

Keynote: Performance

«Что в имени тебе моём?»

Aleksey Shipilëv

aleksey@shipilev.net

@shipilev

Safe Harbor / Тихая Гавань

Everything on this and any subsequent slides may be a lie. Do not base your decisions on this talk. If you do, ask for professional help.

Всё что угодно на этом слайде, как и на всех следующих, может быть враньём. Не принимайте решений на основании этого доклада. Если всё-таки решите принять, то наймите профессионалов.

Крупно

Крупно: критерии успеха в разработке

Крупно: критерии успеха в разработке



Крупно: критерии успеха в разработке

Крупно: критерии успеха в разработке

2. Корректность реализации

Крупно: критерии успеха в разработке

- 2. Корректность реализации
- 3. Безопасность

Крупно: критерии успеха в разработке

1. Соответствие получившегося желаниям пользователя
2. Корректность реализации
3. Безопасность

Крупно: критерии успеха в разработке

1. Соответствие получившегося желаниям пользователя
2. Корректность реализации
3. Безопасность
4. Быстрота и удобство разработки

Крупно: критерии успеха в разработке

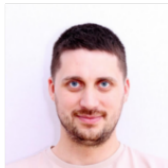
1. Соответствие получившегося желаниям пользователя
2. Корректность реализации
3. Безопасность
4. Быстрота и удобство разработки
5. Производительность

Крупно: критерии успеха в разработке

1. Соответствие получившегося желаниям пользователя
2. Корректность реализации
3. Безопасность
4. Быстрота и удобство разработки
5. Производительность

Чаще всего производительность даже близко не главный приоритет.
А чаще всего её даже в критериях успеха нет.

Крупно: их разыскивает милиция



Siemargl 30 августа 2016 в 18:37 #



То есть из четырех экспертов никто не оценил Java, как быструю. Скорее, как достаточную и удовлетворяющую. Это вполне себе показательно.

Крупно: про критерии

«Корректная программа»:

«Быстрая программа»:



Крупно: про критерии

«Корректная программа»:

1. не видно бесящих
пользователя ошибок

«Быстрая программа»:

1. не видно бесящих
пользователя тормозов



Крупно: про критерии

«Корректная программа»:

1. не видно бесящих
пользователя ошибок
2. в критерии успеха
вложились



«Быстрая программа»:

1. не видно бесящих
пользователя тормозов
2. в критерии успеха
вложились

Крупно: про критерии

«Корректная программа»:

1. не видно бесящих
пользователя ошибок
2. в критерии успеха
вложились
3. количество багов
известно



«Быстрая программа»:

1. не видно бесящих
пользователя тормозов
2. в критерии успеха
вложились
3. перформансные
проблемы известны

Крупно: про критерии

«Корректная программа»:

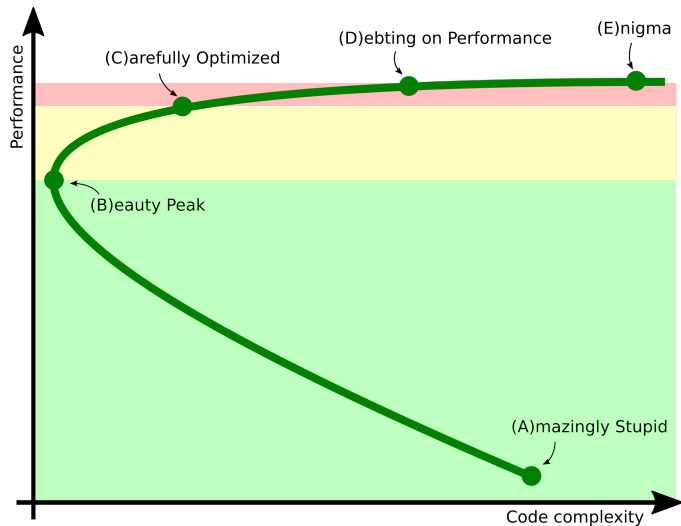
1. не видно бесящих пользователя ошибок
2. в критерии успеха вложились
3. количество багов известно
4. пути обхода и альтернативы известны



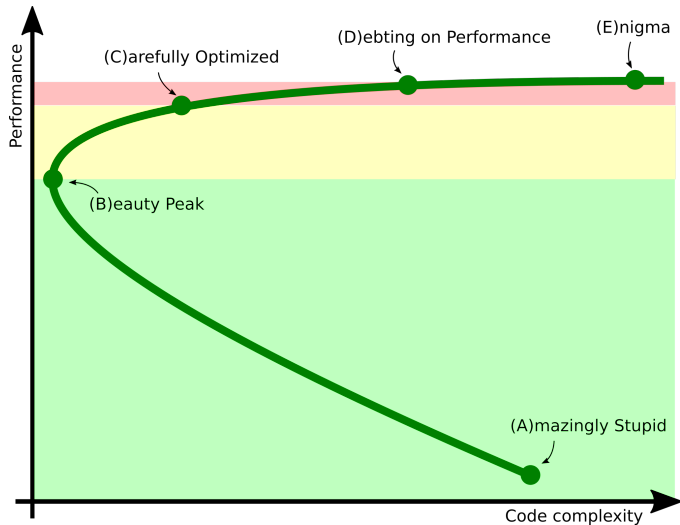
«Быстрая программа»:

1. не видно бесящих пользователя тормозов
2. в критерии успеха вложились
3. перформансные проблемы известны
4. пути обхода и альтернативы известны

Крупно: Кривая им. Ш:



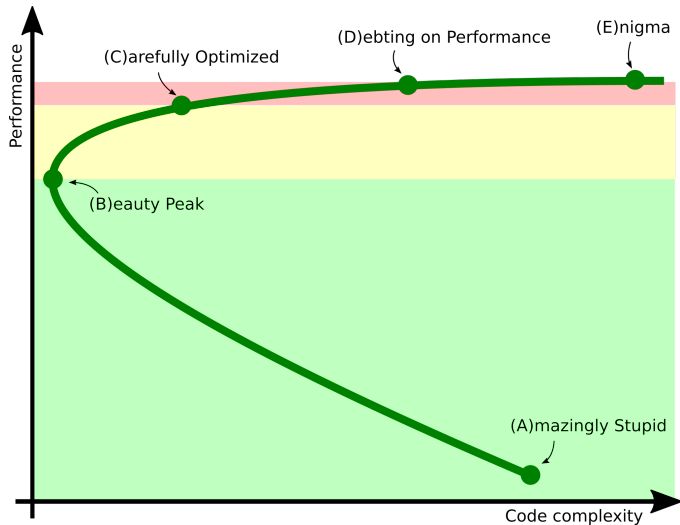
Крупно: Кривая им. Ш:



Вы в зелёной зоне:

Берёте профайлер и
переписываете куски,
которые написаны
очевидно ужасно

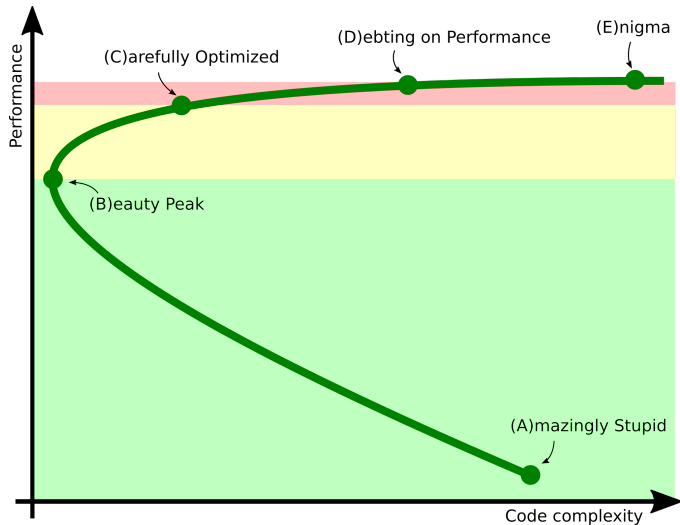
Крупно: Кривая им. Ш:



Вы в жёлтой зоне:

Берёте аккуратный
профайлер, пишете
таргет-бенчмарки,
аккуратно закручиваете
гайки

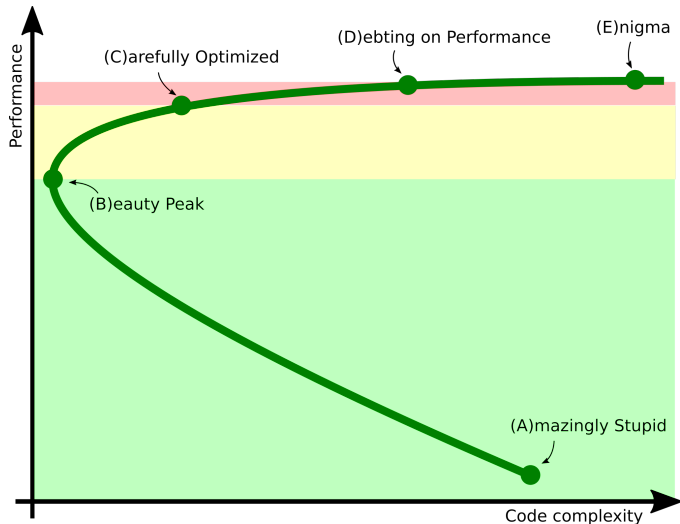
Крупно: Кривая им. Ш:



Вы в красной зоне:

Вас съел огр.

Крупно: Кривая им. Ш:



Вы в красной зоне:

Идёте на Joker<?>, пытаете разработчиков продуктов, как писать код, повторяющий кривизну нижних слоёв

Зелёная зона

Зелёная зона: мотивационная карточка

Зелёная зона:
борьба с говнокодом заусенцами
грубой силой

- *Резать к чертовой матери, не дожидаясь перитонитов!*
- *Ты права, моя дорогая, с этим отростком пора кончать!*

Зелёная зона: профилирование и диагностика



Ментальная ловушка:

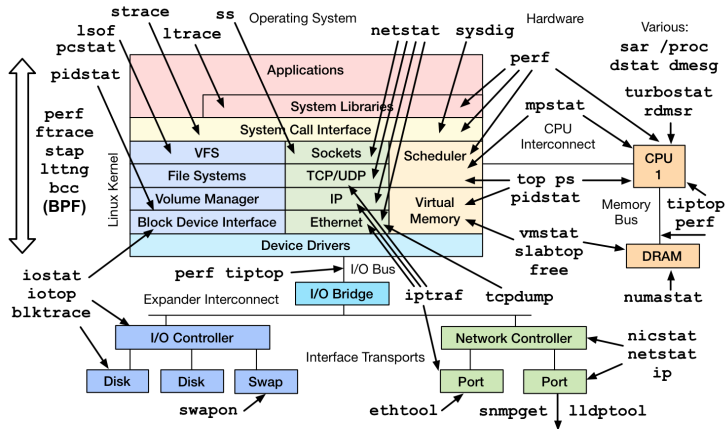
«Профилировать нужно или нормально, или вообще никак»

На самом деле:

В зелёной зоне точность диагностики влияет очень мало! Вам нужно определить, какую часть говнокода выгодно перепилить в первую очередь.

Зелёная зона: диагностика

Linux Performance Observability Tools



<http://www.brendangregg.com/linuxperf.html> 2016

Зелёная зона: диагностика

Не бойтесь смотреть на ваше приложение!

Даже крупноклеточное понимание, какого ресурса не хватает, лучше, чем никакого понимания. Практически сразу показывает, куда копать:

- много `sys%`: копаем в сторону трединга и т.п.
- много `irq%`: копаем в сторону оффлоада прерываний и т.п.
- много `idle%`: ищем, где простаиваем
- много `iowait%`: копаем в сторону оптимизации I/O
- много `usr%`: аттачим профайлер, и...

Зелёная зона: профилирование

Наша цель:

примерно представлять, где мы проводим время

Даже самый наивный профайлер вам покажет все *ужасные* ошибки:

- хоть jstack
- хоть perf top
- хоть VisualVM
- да даже руками расставить Stopwatch-и уже хорошо
- да хоть из палок и желудей его соберите!

Зелёная зона: измерение производительности

Ментальная ловушка:

«Производительность нужно измерять или нормально, или вообще никак»

На самом деле:

В зелёной зоне улучшения в плюс-минус километр, можно и глазками увидеть. Да и какая разница, если всё равно переписываете говнокод?



Зелёная зона: мораль

Даже тривиальные нагрузочные тесты покажут вам *крупные огрехи*:

- Берёте production и публикуете ссылку в Твитторе
- Берёте staging и бешено тыкаете во всё подряд
- Берёте staging и хреначите в него Apache Bench'ем

Чем раньше в разработке вы получите перформансные данные, тем безболезненнее можно будет исправить огрехи (ещё до коммита)!

- Не надо сразу буйствовать и писать огромные сценарии
- Не надо тащить сложные нагрузочные генераторы
- Не надо пытаться обеспечить полную репрезентативность сценария

Зелёная зона: пример

Сюрприз-и-и-из:



JDK / JDK-8153229

JavacFiler.checkFileReopening drowns in exceptions after Modular Runtime Images change

Massive regression! Profiling jdk9b111 case yields this very hot branch:

```
| +- 10.080 (9%) org.openjdk.jmh.generators.annotations.APGeneratorDestinaton.newClass(java.lang.String)
| | +- 10.060 (9%) com.sun.tools.javac.processing.JavacFiler.createSourceFile(java.lang.CharSequence, javax.lang.model.element.Element[])
| | | +- 10.060 (9%) com.sun.tools.javac.processing.JavacFiler.createSourceOrClassFile(boolean, java.lang.String)
| | | +- 10.030 (9%) com.sun.tools.javac.processing.JavacFiler.checkFileReopening(javax.tools.FileObject, boolean)
| | | | +- 9.990 (8%) com.sun.tools.javac.file.JavacFileManager.isSameFile(javax.tools.FileObject, javax.tools.FileObject)
| | | | | +- 9.990 (8%) com.sun.tools.javac.file.PathFileObject.isSameFile(com.sun.tools.javac.file.PathFileObject)
| | | | | +- 9.990 (8%) java.nio.file.Files.isSameFile(java.nio.file.Path, java.nio.file.Path)
| | | | | +- 9.990 (8%) sun.nio.fs.UnixFileSystemProvider.isSameFile(java.nio.file.Path, java.nio.file.Path)
| | | | | +- 7.080 (6%) sun.nio.fs.UnixFileAttributes.get(sun.nio.fs.UnixPath, boolean)
| | | | | +- 7.080 (6%) sun.nio.fs.UnixNativeDispatcher.stat(sun.nio.fs.UnixPath, sun.nio.fs.UnixFileAttributes)
| | | | | +- 6.700 (6%) sun.nio.fs.UnixNativeDispatcher.stat0(long, sun.nio.fs.UnixFileAttributes)
| | | | | +- 2.750 (2%) sun.nio.fs.UnixException.<init>(int)
| | | | | +- 2.750 (2%) java.lang.Exception.<init>()
| | | | | +- 2.750 (2%) java.lang.Throwable.<init>()
| | | | | +- 2.740 (2%) java.lang.Throwable.fillInStackTrace()
| | | | | +- 2.740 (2%) java.lang.Throwable.fillInStackTrace(int)
```



Зелёная зона: пример

Сюрприз-и-и-из:



JDK / JDK-8153229

JavacFiler.checkFileReopening drowns in exceptions after Modular Runtime Images change

Massive regression! Profiling jdk9b111 case yields this very hot branch:

```
| +- 10.080 (9%) org.openjdk.jmh.generators.annotations.APGeneratorDestinaton.newClass(java.lang.String)
| +- 10.060 (9%) com.sun.tools.javac.processing.JavacFiler.createSourceFile(java.lang.CharSequence, javax.lang.model.element.Element[])
| +- 10.000 (8%) com.sun.tools.javac.processing.JavacFiler.createSourceFile(java.lang.CharSequence, javax.lang.model.element.Element[])
```

✓ Aleksey Shipilev added a comment - 2016-05-02 08:31

This issue seems to blow up jcteststress compilation time from 2 minutes to 8 minutes. This is a massive regression, please fix this in 9.

```
|||||| +- 2.750 (2%) sun.nio.fs.UnixException.<init>(int)
|||||| +- 2.750 (2%) java.lang.Exception.<init>()
|||||| +- 2.750 (2%) java.lang.Throwable.<init>()
|||||| +- 2.740 (2%) java.lang.Throwable.fillInStackTrace()
|||||| +- 2.740 (2%) java.lang.Throwable.fillInStackTrace(int)
```

Зелёная зона: оптимизации



Ментальная ловушка:

«Преждевременная оптимизация – корень
всего зла»

На самом деле:

Ну и какое зло в переписывании говнокода?
Использовать удобные модели данных и алгоритмы почти никогда не преждевременно.

Зелёная зона: заходы

Улучшение производительности в основном от переписывания плохого кода на хороший. Но «хорошесть» может быть и вкусовщиной, а может быть выстраданными приёмами:

- эффективнее структуры данных:

`LinkedList` → `ArrayList`

- эффективнее алгоритмы:

`ArrayList` → `HashMap`

`keySet` + `get` → `entrySet`

`bubbleSort` → `Collections.sort`

- меньше работы: что-то посчитать один раз и реиспользовать, етц

Зелёная зона: подитог

Профилирование – необходимая часть ежедневной разработки

Наблюдения:

- >95% проблем находится на первых же заходах
- >90% проблем тривиально разрешимы
- Чёткие инструкции по запуску профилировки **сильно** помогают: отлично, если есть однострочник, или однокнопочник, или АРМ
- Возьмёте девелопера за руку, и с ним один раз попрофилируете – это **уверенно** купирует боязнь базовой перформансной работы¹

¹«Нет-нет, не надо закрывать это окно, оно боится тебя больше, чем ты его»

Жёлтая зона

Жёлтая зона: мотивационная карточка

Жёлтая зона:
~~нефть в обмен на продовольствие~~
усложнение кода в обмен на
производительность

Так вот, в момент, когда в голове у клиента происходит эта реакция, из кустов появляемся мы. Татарскому было очень приятно услышать это «мы». (солнце наше Пелевин, «Generation П²»)

Жёлтая зона: профилирование и диагностика



Ментальная ловушка:

«Сейчас мы возьмём профайлер, посмотрим что где, и как начнём оптимизировать»

На самом деле:

Взросшая цена ошибки предполагает, что мы будем вносить правильные изменения. Правильные изменения требуют продвинутой диагностики, и профилировка – только одна её часть!

Жёлтая зона: что собрались оптимизировать?

Hot Spots - Method	Self Time...	Self Time	Self Time (CPU)
java.lang.Object. wait[native] ()		390,047 ... (26.7%)	0.000 ms
sun.misc.Unsafe. park[native] ()		273,216 ... (18.7%)	0.000 ms
java.net.SocketInputStream. socketRead0[native] ()		244,844 ... (16.8%)	244,844 ms
java.net.PlainSocketImpl. socketAccept[native] ()		97,558 ms (6.7%)	0.000 ms
sun.management.ThreadImpl. dumpThreads0[native] ()		97,558 ms (6.7%)	97,558 ms
java.lang.Object. hashCode[native] ()		27,312 ms (1.9%)	27,312 ms
java.lang.System. identityHashCode[native] ()		25,978 ms (1.8%)	25,978 ms
com.sun.tools.javac.code.Type. hasTag ()		21,945 ms (1.5%)	21,945 ms
com.sun.tools.javac.comp.Attr\$ResultInfo. check ()		17,751 ms (1.2%)	17,751 ms
com.sun.tools.javac.code.Types\$DescriptorCache. get ()		13,326 ms (0.9%)	13,326 ms
com.sun.tools.javac.tree.JCTree\$JCIdent. accept ()		10,730 ms (0.7%)	10,730 ms
com.sun.tools.javac.comp.Resolve\$4. argumentsAcceptable ()		9,868 ms (0.7%)	9,868 ms
com.sun.tools.javac.code.Scope. getIndex ()		7,361 ms (0.5%)	7,361 ms
com.sun.tools.javac.code.Type\$ClassType. accept ()		6,218 ms (0.4%)	6,218 ms
com.sun.tools.javac.code.Types\$18. visitClassType ()		5,807 ms (0.4%)	5,807 ms
java.lang.Object. clone[native] ()		5,435 ms (0.4%)	5,435 ms



Жёлтая зона: *modus operandi*

Теперь, оптимизируя, вы вынуждены объяснять,³
зачем вы это делаете, сколько это будет стоить,
сколько это даст.

При этом желательно:

1. Иметь на руках численные оценки приростов
2. Иметь оценки приростов **до** того, как потратить всё ресурсы
3. Иметь понимание, что это самый дешёвый способ

³хотя бы себе, а может и РМ-у

Жёлтая зона: перформансники

Это подразумевает, что есть люди, которые понимают:

1. ...верить результатам профилировки или нет
2. ...верить результатам тестов или нет
3. ...стек целиком и могут сделать изменение в том месте, где надо
4. ...как изменения в части стека распространяются на остальное
5. ...где ключевые сценарии проводят время, и что на них может повлиять
6. ...и ответам которых **верят**

Это перформансники и есть!

Оценки: Pop quiz

Представим себе приложение с двумя отдельными частями:

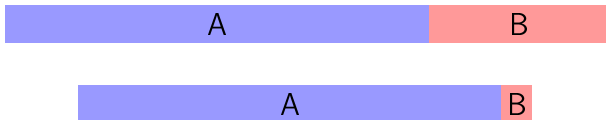
- Часть А занимает 70% времени, разгоняема в 2 раза
- Часть В занимает 30% времени, разгоняема в 6 раз
- Какую часть будем разгонять?



Оценки: Pop quiz

Представим себе приложение с двумя отдельными частями:

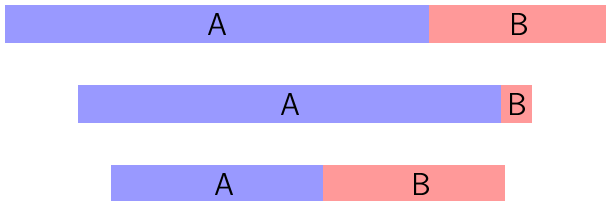
- Часть А занимает 70% времени, разгоняема в 2 раза
- Часть В занимает 30% времени, разгоняема в 6 раз
- Какую часть будем разгонять?



Оценки: Pop quiz

Представим себе приложение с двумя отдельными частями:

- Часть А занимает 70% времени, разгоняема в 2 раза
- Часть В занимает 30% времени, разгоняема в 6 раз
- Какую часть будем разгонять?



Оценки: закон Амдала

$$Part_A = \frac{A}{A+B}$$
$$Part_B = \frac{B}{A+B}$$

Закон Амдала:

$$S = \frac{A+B}{\frac{A}{S_A} + B} = \frac{1}{\frac{Part_A}{S_A} + Part_B}$$

Следствия:

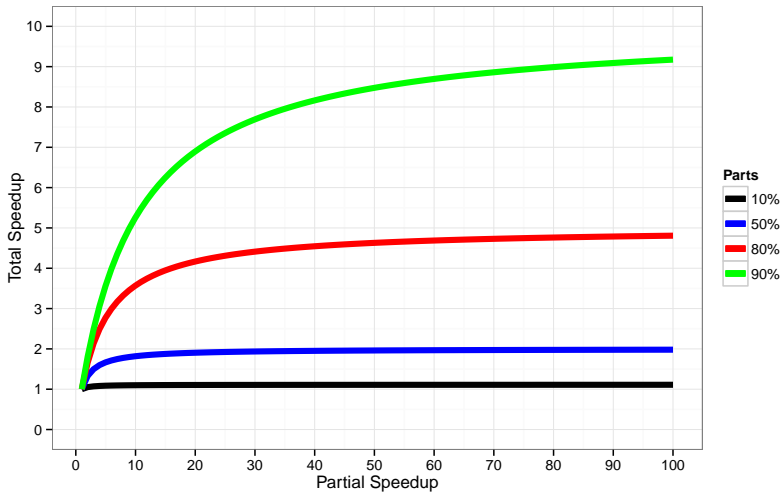
$$\lim_{Part_A \rightarrow 0} S = 1$$

$$\lim_{Part_A \rightarrow 1} S = S_A$$

$$\lim_{S_A \rightarrow 0} S = 0$$

$$\lim_{S_A \rightarrow \infty} S = \frac{1}{Part_B}$$

3-н Амдала: поведение



3-н Амдала: обобщение

Немного поиграем членами:

$$S = \frac{1}{\frac{P_A}{S_A} + P_B} = \frac{1}{\frac{1-P_B}{S_A} + P_B} = \frac{S_A}{1 - P_B + P_B S_A} = \frac{S_A}{1 + P_B(S_A - 1)}$$

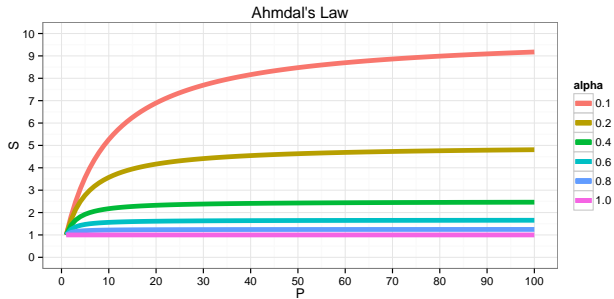
или, после подстановки

$p = S_A$ (во сколько раз ускорили часть),

$\alpha = P_B$ (сколько весит всё остальное):

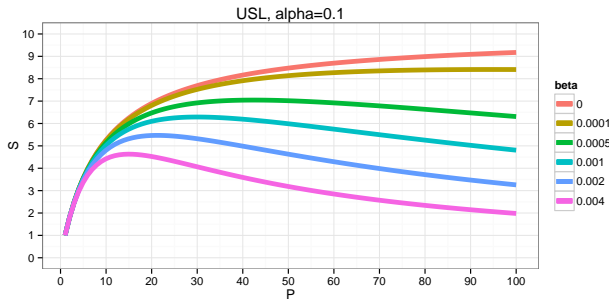
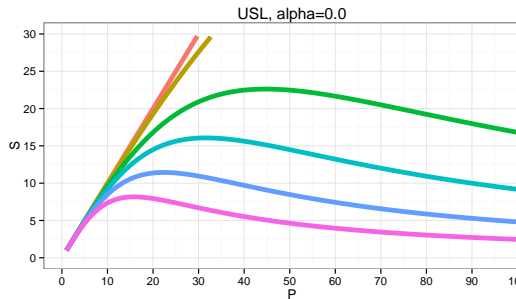
$$S = \frac{p}{\underbrace{1}_{\text{concurrency}} + \underbrace{\alpha(p-1)}_{\text{contention}}}$$

3-н Амдала: поведение



$$S = \frac{p}{\underbrace{1}_{\text{concurrency}} + \underbrace{\alpha(p-1)}_{\text{contention}}}$$

USL: Universal Scalability Law



$$S = \frac{p}{\underbrace{1}_{\text{concurrency}} + \underbrace{\alpha(p-1)}_{\text{contention}} + \underbrace{\beta p(p-1)}_{\text{coherence}}}$$

USL: Universal Scalability Law

$$S = \frac{p}{\underbrace{1}_{\text{concurrency}} + \underbrace{\alpha(p-1)}_{\text{contention}} + \underbrace{\beta p(p-1)}_{\text{coherence}}}$$

Наблюдения:

- USL хорошо натягивается на эмпирические данные
- При $\beta > 0$, с разгоном конкретной части не то, что может не становиться лучше, может становиться **хуже**
- Систем с $\alpha = 0$ и $\beta = 0$ практически не существует

Жёлтая зона: измерение производительности

Ментальная ловушка:

«Да мы наводим и посмотрим, что нам скажут бенчмарки»

На самом деле:

Перформанс-тестирование в норме дико дорогое, и всё не протестируешь.



Жёлтая зона: тестирование

Засада: перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту \Rightarrow сотни машинных часов на один коммит?

Жёлтая зона: тестирование

Засада: перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту \Rightarrow сотни машинных часов на один коммит?
- Требуют изоляции \Rightarrow нельзя шарить HW?

Жёлтая зона: тестирование

Засада: перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту \Rightarrow сотни машинных часов на один коммит?
- Требуют изоляции \Rightarrow нельзя шарить HW?
- Не бинарная метрика \Rightarrow человеко-часы на разбор данных?

Жёлтая зона: тестирование

Засада: перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту \Rightarrow сотни машинных часов на один коммит?
- Требуют изоляции \Rightarrow нельзя шарить HW?
- Не бинарная метрика \Rightarrow человеко-часы на разбор данных?
- Ошибки тестирования находятся только после разбора данных...

Жёлтая зона: тестирование

Засада: перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту \Rightarrow сотни машинных часов на один коммит?
- Требуют изоляции \Rightarrow нельзя шарить HW?
- Не бинарная метрика \Rightarrow человеко-часы на разбор данных?
- Ошибки тестирования находятся только после разбора данных...
- Бенчмарки дают **данные**, а хочется-то **результатов**

Жёлтая зона: тестирование

Засада: перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту \Rightarrow сотни машинных часов на один коммит?
- Требуют изоляции \Rightarrow нельзя шарить HW?
- Не бинарная метрика \Rightarrow человеко-часы на разбор данных?
- Ошибки тестирования находятся только после разбора данных...
- Бенчмарки дают **данные**, а хочется-то **результатов**

Выводы:

1. В активном проекте практически невозможно протестировать всё!
2. Нужно всё-таки уметь разбираться, куда копать.

Бенчмарки: много их, мой друг Горацио

Макробенчмарки

1. берём большой сайт, приложение, библиотеку целиком
2. пишем большой сценарий
3. измеряем от начала и до конца



Микробенчмарки

1. берём маленькую часть сайта, приложения, библиотеки
2. делаем мелкий изолированный тест
3. измеряем конкретную часть

Бенчмарки: макробенчмарки

Голоса в голове у разработчика ему шепчут:

1. Макробенчмарк отражает реальный мир

- ...и любой макробенчмарк – хороший
- ...запустил макробенчмарк – и это «real world»

Бенчмарки: макробенчмарки

Голоса в голове у разработчика ему шепчут:

1. Макробенчмарк отражает реальный мир
 - ...и любой макробенчмарк – хороший
 - ...запустил макробенчмарк – и это «real world»
2. Для любой крутой фишки, макробенчмарк даст крутое улучшение
 - ...если макробенчмарк не показывает улучшения, то фишка – дерьмо
 - ...если макробенчмарк показывает улучшение, то фишка – золото

Бенчмарки: макробенчмарки

Голоса в голове у разработчика ему шепчут:

1. Макробенчмарк отражает реальный мир
 - ...и любой макробенчмарк – хороший
 - ...запустил макробенчмарк – и это «real world»
2. Для любой крутой фишки, макробенчмарк даст крутое улучшение
 - ...если макробенчмарк не показывает улучшения, то фишка – дерьмо
 - ...если макробенчмарк показывает улучшение, то фишка – золото
3. Для любого крутого бага, макробенчмарк даст крутую регрессию
 - ...если макробенчмарк не показал регрессии, то бага и нет
 - ...если макробенчмарк показывает регрессию, то баг жуткий

Бенчмарки: микробенчмарки

Голоса в голове у разработчика ему шепчут:

1. Микробенчмарки – зло

- Очень удобно: раз зло, значит можно не обращать внимания

Бенчмарки: микробенчмарки

Голоса в голове у разработчика ему шепчут:

1. Микробенчмарки – зло

- Очень удобно: раз зло, значит можно не обращать внимания

2. Микробенчмарк можно написать какой угодно

- Регрессия на микробенчмарке ничего не значит
- Улучшение на микробенчмарке ничего не значит

Бенчмарки: микробенчмарки

Голоса в голове у разработчика ему шепчут:

1. Микробенчмарки – зло

- Очень удобно: раз зло, значит можно не обращать внимания

2. Микробенчмарк можно написать какой угодно

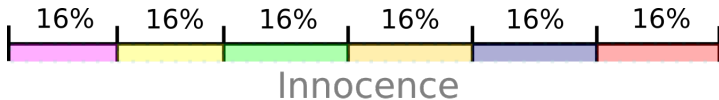
- Регрессия на микробенчмарке ничего не значит
- Улучшение на микробенчмарке ничего не значит

3. Микробенчмарки обычно пишут враги, чтобы опорочить наш продукт

- Говори, что микробенчмарк неправильный – и кодь дальше

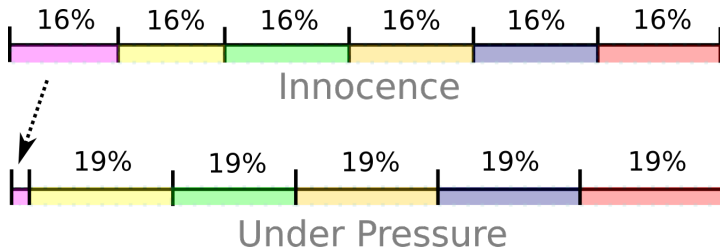
Бенчмарки: жизненный цикл бенчмарков

Все, все, все бенчмарки проходят эти стадии жизненного цикла:



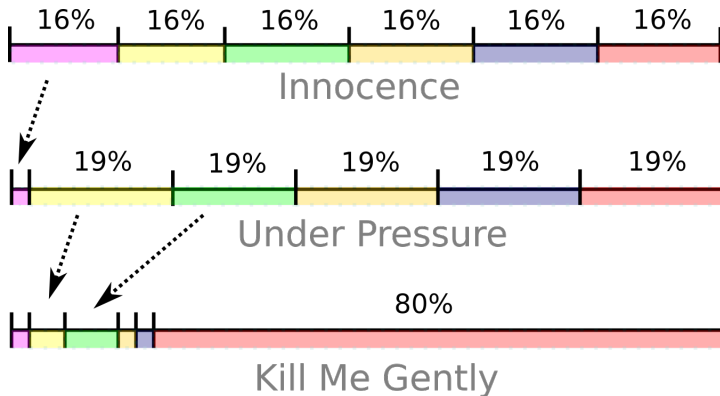
Бенчмарки: жизненный цикл бенчмарков

Все, все, все бенчмарки проходят эти стадии жизненного цикла:



Бенчмарки: жизненный цикл бенчмарков

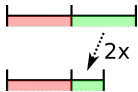
Все, все, все бенчмарки проходят эти стадии жизненного цикла:



Даже если ты начал как макробенчмарк, ты кончишь микробенчмарком

Бенчмарки: но есть и свои плюсы

Улучшения:

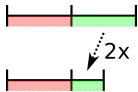


$$S = Speedup$$
$$p = Speedup_{part}$$

$$S = \frac{p}{\underbrace{1}_{concurrency} + \underbrace{\alpha(p-1)}_{contention}}$$

Бенчмарки: но есть и свои плюсы

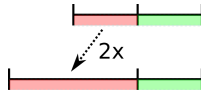
Улучшения:



$$S = Speedup$$
$$p = Speedup_{part}$$

$$S = \frac{p}{\underbrace{1}_{concurrency} + \underbrace{\alpha(p-1)}_{contention}}$$

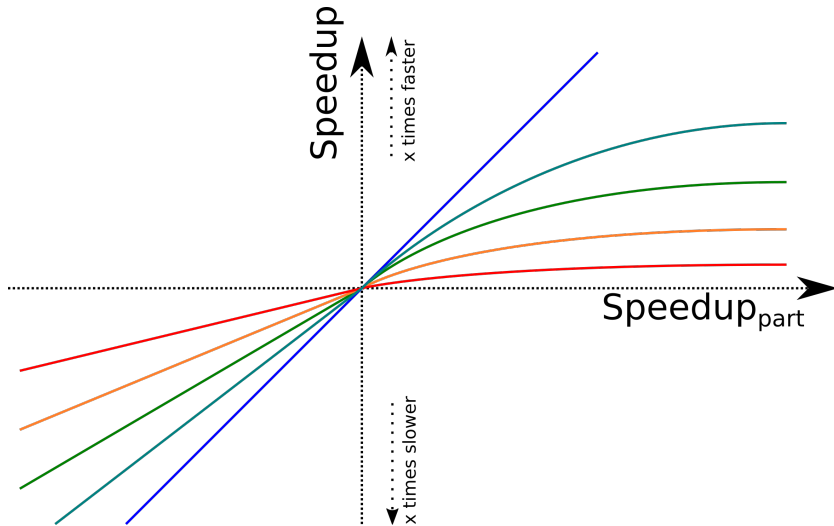
Регрессии:



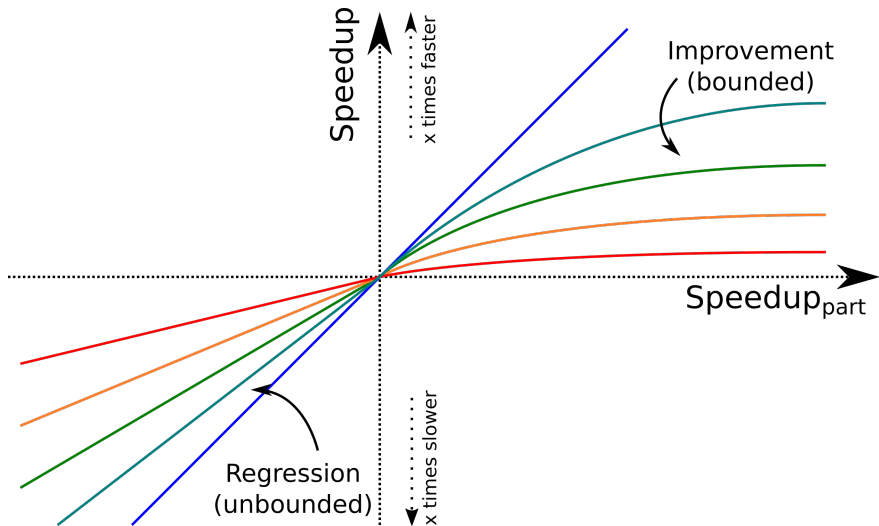
$$R = \frac{1}{S}$$
$$r = \frac{1}{p}$$

$$R = \underbrace{\alpha}_{save} + \underbrace{r(1-\alpha)}_{regressionssion}$$

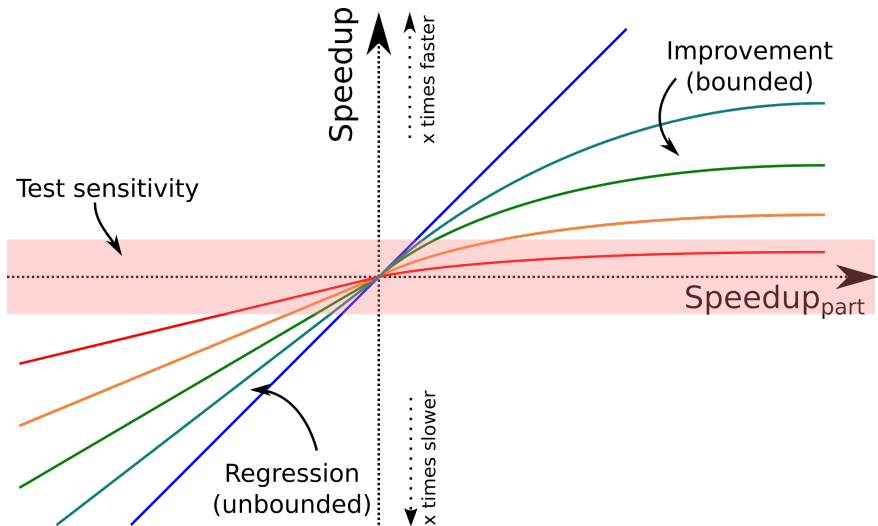
Бенчмарки: подставы



Бенчмарки: подставы



Бенчмарки: подставы



Бенчмарки: наблюдения

Макробенчмарки:

1. Их мало, они написаны непонятно кем и непонятно как
2. Сначала показывают интересные результаты, потом вырождаются
3. В конце жизни применимы для **регрессионного** тестирования

Микробенчмарки:

1. Их много, но их можно *исправить*
2. На всех этапах жизненного цикла показывают интересные результаты
3. На всех этапах отлично реагируют и на **регрессии**, и на **улучшения**

Оба этих класса не взаимозаменяемы!

Бенчмарки: микробенчмарки

Как вы не крутитесь,
а учиться микробенчмаркать вам придётся!

- «Просто возьми JMH» не работает
 - Нужно грамотно планировать эксперименты
 - Грамотно их проводить
 - Грамотно анализировать данные
 - Делать правильные выводы
- Главная задача – **построить правдоподобную модель**
 - Что и как влияет на производительность
 - Как конкретный сценарий прилежит к остальным сценариям

Жёлтая зона: оптимизации



Ментальная ловушка:

«Да мы просто возьмём, и будем делать то, что нам скажут в интернетах об оптимизации Джавы!»

На самом деле:

Глубокое понимание того, как устроено ваше приложение, поможет вам сделать на порядки более элегантное, поддерживаемое, и не гниющее под собственным весом изменение

Жёлтая зона: типичные штуки – опции JVM

Идея: зная что-то специальное о нашем приложении, подскажем JVM, в каком режиме работать

Радости: Синергия, механическая симпатия, всё такое

Проблемы: Ну как бэ...

```
-Xmx1G -Xms1G -Xmn128m -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+UseNUMA -  
XX:+CMSParallelRemarkEnabled -XX:MaxTenuringThreshold=15 -XX:MaxGCPauseMillis=30 -  
XX:GCPauseIntervalMillis=150 -XX:+UseAdaptiveGCBoundary -XX:-UseGCOverheadLimit -XX:+UseBiasedLocking -  
XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=15 -Dfml.ignorePatchDiscrepancies=true -  
Dfml.ignoreInvalidMinecraftCertificates=true -XX:+UseFastAccessorMethods -XX:+UseCompressedOops -  
XX:+OptimizeStringConcat -XX:+AggressiveOpts -XX:ReservedCodeCacheSize=2048m -XX:+UseCodeCacheFlushing -  
XX:SoftRefLRUPolicyMSPerMB=2000 -XX:ParallelGCThreads=10
```

Жёлтая зона: типичные штуки – параллелизм

Идея: неважно, что там где написано, берём
`parallelStream()`, `Executor.submit`, `new Thread`, и
параллелим

Радости:

1. Много ума не надо – раз, и готово!

Проблемы:

1. Синхронизация же – уверены, что всё работает?
2. Оверхеды же – уверены, что работы достаточно?
3. Есть куча внешнего параллелизма – внутренний параллелизм не нужен

Жёлтая зона: типичные штуки – примитивы

Идея: идут в печь эти `Collection<Integer>`,
будем делать `int[]`

Радости:

1. Плотненько так, упаковано, ням-ням

Проблемы:

1. Конверсии обратно во wrappers – уверены, что не сожрёт?
2. Конверсии туда-обратно всех коллекций – уверены?
3. Вставки-удаления-тормошения?
4. Оптимизации самой JDK? К Valhalla готовы?

Жёлтая зона: подитог

Поддержание правдоподобной перформансной модели проекта – **необходимое** условие развития проекта

Наблюдения:

- >50% потенциальных изменений делаются не там, где стоит
- >80% изменений делаются в нужном месте после исследования
- Умение *исследовать* и обновлять свои знания о проекте помогают принимать правильные решения – *поощряйте* это у девелоперов. Если проект большой, то вам нужны *выделенные роли* на такую работу.

Красная зона

Красная зона: мотивационная карточка

Красная зона:
эксплуатация кривизны нижних слоёв,
грязные хаки и залезание в кишки

Здесь Паша Эмильевич, обладавший сверхъестественным чутьем, понял, что сейчас его будут бить, может быть, даже ногами.

Красная зона: внимание

В абсолютном большинстве проектов сюда ходить не надо! Вменяемый техлид, проджект-менеджер или заботливая мама должны сказать:

Красная зона: внимание

В абсолютном большинстве проектов сюда ходить не надо! Вменяемый техлид, проджект-менеджер или заботливая мама должны сказать:

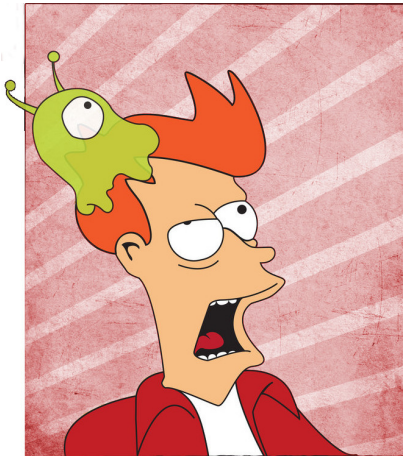


Красная зона: основная идея

Улучшение производительности от эксплуатации
особенностей реализации нижних слоёв

- дёрганье скрытых приватных методов
- дёрганье в «нужном» порядке публичных методов
- доступ к целым кускам приватного API
- хаки для обхода особенностей компиляторов
- микроархитектурные оптимизации

Красная зона: профилирование и диагностика



Ментальная ловушка:

«Если долго смотреть в профайл, можно увидеть там решение»

На самом деле:

Умение хакать вырастает из понимания взаимодействия всех движущихся деталек.

Красная зона: наработка корпуса

Перформансники –

это те люди, которые умеют копать во всех слоях сразу

- Имеют наработанный корпус хаков, знают границы их применимости
- **Не впадают в ступор**, когда видят незнакомую хрень, а начинают её изучать
- «Изучать» = читать документацию, искать упоминаний в статьях, смотреть на другие части кода и историю проекта, делать эксперименты, спрашивать коллег-специалистов, а не *спрашивать на StackOverflow «ой, а чо это такое?»*

Красная зона: подходы к исправлению

Ментальная ловушка:

«Если долго ездить по конференциям, то когда-нибудь там расскажут трюк, дающий 5x перформанса, и мы его тут же применим»

На самом деле:

В конкретном случае спасёт один низкоуровневый трюк из тысячи! Его проще найти самостоятельно, чем ждать у моря погоды.



Красная зона: откроем, значит, StackOverflow...

В программистской тусовке большая часть
обсуждений об этом!

- `(i++)` или `(++i)`?
- `for (int c = 0; c < L; c++)` или `while (c --> 0)?`
- `Math.pow(x, 2)` или `x*x`?
- `(x*2)` или `(x << 1)?`
- `(a*b != 0)` или `(a != 0 && b != 0)?`
- `(a & b)` или `(a && b)?`
- `String.isEmpty()` или `String.trim().length() == 0?`
- `System.arraycopy` или таки моя-любимая-идиома

Красная зона: костыли

Хаки должны быть *временными* заплатками, а не основой вашего проекта



- Указывают на проблему в слое ниже: нужно или исправить, или изучить, или зарепортить!
- Смеее учить хакам студентов – гореть вам в аду за вбивание гвоздей в гроб IT
- Условный Куксенко рассказывает про низкоуровневую вакханалию для наработки корпуса *возможных* хаков, а не чтобы вы их везде использовали

Красная зона: пример: ArrayList.Itr

```
public class ArrayList<E> implements Iterable<E> {  
    public Iterator<E> iterator() {  
        return new Itr();  
    }  
  
    private class Itr implements Iterator<E> { }
```

Красная зона: пример: ArrayList.Itr

```
public class ArrayList<E> implements Iterable<E> {  
    public Iterator<E> iterator() {  
        return new Itr();  
    }  
  
    private class Itr implements Iterator<E> { }  
  
    public [bridge] Itr(java.util.ArrayList, java.util.ArrayList$1);  
}
```

Красная зона: пример: ArrayList.Itr

```
public class ArrayList<E> implements Iterable<E> {  
    public Iterator<E> iterator() {  
        return new Itr();  
    }  
  
    private class Itr implements Iterator<E> { }  
  
    public [bridge] Itr(java.util.ArrayList, java.util.ArrayList$1);  
  
    /*  
    @ 3    j.u.ArrayList::iterator (10 bytes) inlined (hot)  
    - @ 6    j.u.ArrayList$Itr::<init> (6 bytes) unloaded signature classes  
    */  
}
```



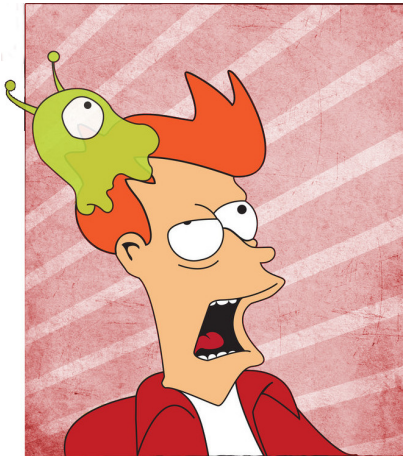
Красная зона: пример: ArrayList.Itr

```
public class ArrayList<E> implements Iterable<E> {  
    public Iterator<E> iterator() {  
        return new Itr();  
    }  
}
```

```
// HACK: Create explicit constructor to avoid generating a bad one  
// Needed because javac generates synthetic bridge with "unloaded class"  
// arguments, see JDK-xxxxxxx  
Itr() {};
```

```
private class Itr implements Iterator<E> { }  
}
```

Красная зона: подходы к исправлениям



Ментальная ловушка:

«Сейчас мы быстро подкрутим вот здесь, и
будет зашибись»

На самом деле:

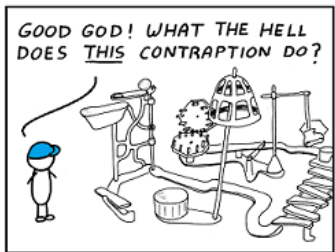
В красной зоне никогда не будет зашибись.



(три года назад)



Красная зона: технический долг!



**I hate reading
other people's code.**

Не обманывайте себя: работая в красной зоне, вы вносите технический долг.

Всегда, всегда, всегда документируйте:

1. Из-за чего хак выполнен
2. В каких условиях он применим
3. Как проверить, что хак больше не нужен
4. Какие upstream-баги вы ждёте, етц

Красная зона: подитог

Удержание численности и плотности хаков в проекте –
необходимое условие выживания проекта

Наблюдения:

- Дай волю, и весь проект порастёт...
- Умение работать с *upstream*-ами и другими компонентами сильно облегчают долговременную судьбу: конверсия хаков в реальные патчи улучшают глобальное положение
- Умение разбираться во всех слоях увеличивает тренируется «на кошках»

Напутствие

Напутствие: на хардкорной конференции

Пользователям продуктов:

1. Обновить запас и применимости хаков из **красной зоны**
2. Найти/обсудить подходы работы в **жёлтой зоне**
3. Обсудить, какие приёмы из **жёлтой зоны** завещать в **зелёную зону**

Напутствие: на хардкорной конференции

Пользователям продуктов:

1. Обновить запас и применимости хаков из **красной зоны**
2. Найти/обсудить подходы работы в **жёлтой зоне**
3. Обсудить, какие приёмы из **жёлтой зоны** завещать в **зелёную зону**

Разработчикам продуктов:

1. Обсудить подходы работы в **жёлтой зоне**
2. Понять, какие хаки из **красной зоны** перетащить в скрижали **жёлтой зоны**, или вообще в **зелёную зону**.

Напутствие: на хардкорной конференции

Пользователям продуктов:

1. Обновить запас и применимости хаков из **красной зоны**
2. Найти/обсудить подходы работы в **жёлтой зоне**
3. Обсудить, какие приёмы из **жёлтой зоны** завещать в **зелёную зону**

Разработчикам продуктов:

1. Обсудить подходы работы в **жёлтой зоне**
2. Понять, какие хаки из **красной зоны** перетащить в скрижали **жёлтой зоны**, или вообще в **зелёную зону**.

Не забывайте: основная перформансная работа проходит в **зелёной зоне**

