.jfrog.com/
[ArtifactoryGN3]	https://artifactory.geant.net/artifactory/
[ATL]	http://www.atlassian.com/
[BeanUtils]	http://commons.apache.org/beanutils/index.html
[Betwixt]	http://commons.apache.org/betwixt/
[BSDTMPL]	http://www.opensource.org/licenses/bsd-license.php
[Castor]	http://castor.codehaus.org/
	http://www.ibm.com/developerworks/xml/library/x-xjavacastor1/
[CHOICE]	http://fsfe.org/projects/gplv3/barcelona-rms-transcript.en.html#q9b-choosing-v2-or-v3
[Cloud Services]	P. Mell and T. Grance, “The NIST Definition of Cloud Computing”, National Institute of Standards and Technology (NIST), Special Publication 800-145, September 2011; available from http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf
[CMM]	Carnegie Mellon Univ. Software Engineering Inst., Marc C. Paulk, et al., “The Capability Maturity Model: Guidelines for Improving the Software Process,” Addison-Wesley Professional, 1994
[CMMI]	http://www.sei.cmu.edu/cmmi/
	CMMI Version 1.3 Information Center http://www.sei.cmu.edu/cmmi/solutions/info-center.cfm
[DevOps]	Damon Edwards, “What is DevOps?” http://dev2ops.org/blog/2010/2/22/what-is-devops.html
[DevOps_1]	http://dev2ops.org/blog/2010/11/7/devops-is-not-a-technology-problem-devops-is-a-business-prob.html
[Digester]	http://commons.apache.org/digester/
[Dozer]	http://dozer.sourceforge.net/
[DUAL]	http://en.wikipedia.org/wiki/Dual-licensing
	http://www.oss-watch.ac.uk/resources/duallicence.xml
[ENDEV]	http://endeavour-mgmt.sourceforge.net
[ERL]	T. Erl, "SOA Principles of Service Design", Prentice Hall 2007
[FOSS]	http://wiki.egee-see.org/index.php/Licensing_and_FOSS_Guidelines (accessed September 2009)
[FOSSOLOGY]	http://fossology.org/
[GÉANT IPP]	https://intranet.geant.net/ipr
[GN3 IPR]	http://www.geant.net/ipr

[GPL]	http://www.gnu.org/licenses/gpl.html
[GTD]	http://en.wikipedia.org/wiki/Getting_Things_Done
[IPRPOL]	http://www.geant.net/About_GEANT/Intellectual_Property/Documents/GN3-10-325%20GEANT%20IPR%20Policy%20v1.2%20-%2030SEP11.pdf
[JAXB]	http://www.oracle.com/technetwork/articles/javase/index-140168.html
[Javolution]	http://javolution.org/
[JBAPI]	http://download.oracle.com/javase/6/docs/technotes/guides/beans/index.html
[JBPerf]	http://www.christianschenk.org/blog/java-bean-mapper-performance-tests/
[JiBX]	http://jibx.sourceforge.net/
[LEAN]	http://en.wikipedia.org/wiki/Lean_software_development
	http://www.leanprimer.com/downloads/lean_primer.pdf
[LGPL]	http://www.gnu.org/licenses/lgpl.html
[LICENCE]	http://fosswire.com/post/2007/04/the-differences-between-the-gpl-lgpl-and-the-bsd/
	http://www.opensource.org/licenses/alphabetical
	http://www.softwarefreedom.org/resources/2008/compliance-guide.html
[LICENCECHOICE]	http://blog.jboss.org/blog/mfleury/2004/08/02/From+GPL+to+BSD+to+LGPL:+On+the+Issue+of+Business+Friendliness.html
	http://www.openoffice.org/FAQs/faq-licensing.html#whylgplv3
	http://www.fsf.org/licensing/licenses/gpl-howto.html
[LICENSING]	http://www.dwheeler.com/essays/floss-license-slide.html
	http://www.fsf.org/licensing/licenses/
	http://www.apache.org/legal/resolved.html
	http://www.gnu.org/licenses/gpl-faq.html
	http://www.gnu.org/licenses/old-licenses/gpl-2.0.html
	http://www.mysql.com/about/legal/licensing/foss-exception/
	http://www.mysql.com/products/which-edition.html
	http://www.mysql.com/about/legal/licensing/oem/
[MDA]	A.T. Rahmani, V. Rafe, S. Sedighian, A. Abbaspour, "An MDA-Based Modeling and Design of Service Oriented Architecture", Springer 2006
[Mockito]	http://code.google.com/p/mockito/
[MockObj]	http://www.sizovpoint.com/2009/03/java-mock-frameworks-comparison.html
[MocksTDD]	http://www.theserverside.com/news/1365050/Using-JMock-in-Test-Driven-Development
[OPENS]	http://www.fsf.org/licensing/essays/free-sw.html
	http://opensource.org/docs/osd
	http://www.fsf.org/licensing/essays/categories.html
	http://www.iosn.net/licensing/foss-licensing-primer/foss-licensing-final.pdf
[OPENUP]	http://www.eclipse.org/epf/general/OpenUP.pdf
[OPENUPLIFE]	http://epf.eclipse.org/wikis/openup/
[**OSI**]	http://www.opensource.org/
[PRINCIPLES]	T. Erl, “Principles of Service Design”, Prentice Hall 2007
[ProBuf]	http://www.predic8.com/protobuf-etch-thrift-comparison.htm
[ProBufComp]	http://www.predic8.com/protobuf-etch-thrift-comparison.htm
[ProBufDev]	http://code.google.com/apis/protocolbuffers/docs/overview.html
[ProBufJava]	http://code.google.com/apis/protocolbuffers/docs/reference/java-generated.html
[ProBufMes]	http://blog.oskarsson.nu/2009/02/compact-and-cross-language.html
[ProBufPerf]	http://hubpages.com/hub/protocolbuffers

[ProBufRMS]	http://jfaleiro.wordpress.com/2009/12/03/rms-and-protocol-buffers/
[QA]	The most recent version of the GN3 *Quality Assurance Best Practice* Guide can be accessed from the GÉANT Intranet at: https://intranet.geant.net/sites/Services/SA4/T1/Documents/Forms/AllItems.aspx
[QUAL]	QualiPSo Deliverable A6.D1.6.3 "CMM-like model for OSS" http://qualipso.org/sites/default/files/A6.D1.6.3CMM-LIKEMODELFOROSS_0.pdf
[REDM]	http://www.redmine.org
[Rest]	http://portal.acm.org/citation.cfm?id=1367606
[SA4IPRSup]	https://intranet.geant.net/sites/Services/SA4/T3/pages/ci-artifactory-license-checks.aspx
[SDG]	The most recent version of the GN3 Software Developer Guide can be accessed from the GÉANT Intranet at: https://intranet.geant.net/sites/Services/SA4/T1/Documents/Forms/AllItems.aspx
[SDP]	http://www.vocw.edu.vn/content/m10078/latest/ http://en.wikipedia.org/wiki/Software_Development_Process
[SEAM]	http://www.jboss.com/products/seam/
[Sharding]	http://www.codefutures.com/database-sharding
[SHNEIDERMAN]	Shneiderman, B., "Designing the user interface: Strategies for effective human-computer interaction" (3rd ed.), Reading, MA, Addison-Wesley Publishing 1998
[SOA]	M. Rosen, B. Lublinsky, K.T. Smith, M.J. Balcer, "Applied Soa: Service-Oriented Architecture And Design Strategies", Wiley Publishing 2008
[SQL]	http://www.linuxjournal.com/article/10770
[TDD]	http://www.agiledata.org/essays/tdd.html http://xprogramming.com/articles/testfirstguidelines/
[TopInf]	Lewandowski, Labedzki, Mazurek, Wolski, "Topology information as a service in a complex networks – cNIS case", Foundations of Computing and Decision Sciences 35/4, 2010
[TRAC]	http://trac.edgewall.org
[Transmorph]	http://transmorph.sourceforge.net/wiki/index.php/Main_Page
[USABILITY]	http://www.useit.com/papers/heuristic/heuristic_list.html http://www-static.cc.gatech.edu/classes/cs6751_97_winter/Topics/design-princ/ http://www.humanfactors.uiuc.edu/Reports&PapersPDFs/humfac01/wroblewskirantanenhf01.pdf http://aralbalkan.com/687 http://www.ibm.com/developerworks/library/w-berry3/index.html http://www.w3.org/TR/WCAG20/ http://msdn2.microsoft.com/En-US/library/aa511327.aspx http://download.microsoft.com/download/e/1/9/e191fd8c-bce8-4dba-a9d5-2d4e3f3ec1d3/ux%20guide.pdf
[WEBDESIGN]	http://webdesign.about.com/od/forms/a/aa050707.htm http://webdesign.about.com/od/examples/l/bl_aa050707.htm
[XML]	http://www.w3.org/TR/REC-xml/ http://monasticxml.org/one.html http://www.xfront.com/BestPracticesHomepage.html http://www.w3.org/TR/xmlschema-0/ http://www.w3.org/TR/xmlschema-1/ http://www.w3.org/TR/xmlschema-2/
[XMLBeans]	http://xmlbeans.apache.org/index.html

[XStream]	http://xstream.codehaus.org/tutorial.html
[ZEN]	http://www.csszengarden.com/

Glossary

AJAX	Asynchronous JavaScript and XML
AL	Activity Leader
API	Application Programming Interface
AUP	Agile Unified Process
BSD	Berkeley Software Distribution
BPM	Business Process Management
CI	Continuous Integration
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
CMS	Content Management System
CRM	Customer Relationship Management
CSS	Cascading Style Sheets
DMS	Document Management System
DRY	Don't Repeat Yourself
DSDM	Dynamic Systems Development Method
DTO	Data Transfer Object
EAI	Enterprise Application Integration
ERP	Enterprise Resource Planning
EssUP	Essential Unified Process
FDD	Feature Driven Development
FOSS	Free and Open Source Software
GIS	Geographical Information System
GN3	The GN3 Project (3rd GÉANT project year)
GPL	General Public License
GUI	Graphical User Interface
DRY	Don't Repeat Yourself
DTO	Data Transfer Object
EJB	Enterprise JavaBeans
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IDL	Interface Description Language
IETF	Internet Engineering Task Force
IP	Intellectual Property

IPR	Intellectual Property Rights
ITIL	Information Technology Infrastructure Library
ITSM	Information Technology Services Management
JCP	Java Community Process
JSF	JavaServer Faces
JSP	JavaServer Pages
JSR	Java Specification Request
JSTL	JavaServer Pages Standard Tag Library
LGPL	Lesser General Public License
MDA	Model-driven architecture
MVC	Model View Controller
OCP	Open-Closed Principle
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
OpenUP	Open Unified Process
ORM	Object-Relational Mapping
OSI	Open Source Initiative
PaaS	Platform as a Service
PIM	Platform-Independent Model
PSM	Platform-Specific Model
QA	Quality Assurance
RC	Release Candidate
RDBMS	Regional database management systems
REST	REpresentational State Transfer
RFC	Request for Comments
RMI	Remote Method Invocation (Java API)
RPC	Remote Procedure Call
SaaS	Software as a Service
SAX	Simple API for XML
SIT	System Integration Testing
SCM	Supply Chain Management
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SRP	Single Responsibility Principle
StAX	Streaming API for XML
SVT	Stress and Volume Testing
TDD	Test-Driven Development
TFD	Test-First Development
TLD	Test-Late Development
TPS	Toyota Production System
TWE	Trustworthy Element
UAT	User Acceptance Testing
UDDI	Universal Description Discovery and Integration
URL	Uniform Resource Locator

WML	Wireless Markup Language
WSDL	Web Services Description Language
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language
XSD	XML Schema Definition
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformations
XP	Extreme Programming