How Pattern Enabled Development Contributes to Software Safety Engineering

Marvin Toll and William R. Minto

Abstract

Pattern Enabled Development® (PED) is an approach to software application development intended for enabling business innovation and empowering project teams. Software safety engineering, or "*anzeneering*" is the discipline of protecting developers (and other constituencies) from hazardous code. We contend that the PED Principles advancing software flexibility concurrently contribute to safe code.

PED: A Brief Retrospective

In 2002, Java was adopted as the standard enterprise platform for in-house application development at a large automotive manufacturing company. In subsequent years, many applications would migrate from Perl/CGI to Java. Marvin Toll joined one such co-located 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.[1] Fortunately, the team included capable individuals that worked well together.

Could a novel approach compensate for the lack of requisite Java knowledge? Would the desired business innovations succeed if the team committed to adopting simple patterns prior to coding? If so, which ones? Would example pattern implementations be effective for communicating to a developer audience without a Java background? How much Test Driven Development (TDD) could a team absorb if they

[1] Hiring new Java developers without relevant business domain expertise would not have improved the situation.

had not learned the programming language? Would a pattern language foster team confidence and a sense of empowerment?

A significant portion of the project's success[2] can be traced to what would 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."[3] Thus launched the "PED Journey," a decade-long pursuit to validate, extend and refine lessons first learned from a single project team.

WHAT IS "ANZENEERING?"

The term "*anzeneering*" was coined by Joshua Kerievsky, CEO of Industrial Logic, to identify safety practices pertaining to software engineering akin to those used in manufacturing and construction.[4] In these two fields, firms like Alcoa have demonstrated leadership with the introduction of practices and cultural norms that protect workers. Why should software developers not be equipped for safety? "*Anzen*" (Japanese for "safety") is not meant metaphorically in the software context—the practices recommended by *anzen* have measurable implications for protecting such things as reputations, budgets, relationships, worker health, and market position.

Would a pattern centric development approach advancing software flexibility concurrently contribute to "safe" code? That is, could an association between safe code and software flexibility be nurtured in the form of a coding principle?

The opposite of safe code is hazardous code—software that is poorly designed, highly complex, and deeply defective. For example, hazardous code might be developed in an environment lacking even basic Agile practices like automated builds and automated tests. In the following sections, we identify several of the hazards encountered in enterprise application development, along with the corresponding PED Principle providing protection, discussed in the context of why the principle emerged during the "PED Journey."

[2] In this case, "success" meant an uninterrupted flow of delivered business value.

[3] Todd Hall, private correspondence, June 30, 2014.

[4] See Joshua Kerievsky, "Anzeneering," retrieved from http://www.industriallogic.com/blog/anzeneering/, June 19, 2014.

SOME HAZARDS AND THE PED PRINCIPLES PROVIDING PROTECTION[5]

PED is based on a set of principles; the names within {curly braces} of the Principle section are the names of the PED Principles.

1. Faux Flexibility

Hazard

To achieve software flexibility, a developer might have in mind a pattern that is overly complex for its immediate purpose. Did the developer assume a relatively complex pattern was needed for application extensibility? Joshua Kerievsky underscores this risk:

> … [over] time, the power of patterns led me to lose sight of simpler ways to write code. After learning that there were two or three different ways to do a calculation, I'd immediately race towards implementing the Strategy pattern, when, in fact, a simple conditional expression would have been easier and faster to program—a perfectly sufficient solution.[6]

Complexity can convey an illusory sense of flexible software, *i.e., faux* flexibility, where the developer spends additional time on testing and implementation (tantamount to over-engineering), with the imagined benefits never coming to fruition.

PED Journey

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." While this serendipitous discovery was made with a technically limited project team, we later found that more seasoned developers benefit from the adoption of simple patterns. In addition, we discovered that adopting simple patterns early during JUnit testing contributed to sustaining flexibility. Our conclusion: to achieve *genuine* software flexibility, a developer should begin with the simplest viable pattern for a given purpose.

[5] See http://pedCentral.com for an articulation of the PED Principles. Throughout this paper, we assume Java as the implementation language. It is tantalizing to suppose that these principles could apply to other languages, however we have insufficient experience working with them to make the case.

[6] Joshua Kerievsky, *Refactoring to Patterns* (Addison-Wesley, 2005), 2.

In recent years, we have observed that enthusiasm for the power of TDD has drawn some towards embracing an ever-expanding number of test tools (e.g. Mockito) and an increasing number of test instructions per functional (production) instruction. For some, the tendency to over-engineer has migrated to test code.

Principle

> {First Principle} **In the beginning, when writing your first line of test code, keep in mind a *simple* pattern. In the end, your customer will have more flexible software.**

"Simplicity can encourage flexibility."[7] "… [T]he flexibility of simplicity and extensive tests is more effective than the flexibility offered by speculative design."[8] The First Principle of PED presumes "software flexibility" is an attribute of computer applications valued by business owners for contributing to business agility. We define software flexibility as:

> [t]he quality attribute of a working software application 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.[9]

There is an additional presumption behind the First Principle, that is, a commitment to a "pattern first" mindset. Beck, writing in 2007, seems to affirm this commitment:

> 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*.[10] 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

[7] Kent Beck, *Implementation Patterns* (Addison-Wesley, 2007), 13.

[8] Kent Beck, *ibid.*, 12.

[9] "What is PED?" Retrieved from http://pedCentral.com/what-is-ped/.

[10] Italics added.

> 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.[11]

2. Inconsistency

Hazard

Software lacking consistency may function properly, but grows increasing inflexible with each release. This risk becomes more salient in time-boxed delivery scenarios (with iterations, releases, *etc*.) where the focus on delivering incremental business value trumps reflection on overall application architecture and design. Delivery pressures could result in:

- Patterns without explicit definitions to support effective application architecture/design
- Uncoordinated pattern selection, resulting in a pattern palette[12] lacking in coherence

These hazards can result in structural and semantic inconsistencies in the code base, leading to, for example, obscuring the relevant similarities that would naturally bind modules together. Without compensating mechanisms, entropy may rob a team of opportunities to realize architectural, design and implementation consistency.

PED Journey

We realized that some patterns carry architectural implications, while others do not. For example, an INBOUND CONTROLLER contains front-end specific logic and belongs to the inbound interface layer, whereas a HELPER could "reside" anywhere in the code base. Therefore, a team pattern palette should be established in concert with the application architecture.

Based on our experience, teams should strive for a palette robust enough so that approximately 80% of the concrete application classes correspond to patterns on the

[11] Kent Beck, *ibid.*, 20.

[12] The metaphor of a pattern "palette" is intended to suggest an adaptive, non-prescriptive approach to establishing a collection of patterns for use on a particular project.

palette. That is, 80% of the classes are structurally consistent with one of the palette patterns. We refer to this percentage as "pattern coverage."[13]

Principle

> {Pattern Palette} **When forming a team, reach consensus on a coherent "pattern palette." Avoid "freezing" the palette at a particular time, leaving the conversation open for palette modification as needed.**

Pattern selection is a design exercise and should be a team effort. A satisfactory pattern palette provides "just enough" (and no more) up-front design to promote long-term flexibility. We submit that all application development teams benefit from conversations aimed at selecting *their own* pattern palette from the PED collections, or from other sources.[14] The patterns included in the palette should carry sufficient structural weight to back a robust architecture and design, and as a whole, represent a coherent set.

3. Unteamful Pattern Selection

Hazard

When individuals or programming pairs are singularly focused on a user story, the "tyranny of the urgent" may supplant team deliberation. That is, implementation may be *ad hoc*, with or without pattern consciousness.

If team pattern selection conversations do occur, more extroverted or more influential team members may have their opinions adopted, over the unvoiced objections of the team's more introverted members.

A third risk, "groupthink", can happen when premature convergence on a pattern palette occurs without adequate thoughtful deliberation. A superficial understanding of how to apply the patterns in different contexts can result.

In each case, unteamful behavior threatens the establishment of a pattern palette that is coherent and structurally oriented.

[13] See page 8 for elaboration on the principle of Pattern Coverage.

[14] PED provides three collections of patterns: Class, User Interface and Method. We are not arguing for the PED collections, specifically. Our bias is, of course, that PED be consulted as a starting point, but other collections of patterns may provide additional inspiration for a team selecting its palette.

PED Journey

Over the past ten years, we have seen how helpful it has been for a team member to play the role of informal "pattern champion"—to facilitate conversations focused on consensus-formation. A related observation: the technical leader on a project team needs to be supportive of the pattern champion, or assume that role themselves.

These observations prompted the quest for a metaphor that would suggest an adaptive, non-prescriptive approach to establishing a collection of patterns for use on a particular project. The metaphor of the artist's palette seemed particularly suggestive.

Principle

> {Culture of Deliberation} **Pattern-selection should be well-considered, and discussions inclusive of all voices on the project team.**

The pattern selection conversation is an opportunity to provide the team with a "psychologically safe" forum early in the project where contrary opinions are welcome. It also helps establish a culture of inclusion, *i.e.*, the presumption that every team member's standpoint has potential value. *Anzeneering* rejects the assumption that silence is implicit consent. [15]

Collaborative problem-solving spaces have built-in mechanisms for filtering complexity; by and large, people prefer the simple to the complex.[16] The palette that emerges will tend to include patterns optimized for structural economy. This encourages developers to remain "on-palette" during coding. Of course, any application will inevitably encounter customer features requiring "off-palette" solutions. However, in a well-functioning team, developers are less inclined to diverge from established norms, including norms concerning what constitutes a simple *vs*. a complex solution.

4. Inscrutable Code

Hazard

In a global enterprise environment where people are onboarded at project inception and later released to other teams, a lack of shared experiences is typical, and the readability of code can vary widely. In addition, the intelligibility of source may suffer when the code is written for a compiler rather than for other human beings.

[15] See "Anzeneering," retrieved from http://www.industriallogic.com/blog/anzeneering/, July 2, 2014.
[16] From the standpoint of Occam's Razor, the simpler solution is preferable to the complex.

PED Journey

Who is the audience? Increasingly we witness concepts such as global sourcing impede the co-located ideal represented by Todd's team. It is no longer imaginable who may maintain an application in five or ten years, what their native language may be, or which cultural context shapes their technical and business view. A team adage, such as "Write Once, Read Many," is a useful reminder that the shelf life of application code is often longer than first imagined. It may have many readers beyond the original development team.

In addition, the technique of "Pattern Encoding," in which an applied pattern is denoted by suffix, is a convention for enhancing readability. The PED pattern collections demonstrate using a suffix for this communication.[17]

Principle

> {Software Understandability} **When authoring a narrative in Java, imagine a global reading audience. Such a narrative—your code—will be understood beyond a pair programmer and immediate project team.**

While we appreciate the value of the standard practices often employed to facilitate understandable code (coding conventions, pair programming, team code reviews, *etc*.), these arguably are inadequate for the long-term global readability of enterprise application code.

5. *Tight Coupling*

Hazard

Tight coupling is a structural, not a behavioral, hazard. It can take two forms: (a) individual modules that grow unmanageably large, and (b) multiple integration points between modules. In both cases, undisciplined emergence contributes to inter-module dependencies.

PED Journey

In 2009, we were introduced to Alexander von Zitzewitz[18] and his taxonomy of structural metrics (*e.g*., quantifying cyclic dependencies). This encounter led to the idea

[17] For example, the NoteDE class is presumed to implement the Domain Entity pattern based on its suffix 'DE.'

[18] Alexander is co-founder and managing director of hello2morrow and CEO of the US subsidiary.

that application architecture for organizing source code into dependency groupings may be use-case-agnostic. That is, structure can be analyzed independent of an application domain.

At the class level, Java code has structural and behavioral (or functional) characteristics. These two characteristics are often conflated when teams perform TDD. Said another way, TDD emphasizes behavior, with code coverage as a key metric. In addition to measuring the extent that JUnit tests cover functional code, we can also measure the extent of (structural) pattern usage, yielding a pattern coverage metric.

Beginning in 2006, Todd's team had begun experimenting with a rudimentary JUnit-based tool[19] while seeking opportunities for independent structural testing.

Principle

> {Pattern Coverage} **When TDD provides insufficient attention to structure, focus on pattern coverage as well.**

PED's flexible technology wrapper (jPED)[20] includes an innovative mechanism 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." The pattern coverage tool iterates

[19] This refers to Marvin Toll's open source TestUtil 2.0 project, circa 2006.

[20] A flexible technology wrapper refers to a layer of code abstracting an API for convenience and reducing duplication. The PED Website (http://pedcentral.com) provides a reference implementation (jPED).

through the classes in the code base and verifies structural compliance for the elements of each pattern on the team-selected palette. Upon completion, jPED reports summary metrics such as what percentage of the classes represent themselves as patterns from the palette (based on suffix) and what percentage do not.

6. Duplicate Code

Hazard

The numerous tools and algorithms available for detection of duplicate—and wasteful—code attest to the widespread recognition of this hazard.[21]

Copy-and-paste is perhaps the most obvious practice yielding duplicate code. Another form of duplication derives from using boilerplate code.[22] For example, two or more occurrences of boilerplate code may perform virtually identical functions while addressing different user stories. The duplication is sometimes a response to the demands of a verbosity-inducing Application Programming Interface (API).

PED Journey

We learned early on with Todd's team that by wrapping the two new (new to our enterprise) technologies,[23] we could realize several benefits, including the reduction of redundant boilerplate code (along with pattern adoption, learning and usage, and consistent source code semantics.)

This approach—wrapping preferred technologies' APIs—has additional virtues related to software flexibility that were originally unrecognized, but were later explicated by Michael Feathers.

> 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.[24]

---

[21] "Sonargraph-Explorer comes with a powerful duplicate code detection algorithm," retrieved from https://www.hello2morrow.com/products/sonargraph/explorer, August 9, 2014.

[22] "…boilerplate code or boilerplate is the sections of code that have to be included in many places with little or no alteration." Retrieved from http://en.wikipedia.org/wiki/Boilerplate_code, August 20, 2014.

[23] In this case, Struts and Toplink.

[24] Michael Feathers, "Error Handling" in Robert C. Martin, ed., *Clean Code* (Pearson Education, Inc., 2009), 109.

Principle

> {Flexible Technology Wrapper} **When coding to an existing API, consider whether a further layer of abstraction will reduce waste and thereby diminish the testing and maintenance burdens.**

Team synergy is amplified when a flexible technology wrapper is intentionally designed to leverage a pattern collection.

7. Lack of Enterprise Scalability

Hazard

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 new hazard emerges: that of divided allegiance.

Anzeneering, as a protection discipline, identifies five constituencies at risk from hazardous code and made memorable by the acronym MUMPS (Makers, Users, Managers, Purchasers, Stakeholders). When the use of patterns does not scale well, these people get hurt.[25]

PED Journey

The PED journey took place within a broader context. During this period our highly regarded CEO championed an enterprise appreciation and belief in collaborating as "one [global] team."

So how would one extend a pattern foundation, first realized by Todd's team, to hundreds of enterprise developers on project teams spread across five continents? How can we foster the broad adoption of a pattern mindset among dispersed individuals?

Beginning in 2002 with the adoption of Java, there was an initial bias for written English-language documentation as a way to communicate to the enterprise development community. After several years we began to contemplate if distributed project teams

[25] A further list is embedded within the Hillside Group mission, "to improve the quality of life of everyone who uses, builds, and encounters software systems—users, developers, managers, owners, educators, students and society as a whole." Retrieved from http://www.hillside.net/home/mission-statement, August 20, 2014.

could benefit from a learning example implementation. Such an example could showcase, among other things, how a flexible technology wrapper could be used.

Others have arrived at similar conclusions.

> 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…. [26]

In addition, we came to the realization that expecting modestly skilled developers to learn via independent study was unrealistic. There appeared to be a need for formal training which:

- Was initially provided as hands-on, lab-based instructor-led classes,
- Challenged participants to extend the learning example implementation's source code to support an additional use case within the same business domain, and
- Was focused on implementing patterns in enterprise software

These discoveries lead us to our last principle:

Principle

> {Enterprise Scalability} **Establish a learning example implementation useable as the centerpiece for formal developer training.**

The Java platform languages provide a globally understood mechanism for communicating specific, contextualized guidance to Java developers. When promoting enterprise team behaviors with developers, doesn't it make sense to use *their* language?

CONCLUSION

PED, like any software engineering refinement, may be subject to both inflated expectations on the part of developers using it, and under-delivery by those promoting it. The situations where PED seems least beneficial include:

- A below average team, including a lack of problem-solving skills

[26] Domain Driven Design. Retrieved from http://dddsample.sourceforge.net, June 20, 2014.

- A high-performance co-located team that thrives on informal collaboration

On the contrary, the situations where PED seems most beneficial include:

- Teams with communication-challenged members
- Geographically dispersed teams, especially those distributed across time zones
- Teams with talented younger developers lacking experience with the complexities of corporate application development

PED does not and cannot compensate for teams with minimal Agile practice skills. PED can help a team progress beyond basic *software* engineering into the world of *safety* engineering.

Appendix: Communicating Patterns

'Seven Habits' of Highly Effective Single-Class Patterns[27]

The successful adoption of 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 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)

Communicating Single-Class Patterns

Over the last decade, the approach to communicating patterns has matured. Currently, the patterns are presented 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. [28]

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

[27] PED includes mostly "single class" patterns, that is, patterns whose scope is one class. PED also addresses UI patterns and method patterns, however these are beyond the scope of this paper.

[28] http://patternenabled.com/source/html/.

Example: The Domain Entity (DE) Pattern

1. Definition

A Domain Entity (DE) represents a physical entity with a unique identity.[29]

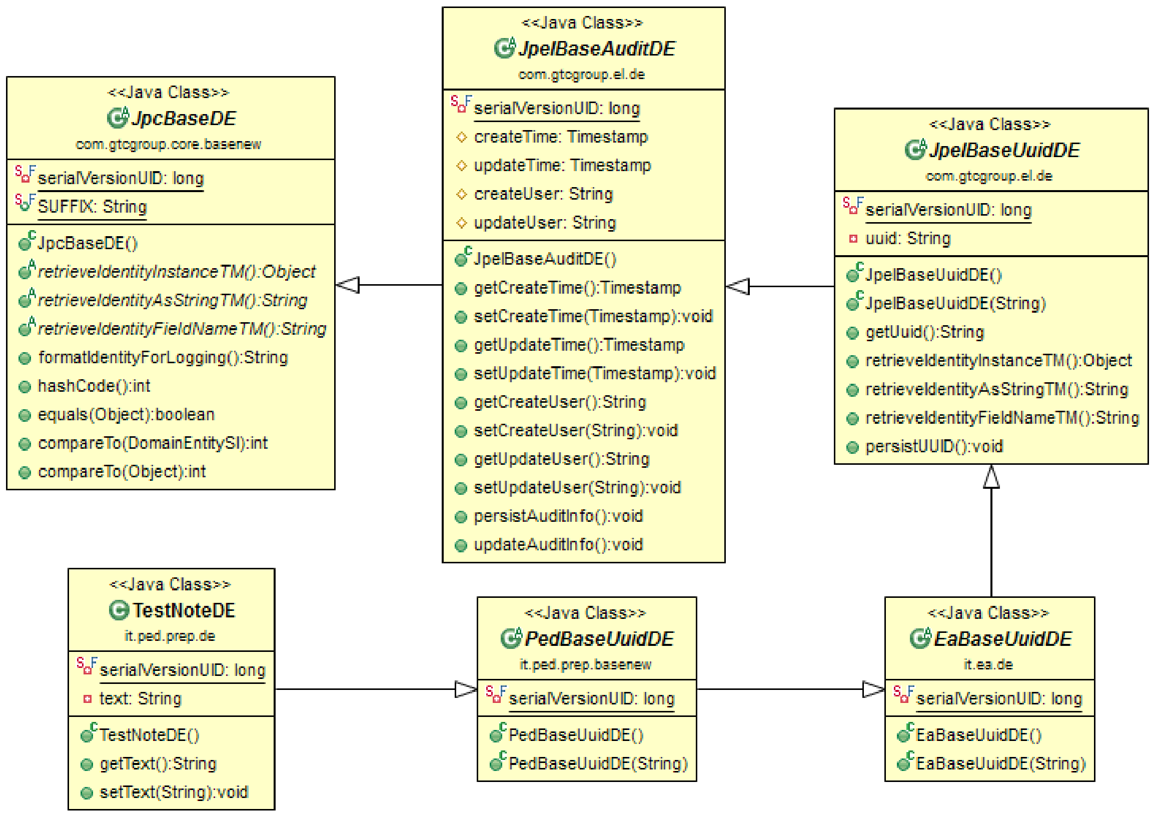
2. Structural Elements

- 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 DE relates to a second DE typically via composition
- Contains a single field for identity (no composite primary keys)
- Implements two constructors and at least one immutable identity field:
    - 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[30]
- Implements the *equals()*, *hashcode()*, and *compareTo()* methods
- Satisfies Template Methods as required

[29] 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 Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Pearson Education, Inc., 2004), 512.

[30] Agreeing whether a DE should contain encapsulated behavior (beyond accessor methods) is a significant concern explicitly settled by an application development team formalizing their own pattern palette.

3. Class Diagram



4. Source Code

This section of each pattern page contains links to relevant source code viewable as HTML pages.[31] DOMAIN ENTITY source can be found at http://patternenabled.com/architecture/application-tier/shareable/domain-entity-de/.

[31] http://patternenabled.com/source/html