Исследования и разработки в области новых информационных технологий и их приложений

https://doi.org/10.25559/SITITO.021.202502.260-271
УДК 004.44:004.43

Модульная архитектура на основе локатора сервисов для рефакторинга монолитной нативной программной системы

А. С. Шумихин

ФГБУ «Всероссийский научно-исследовательский геологический нефтяной институт», г. Москва, Российская Федерация

Адрес: 105118, Российская Федерация, г. Москва, Шоссе Энтузиастов, д. 36

shmkhn@geosys.ru

Оригинальная статья

Аннотация

Монолитные нативные программные системы, отличающиеся высокой связностью компонентов и зависимостью от проприетарных технологий, создают значительные трудности при модернизации и переходе на новые платформы. Данная работа посвящена разработке подхода к инкрементальному рефакторингу такой системы, обеспечивающего слабую связность и кроссплатформенность. Предложенное решение основано на использовании локатора сервисов и генерации программного кода для создания модульных компонентов с чёткими интерфейсами. Предлагаемая архитектурная реализация локатора сервисов использует принципы разделения ответственности и инверсии управления, комбинирует паттерны объектно-ориентированного программирования, обеспечивая безопасность типов времени выполнения, а также корректность управления временем жизни объектов и последовательности их инициализации. Подход поддерживает три цели: добавление новых функций, уменьшение технического долга и миграцию на новые инструменты и платформы для снижения зависимости от поставщиков. Разработанная архитектура позволяет проводить рефакторинг без остановки разработки, упрощая интеграцию и снижая риски. Эмпирическая проверка на реальном кейсе миграции программного обеспечения со стека технологий Embarcadero (Borland) C++ Builder и VCL на Microsoft Visual Studio и Qt с дальнейшим портированием на операционную систему Linux подтвердила применимость решения, которое может быть использовано для других систем с аналогичными ограничениями.

Ключевые слова: рефакторинг, локатор сервисов, генерация кода, модульность, кроссплатформенность, дизайн-ориентированное исследование, слабая связность, паттерны проектирования

Конфликт интересов: автор заявляет об отсутствии конфликта интересов.

Для цитирования: Шумихин А. С. Модульная архитектура на основе локатора сервисов для рефакторинга монолитной нативной программной системы // Современные информационные технологии и ИТ-образование. 2025. Т. 21, № 2. С. 260-271. https://doi.org/10.25559/SITITO.021.202502.260-271

© Шумихин А. С., 2025

Контент доступен под лицензией Creative Commons Attribution 4.0 License.
The content is available under Creative Commons Attribution 4.0 License.

Research and Development in the Field of New IT and Their Applications

Modular Architecture Based on Service Locator for Refactoring a Monolithic Native Software System

A. S. Shumikhin

All-Russian Research Geological Oil Institute, Moscow, Russian Federation

Address: 36 Sh. Entuziastov, Moscow 105118, Russian Federation

shmkhn@geosys.ru

Original article

Abstract

Monolithic native software systems, characterized by high component coupling and dependency on proprietary technologies, pose significant challenges for modernization and migration to new platforms. This work is dedicated to developing an approach to incremental refactoring of such a system, ensuring loose coupling and cross-platform compatibility. The proposed solution is based on the use of a service locator and code generation to create modular components with clear interfaces. The proposed architectural implementation of the service locator employs principles of separation of concerns and inversion of control, combining object-oriented programming patterns to ensure runtime type safety, as well as proper management of object lifecycles and their initialization sequence. The approach supports three goals: adding new functionality, reducing technical debt, and migrating to new tools and platforms to decrease vendor dependency. The developed architecture enables refactoring without halting development, simplifying integration, and reducing risks. Empirical validation on a real-world case of migrating software from the Embarcadero (Borland) C++ Builder and VCL technology stack to Microsoft Visual Studio and Qt, with subsequent porting to the Linux operating system, confirmed the applicability of the solution, which can be used for other systems with similar constraints.

Keywords: refactoring, service locator, code generation, modularity, cross-platform compatibility, design science research, loose coupling, design patterns

Conflict of interests: The author declares no conflict of interests.

For citation: Shumikhin A.S. Modular Architecture Based on Service Locator for Refactoring a Monolithic Native Software System. *Modern Information Technologies and IT-Education.* 2025;21(2):260-271. https://doi.org/10.25559/SITITO.021.202502.260-271

Введение

Монолитные программные системы, характеризующиеся сильной связанностью компонентов и хрупкостью кода, создают значительные трудности для рефакторинга и модернизации из-за накопленного технического долга и сложностей с выделением модулей [1, 2]. Независимо от монолитной архитектуры, системы, привязанные к проприетарным технологиям поставщиков (*vendor lock-in*, привязка к поставщику), сталкиваются с ограничениями портируемости и сложностями интеграции с современными инструментами[1]. Эти две проблемы – монолитность и привязка к поставщику – могут сочетаться, усиливая вызовы, связанные с обновлением архитектуры. В десктопных нативных (разрабатываемых для конкретной операционной системы с использованием её низкоуровневых языков и инструментов) приложениях достижение динамической модульности осложняется необходимостью обеспечения типобезопасности и управления временем жизни объектов, что делает рефакторинг особенно трудоёмким [3].

Данная работа посвящена **разработке** архитектуры геоинформационной системы (ГИС) INTEGRO, поддерживающей ее инкрементальный рефакторинг. Изначально система разрабатывалась с использованием Embarcadero C++ Builder 10.1 и библиотеки Visual Control Library (VCL). В своих ранних версиях она страдала от зависимости от проприетарных технологий и ограниченной поддержки современных стандартов, что препятствовало портированию на альтернативные Windows платформы и создавало риски в случае прекращения поддержки или поставки платформы либо необходимых инструментов разработки. Хотя система INTEGRO поддерживала подключение плагинов для расширения функциональности, эти плагины зависели от VCL и не имели единого механизма загрузки, что создавало значительные сложности в их разработке и интеграции. Изначально целью было повышение расширяемости системы путём поиска решения, которое унифицировало бы работу с плагинами с целью повышения модульности и расширяемости приложения [4, 5]. Ранее был проведен анализ существующих решений, направленных на обеспечение расширяемости (Qt, Poco OSP, SOF, nOSGI, CTK, Celix C++, Celix C, COM/ActiveX, XPCOM, CORBA) [6, 7], который выявил ограничения в их применимости в контексте рассматриваемой системы; чтобы обеспечить возможность использовать для создания подключаемых модулей широкий спектр языков, компиляторов и интегрированных сред разработки, в том числе свободно распространяемых, была разработана технология построения совместимого двоичного интерфейса *(Application Binary Interface, ABI)*, основанная на кодогенерации, управляемой XML-моделями объектно-ориентированных интерфейсов [8]. Выбор этого подхода преодолевал проблему отсутствия стандартизированного ABI для C++ и открывал перспективу использования для разработки модулей современных инструментов, таких как Microsoft Visual C (MSVC) с поддержкой C++11/17, и снижения зависимости от проприетарных технологий путем рефакторинга [9]. В свою очередь, устранение привязки к поставщику могло в перспективе способствовать переходу к кроссплатформенности. С учетом этого были сформулированы три уровня целей, на которые опиралась данная работа: краткосрочная – добавление новых функций через плагины, среднесрочная – уменьшение технического долга путём рефакторинга монолита [2], [10], и долгосрочная – диверсификация платформ для снижения зависимости от проприетарных технологий и минимизации связанных с этим рисков через поддержку операционных систем семейства Linux, и альтернативных оконных фреймворков, таких как Qt. Противоречие между краткосрочной и среднесрочной целями, связанное с замедлением разработки вследствие ограниченных ресурсов и риском дестабилизации системы, вызванным каскадными изменениями в коде при рефакторинге, могло быть снято применением паттерна Душитель (Strangler Fig Pattern)[2] [11]. Этот паттерн позволяет постепенно заменять старый код новыми модулями без остановки разработки, но его использование предполагает архитектуру, обеспечивающую минимальное вмешательство в существующий программный код (неинвазивность) при добавлении новых компонентов системы [12]. Целью представленного исследования было разработать архитектуру, обладающую расширяемостью, обеспечивающую слабую связанность компонентов, поддерживающую инкрементальный рефакторинг в соответствии с упомянутым паттерном Душитель, и независимую от платформы, а также сохраняющую безопасность типов и управления ресурсами (главным образом, памятью) при использовании C++ в работах[3]. Статья описывает процесс разработки, включая анализ альтернатив, детализирует архитектуру, обобщает теоретический и практический вклад разработки.

[1] O'Grady S. The New Kingmakers: How Developers Conquered the World. O'Reilly Media, Inc., 2013. 56 p.

[2] Fowler M. Strangler Fig [Электронный ресурс] // Martin Fowler, 22 August 2024. https://martinfowler.com/bliki/StranglerFigApplication.html (дата обращения: 28.03.2025).

[3] Stroustrup B., Sutter H., Dos Reis G. A brief introduction to C++'s model for type-and resource-safety [Электронный ресурс] // Isocpp.org. October, 2015. URL: https://www.stroustrup.com/resource-model.pdf (дата обращения: 28.03.2025); Stroustrup B., Dos Reis G. Design Alternatives for Type-and-Resource Safe C++ [Электронный ресурс] // Programming Language C++. No. P2687R0. 2022. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2687r0.pdf (дата обращения: 28.03.2025).

Методология

Разработка плагинной архитектуры велась по принципам дизайн-ориентированного исследования (DSR) [13], обеспечивая баланс между научной строгостью и практической ценностью, как отмечено Lee, 1999: «*чрезмерный акцент на строгость в поведенческих исследованиях информационных систем часто приводил к снижению релевантности*» [14]. DSR ориентировано на создание артефактов (концепций, моделей, методов, алгоритмов, процессов, инструментов или прототипов) для решения практических задач. Этапы проведенного исследования включали анализ проблемы, генерацию и оценку архитектурных подходов, проектирование архитектуры, её реализацию и эмпирическую валидацию.

Анализ проблемы

После выбора кодогенерации, управляемой XML-моделями объектно-ориентированных интерфейсов, в качестве технологической базы для обеспечения взаимодействия модулей открытой проблемой оставалась разработка XML-моделей новой плагинной инфраструктуры и компонентов системы, которые реализовывали бы соответствующие интерфейсы. Эти модели и компоненты и составляли искомый артефакт.

Разрабатываемая архитектура должна была обеспечить:

1. Расширяемость системы для добавления новых функций [4];
2. Снижение технического долга за счет введения четких программных контрактов [15];
3. Простоту инкрементального рефакторинга [11];
4. Обеспечение безопасного порядка инициализации зависимых сервисов;
5. Типобезопасность [3];
6. Безопасность и гибкость в управлении временем жизни объектов[4].

Кроме того, архитектура должна была соответствовать ограничениям выбранной технологии:

1. Использование в интерфейсах только простых структур данных (Plain Old Data, POD), к которым относятся и сгенерированные описатели интерфейсов [16];
2. Отсутствие поддержки исключений;
3. Отсутствие поддержки шаблонов;
4. Отсутствие поддержки перегруженных методов;
5. Отсутствие поддержки наследования интерфейсов;
6. Использование кодогенерированного механизма динамической идентификации типов (интерфейсов) взамен предоставляемого языком программирования [17];

Дополнительно, от новой архитектуры требовалось не подавлять преимущества названной технологии ABI-совместимости, такие как:

1. Независимость от проприетарных технологий и кроссплатформенность;
2. Возможность добавления интерфейсов без изменения объекта [18];
3. Возможность привязки к различным языкам программирования;
4. Низкие накладные расходы на вызов метода.

Рассмотрение альтернатив

В первую очередь были проанализированы архитектурные подходы, извлечённые из существующих решений. Следует подчеркнуть дважды, что на представленном этапе работы в качестве альтернатив изучалось не использование сторонних реализаций, а применение заложенных в них принципов, паттернов и общих подходов. Это связано с тем что, как было показано на предыдущем этапе работы, существующие готовые решения не обеспечивают взаимодействие модулей, разработанных с использованием разных компиляторов, на уровне объектно-ориентированных интерфейсов [8]. Рассматривались следующие варианты:

1. Использование спецификации OSGi[5] [6]. Реализации спецификации представлены главным образом архитектурами приложений на Java, такими как Eclipse Equinox и использующий ее сервер приложений Eclipse Virgo. В то же время, существуют и реализации этой спецификации для C/C++, такие как Apache Celix, Poco OSP, SOF, CTK, nOSGi. Здесь модуль – динамическая библиотека (DLL/SO), содержащая код, ресурсы и манифест, который описывает сервисы, их зависимости и конфигурацию. Модуль загружается OSGi-фреймворком во время выполнения. Модуль содержит объект, называемый активатором и отвечающий за жизненный цикл модуля, а также объекты – сервисы, реализующие основную функциональность модуля. Активатор регистрирует сервисы через передаваемый ему при загрузке интерфейс контекста модуля и отменяет регистрацию при выгрузке. Доступ к сервисам осуществляется через посредство сервис-локатора. Клиент запрашивает сервисы по их интерфейсу. Для обеспечения гарантий времени жизни сервиса используется инверсия управления (коллбэк). К ограничениям данного подхода, не соответствующим требованиям проекта, относятся не полностью решенные вопросы безопасности типов и совместимости: в реализациях на языке C используется небезопасное приведение типов, а в C++ – реализациях применяются возможности языка, не обеспечивающие совместимость: шаблоны, RTTI [16]. Кроме того, использование манифестов для управления зависимости усложняет внедрение технологии в существующем проекте.

[4] Там же.

[5] Apache Celix Introduction [Электронный ресурс] // The Apache Software Foundation, 2025. URL: https://celix.apache.org/documentation/ (дата обращения: 28.03.2025).

2. Использование контейнера инверсии управления (DI/IoC)[6] [19], наподобие Hypodermic, MinIoC, iocplusplus, Boost.DI, который берет на себя управление жизненным циклом и связыванием компонентов, гарантируя такой порядок их создания и удаления, при котором неразрешенные зависимости не возникает. Контейнер также обеспечивает загрузку необходимых модулей, независимо от их природы. Для динамического связывания требует механизмов рефлексии и интроспекции, не предоставляемых языками программирования C и C++. Внедрение этой технологии в легаси-системе потребовало бы масштабной переработки кода.
3. Подход COM[7], основанный на компонентах. Компонент – объект конкретного класса, предоставляющий некоторую совокупность интерфейсов и имеющего хорошо известный идентификатор (CLSID, представляющий собой GUID). Класс компонента регистрируется в системном реестре, где CLSID ассоциируется с реализующим этот класс модулем. В случае взаимодействия внутри одного процесса модуль (динамическая библиотека) содержит фабрику класса. Клиент использует интерфейс фреймворка, чтобы создать экземпляр известного ему класса, фреймворк загружает динамическую библиотеку и с помощью ее фабрики класса создает экземпляр объекта, возвращая указанный интерфейс для него. Для обеспечения гарантий времени жизни объектов и модулей используются счетчики ссылок. Применение подхода ограничено зависимостью клиента от конкретной реализации компонента и необходимостью использовать навязанный фреймворком механизм управления временем жизни объектов.
4. Подход Qt Plugins[8], основанный на модулях с метаданными. Плагин представляет собой динамическую библиотеку, которая экспортирует классы с интерфейсами, имеющими хорошо известные строковые идентификаторы (IID). Плагин загружается через интерфейс фреймворка с указанием пути к файлу библиотеки. Загрузчик плагина проверяет метаданные и загружает плагин, возвращая указатель на объект единого базового класса, который может быть преобразован к указателю на требуемый интерфейс. Подход широко полагается на использование расширенного синтаксиса языка и специальных инструментов (так называемого компилятора метаобъектов MOC) для генерации метаданных плагинов и кода, обеспечивающего взаимодействие с ними. Использование ABI-несовместимых типов данных и специфичных механизмов ограничивает портируемость и совместимость с другими инструментами сборки. Использование MOC создает зависимость от стороннего фреймворка.
5. Подход CORBA[9] с распределенными сервисами. Здесь под сервисом понимается система интерфейсов, а реализующие их объекты можно получить по хорошо известным начальным ссылкам или через другие сервисы, например, через сервис именования (CosNaming). Этот подход абстрагирован от способа загрузки модулей и управления жизненным циклом объектов. Для взаимодействия с объектами, с которыми отсутствует ABI-совместимость, используется сериализация параметров, что увеличивает накладные расходы на вызов метода.
6. Подход SOFA [7], основанный на компонентной архитектуре, где компоненты описываются через архитектурные описания и поддерживают динамическое связывание. Однако его сложность и зависимость от специфичных инструментов ограничивают применимость в контексте легаси-систем.

Таблица 1. **Соответствие архитектурных подходов требованиям и ограничениям проекта**
Table 1. **Compliance of architectural approaches with the project requirements and constraints**

Подход	Расширяемость	Неинвазивность	Кроссплатформенность	ABI -совместимость	Типобезопасность	Нейтральность ко времени жизни
OSGi	+	+	+	-	+/-	-
DI/IoC	-	-	+	+	+	-
COM	+	-	-	+	+	-
Qt	+	-	+	-	+	-
ORB	+	-	+	-	+	+
SOFA	+	-	+	-	+	-
Статический шаблонный	-	+	+	-	+	-

Источник: здесь и далее в статье все таблицы и рисунки составлены автором.
Source: Hereinafter in this article all tables and figures were made by the author.

7. Подход с использованием шаблонов[10]. Компонент представляет собой конкретный класс с интерфейсами в виде абстрактных базовых классов. Зависимости

[6] Boost.DI [Электронный ресурс] // GitHub, 2025. URL: https://boost-ext.github.io/di/ (дата обращения: 28.03.2025).
[7] Microsoft. COM: Component Object Model Technologies [Электронный ресурс] // Microsoft, 2025. URL: https://docs.microsoft.com/en-us/windows/win32/com (дата обращения: 28.03.2025); Gruntz C., Murer S. Component Software: Beyond Object-Oriented Programming. Pearson Education, 2002. 589 p.
[8] The Future of Digital Experiences [Электронный ресурс] // Qt Documentation, 2025. URL: https://doc.qt.io (дата обращения: 28.03.2025).
[9] OMG CORBA [Электронный ресурс] // OMG, 2025. URL: https://www.omg.org/ (дата обращения: 28.03.2025).
[10] Farrier J. The Service Locator Pattern: A Robust C++ Implementation [Электронный ресурс] // John Farrier, 2025. URL: https://johnfarrier.com/the-service-locator-pattern-a-robust-c-implementation/ (дата обращения: 28.03.2025).

разрешаются с использованием шаблонов на этапе компиляции, что не позволяет его использовать для динамического подключения модулей.

Как можно видеть, ни один из рассмотренных подходов не удовлетворял одновременно всем сформулированным требованиям (Таблица 1). Тем не менее, эти подходы послужили источником для заимствования отдельных концепций и соответствующей терминологии.

Синтез оригинального подхода

Ограничения существующих альтернатив подтолкнули к разработке оригинальной архитектуры сервис-локатора, вдохновленной некоторыми почерпнутыми из них идеями. Привлекательными для заимствования были найдены следующие элементы рассмотренных подходов:

1. OSGi: Концепции бандлов (модули с жизненным циклом) и сервисов (интерфейсы в реестре) с их терминологией, инверсия управления, сервис-локатор, контекст для управления сервисами [6];
2. COM: использование фабрик для создания объектов[11];
3. SOFA: Архитектурное описание компонентов для поддержки модульности [7].
4. ORB: абстракция локатора от загрузки модулей и управления временем жизни сервиса.

Эти элементы легли в основу подхода с кодогенерацией, XML-моделями и локатором сервисов, обеспечивающим типобезопасность, ABI-совместимость, слабую связанность и кроссплатформенность [9].

Проектирование архитектуры

Предложенная архитектура была призвана сохранить достоинства известных подходов, избегая их недостатков, что обеспечивается следующим рядом решений:

1. Как менее инвазивный по сравнению с внедрением зависимостей *(Dependency Injection, DI)* [19], для обеспечения инкрементального рефакторинга с минимальным вмешательством в существующую кодовую базу был применен паттерн Service Locator[12] [20]. В соответствии с этим паттерном получатель служб запрашивает их через единый интерфейс.
2. Проблема безопасности и гибкости управления временем жизни сервиса решается за счет инверсии управления: интерфейс сервиса передается получателю через обратный вызов (коллбэк). Коллбэк представляет собой интерфейс с методами, которые принимают сервис требуемого типа и выполняют операцию над ним без изменения сервиса и без необходимости передавать ответственность за его время жизни, что соответствует паттерну Visitor [18], [21]. Использование коллбэка также позволяет получать несколько сервисов в одном запросе, следуя паттерну Internal Iteration [22]
3. Для обеспечения гибкости решения, выражающейся в возможности добавлять типы сервисов без изменения локатора, интерфейс обратного вызова передается в локатор с обобщенным типом и должен быть преобразован к конкретному типу на стороне поставщика сервисов. Это позволяет использовать поддержку коллбэком определенного интерфейса как критерия для поиска сервисов по аналогии с паттерном Marker Interface[13], а также избежать приведения типа на стороне получателя сервисов. Поскольку преобразование типов контролируется функциями, полученными генерацией кода из моделей интерфейсов, такая проверка обеспечивает совпадение версий ожидаемого и предоставляемого интерфейсов сервиса [8]. Локатор сервисов, таким образом, выступает в качестве медиатора в соответствии с паттерном Mediator[14].
4. Для безопасного упорядочения инициализации, а также для оптимизации производительности сервисы должны иметь возможность создаваться и инициализироваться лениво, для чего обязанность создания сервисов и передачи их посетителям возлагается на фабрики в соответствии с паттерном Factory[15].
5. Для перечисления предоставляемых модулем фабрик сервисов используется интерфейс обратного вызова. Для дальнейшей оптимизации производительности за счет ленивой загрузки подключаемых модулей и сужения области поиска фабрик им присваиваются строковые идентификаторы категорий.

Реализация

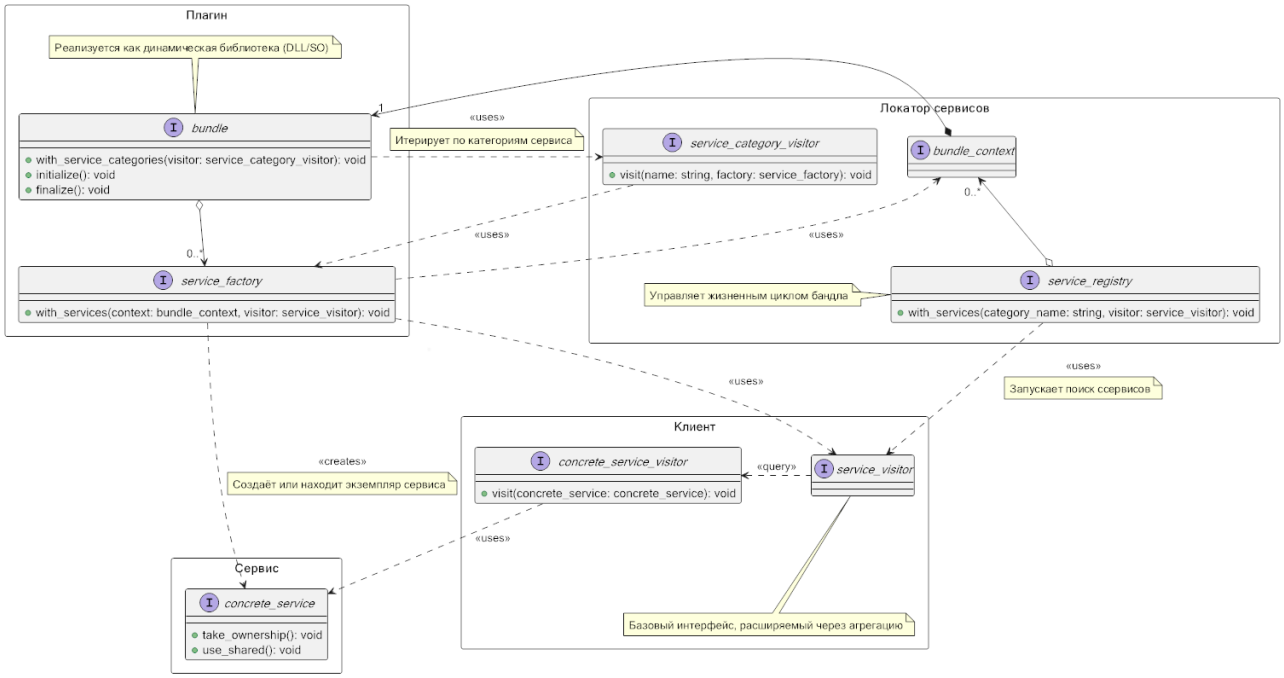
Технология разработки включала следующие шаги:

1. Описание интерфейсов в XML. Интерфейсы для новых сервисов описывались в XML-файлах, определяющих контракты для взаимодействия между ядром и плагинами [15].
2. Кодогенерация интерфейсов. На основе XML-описаний генерировался C++-код, включая C-структуры или производные классы, содержащие указатель на экземпляр и таблицу виртуальных методов [8], обеспечивая совместимость с C ABI [17].
3. Внедрение точек расширения. В монолит добавлялись точки расширения, где код обращался к сервис-локатору для получения сервисов, что что требовало минимальных изменений в существующем коде [12].

[11] Gruntz C., Murer S. Component Software: Beyond Object-Oriented Programming. Pearson Education, 2002. 589 p.
[12] Fowler M. Using A Service Locator [Электронный ресурс] // Martin Fowler. 23 January 2004. URL: https://martinfowler.com/articles/injection.html#UsingAServiceLocator (дата обращения: 28.03.2025).
[13] Design Patterns: Elements of Reusable Object-Oriented Software / E. Gamma [et al.]. Professional Computing Series. Addison-Wesley Professional, 1994. 416 p.
[14] Там же.
[15] Там же.

4. Разработка плагинов. Создавались плагины, дублирующие функциональность ядра, реализованные как динамически загружаемые модули (DLL/SO).
5. Тестирование. Плагины тестировались для проверки корректности дублирования функциональности ядра и стабильности системы.
6. Удаление устаревшего кода. После успешного тестирования устаревший код удалялся из ядра, завершая цикл рефакторинга по паттерну Душитель [11].
7. Вспомогательные библиотеки. Для поддержки процесса были разработаны вспомогательные библиотеки на C++, предоставляющие типовые реализации специфицированных интерфейсов.

В рамках предложенной архитектуры в соответствии с принципом разделения ответственности выделяются такие роли взаимодействующих компонентов, как Клиент, Локатор Сервисов, Бандл и Сервис. Под Сервисом понимается программная сущность, реализующая строго определенный контракт. Природа этой сущности и ее контракта, включая гарантии времени жизни, заранее не определена. В роли Клиента выступает компонент, который при выполнении своей функции зависит от Сервиса, реализующего необходимый контракт, и который инициирует разрешение этой зависимости. Локатором Сервисов называется компонент, который предоставляет функциональность для поиска



Р и с. 1. Компоненты предложенной плагинной архитектуры
F i g. 1. Components of the proposed plugin architecture

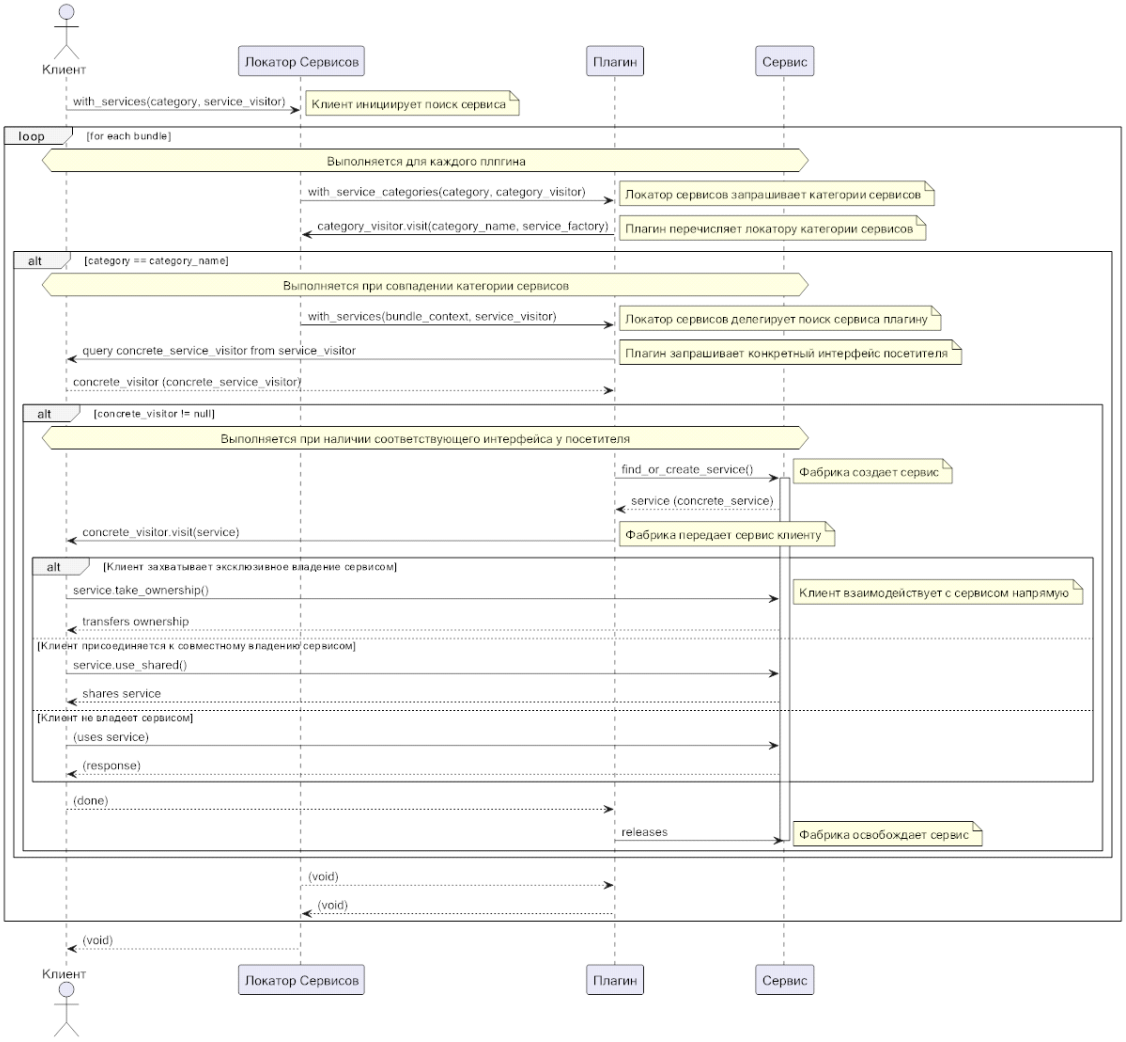
необходимых Сервисов. Бандл представляет собой контейнер, содержащий и предоставляющий сервисы, такой как подключаемый модуль (плагин). Взаимодействие компонентов осуществляется через ABI-совместимые интерфейсы. Рассматриваемая архитектура определяет следующие интерфейсы (рисунок 1):
Для Клиента: интерфейс посетителя сервисов – service_visitor, который служит маркером для обеспечения типобезопасности, и интерфейс посетителя сервисов с заданным контрактом – условно, concrete_service_visitor, предназначенный непосредственно для получения требуемого сервиса;
Для Локатора Сервисов: интерфейс реестра сервисов service_registry, через который Клиент обращается к Локатору Сервисов с запросом на поиск сервисов; интерфейс контекста бандла bundle_context, который идентифицирует контейнер с сервисами; интерфейс посетителя категорий сервисов – service_category_visitor, который служит для получения идентификатора категории сервисов и фабрики сервисов этой категории;
Для Бандла: интерфейс бандла bundle с методами для управления жизненным циклом контейнера и для доступа к категориям предоставляемых сервисов; интерфейс фабрики сервисов service_factory, отвечающий за предоставление сервисов;
Для Сервиса: интерфейс сервиса concrete_service (название приведено условно, поскольку сервис может быть представлен значением любого типа).

Последовательное взаимодействие через интерфейсы

1. Инициализация системы. Ядро содержит синглтон, выступающий в роли Service Locator. На старте ядро сканирует каталоги, загружает плагины (реализованные как DLL/SO), импортирует точку

входа, получает с ее помощью интерфейс bundle и регистрирует в локаторе.

2. Поиск сервисов. Клиент запрашивает необходимый сервис, указывая его категорию и передавая интерфейс service_visitor. Объект, реализующий интерфейс service_visitor, должен также поддерживать интерфейс concrete_service_visitor, имеющий метод для получения необходимого сервиса. Локатор сервисов обходит зарегистрированные бандлы и запрашивает у каждого перечисление фабрик с их категориями. В случае совпадения категории с указанной в запросе локатор передает service_visitor фабрике. Фабрика пытается преобразовать интерфейс service_visitor к concrete_service_visitor, используя сгенерированные функции. Если такой интерфейс получен, фабрика передает посетителю интерфейс сервиса, создавая при необходимости реализующий его объект. Таким образом архитектура избегает небезопасных ручных приведений типа, которые скрыты в автоматически сгенерированном коде и содержат проверку на совпадение версий запрашиваемого и предоставляемого интерфейсов [8]. Кроме того, минимизируется риск ошибок, связанных с отсутствием проверки успешности безопасного приведения типа, поскольку эта успешность является критерием запроса клиента на получение сервиса от фабрики, и необходимость проверки диктуется логикой обработки этого запроса фабрикой.



Р и с. 2. Процесс запроса сервиса через локатор
F i g. 2. Service request process through the locator

Аналогично, исключается необходимость проверки на стороне клиента: фабрика гарантирует, что коллбэк получит валидную ссылку на сервис, а при отсутствии такой возможности коллбэк не будет вызван. Таким образом, компилятор, кодогенератор и архитектура локатора сервисов в совокупности исключают несоответствие ожидаемого и фактического типа объекта, обеспечивая этим типобезопасность времени выполнения.

Возникающие в процессе загрузки модулей события (ошибки загрузки модулей, несовпадение версий интерфейсов, отсутствие необходимых зависимостей) логируется в целях диагностики. При этом обработка ошибок и восстановление после них остаются прозрачными для пользователя и выражаются для

него только в деградации функциональности, связанной с невозможностью получить отдельные сервисы. Причины деградации можно установить по имеющимся логам.

Валидация

Архитектура и технология были протестированы на реальной системе. Эмпирическая валидация включала тестирование плагинов, заменяющих функциональность ядра, успешный переход на современные фреймворки и подготовку к портированию на Linux без каскадных ошибок. Соответствие требованиям расширяемости, снижение технического долга, минимальной инвазивности рефакторинга, безопасного порядка инициализации сервисов, типобезопасности, безопасности управления временем жизни объектов оценивалось качественно по наличию или отсутствию принципиальных ошибок проектирования, препятствующих их выполнению, а также по результатам прохождения системой интеграционных тестов. Результаты подтвердили применимость разработанной архитектуры и технологии, что соответствует принципам оценки решений в разработке программного обеспечения[16]. Итеративная обратная связь позволила оптимизировать реализацию, например, внедрить категории для ускорения поиска.

Результаты

Разработанные архитектура и технология прошли валидацию на реальной системе. В ходе масштабного рефакторинга, проведённого без остановки разработки, функциональность монолитной системы была перенесена в подключаемые модули. Это позволило преодолеть ключевые ограничения исходной платформы, включая отсутствие кроссплатформенности, слабую поддержку современных стандартов C++ и зависимость от проприетарных технологий. После адаптации кода под C++11/17 были обеспечены интеграция с современными инструментами (профилировщиками, IDE) и использование кроссплатформенных библиотек [5].

Показательны следующие численные метрики эффективности рефакторинга (таблица 2), отражающие соответствие решения поставленным целям: количество разработанных подключаемых модулей, свободных от привязки к поставщику компилятора; количество разработанных подключаемых модулей, предоставляющих расширения пользовательского интерфейса, в том числе, интегрируемые с Qt.

Чёткие контракты сервисов, заданные через XML-спецификации, изолировали платформенно-зависимые компоненты (например, GUI на VCL) и позволили заменить их на кроссплатформенные реализации [23]. В рамках миграции основное приложение было перенесено на фреймворк Qt без внесения изменений в существующие модули, после чего система успешно портирована на Linux. Разработано более 100 подключаемых модулей, идентичных на уровне исходного кода для обеих платформ, что подтверждает воспроизводимость подхода.

Т а б л и ц а 2. **Метрики эффективности рефакторинга**
T a b l e 2. **Refactoring efficiency metrics**

	До рефакторинга	После рефакторинга
Количество подключаемых модулей, свободных от привязки к поставщику компилятора	0	150+
Количество подключаемых модулей, предоставляющих расширения пользовательского интерфейса	6	100+
В том числе, количество подключаемых модулей, предоставляющих расширения пользовательского интерфейса, интегрируемые с Qt	0	100+

Обсуждение

Вклад работы заключается в создании методологии рефакторинга монолитных C++-систем, привязанных к проприетарным технологиям, на основе дизайн-ориентированного исследования [24]. Предложенный подход представляет собой композицию паттернов проектирования [23], [25], что поддерживает три ключевые цели: функциональное проектирование, миграцию с устаревших технологий и долгосрочное развитие за счёт кроссплатформенности.

Сравнение с существующими решениями выявило следующие преимущества:

- Против OSGi: Устранена зависимость от манифестов и небезопасных приведений типов, которые нарушают C ABI. Автогенерация интерфейсов заменила RTTI, упростив интеграцию модулей [6], [16].
- Против Boost.DI: реализовано динамическое разрешение зависимостей [19].
- Против COM: Преодолена привязка к Windows-экосистеме благодаря кроссплатформенной кодогенерации и независимости от системного

[16] Reade C. Elements of functional programming. Addison-Wesley Longman Publishing Co., Inc.; 1989; O'Connor P. D. T., Kleyner A. V. Practical reliability engineering. John Wiley & Sons; 2025.

реестра.
• Против SOFA: Упрощена интеграция за счёт исключения сложных архитектурных описаний [7].
• Против CORBA/ORB: Исключены накладные расходы на маршаллинг, так как сервисы доступны напрямую в адресном пространстве.
• Против статических шаблонов: Сохранена ABI-совместимость за применения только простых структур (POD) в API [24].
Эмпирическая валидация, включая портирование на Linux и создание более сотни подключаемых модулей, подтвердила воспроизводимость подхода. Однако выявлены ограничения:
1. Отсутствие межпроцессного взаимодействия (IPC). Сервисы доступны только в пределах одного процесса, что исключает их использование в распределённых сценариях. Например, интеграция с микросервисной архитектурой [11] или взаимодействие с внешними системами требует дополнительных механизмов (RPC, сокеты), которые не предусмотрены текущей реализацией. Это ограничение связано с отказом вводить в условия задачи требование автоматической совместимости интерфейсов с IPC, выполнение которого привело бы к неоправданному усложнению архитектуры и дополнительным накладным расходам во время выполнения за счет необходимости сериализации/десериализации параметров или аналогичных решений.
2. Управление жизненным циклом плагинов:
• невозможность принудительной выгрузки: Модуль невозможно выгрузить, пока все его сервисы не будут освобождены клиентами, из-за чего затруднена реализация горячей замены модулей [25-27];
• отслеживание зависимостей: Подключаемые модули не имеют явной спецификации зависимостей, что создает сложности при инкрементальном обновлении системы;
3. Сложность в отладке
Ленивая инициализация сервисов и многоуровневое делегирование между модулями, разработанными с помощью различных компиляторов, затрудняют совместную отладку основного приложения и плагинов [27].
Практическая значимость работы проявляется в двух аспектах:
• для индустрии – снижение стоимости поддержки legacy-систем за счёт поэтапного рефакторинга;
• для исследований – доказательство возможности управления ABI через кодогенерацию в гетерогенных средах [9].
Перспективы развития системы включают использование манифестов для декларативного описания зависимостей модулей и генерации кода для реализации фабрик. Таким образом, предложенный подход обогащает теорию композиции паттернов и предлагает готовую методологию для модернизации систем, страдающих от монолитности и привязки к поставщику [4, 5].

Выводы

Разработана архитектура локатора сервисов для C++-систем, основанная на кодогенерации ABI-стабильных интерфейсов, которая поддерживает рефакторинг монолитных приложений и позволяет преодолеть зависимость от проприетарных технологий (Borland C++, VCL), а также обеспечить кроссплатформенную совместимость модулей на уровне исходного кода.
1. Ключевые преимущества:
• минимальная инвазивность: Интеграция в legacy-код требует изменений только в точках взаимодействия с плагинами;
• типобезопасность и ABI-совместимость: Автогенерация заменяет небезопасные приведения типов и гарантирует работу модулей между компиляторами (MSVC, GCC) и ОС (Windows, Linux) [16, 17];
• практическая проверка: Успешная миграция системы на Qt и портирование на ОС Linux с созданием 100+ идентичных модулей подтверждает жизнеспособность подхода.
2. Направления для будущих работ:
• расширение для распределённых сценариев (IPC/RPC) через интеграцию с современными протоколами (gRPC, REST);
• реализация механизмов управления зависимостями и жизненным циклом модулей, вдохновлённых OSGi, но без её сложности [6].
Работа показывает, что комбинация кодогенерации и паттернов проектирования позволяет модернизировать legacy-системы с минимальными затратами, сохраняя их работоспособность и открывая путь к эволюции в сторону кроссплатформенности и слабо связанной архитектуры [4, 5].

References

1. Lehman M.M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*. 2005;68(9):1060-1076. https://doi.org/10.1109/PROC.1980.11805
2. Politowski C., et al. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. *Information and Software Technology*. 2020;122:106278. https://doi.org/10.1016/j.infsof.2020.106278
3. Meyers S. Effective C++: 55 specific ways to improve your programs and designs. Pearson Education; 2005.
4. Oliveira B.C.S., Cook W.R. Extensibility for the masses: Practical extensibility with object algebras. In: Extensibility for the Masses. In: Noble J. (eds.) ECOOP 2012 – Object-Oriented Programming. ECOOP 2012. *Lecture Notes in*

Computer Science. Vol. 7313. Berlin, Heidelberg: Springer; 2012. P. 2-27. https://doi.org/10.1007/978-3-642-31057-7_2
5. Wiegand J., et al. Eclipse: A platform for integrating development tools. *IBM Systems Journal*. 2004;43(2):371-383. https://doi.org/10.1147/sj.432.0371
6. Kächele S., et al. nOSGi: a posix-compliant native OSGi framework. In: Proceedings of the 5th International Conference on Communication System Software and Middleware (COMSWARE '11). New York, NY, USA: Association for Computing Machinery; 2011. Article number: 4. https://doi.org/10.1145/2016551.2016555
7. Kalibera T., Tůma P. Distributed component system based on architecture description: The sofa experience. In: OTM Confederated International Conferences "On the Move to Meaningful Internet Systems". Berlin, Heidelberg: Springer Berlin Heidelberg; 2002. p. 981-994. https://doi.org/10.1007/3-540-36124-3_63
8. Shumikhin A.S. Development of elements of technology for transferring the INTEGRO geoinformation system to Linux based on a systems approach. *Modeling, Optimization and Information Technology*. 2024;12(2). (In Russ., abstract in Eng.) https://doi.org/10.26102/2310-6018/2024.45.2.040
9. Dig D., Johnson R. How do APIs evolve? A story of refactoring. *Journal of software maintenance and evolution: Research and Practice*. 2006;18(2):83-107. https://doi.org/10.1002/smr.328
10. Liu A., et al. Prevalence and severity of design anti-patterns in open source programs – A large-scale study. *Information and software technology*. 2024;170:107429. https://doi.org/10.1016/j.infsof.2024.107429
11. Ma S.P. et al. Microservice Migration Using Strangler Fig Pattern and Domain-Driven Design. *Journal of Information Science & Engineering*. 2022;38(6). https://doi.org/10.6688/JISE.202211_38(6).0010
12. Parnas D.L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*. 1972;15(12):1053-1058. https://doi.org/10.1145/361598.361623
13. Hevner A.R., et al. Design science in information systems research. *MIS quarterly*. 2004. p. 75-105. https://doi.org/10.2307/25148625
14. Lee A. Inaugural editor's comments. *MIS Quarterly*. 1999;23(1):v-xi.
15. Liskov B., Zilles S. Programming with abstract data types. *ACM Sigplan Notices*. 1974;9(4):50-59. https://doi.org/10.1145/942572.807045
16. Petrescu C.C., et al. Do names echo semantics? A large-scale study of identifiers used in C++'s named casts. *Journal of Systems and Software*. 2023;202:111693. https://doi.org/10.1016/j.jss.2023.111693
17. Solodkyy Y., Dos Reis G., Stroustrup B. Open and efficient type switch for C++. *ACM SIGPLAN Notices*. 2012;47(10):963-982. https://doi.org/10.1145/2384616.2384686
18. Oliveira B.C.S., Wang M., Gibbons J. The visitor pattern as a reusable, generic, type-safe component. The visitor pattern as a reusable, generic, type-safe component. *ACM SIGPLAN Notices*. 2008;43(10):439-456. https://doi.org/10.1145/1449955.1449799
19. Laigner R., et al. Cataloging dependency injection anti-patterns in software systems. *Journal of Systems and Software*. 2022;184:111125. https://doi.org/10.1016/j.jss.2021.111125
20. Zhang W., Oliveira B.C.S. EVF: An extensible and expressive visitor framework for programming language reuse. In: 31st European Conference on Object-Oriented Programming (ECOOP 2017). Schloss Dagstuhl – Leibniz-Zentrum für Informatik; 2017. p. 29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.29
21. Oliveira B.C.S. Modular visitor components: a practical solution to the expression families problem. In: European Conference on Object-Oriented Programming. Berlin, Heidelberg: Springer Berlin Heidelberg; 2009. p. 269-293. https://doi.org/10.1007/978-3-642-03013-0_13
22. Martin R. Discovering patterns in existing applications. In: Pattern Languages of Program Design. ACM Press/Addison-Wesley Publishing Co., USA; 1995. p. 365-393.
23. Riehle D. Composite design patterns. In: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '97). New York, NY, USA: Association for Computing Machinery; 1997. p. 218-228. https://doi.org/10.1145/263698.263739
24. Alfadel M., Aljasser K., Alshayeb M. Empirical study of the relationship between design patterns and code smells. *PLOS One*. 2020;15(4):e0231731. https://doi.org/10.1371/journal.pone.0231731
25. Hicks M., Moore J.T., Nettles S. Dynamic software updating. *ACM SIGPLAN Notices*. 2001;36(5):13-23. https://doi.org/10.1145/378795.378798
26. Neamtiu I., et al. Practical dynamic software updating for C. *ACM SIGPLAN Notices*. 2006;41(6):72-83. https://doi.org/10.1145/1133981.1133991
27. Hayden C.M., et al. Kitsune: Efficient, general-purpose dynamic software updating for C. *ACM SIGPLAN Notices*. 2012;47(10):249-264. https://doi.org/10.1145/2398857.2384635

Поступила 28.03.2025; одобрена после рецензирования 14.05.2025; принята к публикации 08.06.2025.
Submitted 28.03.2025; approved after reviewing 14.05.2025; accepted for publication 08.06.2025.

Об авторе:

Шумихин Александр Сергеевич, старший научный сотрудник, ФГБУ «Всероссийский научно-исследовательский геологический нефтяной институт» (105118, Российская Федерация, г. Москва, Шоссе Энтузиастов, д. 36), **ORCID: https://orcid.org/0009-0008-2783-3594**, shmkhn@geosys.ru

Автор прочитал и одобрил окончательный вариант рукописи.

About the author:

Aleksandr S. Shumikhin, Senior Researcher, All-Russian Research Geological Oil Institute (36 Sh. Entuziastov, Moscow 105118, Russian Federation), **ORCID: https://orcid.org/0009-0008-2783-3594**, shmkhn@geosys.ru

The author has read and approved the final manuscript.