DOI 10.37882/2223-2966.2025.06.28

АНАЛИЗ СПОСОБОВ ИНТЕГРАЦИИ ТЕХНОЛОГИЧЕСКИХ СРЕДСТВ ПРИ ТЕСТИРОВАНИИ И ЭКСПЛУАТАЦИИ СЛОЖНЫХ ИНФОРМАЦИОННЫХ СИСТЕМ

ANALYSIS OF METHODS FOR INTEGRATING TECHNOLOGICAL TOOLS IN THE TESTING AND OPERATION OF COMPLEX INFORMATION SYSTEMS

S. Kachalov
A. Zavjalov

Summary. The paper considers the problem of integrating technological tools for debugging, testing, evaluation, monitoring, and various other tasks into complex information systems. The complexity of integration is due to the presence of heterogeneous components, time constraints in the scenarios implemented by the system, requirements for reliability, security, and others. The more complex the system, the less development needs are met by universal tools. However, adding special instructions to the code can negatively affect the fulfillment of requirements and distort the operating modes of the system. Therefore, the choice of a method for integrating such infrastructure into existing code becomes a non-trivial task. The aim of the article is to generalize the experience of using various debugging integration techniques, to identify the most effective methods, and to evaluate their positive and negative effects. As a result, the authors propose recommendations for the use of various methods and minimizing their impact on the system.

Keywords: debugging, program behavior analysis, integration, C++, software architecture, system programming, code analysis, implementation tactics.

Качалов Сергей Константинович
Старший преподаватель, МИРЭА — Российский технологический университет
kachalov@mirea.ru
Завьялов Антон Владимирович
Кандидат технических наук, МИРЭА — Российский технологический университет
zavjalov@mirea.ru

Аннотация. В данной статье рассматривается задача интеграции в сложные информационные системы технологических средств, предназначенных для отладки, тестирования, оценки, мониторинга и различных других задач. Сложность интеграции обусловлена наличием разнородных компонентов, временных ограничений в реализуемых системой сценариях, требований к надежности, безопасности и других. Чем сложнее система, тем меньше потребностей разработки помогают закрывать универсальные инструменты. Однако добавление в код специальных инструкций может негативно сказаться на выполнении требований и искажать режимы функционирования системы. В связи с этим, выбор способа интеграции такой инфраструктуры в существующий код становится нетривиальной задачей. Цель статьи — обобщить опыт использования различных приемов интеграции средств отладки, определить наиболее эффективные способы и оценить их положительные и отрицательные воздействия на систему. В результате авторами предложены рекомендации по применению различных способов интеграции и минимизации их воздействия на различные характеристики системы.

Ключевые слова: отладка, анализ поведения программ, интеграция, C++, программная архитектура, системное программирование, анализ кода, тактики реализации.

Введение

Разработка сложных информационных систем, как правило, не позволяет ограничиваться использованием универсальных средств отладки, тестирования и оценки полученных характеристик. Наличие разнородных компонентов, распределенной структуры, сложной логики, параллельного исполнения, синхронизации и другие характерные для них черты требуют дополнительных технологических средств. Это требует как автономных инструментальных средств, так и встраивания элементов такой технологии в целевой, основной программный код комплекса.

В то же время добавление в код дополнительных конструкций может приводить как к искажению временных параметров работы комплекса, так и снижению надежности, безопасности и многих других характеристик. Зачастую при добавлении в код дополнительных инструкций, предназначенных для тестирования, мониторинга, журналирования, разработчик использует те решения, которые ему наиболее знакомы и понятны. Однако необходимость соблюдения жестких требований к качеству системы все-таки приводит его к необходимости анализа и осознанного выбора способа интеграции подобной инфраструктуры в существующий программный код.

Приемов для интеграции средств отладки в программный код существует достаточно много, поэтому статья посвящена попытке обобщить опыт их использования, определить основные, наиболее распространенные и эффективные способы, и выработать универсальные рекомендации по их применению при разработке сложных информационных систем.

Проблемами отладки и тестирования также занимались такие зарубежные авторы, как Гради Буч в книге «Объектно-ориентированный анализ и проектирование с примерами приложений» [1], а также Мартин Фаулер и Кент Бек в книге «Экстремальное программирование: планирование» [2].

В рамках статьи авторы поставили перед собой следующие основные задачи:

1) перечислить основные существующие тактики интеграции технологического кода в основной программный код;
2) исследовать положительные и отрицательные аспекты каждой из тактик интеграции;
3) выявить проблемы, возникающие при интеграции технологического кода в основной программный код;
4) сформулировать решения, минимизирующие нежелательное влияние технологического кода на различные характеристики системы.

Способы внедрения технологического кода в основной код

Для интеграции различного технологического кода в основной программный код наиболее часто используют следующие основные способы [3]:

1. Добавление служебных вставок напрямую в существующий код.
2. Перегрузка имеющихся методов или их замена на аналогичные с дополнительными параметрами.
3. Создание классов-обёрток над имеющимися в программе основными классами с помощью наследования, полиморфизма и инкапсуляции.
4. Использование директив препроцессора, реализующих обращение к дополнительному служебному коду.

Каждый из приведённых методов имеет свои достоинства и недостатки в рамках решения различных задач, что влияет на целесообразность их применения в каждом конкретном случае.

Добавление служебных вставок напрямую в существующий код

Вставка в основной программный код служебных конструкций — наиболее очевидный и широко распространенный способ — может выполняться, например, для логирования состояния отслеживаемых объектов, вызова специальных интерфейсов управления внутренним состоянием, логирования этапов работы программы или конкретных функций и множества других целей.

На рисунке 1 на примере функции переноса сообщений между двумя очередями показан пример реализаций подобных отладочных вставок.

```
bool queueTransfer(Que q1, Que q2)
{
    QueMsg tmp;
    debugToolLog("start");
    debugToolCall(q1);
    debugToolCall(q2);
    while (q1.Read(&tmp))
    {
        if(!q2.Write(tmp))
        {
            debugToolLogError("write");
            break;
        }
    }
    debugToolLog("end");
}
```

Рис. 1. Функция со встроенными отладочными вставками

Создание классов-обёрток над существующими в программе классами

Объектно-ориентированные языки программирования предлагают механизмы наследования, полиморфизма и инкапсуляции [4], которые эффективно можно использовать, в том числе, для реализации необходимой инфраструктуры мониторинга и отладки. Данные методы позволяют не изменять, или минимально изменять программный код изначальной реализации при добавлении технологического кода.

На рисунках 2 и 3 приведён пример интеграции дополнительного отладочного кода для класса MyLock, который реализует обёртку для системного объекта синхронизации мьютекс (SysMutex) при помощи идиомы pImpl (англ. «pointer to Implementation» — указатель на реализацию) [2].

```
class MyLock {
public:
  MyLock(): pImpl(new SysMutex) {}
  virtual ~MyLock();
  virtual bool Lock() { pImpl→lock(); }
  virtual void Unlock() { pImpl→unlock(); }
protected:
  std::unique_ptr<SysMutex> pImpl;
};
```

Рис. 2. Реализация класса-обёртки без технологического кода

К минусам такого подхода можно отнести значительное увеличение общего объёма исходного кода программы, а также появление ещё одного промежуточного класса при наследовании, как показано на рисунке 4, что может отразиться на сложности отладки и необходи-

мости незначительно изменять основной программный код.

```
class DebugLock : public MyLock
{
public:
    DebugLock(): MyLock() {}
    ~DebugLock();
    bool Lock() override
    {
        debugToolCall1();
        pImpl→lock();
        debugToolCall2();
    }
    void Unlock() override
    {
        debugToolCall3();
        pImpl→unlock();
        debugToolCall4();
    }
};
```

Рис. 3. Реализация класса-наследника с интегрированным технологическим кодом

Перегрузка существующих методов или их замена на аналогичные методы с дополнительными аргументами

Дополнительные аргументы позволяют получить такую отладочную информацию, как имя вызвавшей функции, имя файла и номер строки, идентификатор потока и другие системные и заданные пользователем значения.

На рисунках 5 и 6 приведён пример перегрузки метода переноса содержимого одной очереди в другую с технологическим кодом, логирующим дополнительную отладочную информацию.

Использование макросов для обращения к технологическому коду

Использование макросов позволяет сохранить исходный программный интерфейс, подменяя вызов исходной функции [5]. Пример реализации подобного макроса для вызова функции queueTransfer продемонстрирован на рисунке 7.

Данный метод позволяет практически полностью визуально скрыть из основного кода добавленные технологические вставки, но из-за принципа работы препроцессора языков C и C++ он значительно понижает удобство обслуживания и отказоустойчивость основного кода. Так как в данном случае вызов функции будет заменён на необходимый служебный вызов в текстовом виде до начала работы компилятора. Это приводит к трудности поиска ошибок в интегрируемом коде, так как он поставляется в код только на этапе работы препроцессора.

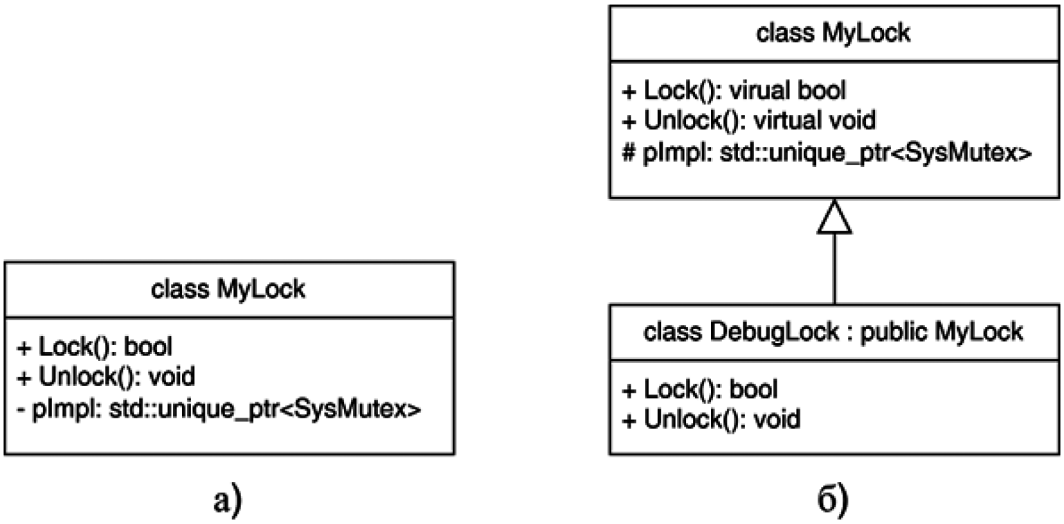

Рис. 4. Структура классов до и после применения тактики наследования

Проблемы, возникающие при внедрении технологического кода

Из возникающих при внедрении технологического кода проблем можно выделить следующие основные:

1. Ухудшение читаемости кода.
2. Увеличение сложности отладки.
3. Усложнение обслуживания кода.
4. Увеличение объема кода.
5. Дублирование кода.
6. Увеличение нагрузки на системные ресурсы.
7. Нарушение логики работы программы.
8. Нарушение архитектуры программы.

Отсутствие чёткой организации при реализации средств мониторинга и отладки в программе может зна-

```
bool queueTransfer(Que q1, Que q2)
{
    QueMsg tmp;
    while (q1.Read(&tmp))
    {
        if(!q2.Write(tmp)) break;
    }
}
```

Рис. 5. Исходная функция

```
bool queueTransfer(Que q1, Que q2,
    int debugFlag,
    std::string filename = __FILE__,
    int strNum = __LINE__)
{
    debugLog("queTransfer", debugFlag, filename, strNum);
    QueMsg tmp;
    while (q1.Read(&tmp))
    {
        if(!q2.Write(tmp)) break;
    }
}
```

Рис. 6. Перегрузка функции с логированием дополнительных аргументов

чительно ухудшить его читаемость для новых разработчиков, а также удобство его сопровождения и модификации.

Добавление служебных вставок и классов и изменение архитектуры целевого ПО для реализации механизмов мониторинга и отладки, а также возникновение из-за них связей и зависимостей между разными уровнями архитектуры программы может значительно ухудшить удобство обслуживания и модификации данной архитектуры [6].

Внедрение в исходный код программного обеспечения (ПО) дополнительных средств мониторинга и отладки сопряжено с увеличением затрат системных ресурсов, что также может приводить к значительному ухудшению производительности отлаживаемого ПО. При использовании таких стандартных и широко распространённых средств, как отладчики и динамические анализаторы кода, затраты ресурсов могут возрасти в два раза. В случае высокопроизводительного ПО и систем реального времени это может быть недопустимо и препятствовать отладке.

Наличие подробной отладочной информации в доступных конечному пользователю файлах лога и других источниках может перегружать лог и усложнять отладку. Наличие открытых служебных интерфейсов для мониторинга и отладки порождает риски, связанные с безопасностью приложения и возможностью вмешательства в его работу извне. Поэтому необходимо либо защищать доступ к подобным интерфейсам, либо исключать их из сборок программ, используемых конечным пользователем.

```
#define queueTransfer(q1, q2) \
  do { \
    debugToolCall(); \
    ::queueTransfer(q1, q2); \
  } while (0)
```

Рис. 7. Макрос для подмены вызова функции queueTransfer

Данные проблемы могут привести к ситуациям, когда внедрение технологического кода в программу будет сопровождаться настолько большими накладными расходами, что само внедрение будет нецелесообразно.

Тактики решения проблем интеграции

В таблице 1 приведены тактики решения существующих проблем для соответствующих тактик интеграции, а также описаны возможные результаты после их применения, в том числе и нерешенные проблемы.

Для решения проблем, возникающих при использовании различных тактик интеграции необходимо использовать следующие тактики решения:

1. Написание подробных комментариев.
2. Реализация возможности полного отключения отладочной конфигурации.
3. Изоляция внедряемого технологического кода.
4. Использование во внедряемом коде неблокирующих вызовов.
5. Использование встраиваемых (inline) функций.
6. Использование идиомы plmpl для реализации обёртки над объектами в основном коде, а также использование виртуальных методов в реализации классов основного кода.

Написание подробных комментариев в местах внедрения технологического кода является важной тактикой решения проблем внедрения, так как позволяет снизить ущерб для читаемости кода и простоты его последующего сопровождения другими разработчиками. Поэтому данную тактику решения следует применять при использовании любой из тактик интеграции [7].

То же относится и к реализации возможности полного отключения технологического кода, так как возможность исключить влияние данных вспомогательных инструкций позволит быть уверенным, что исходная конфигурация никак не пострадала в ходе интеграции и ее поведение никак не изменится.

Изоляция внедряемого технологического кода на уровне архитектуры, функций и файлов позволит в меньшей степени снижать общую читаемость кода и удобство его обслуживания. Данную тактику стоит использовать, когда объём внедряемого кода сравним или превосходит объём основного.

Таблица 1.

Тактики решения проблем интеграции

Тактика интеграции	Проблемы	Тактики решения	Результаты
Макросы	— ухудшение читаемости кода сложность отладки — неприменимость при работе с классами — увеличение нагрузки на системные ресурсы — нарушение логики работы программы	— изоляция макросов в отдельном файле — использование неблокирующих вызовов в отладке	— возможны ошибки препроцессора при замене фрагментов исходного текста — сложность отладки — читаемость кода не снижается
Вставки в код	— ухудшение читаемости кода — увеличение нагрузки на системные ресурсы — нарушение логики работы программы	— изоляция отладочных вызовов в отдельных функциях — использование встраиваемых (inline) функций — использование неблокирующих вызовов	— простота интеграции — снижение накладных расходов
Наследование и инкапсуляция	— ухудшение читаемости кода — усложнение обслуживания кода — увеличение объема кода — увеличение нагрузки на системные ресурсы — нарушение логики работы программы — нарушение архитектуры программы	— изоляция отладочной инфраструктуры — использование неблокирующих вызовов — использование идиомы pImpl для создания классов-обёрток — использование виртуальных функций	— сложность реализации — максимальные возможности интеграции — большой объем и дублирование кода — изменения в программных интерфейсах
Перегрузка и переопределение функций	— ухудшение читаемости кода — дублирование кода — увеличение нагрузки на системные ресурсы — нарушение логики работы программы — нарушение архитектуры программы	— использование неблокирующих вызовов	— опасность некорректного переопределения методов — читаемость кода не снижается
Общие рекомендации		— подробные комментарии — возможность полного отключения	— простота сопровождения кода — простота переключения в исходную конфигурацию — отсутствие вреда для исходной конфигурации

При реализации различных технологических вставок следует использовать только неблокирующие вызовы. Это позволит снизить влияние внедряемого кода на логику программы и позволит избежать зависания и некорректной работы программы из-за возникших блокировок в технологическом коде.

Ещё одной тактикой решения является использование механизма встраивания (inline) [5]. При комбинации данной тактики с тактикой изоляции внедряемого технологического кода получится избавиться от лишнего вызова функции и упростить компилятору задачу оптимизации встраиваемого кода, так как вместо вызова в код будет подставлено сразу тело функции. Это позволит немного повысить производительность выбранной тактики интеграции и снизить нагрузку на ресурсы системы. Данную тактику следует применять тогда, когда весь технологический код можно изолировать по отдельным функциям, которые будут встраиваться в основной код.

Последней тактикой решения является проектирование необходимой инфраструктуры для добавления технологического кода ещё при разработке самой программы. К этому можно отнести использование идиомы pImpl и виртуальных методов для создания классов-обёрток над необходимыми в исследовании классами, что кроме упрощения интеграции различных технологических вставок также позволит значительно проще реализовывать кроссплатформенность приложения, а также повысит его модифицируемость. Данную тактику следует применять ещё на ранних стадиях реализации.

Заключение

При реализации тактик интеграции технологического кода в целях добавления механизмов мониторинга, оценки, тестирования информационных систем и их компонентов разработчики сталкиваются со следующими проблемами:

1. Ухудшение читаемости кода.
2. Увеличение сложности отладки.
3. Усложнение обслуживания кода.
4. Увеличение объема кода.
5. Дублирование кода.
6. Увеличение нагрузки на системные ресурсы.
7. Нарушение логики работы программы.
8. Нарушение архитектуры программы.

Для минимизации их нежелательного влияния на характеристики системы чаще всего применяют следующие тактики:

1. Написание подробных комментариев, реализацию возможности полного отключения отладочной конфигурации и использование во внедряемом технологическом коде неблокирующих вызовов стоит делать при использовании любой из тактик интеграции;
2. Изоляция внедряемого технологического кода должна применяться в тех случаях, когда объём внедряемого кода сравним или превосходит объём основного;
3. Встраиваемые (inline) функции необходимо использовать при возможности изолирования технологического кода по отдельным функциям, встраиваемым в основной код;
4. Использование идиомы pImpl и виртуальных методов для реализации обёртки над объектами в основном коде допустимо применять на этапе разработки основного кода для избежания значительных накладных расходов при попытке изменения основного кода уже законченной программы.

Применение этих тактик позволяет реализовать необходимую функциональность отладки и анализа с минимальным влиянием на производительность, безопасность программы, читаемость исходного кода, а также другие характеристики программной архитектуры. В итоге это позволяет упростить разработку и обслуживание сложных информационных систем, в компоненты которых были внедрены необходимые технологические средства.

ЛИТЕРАТУРА

1. Гради Буч, Роберт А. Максимчук, Майкл У. Энгл, Бобби Дж. Янг, Джим Коналлен, Келли А. Хьюстон Объектно-ориентированный анализ и проектирование с примерами приложений. — 3-е изд. — Москва.: Вильямс, 2010. — 720 с. — ISBN 978-5-8459-1401-9.
2. Бек Кент Экстремальное программирование. Разработка через тестирование TDD — СПб.: Питер, 2024. — 224 с. — ISBN 978-5-4461-1439-9.
3. Качалов С.К., Завьялов А.В. Методы интеграции средств мониторинга и отладки в сложную многоуровневую архитектуру // Перспективы науки. — 2023. — № 7(166). — С. 42–44. — EDN KURICG.
4. Лафоре Р. Объектно-ориентированное программирование в C++. — 4-е изд. — СПб.: Питер, 2016. — 928 с. — ISBN 978-5-4461-0927-2.
5. Страуструп Б. Язык программирования C++. — 4-е изд. — М.: Бином, 2022. — 1216 с. — ISBN 978-5-6045724-6-7.
6. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. — СПб.: Питер, 2022. — 352 с. — ISBN 978-5-4461-0772-8.
7. Мартин Р. Чистый код: создание, анализ и рефакторинг. — СПб.: Питер, 2022. — 464 с. — ISBN 978-5-4461-0960-9. — (Серия «Библиотека программиста»).

Журнал «Современная наука: актуальные проблемы теории и практики»