Code Organization Guidelines for Large Code Bases

Jürgen Höller
Interface21

Agenda

- Why worry about code organization?
- Package interdependencies
- Module decomposition and layering
- Evolving a large code base
- Case study: the evolution of Spring
- Tools for architectural analysis

Why worry about code organization?

- The obvious:
  - Code needs to be logically organized in units, to allow for easier understanding of the overall code base, and for easier navigation within the code base.
    - Only the most trivial applications can get away with keeping all code in a flat single unit...
  - Java offers the well-known package / sub-package concept
    - However, without any strong recommendations on how to apply it...

Why worry about code organization?

- The not-so-obvious:
  - A code base needs to be able to evolve based on its original structure!
    - Even years later, based on completely new requirements...
  - Refactoring and agile development are fine, but how do you preserve backwards compatibility once the code is released?
    - You might be pretty free to restructure the code base of an application that you are in full control of (i.e. have full ownership of)...
    - But what about published code with strong backwards compatibility requirements?

Why worry about code organization?

- The not-so-obvious continued...
  - Separate modules might need to interact in a later revision, despite the original design not having intended it.
    - introducing new interdependencies at the module / package level
  - Does the code base allow for repackaging into more fine-grained modules, if the need arises, while preserving the API?
    - even if you introduced new interdependencies in the meantime?

Why worry about code organization?

- The focus of this presentation:
  - package interdependencies
    - in particular in the context of evolving a code base
  - lessons learned from the evolution of the Spring code base
    - some anecdotes...
  - using tools to validate architectural soundness of a code base

Package interdependencies

- Designing a package structure is surprisingly non-trivial
  - The first cut of a package structure is always quite straightforward...
  - then along comes an unexpected new requirement...
  - how to fit it into the existing structure?
- Common code bases out there are often a less-than-ideal role model
  - starting with the JDK libraries...
  - as well as many open source projects

Package interdependencies

- Typical scenario:
  - Package B depends on package A according to its role in the architecture
  - but A could use a little piece of code from B in its own implementation...
  - don't want to duplicate code, hence just call that code in B from A...
  - now you got: B -> A -> B
    - a circle!
    - even if the involved classes differ, a circle emerges at the package level

Package interdependencies

- Central rule:
  - *Packages should have (at most) one-way dependencies between each other!*
  - clear architectural place for each package
  - in particular: no circular dependencies between packages
- Often violated...
  - example: java.lang <-> java.util
  - another example: Hibernate
- Counter example: Spring does not contain a single package circle!

Package interdependencies

- Why are one-way dependencies between packages so important?
  - a.k.a. Why are circles so undesirable?
- Nobody introduces circles deliberately...
  - they rather emerge over time
  - *indicating code deterioration*
- Circles limit reuse of packages
  - Try splitting one of the affected packages out into its *own build module*...
  - B needs to compile against A, but A needs to compile against B as well...

Package interdependencies

- Essence: *Avoid circular dependencies between packages!*
  - However, that is easier said than done...
    - new requirements might imply new interconnections between packages
  - often requires creative refactoring
    - which in turn imposes backwards compatibility challenges...
  - Nevertheless: Also avoid code duplication!
    - Do not take the easy way out ☐

Module decomposition and layering

- One step up in granularity: assemble packages into *conceptual modules*
  - with reasonably natural boundaries
  - Generally, modules are a collection of specific packages...
    - collaborating and/or conceptually related
    - might live in separate source directories, but do not have to
  - Some modules might consist of a single package only...
    - or even a single sub-package

Module decomposition and layering

- Modules are often driven by *deployment* needs as much as conceptual boundaries
  - multi-tier separation
    - often unnatural, since it does not match conceptual boundaries
  - runtime dependencies
    - isolate specific dependencies into their own modules (e.g.: JDK 1.5, Hibernate)
  - jar size
    - keep content as minimal as possible
    - tailored for specific use case scenarios

Module decomposition and layering

- One step up in granularity: assemble packages into *conceptual modules*
  - with reasonably natural boundaries
  - Generally, modules are a collection of specific packages…
    - collaborating and/or conceptually related
    - might live in separate source directories, but do not have to
  - Some modules might consist of a single package only…
    - or even a single sub-package

Module decomposition and layering

- Modules are often driven by *deployment* needs as much as conceptual boundaries
  - multi-tier separation
    - often unnatural, since it does not match conceptual boundaries
  - runtime dependencies
    - isolate specific dependencies into their own modules (e.g.: JDK 1.5, Hibernate)
  - jar size
    - keep content as minimal as possible
    - tailored for specific use case scenarios

Module decomposition and layering

- Desirable characteristics of a module
  - low coupling
    - to other modules
  - high cohesion
    - within the module
- Modules are a *conceptual unit* as much as a source management & deployment unit
  - avoid cognitive overload
  - should allow for individual use
    - or a distinct role within a larger system

Module decomposition and layering

- Layering is essentially a logical view on the package structure
  - higher layers build on lower layers = higher-level packages depend on lower-level packages, not the other way round
- The module structure might have a straightforward mapping onto layers
  - However, this is not strictly necessary...
  - since modules might be a vertical slice
  - Modules are often driven by deployment considerations more than by layering!

Module decomposition and layering

- Essence: *Establish natural conceptual boundaries in your code base!*
  - It does not matter (much) where the source code resides...
    - single shared source root
    - or one source root per module
  - Although it does help if the source code structure mirrors the conceptual structure
    - natural package naming
    - easy navigation!

Evolving a large code base

- The hardest challenge is *evolving the code as well as the architecture over time...*
  - without letting the code deteriorate
  - not compromising on architectural quality
- This becomes exponentially harder with growing size of the overall code base!
  - many developers involved
  - often no single point of architectural management and enforcement anymore
    - at the fine-grained artifact / module level

Evolving a large code base

- Tradeoff between backwards compatibility and architectural quality?
  - strict 100% backwards compatibility might not allow for sustaining the architectural quality level
- *Nevertheless, there is (almost) always a better solution than compromising on architectural quality!*
  - e.g. a creative internal refactoring that allows to preserve compatibility as well as well-defined package dependencies

Case study: the evolution of Spring

- Example: the Spring core
  - origins date back to 2001
  - first public release as download on Wrox website in late 2002
  - first public release as open source project in mid 2003
  - went 1.0 final in early 2004, implying backwards compatibility guarantees
  - 2.0 came in 2006, allowing for some isolated compatibility breakages, but largely compatible with 1.2

Case study: the evolution of Spring

- The Spring project faces many code evolution challenges
  - broad public API, used by applications
  - sophisticated SPI, used by advanced applications as well as sister products and third-party frameworks
  - new requirements addressed in every release, often implying some refactoring
  - How has the Spring code base survived in its original shape for 3.5 years already?

Case study: the evolution of Spring

- Clue: strict architecture management
  - loosely coupled packages with well-defined interdependencies
    - org.springframework.util
    - org.springframework.core
    - org.springframework.beans
    - org.springframework.aop
    - ...
  - no circles allowed at package level, not even as a temporary measure
    - if it looks like we need a circle, we force ourselves to look again – and think harder

Case study: the evolution of Spring

- Stories from Spring's history...
  - core <-> util
  - beans <-> aop
  - beans <-> context
  - transaction <-> dao
  - transaction <-> jdbc
- Special challenge: global configuration
  - low coupling through dependency injection
  - Spring 2.0 XML namespaces
    - namespace discovery at runtime

Case study: the evolution of Spring

- Ever-changing third-party libraries
  - Hibernate 2.1 -> 3.0 -> 3.1 -> 3.2
  - iBATIS 2.0 -> 2.1 -> 2.2 -> 2.3
  - Quartz 1.3 -> 1.4 -> 1.5 -> 1.6
- What to do in case of incompatible API changes in such libraries?
  - while still preserving compatibility with previous versions
    - binary compatibility required!
  - solution: reflective checks and invocations
    - where necessary

Case study: the evolution of Spring

- Ever-changing third-party libraries
  - Hibernate 2.1 -> 3.0 -> 3.1 -> 3.2
  - iBATIS 2.0 -> 2.1 -> 2.2 -> 2.3
  - Quartz 1.3 -> 1.4 -> 1.5 -> 1.6
- What to do in case of incompatible API changes in such libraries?
  - while still preserving compatibility with previous versions
    - binary compatibility required!
  - solution: reflective checks and invocations
    - where necessary

Case study: the evolution of Spring

- How do we make sure that no architectural violations, such as circles between packages, slip in?
  - Manual analysis only gets you so far...
    - it's a bit like manual testing versus automated testing
  - Solution: use tools!
    - since 2003: JDepend
    - new in the toolbox: SonarJ
  - We at least run JDepend before every public release, as a sanity check!

Tools for architectural analysis

- JDepend
  - http://clarkware.com/software/JDepend.html
  - open source tool
    - by Mike Clark
  - the classic candidate
    - been around since 2001
  - typically used as command line tool
    - trivial to install
  - generates analysis report
    - including package dependency cycles

Tools for architectural analysis

- SonarJ
  - http://www.hello2morrow.com/en/sonarj/sonarj.php
  - commercial tool
    - by hello2morrow
  - GUI-driven architecture introspection
    - including package dependency analysis
  - custom architectural constraints
    - allowed imports, etc
  - on-the-fly analysis
    - change code, check architectural soundness

Tools for architectural analysis

- DEMO
  - evolving the Spring code base
  - checking it with JDepend and SonarJ
  - also calculating some metrics
- Let's do some comparisons...
  - Spring 2.0.1
  - Spring 1.2.8
  - Hibernate 3.2.1