Software Portability: Forty Years Later

Alexey Khoroshilov
Institute for System Programming
Russian Academy of Sciences
email: khoroshilov@ispras.ru

Denis Silakov
Institute for System Programming
Russian Academy of Sciences
email: silakov@ispras.ru

Abstract (English)

The software portability problem is a well-known problem in software engineering, and there are many approaches to fix it. Nevertheless, when the portability question is not considered in time and unpleasant consequences happen as a result, these situations perpetually occur.

This report draws attention to this problem. We hope to persuade software developers and their customers to more carefully consider the portability requirement at the initial stage of a project. This report discusses several examples of incorrect solutions taken. We consider the most popular approaches to fix the portability problem and discuss their virtues and shortcomings.

***Keywords**: Portability; interoperability.*

Проблема переносимости приложений – сорок лет спустя

Денис Силаков
Институт системного программирования
Российская Академия Наук
email: silakov@ispras.ru

Алексей Хорошилов
Институт системного программирования
Российская Академия Наук
email: khoroshilov@ispras.ru

Abstract (Russian)

Проблема переносимости приложений между программно-аппаратными платформами имеет длинную историю. За это время появилось множество подходов к её решению. Тем не менее, постоянно возникают ситуации, когда выясняется, что данному вопросу вовремя не было уделено должное внимание и это привело к неприятным последствиям.

Настоящий доклад ставит своей целью привлечь внимание к вопросу переносимости программного обеспечения и добиться, чтобы и заказчики, и разработчики ПО более серьезно подходили к рассмотрению требования переносимости на начальных этапах проекта. Мы представим примеры последствий недальновидных решений этого вопроса, рассмотрим наиболее распространенные подходы к обеспечению переносимости ПО, а также обсудим их достоинства и недостатки.

***Keywords**: Переносимость; мобильность; интероперабельность.*

1. Введение

Проблема переносимости приложений между различными программно-аппаратными платформами ненамного моложе собственно компьютерных программ. Еще в конце шестидесятых годов озабоченность некоторых сотрудников AT&T Labs проблемой переносимости ОС UNIX на новые аппаратные платформы привела к созданию языка Си; однако темпы развития компьютерной индустрии таковы, что проблемы сорокалетней давности кажутся достаточно простыми и решаемыми, по сравнению с тем, что мы имеем сегодня.

Стремительное развитие связанных с компьютерами отраслей приводит к постоянному появлению новых программно-аппаратных платформ, информационных систем, и т.п., в то время как устаревшие комплексы уходят в небытие.

Производители ПО, как правило, заинтересованы в быстром переносе своих продуктов на новые системы, чтобы захватить

соответствующую долю рынка. Если приложение изначально проектировалось с оглядкой на возможность портирования, то этот процесс может оказаться существенно дешевле создания нового продукта. Будет проще и конечным пользователям, которые в новой системе увидят то же самое приложение, с которым работали раньше, что также способствует популярности продукта.

Исчезновение же с рынка тех или иных платформ приводит к появлению унаследованного ПО — программных продуктов, необходимых для функционирования той или иной организации, но требующих для работы устаревшей программно-аппаратной платформы. В случае выхода из строя аппаратного обеспечения может оказаться, что найти ему замену очень сложно, дорого, а иногда и попросту невозможно, так как устаревшая ОС не работает на современном оборудовании ввиду отсутствия необходимых драйверов. Для многих предприятий задача переноса таких приложений в более современное окружение является крайне актуальной [1,2].

2. Примеры из современности

Несмотря на то, что о проблеме переносимости известно достаточно давно и ее решению посвящено множество работ и исследований, постоянно возникают ситуации, когда выясняется, что данному вопросу вовремя не было уделено должное внимание и это привело к неприятным последствиям. Рассмотрим несколько реальных примеров таких ситуаций.

Некоторая американская компания разрабатывала ряд программных продуктов для платформы Microsoft Windows. Не так давно руководство компании осознало, что активно растущий рынок пользователей MacOS в США также представляет для них лакомый кусочек, в особенности, учитывая бо́льшую заинтересованность пользователей в их продуктах.

Руководство инициировало анализ возможности выпуска продуктов для MacOS, который выявил следующую картину. Ведущий разработчик одной из линеек продуктов, имевший опыт разработки переносимых приложений, по собственной инициативе организовал внутреннюю архитектуру этих приложений с четким выделением платформенно-зависимой функциональности. В других приложениях компании собственная функциональность оказалась тесно завязана на особенности целевой операционной системы. В результате затраты по доработке, тестированию и выводу на рынок первого продукта были признаны окупаемыми в краткосрочной перспективе. А вот экономическая целесообразность портирования других продуктов была поставлена под сомнение, ввиду необходимости практически полного переписывание их кода. Таким образом, невнимание к проблеме переносимости обернулось потерей части прибыли и проблемам с захватом позиций на перспективном рынке.

Другой пример недальновидного подхода к этому вопросу проявился в ходе организации проекта по разработке пакета свободного программного обеспечения (СПО) для образовательных учреждений России. Практически все программное обеспечение, которое разрабатывалось по заказу Министерства образования РФ в последние годы было предназначено для работы исключительно на платформе Microsoft Windows. Поэтому при внедрении пакета СПО на основе операционной системы Linux бо́льшая часть разработанных ранее образовательных программ оказалась недоступна, и только часть из них удавалось запустить с помощью эмулятора wine [3].

Схожие проблемы возникали и в стане коммерческих заказчиков. Известно несколько случаев, когда при разработке интернет-сервисов заказчик ограничивался требованием совместимости с браузером Internet Explorer, а через некоторое время под давлением клиентов был вынужден дорабатывать ПО для поддержки Mozilla Firefox.

Приведенные примеры демонстрируют, как невнимание к проблеме переносимости может привести к различным негативным техническим, экономическим и политическим последствиям:

- проблемам с поддержкой ПО в долгосрочной перспективе;
- сокращению доступных рынков и недополучению прибыли;
- попаданию в зависимость от одного поставщика.

Как мы уже отметили, проблема известна давно, и за время эволюции программного обеспечения появилось достаточно много подходов к созданию приложений, переносимых между различными системами; рассмотрим те, что представляются наиболее популярными.

3. Бинарная совместимость

Перенос приложения на новую программно-аппаратную платформу может пройти безболезненно для разработчиков, если старая и

новая системы совместимы на бинарном уровне, то есть новая система позволяет использовать старые двоичные файлы приложения без каких-либо изменений.

Для реализации такой возможности целевая платформа должна поддерживать соответствующий формат исполняемых файлов и файлов разделяемых библиотек, а также обладать совместимостью на уровне двоичного интерфейса приложений (Application Binary Interface, ABI).

Одной из важных составляющих бинарной совместимости является набор функций, предоставляемых системой. Полной идентичности таких наборов ожидать трудно, однако во многих случаях пересечение множеств предоставляемых функций является достаточно большим. Многие производители операционных систем в настоящее время заботятся об обратной совместимости своих продуктов на бинарном уровне – гарантируется, что приложение, работающее в некоторой версии ОС, будет работать без перекомпиляции в более новых версиях системы.

Свойство бинарной совместимости жестко связано с аппаратной частью платформы — в силу технических причин в общем случае сложно достичь совместимости систем, работающих на различных архитектурах. В некоторых конкретных ситуациях операционные системы предоставляют возможность запуска приложений, написанных для той же системы, но на другой платформе — так, операционная система MacOS X, работающая на компьютерах Apple с процессорами Intel, использует динамический транслятор Rosetta для выполнения программ, предназначенных для машин с процессорами PowerPC [4]. Однако пользоваться такой возможностью следует с осторожностью. Во многих случаях совместимость обеспечивается за счет некоторого дополнительного слоя совместимости между системой и приложением, который может и не гарантировать полной совместимости — так, уже упомянутая Rosetta позволяет исполнять код для процессоров G3, G4 и AltiVec, но не для G5. Кроме того, дополнительный компонент системы является дополнительным потенциальным источником ошибок, а использование посредника в общем случае снижает производительность.

4. Переносимость исходного кода

К сожалению, многие существующие платформы не совместимы на бинарном уровне и не способны загружать исполняемые файлы друг друга (хотя бы потому, что используют различные форматы файлов, соглашения о вызове функций и т.п.). Альтернативой использованию одних и тех же бинарных файлов явилось использование одного и того же исходного кода для сборки приложения на различных системах.

В те времена, когда программы писались на ассемблере конкретной аппаратной платформы, добиться компиляции приложения на системе с другой архитектурой было практически нереально. Существенной подвижкой в этом направлении стало создание высокоуровневых языков программирования, не привязанных к конкретной архитектуре и позволяющих использовать одни и те же конструкции на различных системах.

Для обеспечения такой возможности целевые системы должны предоставлять компиляторы для используемого языка программирования. Естественно, что компиляторы для различных систем создаются различными производителями и могут достаточно сильно отличаться друг от друга.

Частично эту проблему решают международные стандарты, существующие для многих языков программирования. Однако далеко не все компиляторы поддерживают стандарты в полном объеме (хотя, как правило, не поддерживаются некоторые специфические конструкции языка, в плане же предоставления библиотечных функций ситуация гораздо лучше). Другой проблемой является относительная узость стандартов — несмотря на наличие во многих из них перечня функций, которые должны предоставляться любой удовлетворяющей стандарту средой разработки, эти перечни не описывают существенную часть функциональности, которая могла бы быть полезна при создании приложений - например, функции графического интерфейса.

Помимо стандартов языков программирования, существуют стандарты, описывающие интерфейс прикладных программ (API, Application Programming Interface) — например, POSIX. Однако такие стандарты также, достаточно узки, и являются недостаточными для написания большинства приложений.

Для реализации функциональности, не охваченной никакими стандартами, полезными могут оказаться кросс-платформенные библиотеки, предоставляющие необходимый набор функций — например, существует ряд библиотек для создания графического интерфейса, которые можно использовать как в Windows, так и в Linux.

5. Интерпретируемые языки

Еще одним шагом по обеспечению переносимости приложений явилось использование интерпретируемых языков – т.е. языков, использование которых не подразумевает создания исполняемых файлов в формате целевой операционной системы. Вместо этого интерпретатор последовательно считывает и выполняет инструкции непосредственно из текста программы.

Однако прямолинейная интерпретация достаточно неэффективна – у интерпретатора практически нет возможностей для оптимизации кода. Для повышения эффективности во многих языках (Java, Perl, семейство .NET) исходный код сначала транслируется в некоторое промежуточное представление (байт-код), который уже подается на вход интерпретатору.

Еще большей производительности удается достигнуть при использовании компиляции “на лету” (Just In Time compilation), когда байт-код транслируется в машинный код во время работы программы. Однако разработчикам JIT-компиляторов также приходится идти на компромисс между временем, уходящим на оптимизацию, и эффективностью получаемого кода, в результате чего производительность приложений может уступать коду, создаваемому “обычными” компиляторами. Кроме того, использование данной технологии увеличивает потребление памяти приложением, поскольку оттранслированный код также хранится в оперативной памяти. Также следует иметь в виду, что эффективность реализации интерпретаторов, также как и JIT-компиляторов, на различных платформах может отличаться.

6. Эмуляция ABI других систем

Говоря о бинарной переносимости, мы уже упомянули, что в ряде случаев операционная система может обеспечивать бинарную совместимость с другой системой за счет дополнительного слоя совместимости. Возможности самих систем в этом плане достаточно ограничены — как правило, такой слой совместимости вводится для поддержки приложений той же самой системы, но для другой архитектуры.

В то же время существует много разработок, позволяющих загружать файлы других операционных систем путем использования транслятора, способного загружать файлы требуемого формата, преобразуя вызовы функций, осуществляемые внутри файла, в соответствующие вызовы текущей ОС (фактически, такой транслятор реализует ABI старой системы в новой системе).

В качестве примера можно привести wine [3], предназначенный для запуска Windows-приложений в Linux, а также cygwin [5], обеспечивающий переносимость в обратную сторону. При этом, например wine достаточно легко использовать как часть приложения, не полагаясь на его доступность в целевой системе.

Недостатком использования такого рода эмуляторов является потенциальная неполнота реализации интерфейса, необходимого приложению. Так, разработчики того же wine ориентируются на публично доступную информацию об API Windows (например, стандарт ECMA-234); так что если приложение использует какие-то недокументированные возможности этой ОС, то попытка его запуска в wine может оказаться неудачной.

7. Веб-сервисы и сервисно-ориентированная архитектура

Необходимость создания сложных распределенных систем привела к созданию парадигм, подразумевающих разбиение системы на отдельные компоненты, взаимодействующие друг с другом по строго определенным протоколам. При таком подходе каждый компонент может разрабатываться независимо, а процесс его переноса на другую платформу никак не затрагивает других частей системы.

К одной из первых попыток описания механизма взаимодействия компонентов распределенной системы можно отнести спецификацию CORBA (Common Object Request Broker Architecture), разработанную консорциумом OMG [6].

Однако CORBA в силу ряда причин не снискала большой популярности, и в настоящее время гораздо больший интерес проявляется к веб-сервисам, использующим протоколы обмена сообщениями на базе XML. При проектировании сложных распределенных программных комплексов используется парадигма сервисно-ориентированной архитектуры (Service-oriented architecture, SOA [7]); при этом программные комплексы часто реализуются как набор веб-сервисов.

8. Виртуализация

В ряде случаев для запуска приложений необходимо аппаратное обеспечение, которое в силу ряда причин не доступно человеку, запускающему программу. В таком случае на

помощь приходят виртуальные машины, эмулирующие работу некоторой аппаратной платформы. На такой виртуальной машине устанавливается операционная система и другое окружение, необходимое приложению, а само приложение запускается уже в родной для него среде.

Возможность запуска приложения на виртуальной машине зависит, в основном, от возможностей самой машины, нежели от разработчиков приложения. Тем не менее, программы, работающие с аппаратурой напрямую, могут встретить определенные трудности, поскольку им будет предоставлен доступ к устройствам виртуальной машины, а не непосредственно компьютера. Такая особенность ограничивает, например, возможность работы с графическими ускорителями изнутри виртуальных машин.

Использование виртуальной машины достаточно ресурсоемко — ведь помимо собственно приложения, ресурсы компьютера потребляются самой машиной, а также работающими внутри нее программами, необходимыми для функционирования приложения (например, операционной системой). Поэтому выигрыш в производительности достигается, как правило, только в случае запуска машин, эмулирующих достаточно маломощные платформы на более производительных системах.

9. Заключение

Если невнимание к проблеме переносимости приводит к негативным последствиям и существует множество путей по ее решению, то возникает вопрос: так почему же ИТ индустрия не переориентируется на разработку переносимого ПО. Несложно догадаться, что разработка переносимого ПО имеет свои недостатки.

Среди рассмотренных видов переносимости приложений очень привлекательным с точки зрения разработчиков является перенос непосредственно бинарных файлов на новую систему, позволяющий при относительно небольших затратах (в основном уходящих на тестирование) получить на новой системе приложение, имеющее всю необходимую функциональность. При этом потери в производительности если и возникают, то совсем небольшие.

Однако для любой ОС число платформ, совместимых с ней на бинарном уровне, достаточно невелико. Использование эмуляторов может расширить их круг, однако эмулятор — дополнительный потенциальный источник ошибок, который при этом может и не предоставлять всех необходимых функций.

Потенциально больший охват дает переносимость исходного кода. Сложность портирования в этом случае может варьироваться в зависимости от того, насколько такая возможность учитывалась при разработке приложения; полезной с этой точки зрения является ориентация на различные интерфейсные стандарты, регламентирующие взаимодействие приложения с окружающей средой. Но существующие стандарты охватывают достаточно небольшую функциональность; в ряде случаев может помочь использование кросс-платформенных библиотек, другой же альтернативой является использование интерпретируемых языков.

Спецификации таких языков не привязаны к конкретной платформе и можно полагаться на то, что интерпретаторы на разных системах поддерживают один и тот же набор функций. Среди недостатков подхода можно выделить меньшую производительность по сравнению с бинарным кодом.

Архитектура SOA затрагивает более сложную проблему организации сложных программных комплексов, предлагая строить их в виде набора достаточно изолированных компонентов, каждый из которых может работать на своей собственной платформе и в случае необходимости может быть перенесен на другую (либо заменен на альтернативную реализацию).

Использование виртуальных машин также не требует больших усилий со стороны разработчиков ПО, хотя этот способ достаточно накладен, как в смысле производительности, так и ввиду необходимости иметь лицензии на все используемые операционные системы. Применение виртуализации оправдано в тех случаях, когда перенос приложения каким-то другим способом представляется экономически неэффективным. В частности, это относится ко многим унаследованным системам, для которых портирование на новую платформу означало бы практически полное переписывание приложения.

В таблице 1 мы постарались отразить влияние различных методик, используемых для увеличения переносимости приложения, на процесс разработки, а также на характеристики получаемого продукта.

Ориентация на существующие стандарты ПО полезна в любом случае, однако не является панацеей от всех бед ввиду относительной узости (или даже отсутствию) стандартов во многих областях. Использование различных медиаторов

(библиотек, интерпретаторов, дополнительных слоев совместимости), отделенных от самого приложения, расширяет число потенциальных целевых систем. Однако разработка медиаторов собственными силами ведет к увеличению затрат на производственный процесс; использование же сторонних продуктов ставит разработчика в зависимость от других поставщиков. То же самое верно и для эмуляторов и виртуальных машин — использование готовых решений достаточно дешево (фактически, при этом необходимо лишь дополнительное тестирование в новой системе плюс соответствующие лицензионные отчисления), однако ставит в зависимость от их производителей. Риски зависимости снижаются при использовании свободных компонентов с открытым кодом, так как перспективы развития этих компонентов становятся более управляемыми.

Подводя итоги, отметим, что возможны ситуации, когда отказ от обеспечения переносимости разрабатываемого ПО оправдан; например, с достаточно большой долей вероятности такой выбор будет выгоден в краткосрочной перспективе. Однако при создании продуктов, которые планируется поддерживать в течении достаточно длительного периода, обеспечение переносимости может оказаться одним из ключевых факторов успеха. Повышение мобильности приложения в общем случае всегда связано с увеличением расходов на его разработку; однако чем раньше об этой проблеме задумаются разработчики и архитекторы, тем меньше будет стоимость переноса приложения на новую платформу. Поэтому хотелось бы подчеркнуть, что вопрос обеспечения переносимости следует рассматривать в самом начале проекта, на стадии проектирования и выбора технологий и инструментов, которые будут использованы при его реализации.

Таблица 1. Сравнительная характеристика путей достижения переносимости

	Ориентация на стандарты	Медиаторы сторонних разработчиков	Собственные медиаторы	Эмуляция	Виртуализация
Производительность	*Не влияет*	*Возможно снижение*	*Возможно снижение*	*Снижается*	*Снижается*
Удорожание разработки	*Доп. затраты из-за узости стандартов*	*Нет (при наличии готовых решений)*	*Да, зависит от числа целевых систем*	*Практически нет (при наличии готовых решений)*	*Практически нет (при наличии готовых решений)*
Увеличение сроков разработки	*Доп. затраты из-за узости стандартов*	*Незначительно*	*Да, зависит от числа целевых систем*	*Практически нет*	*Нет (при наличии готовых решений)*
Расширение рынка	*В рамках охвата стандарта*	*На системы, охваченные медиаторами*	*В соответствии с собственными возможностями*	*На системы, содержащие эмулятор*	*На системы, содержащие виртуальную машину*
Долгосрочная поддержка	*Существенно облегчается*	*Зависит от поддержки медиаторов*	*Доп. затраты на поддержку медиаторов*	*Зависит от поддержки эмуляторов*	*Зависит от поддержки виртуальных машин*

10. Литература

[1] James D. Mooney. "Bringing Portability to the Software Process". Technical Report TR 97-1, Dept. of Statistics and Computer Science, West Virginia University, Morgantown WV, 1997.

[2] Ian Sommerville. *Software Engineering*, *8th Edition*, Addison Wesley, 2006.

[3] Wine - Open Source implementation of the Windows API. http://www.winehq.org/

[4] Apple Rosetta. http://www.apple.com/rosetta/

[5] Cygwin - Linux-like environment for Windows. http://www.cygwin.com/
[6] OMG CORBA 3.0. http://www.omg.org/technology/documents/formal/corba_2.htm
[7] Ben Margolis, Joseph Sharpe. *SOA for the Business Developer: Concepts, BPEL, and SCA*. MC Press, 2007.