Principles of Pattern Enabled Java Development

MARVIN TOLL, GTC Group
WILLIAM R. MINTO, Center for Effective Thinking

Pattern Enabled Development® (PED) seeks to advance Java application construction by inspiring Pattern First Thinking among software engineers, enabling project teams with a Pattern Palette, and harmonizing enterprise communication with a Pattern Language.[1]

1. INTRODUCTION

Contemporary software engineering professionals can point to at least five landmark advances in the field: object-oriented software (Meyer), design patterns (Gang of Four), enterprise architecture patterns (Fowler), implementation patterns (Beck 2007), and the practices associated with "Agile" software development (Agile Manifesto signatories). PED is a development within this tradition that seeks to extend proven practices by introducing a set of principles targeted specifically for Java application development with patterns.

Since PED emerged from experiences with multiple teams (albeit within a single successful large enterprise), we are reasonably confident that the principles are potentially transferable, provided PED's adopters are sensitive to the context in which they are applied and make appropriate adjustments. We begin with a brief retrospective from which Pattern Enabled Development arose. We then explore the principles in three movements: The Individual, The Team, and The Enterprise.

2. PED: A BRIEF RETROSPECTIVE (2004-2014)

In 2002, Java was adopted as the standard platform for in-house development at a successful large automotive manufacturing enterprise ("the Company"). In the ensuing years, many applications would migrate from Perl/CGI to Java. Marvin Toll joined one such co-located project team in 2004 as a consultant to migrate a significant global dealer application to the new platform.

Despite considerable Perl/CGI, and (to a lesser degree) Smalltalk expertise and a deep understanding of the business domain, the team was about to take on a technical challenge without the correct skill set! A tight project timeline precluding formal Java language training compounded the skills deficit.[2] Fortunately, the team included capable performers that worked well together.

In 2004, we asked: Could a novel approach compensate for the lack of requisite Java knowledge? Would the project succeed if the team committed to adopting simple patterns prior to coding? If so, which ones? Would individual example pattern implementations, or snippets, be effective for communicating to an audience without a Java background? How much Test Driven Development (TDD) could a team absorb when they had not learned the programming language?

A significant portion of the project's success[3] can be traced to what would years later be codified as Pattern Enabled Development® (PED). As Todd Hall, technical lead and architect, reflects: "without patterns, the team would not have created a solution or an application architecture that is still in use ten years later."[4] Thus launched the PED Journey, a decade-long validation and refinement of lessons first learned from a single project team, and the pursuit of several additional questions:

[1]The authors would like to thank the following for their generous and insightful comments on drafts of this paper: Robert (Bob) Hanmer, Sorin Costea, Rebecca Wirfs-Brock, Joseph Yoder, and Eduardo Guerra.

[2] Hiring experienced Java developers without relevant business domain expertise would not have improved the situation.
[3] In this case, "success" meant an uninterrupted flow of delivered business value.
[4] Todd Hall, private correspondence, June 30, 2014.

- 2005 – Would a monolithic abstraction (framework) wrapping primary technologies (in this case, both Struts and Toplink) be beneficial? (Although we tried this option with several teams, we concluded that a monolithic approach was a failed strategy. This initiative was abandoned in favor of a revised approach in 2007.)
- 2006 – Would a reference application based on existing production use cases serve to connect the efforts of multiple globally distributed teams? (We soon realized the complexity of 'real world' use cases was inhibiting the learning of fundamentals. This initiative was abandoned in favor of a revised approach in 2008.)
- 2007 – Could loosely-coupled "adaptable technology wrappers" provide development convenience *and* insulate "the Company" from technology change? (See Adaptable Technology Wrapper (ATW) on p. 10.) Our experience suggests an affirmative answer. We later wondered if an ATW could foster alignment with a Pattern Language.
- 2008 – Could a "reference application" supersede comprehensive documentation as the primary vehicle for providing architectural and design guidance? (See Reference Application on p. 14.)
- 2009 – Could "instructor guided training" inspire use of the reference application as a self-study resource for a typical developer? (See Instructor Guided Training on p. 15.)
- 2010 – Would providing a website where "pattern collections" are published encourage project teams to formulate their own Pattern Palette?" (See Pattern Palette Adoption on p. 4 and Pattern Collection Publication on p. 10.)
- 2011 – Could we use a "real services" approach to accelerate performance of JUnit testing while reducing a dependence upon mock-related objects? (See Testing in Unmanaged Mode [TUM] on p. 8.)
- 2012 – Could a "pattern coverage" tool provide effective metrics for enhancing code quality? (See Pattern Coverage Measurement on p. 7.)
- 2013 – Would Pattern First Thinking be useful for coding client-centric JavaScript Responsive Web Design (RWD) applications? (Although we are actively pursuing this question with an initial set of UI patterns, it is too early to assert a definite conclusion.)
- 2014 – Are there ways to increase the value of Pattern Language usage for "the Company?" (Since we have only begun to answer this question, it is not addressed in this paper.)

We have codified the lessons learned during "the Company" journey into ten principles, explicated in the following sections.

3. THE PED PRINCIPLE FOR INSPIRING SOFTWARE ENGINEERS

PED advocates one principle for inspiring software engineers: Pattern First Thinking.

3.1 Pattern First Thinking

> Principle 1. *In the beginning, when writing your first line of test code, keep in mind a simple class pattern. In the end, your software will be more flexible.*

Based on experience with "the Company," we submit that a "pattern first" mindset: is compatible with Test Driven Development (TDD), delivers *flexible* software to stakeholders, prefers *simple* class patterns, and provides additional benefits.

The following graphic illustrates where Pattern First Thinking integrates with TDD:[5]

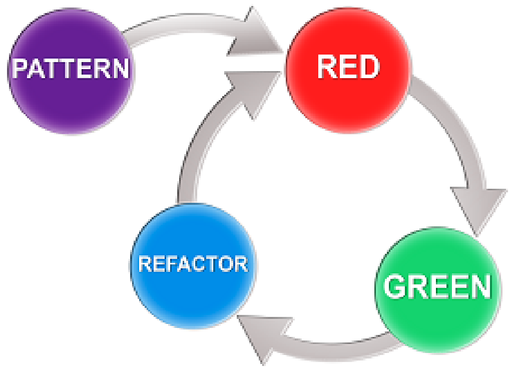


Fig. 1. Pattern First Thinking Integrates with TDD

Pattern First Thinking makes explicit what many seasoned developers already practice; that is, an ability to mentally conceptualize target patterns concurrently with constructing test code. However, we have observed that unseasoned TDD developers do not always engage in pattern reflection synchronously with JUnit test implementation. Said another way, an experienced practitioner may think of a simple pattern for production code while writing test code in a single step. An inexperienced developer may need prompting to do the same thing in two discrete steps.

It is not the intent for an emphasis on "pattern first" thought to supplant the traditional Red-Green-Refactor sequence. Specifically, the Refactor step should continue to enable pattern emergence. Emergence can take the form of moving from a simple to more complex pattern, moving from an undetermined pattern to an emerged pattern, or moving from an initial anticipated pattern to an alternative.

PED presumes "software flexibility" is an attribute of computer applications valued by stakeholders. We define software flexibility as:

> [t]he quality attribute of working software whereby changes to its code base are introduced rapidly while preserving the overall functional integrity of the application. Software flexibility is dependent upon a number of other application attributes that ensures its resiliency. These include the ease with which changes are understood, tested and debugged by the development team. Software flexibility is prized where it is critical that the software be quickly restored to a fully functional condition despite frequent iterations throughout its maturation.

Todd Hall might well have said "without *simple* patterns, the team would not have created a solution or an application architecture that is still in use ten years later." We wish to promote simplicity of *structure*. We regard patterns as structurally complex if they require multiple Java classes for their implementation, *e.g.*, Model-View-Controller or the Strategy and Adapter patterns. Simple patterns are generally implementable as a single class.

The common historical division of patterns into structural, behavioral, *etc.*, *categories* is not our concern. Rather, our emphasis is upon how patterns manifest both structural and behavioral *elements*. It is this sense of simplicity that we read into Kent Beck's contentions that "simplicity can encourage flexibility" (Beck 2007, 12) and "… the flexibility of simplicity and extensive tests is more effective than the flexibility offered by speculative design." (Beck 2007, 12)

To achieve software flexibility, a developer might have in mind a pattern that is overly complex for its immediate purpose. Complexity can convey an illusory sense of flexible software, *i.e., faux* flexibility, where a developer spends additional time on testing and implementation (tantamount to over-engineering), with the imagined benefits never coming to fruition.

[5] We recognize the validity of the technique of "refactoring to patterns," although this diagram does not explicitly represent it.

We adopted structurally simple patterns in 2004 due to the technical limitations of a specific project team. Only over the next decade did we discover that a structurally simple foundation helps sustain maintenance and enhancement of an application. Our conclusion: *to achieve genuine software flexibility, a developer should begin with the simplest viable pattern for a given purpose.*

Although our focus is how Pattern First Thinking contributes to software flexibility, Kent Beck, writing in 2007, affirmed an additional benefit (*i.e.,* speed):

> Once a set of implementation patterns has become habitual, I program faster and with fewer distracting thoughts. When I began writing my first set of implementation patterns (1996) I thought I was a proficient programmer. To encourage myself to focus on patterns, *I refused to type a character of code unless I had first written down the pattern I was following*. It was frustrating, like I was coding with my fingers glued together. For the first week every minute of coding was preceded by an hour of writing. The second week I found I had most of the basic patterns in place and most of the time I was following existing patterns. By the third week I was coding much faster than I had before, because I had carefully looked at my own style and I wasn't nagged by doubts.

4. PED PRINCIPLES FOR EMPOWERING PROJECT TEAMS

Principles 2 through 6 are relevant for empowering project teams:

- Pattern Palette Adoption
- Pattern Champion Acknowledgement
- Source Code Understandability
- Pattern Coverage Measurement
- Testing in Unmanaged Mode (TUM)

In the following sections, we discuss these in turn.[6]

4.1 Pattern Palette Adoption

> Principle 2. *Starting with project inception, adopt a coherent Pattern Palette from proven pattern collections.[7] Continue to mature your team palette throughout the project life cycle.*

Based on experience with "the Company," we submit that successful "Pattern Palette adoption"[8] is: based on coherent patterns from a proven collection, aligned with application architecture, a conversation best started during project inception, and usually not "frozen" after initial codification.

Coherence is a necessary condition for an unrelated set of patterns to be considered a Pattern Language, that is, patterns that "work together to solve problems." (Manns and Rising 2005, 14) A judiciously adopted Pattern Palette can inherit the coherence of its parent collection(s), and thus become a "dialect" of the original language(s). Additionally, the pattern collection should be "proven"—it has been previously used with success to deliver similar (business) solutions in equivalent (technical) contexts.

Some patterns carry architectural implications, while others do not. For example, a BUSINESS FACADE class contains reusable logic that belongs within the business layer, whereas a HELPER class could "reside" within any layer in the code base. Therefore, a team Pattern Palette should be established in concert with the application architecture. Vetted patterns (or a vetted pattern collection) nudge a team towards application architecture durability while stimulating design inspiration. A suitable Pattern Palette incorporates "just enough" (and no more) up-front architecture as a foundation for long-term adaptability, and demonstrates "just enough" (and no more) richness and variety to kindle design insight.

[6] The principles pertaining to enterprise communication begin on page 10.
[7] See Pattern Collection Publication, p. 10.
[8] The metaphor of a Pattern *Palette* suggests an artist's palette, and implies an adaptive, non-prescriptive adoption of patterns from one or more pattern collections, for use on a project.

Pattern adoption is a team design exercise that begins the process of aligning divergent technical perspectives. There is no need to wait for key stakeholders' to fully articulate their complete set of project requirements. Technical alignment is particularly important for newly constituted teams during their "norming" phase.

We have observed project teams benefiting from face-to-face conversations when selecting patterns from the PED collections, or from other sources.[9] We concur enthusiastically with the sixth principle of the Agile Manifesto: "the most efficient and effective method of conveying information to and within a development team is face-to-face conversation."[10] A mature self-organizing team will leverage its collective wisdom by seeking opportunities to maximize the amount of work not done; adopting patterns that guide effective development realizes one such opportunity.

In the hands of an experienced artist, colors may be introduced (while preserving chromatic coherence) or eliminated from her *palette* throughout the composition process. Likewise, seasoned developers may add or subtract patterns from their team *Pattern Palette* over the course of iterations.

4.2 Pattern Champion Acknowledgement

> Principle 3. *Consider acknowledging a team champion to promote pattern adoption. Your code base is more likely to remain internally coherent as a result of your champion's efforts.*

Based on experience with "the Company," we submit that successful pattern reuse is aided by the acknowledgement of an informal "pattern champion." A pattern champion should be well versed in the target technology platform (*i.e.*, Java) and savvy about application architecture, design and testing. An effective pattern champion possesses pragmatic knowledge of commonly used industry patterns and is familiar with the enterprise Pattern Language.

Over the past ten years, we have seen how helpful it was when acknowledging a team member in the role of informal pattern champion—to inculcate pattern awareness within the team. A project benefits when the technical leader of the team values the pattern champion, or the technical leader assumes the pattern champion role themselves. The roles of an pattern champion may include:

- Suggesting a viable initial palette
- Responding with palette adjustments for team consideration as business requirements emerge
- Making the pattern adoption process "big and visible"
- Ensuring that the pattern selection conversation within the team begins early
- Facilitating the revisiting of assumptions as circumstances warrant
- Focusing the team on closing emergent gaps between the Pattern Palette and the code base
- Leading a team towards discussions concerning the positive and negative consequences of implementation choices among design alternatives

We refer to patterns that the team has agreed to use as "on-palette," whereas other patterns that are reflected in the code are labeled as "off-palette." There is a natural tendency for the Pattern Palette and the code base to diverge over iterations, giving rise to two risks worth noting.

First, a team member may feel compelled to code to an on-palette pattern when an off-palette alternative is a better fit for their user story. The pattern champion helps the team recognize that, at times, an "off-palette" approach is appropriate. For example, if using an on-palette pattern would constitute its misuse, the champion may encourage an off-palette alternative.

Secondly, a team member could write code that incorporates an off-palette pattern, which is unknowingly picked up by other team members. This *re*use of an off-palette pattern is presumably a good reason for including the pattern on the Pattern Palette. Should such circumstances materialize, the pattern champion facilitates a discussion about whether the pattern warrants promotion to the Pattern Palette, or remains a valid, albeit off-palette, alternative.

[9] See http://pedCentral.com. We are not arguing for the PED collections, specifically. Our bias is, of course, that PED be consulted as a starting point, but other pattern collections may provide additional inspiration for a team adopting its own Patten Palette.

[10] http://agilemanifesto.org/principles.html

4.3 Software Understandability

> Principle 4. *When authoring a narrative in Java, imagine the potential reading audience. Such a narrative—your code—will be understood beyond a pair programmer and immediate project team.*

Based on experience with “the Company,” we assert that “software understandability” is aided by: gauging the (code) reading ability of current and future readers, techniques for expressing code structure, the incorporation of business context when naming Java constructs, and overcoming a limitation of pair programming.

We have witnessed first-hand the emergence of global sourcing as an impediment to the co-location ideal represented by Todd’s team. Can anyone have confidence regarding who will maintain their applications in five or ten years? What native languages will they speak? Where will they reside? What cultural contexts will shape their technical and business views? How much will the enterprise be willing to invest in developer talent?

A team adage, such as “Write Once, Read Many,” is a useful reminder that application code has a longer shelf life than first imagined, and a diverse readership —well beyond the original development team.

We agree with the spirit of the notion that “source code is living documentation.” Accordingly, PED seeks to incorporate a convention that enhances the capacity of code to communicate its structure to a reader. We dub our technique "Pattern Encoding," where an applied pattern is denoted by a suffix. For example, the NoteDE class is recognized as an implementation of the Domain Entity pattern based on its suffix ‘DE.’[11]

In the early days of information systems, COBOL (COmmon Business-Oriented Language) was intended as a tool for the automation of business processes. COBOL syntax is intentionally verbose, reflecting its designer's desire for readability (by both technical and non-technical staff), with English-like syntax and structural elements, such as nouns, verbs, clauses, sentences, *etc*.

Java, as an intentionally more general-purpose language, has neither the English-like grammatical structure nor the incorporation of English business operations into its language elements. To compensate for its native lack of readability, the communication of business-related functionality in Java relies on the developer to effectively name classes, methods, and variables.

We encourage thinking about source code, including JUnit test code, as a business narrative for humans, not compilers. When that kind of thinking occurs, we get code that is understandable in the sense that it reveals the business narrative when read. Martin Fowler (1998, 55) is more direct: “Any damn fool can write code that a computer can understand. The trick is to write code that humans can understand.”[12] Inscrutable code demands that the new developer regularly interrupt veteran team members with questions. Moreover, if business domain knowledge cannot be gleaned by reading the code, the ability of software engineers to move from project to project suffers.

While we appreciate the value of standard practices employed to facilitate understandable code – pair programming in particular – they arguably are inadequate for long-term global readability of enterprise source code. Pair programming, and even a team code review, sometimes demonstrates a kind of parochialism – the native dialect of the current team. That is, teams will evolve to using shorthand communication mechanisms (*e.g.*, acronyms) and the assumptions that go with “team-speak” find their way into the source code.

The antidote is a viable Pattern Palette derived from pattern collections expressing a more universal language. While some team idioms are inevitable, we prefer code produced by pairs when its dialect is intelligible to developers on other teams.

---

[11] Despite superficial similarities with Hungarian notation, none of the well-rehearsed objections to that convention apply to Pattern Encoding. (Search for “Pattern Encoding” at http://pedCentral.com.)

[12] Fowler revised this to “Any fool can write code that a computer can understand. Good programmers write code that humans can understand” in later publications.

4.4 Pattern Coverage Measurement

> Principle 5. *Throughout the project, when measuring code quality, keep in mind Pattern Coverage. This structural measure will enrich your suite of quality metrics.*

Based on our limited experience with "the Company," we have begun seeing the advantages of measuring "pattern coverage" with a technique that provides: low-cost direct testing of accessor methods, automated feedback on pattern reuse, and visible measures.

It is typical for teams committed to TDD to forego *direct* testing of getter/setter (accessor) methods in favor of *indirect* testing. Beginning in 2006, Todd's team had begun experimenting with a JUnit-based tool, TestUtil,[13] to provide direct testing of accessor methods.

With indirect testing, coverage of getter/setter methods is haphazard, leading to the lack of comparability between different teams' measurements. On the other hand, direct testing enables commensurable code coverage metrics. That is, each team's measurements of code coverage are comparable because they are being measured using the same standard. Direct testing of getter/setter methods has subsequently been incorporated into PED's ATW (jPED).[14]

What began as a modest test goal has matured into an innovative approach for: (i) detecting Pattern Palette occurrences within the source code, (ii) verifying the class complies with pre-defined structural elements, and (iii) reporting results. These three steps can be wrapped in a single test method for interrogating an entire code base.

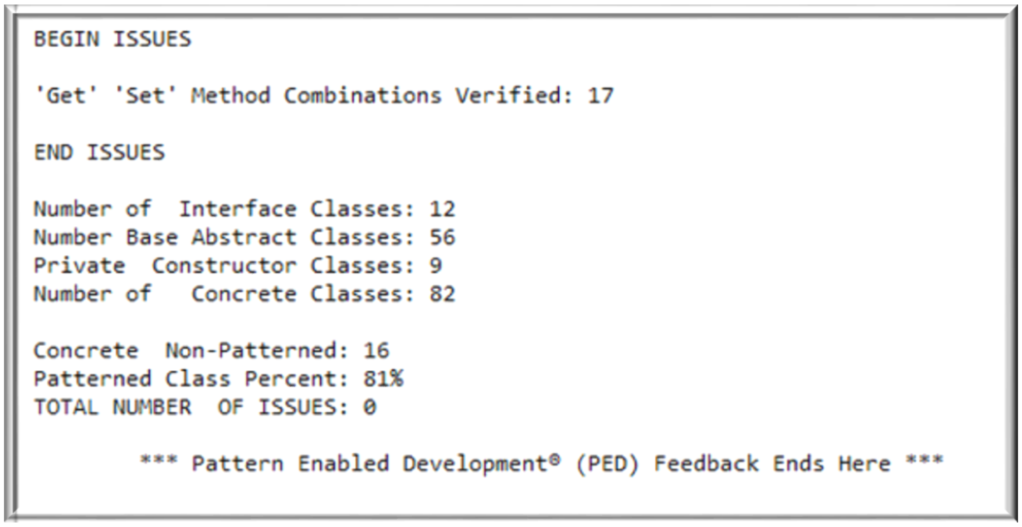
For example, suppose a project has adopted PED's DOMAIN ENTITY pattern (see the Appendix for an elaboration of this pattern). Classes implementing the pattern contain a no-argument constructor (a structural characteristic) and end in the standard suffix 'DE.' The test method executes three steps to verify constructor compliance:

1. Interrogate the code base to identify classes ending in 'DE.'
2. Attempt to instantiate each concrete DE class using its no-argument constructor.
3. Report whether the attempt was successful (classes without a no-argument constructor fail step 2).

Notice that none of these steps have a direct relationship to specific application functions, thus the designation "use-case-agnostic." That is, pattern reuse can be analyzed regardless of business domain.

The Pattern Coverage assert method iterates through the classes in the code base and verifies the presence of key elements of each pattern on the Pattern Palette. Upon completion, jPED reports summary metrics such as what percentage of the classes are derived from the Pattern Palette and what percentage are not, according to how they represent themselves (based on suffix).

[13] This refers to Marvin Toll's open source TestUtil 2.0 project, circa 2006.
[14] An adaptable technology wrapper (ATW) refers to a layer of code abstracting an API for convenience and reducing duplication. The PED Website (http://pedCentral.com) provides an open source reference implementation (jPED).



```
BEGIN ISSUES

'Get' 'Set' Method Combinations Verified: 17

END ISSUES

Number of  Interface Classes: 12
Number Base Abstract Classes: 56
Private  Constructor Classes: 9
Number of   Concrete Classes: 82

Concrete  Non-Patterned: 16
Patterned Class Percent: 81%
TOTAL NUMBER  OF ISSUES: 0

        *** Pattern Enabled Development® (PED) Feedback Ends Here ***
```

Fig. 2. Output from jPED Pattern Coverage Tool

Based on our initial experience, teams should strive for a Pattern Palette robust enough so that approximately 80% of the concrete application classes correspond to palette patterns. We refer to this percentage as the "pattern coverage."

Teams with the best of intentions may fall short of their aspirations when delivery pressures mount. Metrics reported on "big visible charts" or "information radiators" help sustain a culture of accountability to those aspirations. When the radiator shows that coverage metrics slip out of the ideal range, this fact is made more difficult to ignore, and encourages the team to refocus.

4.5 Testing in Unmanaged Mode (TUM)

> Principle 6. *Ensure pattern implementations are conveniently testable. An unmanaged mode encourages frequent running of the complete test suite using real services.*

Based on our initial experience with "the Company," we encourage a "TUM" approach, which seeks to: use an unmanaged mode, leverage indirect testing when possible, and encourage running the full suite of JUnit tests frequently.

We have observed that irrational exuberance for mock testing can lead to a disproportionate ratio of test operations to production code operations. This ratio can serve as a "code smell" for the potential elimination of test code waste. We sympathize with sentiments expressed (Goncalves, 2012) at the launch of the "NoMock Movement" ('No' in 'NoMock' meaning *N*ot *O*nly):

> This movement is for people [who have stopped] being ashamed of not mocking every single aspect of their code and [who are] starting to write more ... tests. Join the movement, don't be ashamed, and repeat after me: 'Stop mocking, start testing!'

We advocate launching JUnit tests using a *real* service, thereby allowing their execution in a production-like context without needing a Java application server. This can be accomplished in the localhost environment as well as on a Continuous Integration server.[15]

We also advocate pursuing creative techniques for direct *and* indirect testing of production code. "Uncle Bob" (Martin 2013) remarks on when TDD is *not* to be practiced:

[15] For example, a CDI *BeanManager* is supported by a modern Java application server. A BeanManager can also be supported by the DeltaSpike testing framework, which provides a CDI service without requiring an application server. By launching the DeltaSpike testing framework in a JUnit JVM, the tests benefit from a production-like, or real, CDI service.

So, when do I not practice TDD? I don't write tests for getters and setters.
Doing so is usually foolish. Those getters and setters will be indirectly tested by the tests of the other methods; so there's no point in testing them directly.
I don't write tests for member variables. They too will be tested indirectly.
I don't write tests for one line functions or functions that are obviously trivial.
Again, they'll be tested indirectly.

Test performance typically degrades when exiting the JUnit JVM. Two alternatives exist for acceptable test performance: "unmanaged mode" or "mock resources." We advocate the unmanaged mode alternative so that JUnit tests run in a context more like that provided by a (real) application server. This reduces the volume of test code, thus maximizing the amount of work not done. Unmanaged services include:

- CDI containers
- in-memory databases
- JAX-RS RESTful clients
- JPA persistence engines
- JNDI trees
- security subsystems

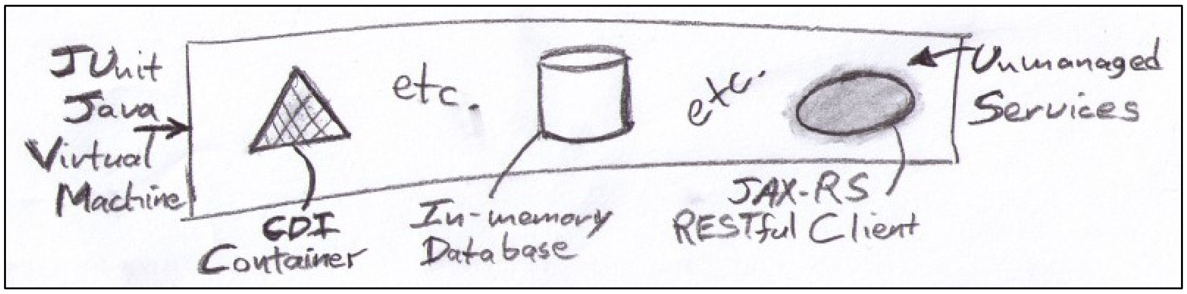


Fig. 3. JUnit Java Virtual Machine (JVM) containing an unmanaged CDI Container, In-memory Database and JAX-RS RESTful Client

The TUM approach nearly matches the historical performance advantage that testing with mock resources has enjoyed *vis-à-vis* tests using a managed Java application server. Using unmanaged real services avoids the performance penalty of leaving the JVM (*e.g.*, to make an application server trip).

High performance testing encourages frequent execution of the complete JUnit suite. Accordingly, the TUM approach enables test execution without requiring network connectivity, thus mitigating one of the many challenges of globally sourced anywhere/anytime development. Mock resources are used as a last resort.

5. PED PRINCIPLES FOR HARMONIZING ENTERPRISE COMMUNICATION

Principles 7 through 10 are relevant for harmonizing enterprise communication:

- Pattern Collection Publication
- Adaptable Technology Wrapper (ATW)
- Reference Application
- Instructor Guided Training

These are discussed in turn below.

5.1 Pattern Collection Publication

Principle 7. *When promoting a shared Pattern Language, publish a pattern collection. The project teams throughout the enterprise will have a starting point for adopting their own Pattern Palette.*

In experience with "the Company," we have seen that communication among diverse teams has been harmonized by: an empowered center of excellence, publishing a "pattern collection," and socializing the Pattern Language.

"The Company" has supported a Java Center of Excellence (JCoE) since 2002, when it adopted the Java platform for enterprise applications. The JCoE's official charter included application consulting and framework development. Over time, it has become a nexus for the germination, maturation and dissemination of an enterprise Pattern Language.

One ancillary function of the JCoE is knowledge management, including the collaborative authorship and online publication of recommended patterns and practices. The genesis of "the Company" pattern collection was originally labeled as the enterprise application architecture. The document evolved to include patterns addressing design and implementation concerns.

The original motivation for one-off patterns was to bring efficiency to technical communication between team members on opposite sides of the planet. We began to view the expanding lexicon of patterns published by the JCoE as key components of an enterprise Pattern Language, thanks to our rather recent engagement with the PLoP[16] community.

Socialization and enrichment of the Pattern Language is ongoing, spread through conversation between the JCoE and project teams. Those conversations are two-way: while the JCoE disseminates recommendations based on prior learning, it also aggregates new knowledge by "harvesting" learning from individuals and teams with which it interacts.

5.2 Adaptable Technology Wrapper (ATW)

Principle 8. *When aligning an API with a Pattern Language consider using adaptable technology wrappers as a way to increase programming consistency regardless of technology choices.*

Based on experience with "the Company," an "adaptable technology wrapper" (ATW) can: align Application Programming Interfaces (API) with chosen technologies, provide pattern enforcement, limit API complexity via filtering, reduce duplicate code, assist with JUnit testing, and provide feedback of the runtime execution of a chosen technology.

Perhaps the earliest widespread example of intentionally aligning an API with a pattern collection was provided by the Spring framework. In the following Javadoc example, the *@Repository* annotation makes the alignment clear by stating the origin of the REPOSITORY pattern:[17]

```
@Target(value=TYPE)
@Retention(value=RUNTIME)
@Documented
```

[16] The centerpiece of the PLoP community is their face-to-face conferences (http://www.hillside.net/conferences). In addition, there is a LinkedIn group entitled "Pattern Languages of Programs."

[17] http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Repository.html

```
@Component
public @interface Repository
```

Indicates that an annotated class is a "Repository", originally defined by Domain-Driven Design (Evans, 2003) as "a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects."

A *technology* wrapper may be understood by analogy; instead of wrapping a single Java object (class) a technology wrapper wraps an entire API. A PED technology wrapper aligns an API with a Pattern Language.

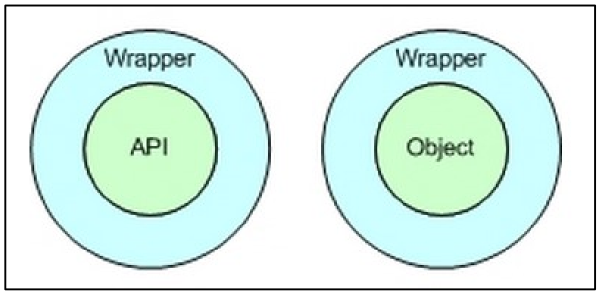
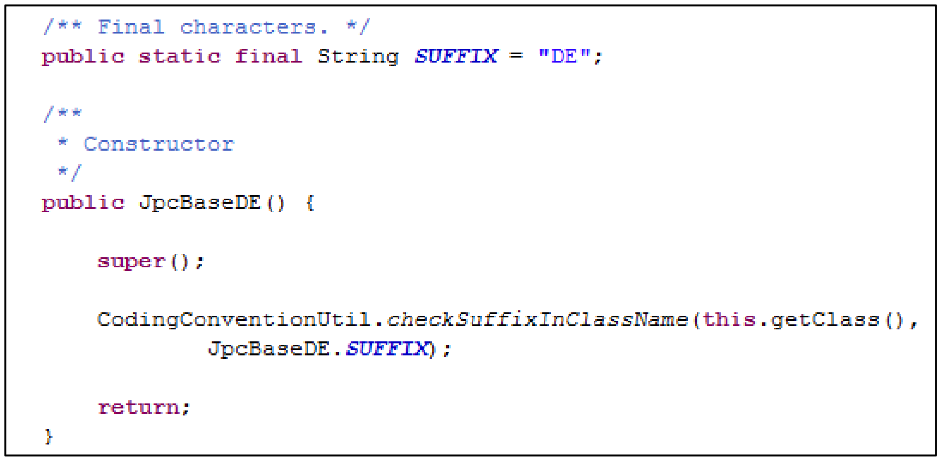


Fig. 4. An API Wrapper is Analogous to a Java Object Wrapper

In addition to pattern alignment, a creative wrapper author can implement pattern enforcement. For example, at "the Company" we have used a coding convention runtime exception if a concrete class name lacks the published suffix from the pattern collection. The following source code is from the open source jPED project for enforcing the DOMAIN ENTITY (DE) pattern suffix:

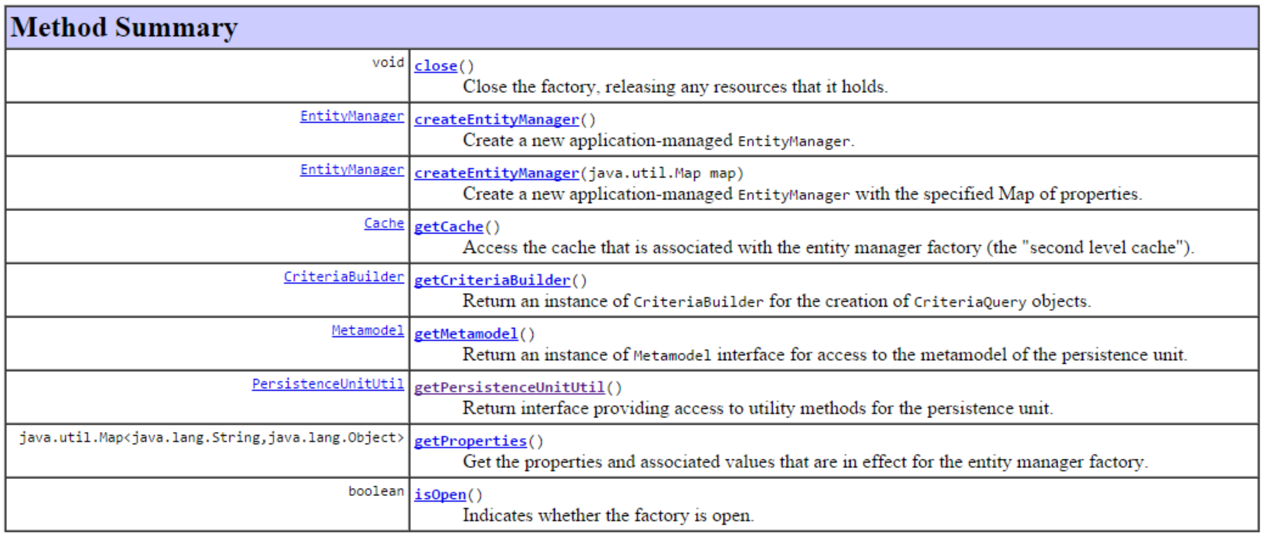


```
/** Final characters. */
public static final String SUFFIX = "DE";

/**
 * Constructor
 */
public JpcBaseDE() {

    super();

    CodingConventionUtil.checkSuffixInClassName(this.getClass(),
            JpcBaseDE.SUFFIX);

    return;
}
```

Fig. 5. jPED Source Code for Enforcing the Domain Entity (DE) suffix

The 2013 version of *Microsoft Word* for *Windows* has 39,900 entries in the Help system. If you studied one of those entries per day it would take over 109 years to learn all the features of this one piece of software! *Word* is hardly unique; many technologies are feature-bloated beyond an ability to learn them quickly. An ATW can limit API complexity by filtering out seldom used features that don't align with the pattern collection.

As an example, consider the following diagram listing the nine methods of the API for the JPA EntityManagerFactory class:[18]



Method Summary	
void	close() Close the factory, releasing any resources that it holds.
EntityManager	createEntityManager() Create a new application-managed EntityManager.
EntityManager	createEntityManager(java.util.Map map) Create a new application-managed EntityManager with the specified Map of properties.
Cache	getCache() Access the cache that is associated with the entity manager factory (the "second level cache").
CriteriaBuilder	getCriteriaBuilder() Return an instance of CriteriaBuilder for the creation of CriteriaQuery objects.
Metamodel	getMetamodel() Return an instance of Metamodel interface for access to the metamodel of the persistence unit.
PersistenceUnitUtil	getPersistenceUnitUtil() Return interface providing access to utility methods for the persistence unit.
java.util.Map<java.lang.String,java.lang.Object>	getProperties() Get the properties and associated values that are in effect for the entity manager factory.
boolean	isOpen() Indicates whether the factory is open.

Fig. 6. API for the JPA EntityManagerFactory Class

There is an opportunity to reduce the number of methods in this class, assuming that the EntityManagerFactory remains durable for the life of the application. jPED eliminates the need to invoke the `close()` and `isOpen()` methods. This illustrates the point that an ATW can filter API verbosity.

Copy-and-paste is perhaps the most obvious practice yielding duplicate code. Another form of duplication derives from using boilerplate code.[19] For example, two or more occurrences of boilerplate code may perform virtually identical functions while addressing different user stories. Duplication is sometimes a response to the demands of a verbosity-inducing API. That is, some APIs are poorly designed for testability or maintainability although the technology may be very useful.

The numerous tools and algorithms available for detection of duplicate—and wasteful—code attest to the widespread recognition of this hazard.[20] We learned early on with Todd's team that by wrapping two new (new to "the Company") technologies,[21] we could realize a significant reduction in source code operations required to implement and maintain boilerplate code.

It has been interesting to observe that as ATWs become more powerful—powerful in that operations accomplish greater amounts of work—the JUnit test code becomes more verbose by comparison. Tests requiring excessive lines of code constitute a test code "smell" indicative of an overlooked opportunity to make effective use of an ATW.

[18] http://docs.oracle.com/javaee/6/api/javax/persistence/EntityManagerFactory.html
[19] In computer programming, boilerplate code or boilerplate is the sections of code that have to be included in many places with little or no alteration.
[20] "Sonargraph-Explorer comes with a powerful duplicate code detection algorithm." https://www.hello2morrow.com/products/sonargraph/explorer.
[21] In this case, Struts and Toplink.

An ATW can include “assert” methods that dramatically ease JUnit test design and construction complexity in combination with the previously articulated TUM approach. The source code example in Figure 7 is from the open source jPED wrapper enabling asserting JSR-303 validations with a single line of test code.

```java
/**
 * This method performs validation using a validator plugin.
 *
 * @param validatorRM
 * @param expectedErrors
 * @param validatorPO
 */
public void assertValidate(final JpcBaseValidatorRM validatorRM,
        final int expectedErrors, final JpcBaseValidatorPO validatorPO) {

    final JpcErrorMessagesTBD errorMessages = validatorRM
            .validate(validatorPO);

    if (errorMessages.isEmpty()) {
        return;
    }

    for (final JpcErrorMessageTBD messageCO : errorMessages
            .getErrorMessageList()) {

        System.out.println("Validation Message: "
                + messageCO.getErrorDescription());
    }

    Assert.assertEquals(expectedErrors, errorMessages.getErrorMessageList()
            .size());
}
```

Fig. 7. jPED Wrapper Enabling Asserting JSR-303 Validations

jPED wrappers also provide integrated logging as feedback from the runtime execution of native technologies.[22] In recent years, some newer technologies such as JSF (with its *Development Project Stage*) have begun to incorporate using additional runtime logging as development feedback. The jPED code example in Figure 8 establishes four discrete modes that augment exception, performance, lifecycle, and trace logging:

```java
/** XML property key - normally true to log exceptions. */
public static final String JPC_LOGGER_EXCEPTION_MODE = "logger.exception.mode";

/** XML property key - normally false except for development/testing/tuning. */
public static final String JPC_LOGGER_PERFORMANCE_MODE = "logger.performance.mode";

/** XML property key - normally false except for development/testing. */
public static final String JPC_LOGGER_LIFECYCLE_MODE = "logger.lifecycle.mode";

/** XML property key - normally false except for debugging. */
public static final String JPC_LOGGER_TRACE_MODE = "logger.trace.mode";
```

Fig. 8. Four Discrete Modes of jPED Logging

These logging modes enable explicit invocation for production and test code during development, continuous integration and production execution. jPED also incorporates multicore asynchronous logging, enticing developers to benefit from increased logging verbosity without a performance penalty.

[22] A more technical elaboration of these benefits is presented at http://pedCentral.com in the context of JPA 2.

At least two additional benefits accrue to this principle of wrapping preferred technology APIs. One relates to software flexibility:

> Wrapping third-party APIs is a best practice. When you wrap a third-party API, you minimize your dependencies upon it: You can choose to move to a different library in the future without much penalty. One final advantage of wrapping is that you aren't tied to a particular vendor's API design choices. You can define an API that you feel comfortable with. (Feathers 2009, 109)

A second benefit can derive from decoupling a technology and its API: insulation from security vulnerabilities. As previously mentioned, in 2006 "the Company" began work on an ATW around Struts intended to enhance convenience and safety. In 2013, a Struts vulnerability was discovered and options for remediation subsequently emerged within the open source community. Because many teams had elected to use the Struts ATW, "the Company" was able to apply remediation across six continents with a simple JAR update.

5.3 Reference Application

> Principle 9. *When promoting application architecture, design or implementation recommendations, establish a reference application as an alternative to comprehensive documentation.*

Based on experience with "the Company," we submit that a "reference application" such as PED's "Learning eXample Implementation" (LeXI) can: demonstrate ATWs in the context of a deployable application, promote synergy among distributed teams, and support informal discovery as an alternative to extensive written documentation. Several years in the making, the LeXI now showcases the current PED pattern collection while highlighting its aligned APIs. It is viewable and downloadable.

In late 2006, the CEO championed collaborating as "one team." We were prompted to ask, how could we globalize a common set of patterns to hundreds of developers on project teams spread across six continents? How do we foster the broad adoption of a pattern mindset among dispersed individuals?

Norms governing "teamful" behavior are widely accepted for project teams of "seven-plus-or-minus-two." While individuals have one set of commitments as part of small self-organizing work groups, could they have a different set when they regard themselves as members of an enterprise team? When conditions require the scaling of teamful behaviors to 70 ± 20, or even 700 ± 200 developers, a software engineering safety hazard emerges: that of divided allegiance.

A reference application can help mitigate this hazard; when multiple teams compare their code against a prevailing standard, that process tends to "normalize" designs and implementations. A reference application helps pave the way for interoperating as that elusive "one [global] team."

Beginning in 2002 with the adoption of Java, there was an initial bias for written English-language documentation as a tactic for communicating software design guidelines to the enterprise development community. There was continual angst over whether or not code examples should be maintained within this documentation, and whether the code examples should be use-case independent or share a similar context.

Comprehensive written documentation never seems to resolve the age-old dilemma: its statements are either too abstract to provide implementation guidance, or too specific to be maintainable. If written documentation becomes subordinate to a learning example implementation, it can focus on guiding the learner through the structure of the application and the manifest patterns, within the context of a user story that grows increasingly familiar with exploration. Additionally, investment in authoring comprehensive documentation in the form of software design guidebooks depreciates rapidly with technology obsolescence.

As a technical communications tool, Java and its allied languages (CSS, JavaScript, XML, *etc.*) offers a degree of precision unmatched by natural languages. When promoting enterprise team behaviors among developers, why not use *their lingua franca*? Others have arrived at a similar conclusion:

> One of the most requested aids to coming up to speed on DDD [Domain Driven Design] has been a running example application. Starting from a simple set of functions and a model … we have built a running application with which to demonstrate a practical implementation of the building block patterns….[23]

5.4 Instructor Guided Training

> Principle 10. *When introducing a reference application into the enterprise, consider offering hands-on face-to-face instructor guided training so that participants are comfortable using the reference application as a self-study resource.*

Based on our experience with "the Company," a reference application like LeXI is an effective enabler of independent learning—for a small minority of bright, self-motivated developers. In other words, it is limited in its appeal. To expand the reach, we introduced instructor guided training.

Offering training is a business decision, and companies differ on the value they perceive in it. We question whether the costs of developing, maintaining and delivering in-house technology-specific training programs are sustainable, due to the rapid pace of technology change. Having experienced the challenge of maintaining dynamic course content first-hand, we posit that teaching people *how* to use a reference application as a learning tool is a more sustainable model. Investment in the maturation of a working reference application should be prioritized, *because* it serves as a foundation for an effective training program that enables subsequent independent learning. Said another way: we believe in teaching the *how*, not the *what*.

We recommend offering formal Java pattern classes that are: instructor-guided, face-to-face, limited in duration and class size (*i.e.,* eight to twelve attendees), provide opportunities for hands-on practice at using a reference application, and impart a familiarity with the enterprise Pattern Language. Reliance on informal-only learning is inferior to this type of formal guidance for competency growth.

6. CONCLUSION

While the ten principles presented in this paper are abstractions describing a unique ten-year journey at a single successful large enterprise, it is tempting to ask whether these principles carry some degree of normative import. That is, can we confidently recommend that IT organizations immediately begin adopting some of the principles?

One note of caution is in order: some of the principles assert benefits that may not be generally applicable. To illustrate, consider Principle 10 (Instructor Guided Training). "The Company" introduced an instructor guided training curriculum in 2009, focused on communicating the Java patterns using a reference application similar to LExI. Between 2009 and 2014, several hundred developers took at least one of the six courses offered, with a large minority taking three or more. Despite receiving high ratings from course attendees gathered in post-class surveys, the long-term benefits of this training program have proven difficult to ascertain. It is more obvious when enterprise software is hampered by low code quality than when it is helped by a pattern-oriented approach to communication. At "the Company," low quality code, inflexible design, and indifference to testability persisted despite widespread pattern adoption.

On the other hand, the value of Principle 7 (Adaptable Technology Wrapper) is readily demonstrable, and the benefits recognized in the industry. In short, we suggest that some of the principles are, at this point, subject to revision in the light of future discoveries, while others are broadly applicable.

Our experience suggests that fundamental deficiencies in software engineering skills cannot be overcome by our approach; Pattern Enabled Development *presupposes* and does not *result in* the ability of developers to write clean, test-driven code. At several points, we alluded to the broadly

[23] http://dddsample.sourceforge.net.

“Agile” presuppositions of PED. For example, Principle 1 integrates Pattern First Thinking into the traditional red-green-factor TDD cycle. However, when project managers promote a “chilly climate” for JUnit testing, code quality suffers considerably, even if Pattern First Thinking is accepted.

Like any software engineering refinement, PED may be susceptible to inflated expectations. We do not expect that the PED principles will be effective without adapting them for your enterprise. Since the PED principles emerged from experiences with multiple teams we believe they may have transferability to teams in other business environments if applied thoughtfully.

Our hope is that we advance Java application construction by inspiring software engineers with Pattern First Thinking, enabling project teams with a Pattern Palette, and harmonizing enterprise communication with a Pattern Language.

APPENDIX: Communicating PED Patterns[24]

The selection of our PED patterns can be ascribed to their shared characteristics. Patterns are:

- i. Scoped to a single class (*e.g.*, INBOUND CONTROLLER), not multiple classes (*e.g.,* MVC)
- ii. Language-specific*, i.e.,* demonstrated in Java
- iii. Relatively few in number to facilitate learning and recollection
- iv. Focused on structural elements for simplicity
- v. Used/understood daily when coding working software
- vi. Readable within source code by class naming convention
- vii. Assigned an architectural context (*i.e.,* they play an architectural role)

Over the last decade, our approach to communicating patterns as a language has matured. Patterns are currently presented at a site called *PED Central* on the Web, with each pattern characterized as follows:

1. A brief definition of the pattern.
2. An inventory of its structural elements.
3. A class diagram representing a contextualized use of the pattern within working software.
4. Source code (*i.e.,* links to the Learning Example Implementation (LExI)) illustrating how the pattern is applied.[25]

PED Central currently documents approximately 20 single-class patterns. The following example pattern is taken from PED Central.

Example: The Domain Entity (DE) Pattern

Behavioral Definition

A DOMAIN ENTITY (DE) represents a physical entity with a unique identity.[26]

Structural Elements

- Extends a base DE class (inheritance) such as:
    - AuditDE
    - UuidDE
        - The lastUpdate field is used for both auditing and versioning
            - Versioning supports both optimistic locking and cache refresh
    - UuidNoAuditDE
- One concrete DE relates to a second DE typically via composition
- Contains a single field for identity (avoid composite primary keys)
- Implements two constructors and the identity field is immutable:
    - A no-argument constructor
    - A second constructor with the identity parameter
    - A 'getter' implemented for the identity field with no 'setter'
- Typically contains field annotations and accessor methods only[27]
- Implements the `equals()`, `hashcode()`, and `compareTo()` methods in a manner that avoids boilerplate code duplication
- Satisfies TEMPLATE METHODS as required

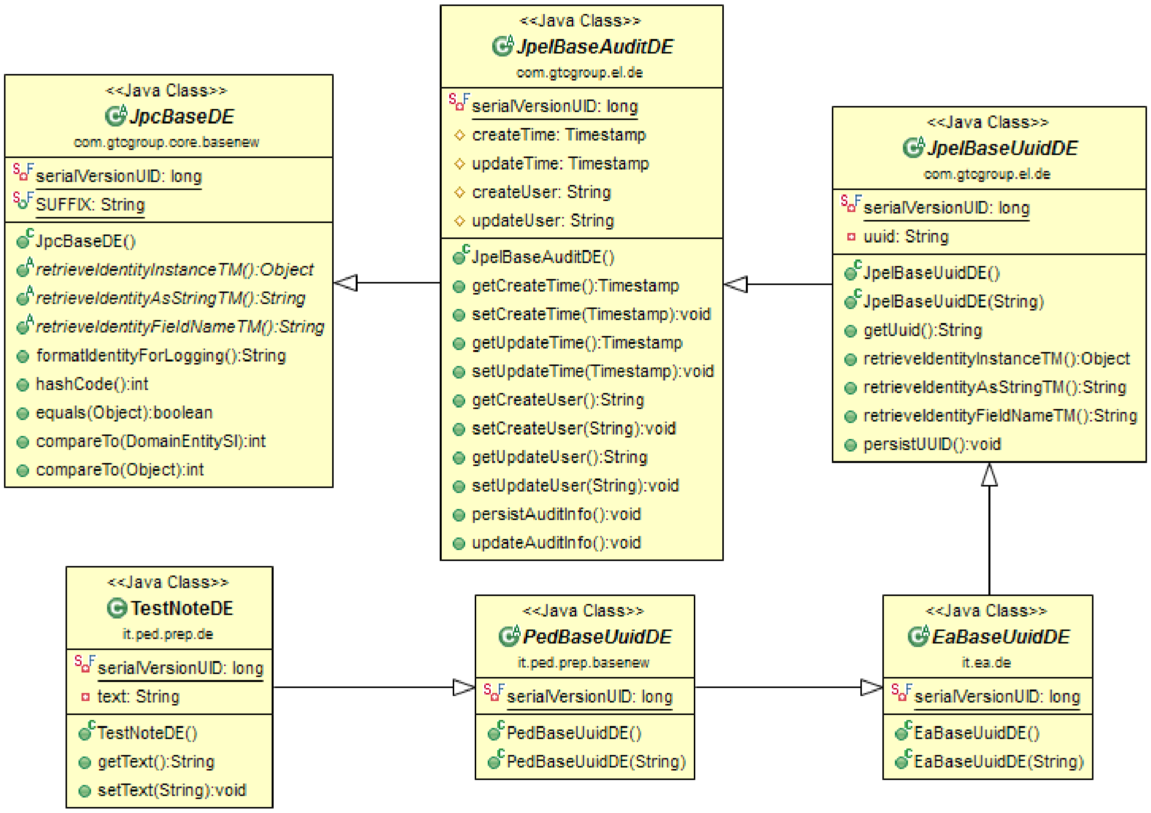
[24] PED includes mostly "single class" patterns, that is, patterns made simple because their scope is one class. PED has begun to addresses UI patterns, however these are beyond the scope of this paper.

[25] http://pedCentral.com.

[26] The PED DE pattern is loosely informed by Eric Evans' ENTITY: "an object fundamentally defined not by its attributes, but by a thread of continuity and identity." See Evans 2004, 512.

[27] Agreeing whether a DE should contain encapsulated behavior (beyond accessor methods) is a concern explicitly settled by a project team while adopting their own Pattern Palette.

Class Diagram



Source Code

This section of each pattern description contains links to relevant source code viewable as HTML pages.[28]

REFERENCES

Beck K. 2007. *Implementation Patterns*. Addison-Wesley.
Evans, E. 2004. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Pearson Education, Inc.
Feathers, M. 2009. Error Handling, In R. C. Martin, ed., *Clean Code*. Pearson Education, Inc., 103–112.
Fowler, M. 1998. After the Program Runs. In *Distributed Computing* (September).
Goncalves, A. 2012. Launching the NoMock Movement. http://antoniogoncalves.org/2012/11/27/launching-the-nomock-movement/.
Manns, M. L. and L. Rising. 2005. *Fearless Change: Patterns for Introducing New Ideas*. Addison-Wesley.
Martin, R. 2013. The Pragmatics of TDD. http://blog.8thlight.com/uncle-bob/2013/03/06/ThePragmaticsOfTDD.html.

[28] http://pedCentral.com