Coherent Generational Design: A Pattern Language

JUAN R. REZA, Research Consultant.

A set of design patterns describing problems and solutions in multi-generation software systems is presented. A new generation of a system is seen when a software system goes through a quantum leap, or a major new release, or technology shift. A central design pattern called Generatrix seeks to promote cohesion from one generation of a system to the next by explicitly capturing design abstractions and their intent and part of the applicable design. It offers a simple vocabulary to help communication among developers, business analysts, and other stakeholders. The purpose of the supporting design patterns should be familiar to developers. The patterns take concepts that are usually very informal, or known by other names and brings them together in one pattern language. The artifacts of the language can become an integral part of the design and help make a more coherent transition to future next generations of the system. The concepts presented are framed in terms of UML and the Java programming language for convenience and do not imply a limitation of the patterns to any particular language.

Categories and Subject Descriptors: **D.2.7 [Distribution, Maintenance, and Enhancement]:** Restructuring, reverse engineering, and reengineering; **D.2.9 [Management]**: Software configuration management; **D.2.13 [Reusable Software]** : Reuse models.

General Terms: Design Patterns

Additional Key Words and Phrases: Generatrix, design abstraction, generation sensitivity, design provenance, maintainability, strategic design pattern, source control, versioning, software evolution.

ACM Reference Format:

Reza, J. 2012. Coherent Generational Design: A Pattern Language (PLoP), Tucson, AZ (October 2012), 26 pages.

1 INTRODUCTION

When planning a major new release of a software system, or next generation, developers have an opportunity to use the existing code base and make some architectural changes. The changes may solve chronic problems of the existing design by reorganizing hierarchies, change languages, and add or eliminate framework.

The developers may decouple the new components from their original versions when names and hierarchies are changed. Although this clean-up is beneficial, it can also result in loss of design knowledge. A class that grew to thousands of lines may now be split into a number of more concern-focused units. The builder of a complex object can be split off into a factory. An implied dependency between remotely collaborating objects may be known to the original developer and not apparent to future personnel. Refactoring for efficiency might obscure a previously more-readable algorithm.

Simply asking the developers to design for maintainability does not provide specific techniques for doing so. The Generatrix design pattern describes a way of retaining design knowledge in a form that guides and is useful to design of the next generation of a software system. The other design patterns support the application of the Generatrix pattern and provide a way of thinking about a system as a multi-generation product.

1.1 The Issues

An important issue in large MIS-based institutions is the need to reproduce results from past generations of the system. Maintenance can degrade the capability to reproduce results in past generations (by re-generating, not just by retrieving results). This problem is mitigated in part by data provenance. Laws are enacted to regulate financial institutions, leading to large, abrupt demands on their MIS systems. In a comparable way, science journals have withdrawn research papers after discovering that research results

This research is in partial fulfillment of the recertification requirements of the author's CSDP credential.

Author's address: J. R. Reza, 2001 S Magnolia Dr., Tallahassee, FL 32301; email: Juan.Reza@ieee.org

could not be reproduced in later generations of studies. The problem has been recognized by database vendors as evidenced by their release of products attempting to address the issue such as historical table statistics (Ural). Often, maintaining legacy systems general leads to ever increasing cost (Defense Logistics Agency).

1.1.1 Design Provenance

This pattern language puts a name on the kind of preservation of past functionality: Design Provenance is a systematic trace from design artifacts to their predecessors and the underlying design abstractions which explain the reasoning, analysis, and concepts behind the particular design artifact. This is different from Software Provenance which addresses purposes like security where you only want to know if a product is authentic. It is also different from Data Provenance which is concerned with a record of data handling to ensure that the data is not corrupted and its sources verifiable.

The ultimate predecessor of a component may be a completely abstract design sketch that represents intention, purpose, scope, leaving the choice of a particular design element to another step in the design process. Design Provenance says that we do not discard that Design Abstraction, which is often done by replacing its symbolic “place holder” with a particular design choice. At that point, trade-offs and specific protocols are decided. The considerations of architecture and quality that went into the design can be lost except in the mind of the designer. It may still exist in the clutter of source code control, but not readily findable from the derivative designs.

1.1.2 The supporting design patterns

This pattern language involves are second-order design patterns. Some of the elements of the Generatrix design pattern are themselves design patterns rather than classifier-level symbols. The structure diagram for the Generatrix requires a way of depicting certain elements by their abstract role in the pattern, rather than “hard coding” them as a particular class symbol, say. This level of abstraction cannot be represented by a simple abstract class, as in the Java language. Specifically, the Generatrix is directly concerned with the role of creating objects but is also concerned about the role of creator element can be fulfilled by a variety of different creational pattern elements. This is different from the challenges of the GoF Structural and Behavioral design patterns. In GoF (E. Gamma, R Helm, R. Johnson, J. Vlissides), those patterns simply leave out a symbol for the component that creates or instantiates the objects that are central to the concern of the pattern. It is left to the software designer to decide how the objects are created. In an implementation of a given GoF pattern the creator of objects is changed, the pattern itself is not disturbed. In the Generatrix, however, the role of create is one of the central concerns. It directly speaks to the creator and components that it creates, how their coupling varies or remains through iterations, and how their abstraction is invariant through iterations.

1.2 Intent

Software designers use a variety of techniques for documenting their design decisions. They are aware of the need to code for maintainability and to communicate with stakeholders of different technical backgrounds. They use previous versions of code and often change names, implementations within defined interface definitions, and refactor code. These artifacts and activities are known in different forms and names. They are not viewed as a unified and universally practicable pattern. These concepts are brought together as part of a single design pattern language. It identifies several of the familiar solutions.

- Maintain knowledge of the design concepts of previous generations of the system in a form that can be found and understood when application to redesign.
- Promote the separation of the design of a component from how it is constructed, built, assembled.
- Promote the explicit knowledge-capture of components that are coupled within their generation and progress together.
- Establish design practices that will make reverse-engineering in the future unnecessary.

Promote the use of design abstractions that are not discarded as detail designs are created but retained and attached to the artifacts they lead to.

1.3 Motivation

- The problems experienced in the maintenance phase of a software development lifecycle are amplified when a significant new generation of the project is undertaken.
- A variety of solutions to specific kinds of maintenance projects are used in re-engineering a system for a new generation. But they are separate solutions and not part of a cohesive design pattern strategy for designing.
- We desire to promote planning for the next generation during current development. This pattern language helps a project to be more "plan-like".
- When this set of patterns was first developed, there was no real-world experience with attempting to apply them. It was difficult to advocate its use without an example of its application.

The Generatrix design pattern was presented in stand-alone form recently (J. R. Reza, The Generatrix Design Pattern) as an attempt to address the (above) stated intent. Based on discussion of that pattern the it was decided to express separate supporting design patterns around the Generatrix which became this pattern language.

The earlier paper revealed that the concept of "a generation" was not easy for the reader to accept in the way that it is intended here. The tendency was to think of the word "generation" as an awkward attempt to use a different word for the familiar "stable release," "redesign,", "evolution" or a way of automatically generating code from sketches or specs. There is no formula for deciding when a system is "a new generation", being an intuitive concept that depends on the kind of system. Section 1.8 illustrates what is meant by a generation in this paper.

1.4 Summary of solution

The problems are addressed by the design patterns comprising this pattern language applied individually and in combination. The patterns are:

- Section 2. Generatrix design pattern
- Section 3. Design Abstraction design pattern
- Section 4. Design Provenance design pattern
- Section 5. Creator design pattern
- Section 6. Generation-sensitivity design pattern
- Section 7. Accord design pattern

1.5 Structure

The diagram of the pattern language shows the Generatrix as comprised of several patterns and their basic relationship to a software system, Figure 1. Together they comprise the pattern language. Each pattern is elaborated in the upcoming sections. First, a walkthrough of the diagram will show how the patterns and design artifacts interrelate. See also the specialized symbols in Figure 1a.

Design Abstraction patterns capture "purpose" and technique without committing to a particular form of implementation. By enabling us to create *Design Abstractions* explicitly, the design implementations need not become disassociated from the purpose expressed by their abstractions, or cluttered with explanations.

The **Creator design pattern** represents any creational design pattern and is not limited to those in the original GoF. As such we can think of Creator as not just a Design Abstraction pattern but a second-order pattern.

The **Provenance** design pattern is a key feature of this pattern language. It defines a **predecessor** relationship between a *Component* and either its Design Abstraction or an earlier design addressing the same or earlier set of requirements, or both.

The **Accord design pattern** is presented in brief form in order to show that the coherent generational design pattern language is not a closed system. It invites the discovery of additional patterns that address intergenerational concerns. The Accord discusses designs where two or more components that are coupled within the same generation of the system. One or both of the components can only interact properly with its

partner. Each component is part of a family of components that are changed with each new generation of the system.

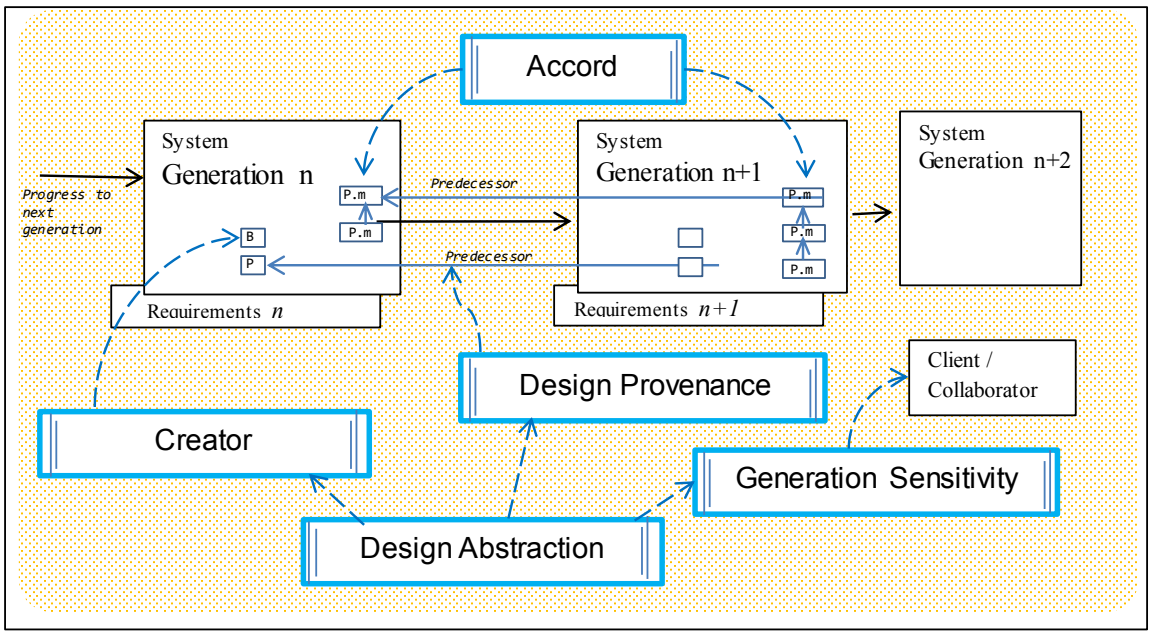

Fig. 1. Informal view of Generatrix as a collaboration of patterns and classes across generations of a software system.

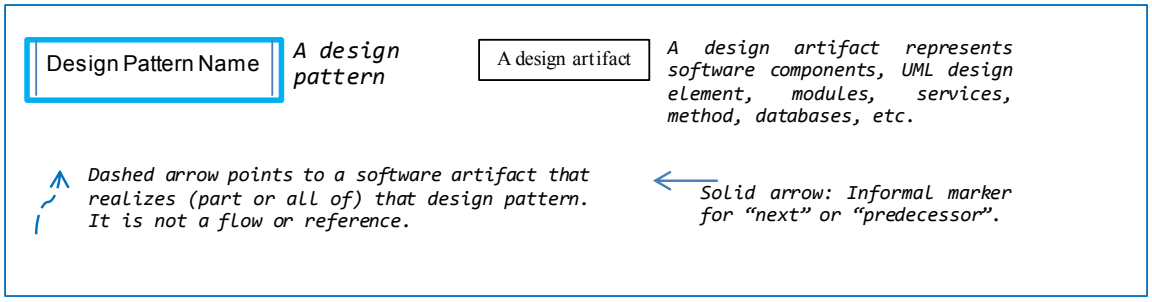

Fig. 1a. Notation used in Figure 1.

The System in Generation n, n+1, etc. is shown containing a couple of representative components labeled B and P. Very concisely, the B component is intended to represent the Builder pattern as an example of a particular creational-patterned component. This construct suggests a pattern construct with a strongly explicit Creator-Component relationship. Notice the it is a Creator design pattern that points to the Builder to indicate that the abstraction of a Creator gives rise to the particular design choice of builder. The P similarly represents a particular Part component. In both builder and part, the derivation of their design is indicated with the Predecessor arrow.

1.6 Terminology

A few words have specialized meaning within this pattern language. Along with the six constituent patterns described in this paper, these terms comprise much of the vocabulary of the pattern language whereby generations of a software system may be developed and maintained more coherently.

- Generation - a software system derived from an earlier product which represents a very significant change of form or functionality, perhaps exhibiting a "quantum leap" in technology or set of requirements. It is not simply software evolution since that term suggests a continuous pattern of

small changes. A series of generations is more like punctuated equilibrium (PBS staff. Credits to Scott Freeman and Jon C. Herron), to borrow a term from evolutionary biology.

- Generatrix – a term borrowed from geometry. The Generatrix design pattern is the central feature of this pattern language. It is concerned with overcoming the loss of quality and knowledge as a software system progresses in quantum leaps, or large scale iterations, from one generation of a product to the next.
- Predecessor - An abstract reference that identifies the component or components of a previous generation of the system from which a component was derived, redesigned, or augmented (in a generation subsequent to the previous generation). The predecessor relationship can exists between any software artifacts including a method, a class, interface, package or group of artifacts (and not necessarily between artifacts of the same kind).
- Design Abstraction - Knowledge that is captured in documentation or other forms becomes detached from design and software artifacts. The Design Abstraction design patterns seek to remedy this be supporting more design knowledge in the same kinds of artifacts as the conventional design and implementation.
- Form-Agnostic – A kind of abstraction that does not specify all of the format, or calling protocols but defines the functionality of a design component. It defines what must be done to connect or interface with it but not how. Contrast this with an abstract method which specifies how to call it, and not much about what it does.
- Creator-Component relationship – a more-general kind of builder-part relationship. Creator-Component is the Design Abstraction for a component that somehow constructs, retrieves, builds, or initializes another Component. The created Component is, in this context, also a Design Abstraction for some kind of software component in the post general sense: a Component refers to a specific design involving one or a collaboration of classes and objects, as well as resources such as database, framework functionality, files, and so forth.
- A Creator is itself a Component of the system. A Component may be designated in a diagram as a Component or Creator to indicate the point being made (not to say that it isn't the other).
- Resource - a component that is designed to be referenced by Components of more than one generation. A resource obtains no generational awareness merely by its role as a resource. It can be an ordinary component.
- Accord relationship – one or more components with a dependency on a component of the same generation.

1.7 Audience

This Pattern Language is intended to "speak to you" if you are . . .

- A ***manager or customer*** of MIS or applications in the broad sense: You are a user, marketer, financial agency or business, information-security, compliance mgr., SME [1], contract analyst, or other participant. Or,
- A **Scientist or research institution** that needs to maintain scientific provenance and reproducibility. You need to prove how your system delivered results in the past or to enable other researchers to reproduce your analysis. Or,
- A **software engineer, or technical team** member in the broad sense: programmer, computer scientist, business analyst, QA tester, build engineer, network admin, system admin, database admin, marketing rep., security admin.
- Developers and researchers whose coding effort is solely for their own academic work might not have the problems addressed by this pattern language. If you write small, one-shot, or stand-alone

[1] SME - Subject Matter Expert, either formally in an CMMI or ITIL environment, or similar situation.

application with no major generations and user base, the problems and solutions presented here might be rather alien.

1.8 What is a Generation?

Very succinctly, a new generation of a software system is characterized as a quantum leap in features, requirements, or underlying technology. The scope of the change in the product is usually dramatized by a big change in its packaging. The concept is intuitive and has no general formula for distinguishing a generation from a routine iteration or release. But a generation would normally not apply to iterations of a system under development that is a step closer to release into production. It is different for different kinds of applications. One consideration is whether the system has been published (Orchard). The publication and subsequent editions of a book would be a reasonable basis of comparison.

This section describes two very different kinds of product line going through multiple generations. This extended explanation of the concept is a response early reviews of this pattern language which revealed that the intent of "generation" needed much clarification. The story of multi-generation product lines is illustrated with the Sony Walkman, and Generations of a typical insurance-industry product..

1.8.1 Multi-generation product line; Sony Walkman

The Sony Walkman product line had 3 or 4 generations of models called Walkman (Wikipedia, with contributions from Sony Inc.), pictured in Figure 2. These generations could be called, in summary fashion, (1) the cassette-based Walkman, (2) the flash memory-based MP3 player, and (3) the disk-based Walkman with several codecs. A 4th generation would be the ear-bud format, not discussed here.

Fig. 2. Three generations of the Sony Walkman

The three generations of the Sony Walkman are shown in 2 (Haire). By casual inspection, you would be correct in guessing that these Walkman units represent a series of generations of the product line. Inside, the first (left) is built on tape recording technology. The next (center) is clearly a new generation of the product. The visible change is due to a major technological shift: flash memory. Next (right) is a generation that looks more slick. Not visible, though, is that it has additional features which are achieved by using a micro disk.

<u>The Walkman illustrates 3 kinds of "quantum leap."</u>

i) Creation. A new set of technologies and requirements are established (from no comparable predecessor).

ii) Technology is the quantum leap. Replace tape with flash memory. Requirements basically follow.

iii) Requirements take a quantum leap. Replace flash memory with micro-disk drive to meet high-capacity and cost requirements; also add audio formats, slick appearance, and interoperability features for competitive reasons. The basic features (recording and playing music) are unchanged except in capacity and media formats.

During the life of each generation, there were undoubtedly new iterations of internal software, stable or tagged releases, some hardware changes, and fixes, and iterations of the manufacturing process involving

the code base. Internal to the development team, some of those iterations might have embodied a new generation of software architecture such as change of language, compiler, test processes, and out-sourcing.

Software systems can have large game-changing requirements placed on them such as requirements beyond configuration and refactoring to implement them (Fowler, Beck and Brant). This fosters a new generation of the third kind (requirement quantum leap).

The example of the Walkman illustrates generations of a product that may be marketed concurrently for a time. But in this kind of stand-alone product there was never a need for more than one generation to be interoperable with, or even operable directly with models of another generation as illustrated in the next example of a pan-generation system.

1.8.2 Pan-generation insurance product line system

This example of a real-world insurance product line has no gadgets or visible packaging to illustrate. The quantum leaps arise from problems unique to service organizations that have new requirements on an annual or periodic basis.

1) Periodically a dramatic new set of requirements is imposed on the MIS system that supports a particular property insurance package. Requirements come from annual legislative actions (laws and regulations) of the State. They come from unplanned court decisions, which may result in rerunning analytic and decision-making components as it would have been in a year past. Simply retrieving a code base and setting system times on a database is not feasible.
2) The changed requirements also come from new compliance mandates of Federal and consortium sources (Conference of State Bank Supervisors).
3) Important DSS[2] functions run 3 consecutive "years" of the system in order to determine charges, trends, and various statistics about groups of clients. In the example of Figure 3, each yearly generation of an insurance policy product is a near-copy of the same system implementing the policy. This prevents the large database from being reengineered into separate instance "years". Changes to the schema to accommodate this year's requirements make legacy code inoperable. Consequently, each year hundreds of lines of code across the whole system are tweaked with conditional logic (e.g. if year < '2009' and price(year-1) > price(year)…). The system is becoming impossible to maintain. Testing cannot be done on small subsets of the database which now contains over 30 years of data with millions of accounts.

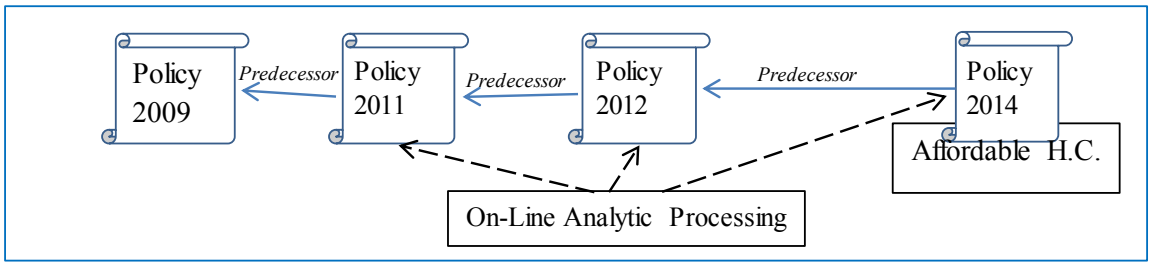

Fig. 3. Pan-generational design: insurance system example.

Each generation of the product adds custom code and new configuration items to support 3rd party productivity products. Development and maintenance is usually scoped into projects that live through a small portion of the life of the product line. There is no time to assess the global situation. As such, the system can build up toxic code over time and periodically experiences crises.

4) In a large institution, the system might experience personnel upheavals, change of service providers, and lack of resources appropriate to rapid technological change. As such the designation of a "next generation" might be organizational and not due to a significant sudden change of scope or

[2] DSS: decision-support system

technology. If the new staff faces a lot of reverse-engineering, rediscovering underlying system concepts, and having to fix broken and missing pieces, *to them* maintenance is a quantum leap.

5) Simple details in code can have rippling impact on stored procedures and logic. Suppose that in year 1, a field for dollars is declared as NUMBER(8). The next year it was changed to allow an embedded dollar type, VARCHAR(12), the next year it splits into a numeric and currency type field. Consequently the creation of the reference in a class has to change. However, the design abstraction is stable: there is a currency value. The creator must be implemented with a different design. But the dollar value field in a corresponding class, and computations on it, are unchanged from year 1 (NUMBER) to year 2 (VARCHAR containing digits plus a code), and in year 3 the value field can still be unchanged while a new field (currency type) is added if it was not anticipated beforehand.

Multiply the effect of the above field-level example by tens or hundreds of such fields referenced and created in many places in the application code.

1.9 Consequences

The design patterns supporting coherent generational design have been applied by the author in two large MIS systems under continuing development. The first objective was to exercise the patterns and confirm that a developer can see the components and design of a system in terms of these patterns. The effort was informal and not systematic, as would often be the case in introducing a new strategic design pattern. In the situation where the concepts and constructs were implemented, the intended benefits were observed with the natural bias of one who is eager to see confirmation but strives to be objective. The informal results, advantages and disadvantages are summarized as follows.

The separation of creator and component largely achieved the ease of updating fewer artifacts and only what needed updating for implementation of the new generation. The Generatrix can viewed in tabular form, a matrix of components by generations. In matrix form it has been easy to communicate with stakeholders about the cumulative impact of numerous small changes. The means of implementing the Predecessor concept was not supported directly by the existing infrastructures. The informal use of comments and annotations begins to help in reverse-engineering the code base.

The Generatrix and related design patterns were introduced along with a number of well-known design patterns without the scope of the consulting contract. The use of design patterns was viewed as strange and unnecessary by some legacy personnel. The Generatrix in tabular form was accepted as easy to understand, however. The concept of a next generation was familiar to people in an area like insurance and tax, but others saw no difference between a generation of a product and a tagged release. Some developers thought that the concept of a Design Abstraction was actually due to misunderstanding the Java abstract class construct.
The most significant obstacle to successful application of these patterns is the education level and cultural shift that comes with it.

1.10 Summary

With the foregoing examples of generations of a product in mind, generational design can be thought of as designing with an awareness of how subsequent generations may benefit, and benefitting from such efforts.

The design patterns following this section may omit the customary Known Uses, Applicability and Consequences since the patterns have been developed and applied together with the Generatrix, hence this pattern language as a whole. The Design Abstraction, Provenance, and Creator patterns could be used independently, perhaps as a way of easing the language into an existing development environment.
.

2 GENERATRIX DESIGN PATTERN

A set of artifacts in the grand design of a software system which support the creation, capture, and transfer of design knowledge and reasoning across generations, or major releases, of a product line.

2.1 Intent

Conventional practices involved in software maintenance may complicate development of multi-generation software systems. In the interest of changing and adding features to a system for the set of requirement in hand, the existing system is, in a sense, destroyed. Knowledge of how to run the system in a previous configuration of data and functionality is lost. How to run the current system is alive in the form of institutional knowledge and so is not captured deliberately for the next generation, the next crop of employees. The cause of irreproducible results becomes an untraceable mystery. Code is duplicated, dead code is left around, place-holders are discarded.

Generatrix is a design pattern for designing components of a system with the next and previous generation in mind. It provides a pattern in which the derivation of a component from a predecessor in the previous generation is explicitly recorded. The pattern supports the separation of the creator of a component from the component so that each may be iterated independently in the next generation as needed. The pattern supports and encourages the capture of higher level abstractions as part of the design base so that the knowledge of design decisions is not lost in the detail designs.

2.2 Motivation

Certain problems arise when developing a new generation of an existing software system.

- Typical design tools such as UML capture a design at a “point in time”. Upon modification, a given component only exhibits one form and functionality of the component. While a series of point-in-time diagrams may be accessible to a software engineer, there is no single design artifact capturing the change and reasoning or trace behind the transition.
- A new generation of a software system often involves a lot of reverse-engineering because the reasons for the legacy design are not apparent; the thought and analysis are lost.
- Software development practices often neglect opportunities to design for the next generation of the system. Code is often duplicated in order to implement new variants of a set of requirements in the new generation.
- Multiple generations of a system are sometimes used together for multi-year business analysis and regulatory mandates. The legacy system’s architecture remains the dominant structure of the system but it was not designed for the multi-year purposes. Versioning systems do not capture or enforce this knowledge.
- The legacy code clutters the current design, making it hard to understand what it is supposed do. There is no clear separation between legacy code that must be retained and new artifacts.
- Interdependencies between components that were intended to work together in the same generation may break.
- A key reason for poor maintainability is the diffuse repetition of large and small chunks of code that do not adhere to “separation of concerns” or the concept “don’t repeat yourself.” (Steve Smith).
- The name of a component may be changed so that if it existed in the previous generation under a different name, that fact is obscured. The same name may be used for a completely different purpose of a component, creating confusion. Name-changes solely to avoid name collisions is a waste and introduces unnecessary complications.
- When a bug is discovered in a released product, the tendency is to fix it in-place and not preserve the released system in a form that can be rerun (with the bug) if required.
- Conventional source control systems alone do not effectively capture the evolving knowledge. Simple design rules such as in “checkstyle” (Pomeroy-Huff, Cannon and Chick 1.2.3. Forms) do not adequately address architecture-level design change issues.
- Source control systems alone to not provide for the above-mentioned problems especially when different languages, perhaps with their own versioning systems, work together in the same system.se
- There is no simple metaphor and vocabulary that could be included in requirements to make the future concern a present concern that developments can follow and check objectively.
- Requirements traceability is not enough. Even good traceability is often not granular and does not convey the reasoning behind technical design choices, just the requirements being addressed by a component.

The list of problems includes small coding issues as well as concerns at the large-scale business level. The various concerns are address by the several design patterns comprising this pattern language.

2.3 Summary of Solution

The Generatrix design pattern identifies several techniques that work together to avoid the problems mentioned above. It provides an explicit Design Abstraction for capturing knowledge about design intentions that are often lost. It provides a clear element (Predecessor) for associating design artifacts with their Design Provenance, both abstract and non-abstract. It raises awareness of the benefit of separating the creator of a component from the component, at least in the abstract if the language does not always support actual separation (e.g. constructors). This pattern largely provides simple archetypal names for concepts that are familiar in other forms.

The supporting design patterns in rest of this paper appear in order in which they were just described. The Design Abstraction supports (is used by) Design Provenance, which in turn supports the use of the Creator design pattern, followed by then Accord design pattern.

2.4 Structure

Throughout this paper, the UML-like class diagrams include informal notations that are not UML compliant for illustration purposes.

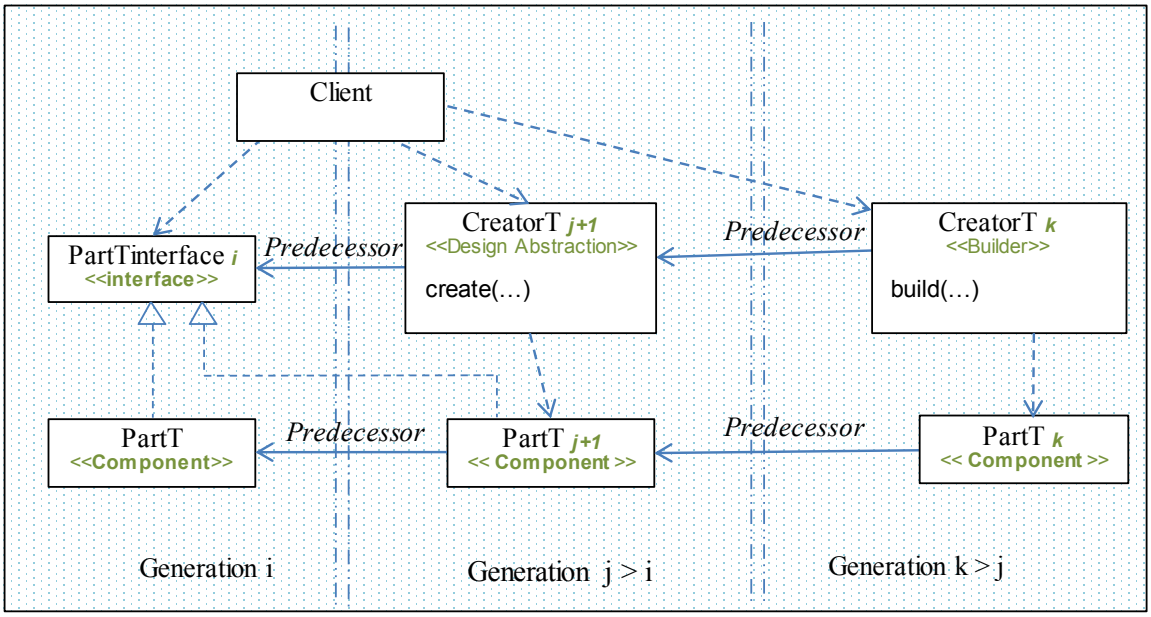

Fig. 4. Illustration: In an abstract-level design, a Client invokes the generational system.

The configuration in Figure 4 illustrates a complex of components. This is an example of a combination of Creator-Component pairs with their relationships across generations. The separation of Creator and Component is intended to facilitate large change and retention of design knowledge with flexibility. The diagram here is just one configuration, not the template for the Generation design pattern in general.

The Client component is shown disregarding the generation borders to indicate that the client is completely unaware of the generation nature or version. Component T_j was derived from, and might even delegate some function to Component T_i in a previous generation of the system. In generations *j* and *k*, the Component and Creator are shown as iterated for their respective generation.

The scenarios in Figure 5 illustrate the basic configurations in which a Creator and it Component are designed across generations. The diagram calls attention to the use of a Resource to help a component when it needs to "know" which generation is in play.

Scenario A. There is a completely different Creator design choice in the next generation. It also builds a different Component. But that Component inherits or in some form derives from its predecessor.

Scenario B. A Component is modified in the next generation, but the same Creator is able to create (initialize, inject, load, etc.) its subject component. Presumably, the Creator can use some information to build the Component that makes the Component different. That difference could be something as simple as the release version of the system.

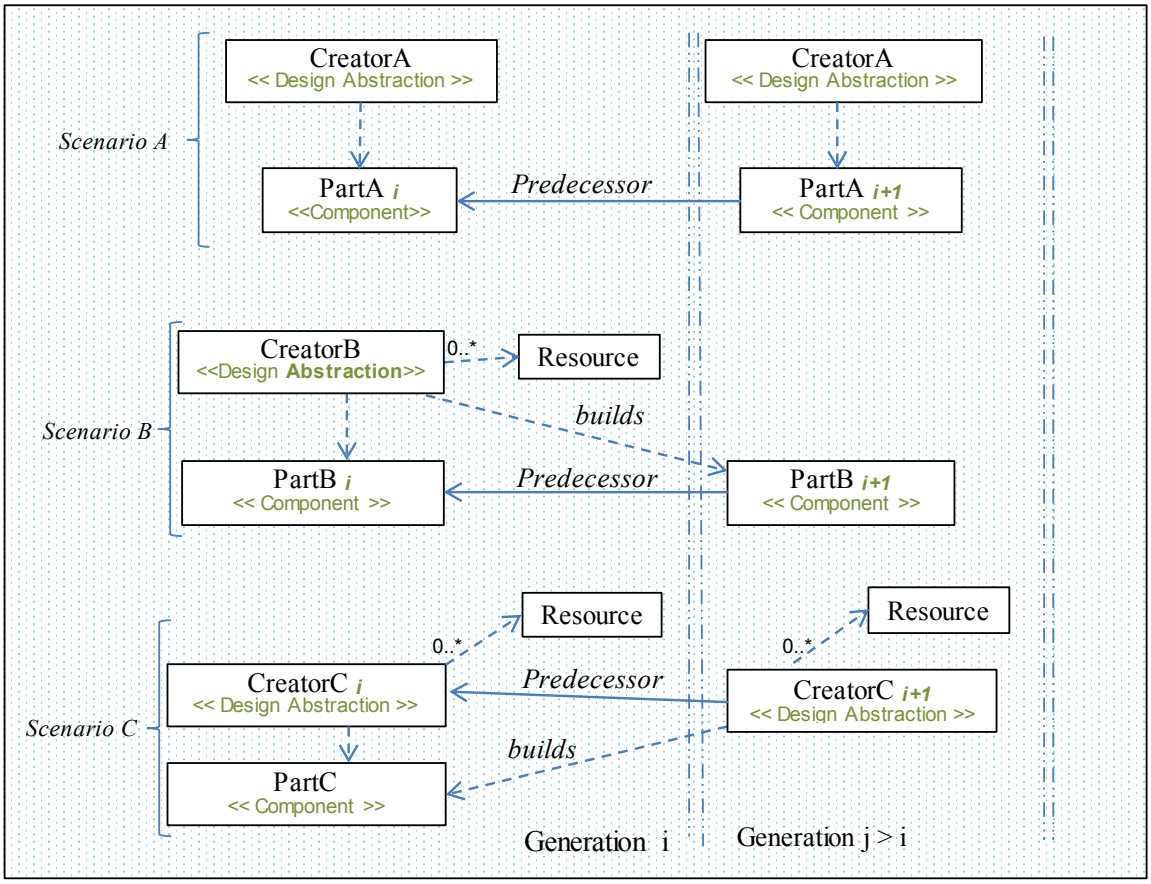

Fig. 5. Creator-Component scenarios.

Scenario B would seem to defy the rule that a previous generation's elements must be unaware of elements in any subsequent generation. The Resource must be supplying something to the Generation $_{j}$ Creator that causes it to build a Generation $_{j+1}$ Component. That could simply be the Component to the implementation of the Generation $_{j+1}$ Component.

Scenario C. A Component is built differently, but its design did not change in the next generation. The Creator was redesigned for reasons that could include defect correction, completing requirements, or improved qualities. The notation <<Design Abstraction>> is an archetype designate the role the classifier (the box).

2.5 Applicability

Use the Generatrix pattern when . . .

1. It is desirable to maintain and develop the system with the next generation's development in mind.

2. The cost of doing work in anticipation of future pay-off would benefit from a simple compelling metric. It would also help if specific artifacts which address next-generation concerns can be included in present requirements.
3. The problems listed as motivating factors are present.

2.6 Related Patterns

- Abstract Factory, (E. Gamma, R Helm, R. Johnson, J. Vlissides).
  - Different implementations under a common set of interfaces are supported, as does Generatrix. Generatrix differs from Abstract Factory*:*
  - The different families of components of Generatrix are not parallel alternatives to each other as in Abstract Factory. Generations may be concurrently supported "alternatives" but with a sense of progression. Much of the next generation is the same as the previous.
  - The Abstract Factory conceives of providing one Factory for each alternative family of Components. All components of each alternative are different and specific to the whole variant under the Abstract Factory.
- Chain of Responsibility, (supra).
  - A generational component may delegate all or part of its functionality to a predecessor. It is more like a general chain since the component can choose to also filter and modify its request and response.
  - A component, in Generatrix, may implement the "chain" concept is ways other than runtime references to components having a compatible interface with a handler method. It can instead use standard subclassing, for purposes expected to be rather stable. It can also use other referencing mechanisms, loading, or distributed communication.
- Façade, (supra).
  - A generational component can be designed to accept requests that are serviced by a particular past generation implementation. As such, it has the quality of a Façade. The component can even be designed to accept requests created by a generation of the system that is newer than that of the component. Here delegation would be necessary.
  - A generational component could instead reject requests that cannot be properly serviced except by the intended generation.
- Intercepting Filter, (Sun Developer Network (SDN)).
  - Filter chains in SOA, can modify messages along the way. Here the component can consume a request of one generation and modify it for servicing by the service of another generation, appropriately.

2.7 Implementations

We have two approaches for implementing a simple Generatrix.

In a system that is based largely in a language such as Java, we can easily adopt a convention based on annotations. The Predecessor construct is easy to represent as an annotation such as @Predecessor with various attributes for pointing out the elements that we wish to consider. It is flexible in that the attributes can be of the same or of a different kind than the element it is applied to. Example: The predecessor of a class containing a factory method may be a set of constructors in a class residing in a previous generation of the system. When inheritance is the mechanism through which a new unit is derived, in this case extending rather than replacing, the @Predecessor can also identify the new items and the unit in the extends clause.

The concept of a Design Abstraction can also take advantage of constructs such as **`enum`** or interface. The substance of the abstraction should not include method signatures since signatures create form-dependence. The Design Abstraction may include verbal descriptions and links to any external artifacts appropriate to the project.

This approach captures the Design Provenance of any element of a design to the extent that developers have the self-discipline to carry it out. The limitation of this approach is that it fails to consider systems with multiple languages and is not readily understood by non-programmers.

The second mechanism for implementing a simple Generatrix is based on a matrix or spreadsheet. It is more useful for database-centric systems. In this situation, a schema is altered periodically to support drastic new requirements. The spreadsheet lists each field of the (pertinent subset of) the system on a separate row. The row represents the design concept for that field as the schema evolves. One or more related database tables may be represented in the same spreadsheet. Each column represents a new generation of the schema. With this layout, developers, database administrators and subject-matter (business) experts readily see where the fields are changed. The entries should contain the field names, which may evolve, the field types as known in the database and if useful as known by the programming language using the data. The particular way of building complex objects and records should be indicated. When a field is split, or combines others, or is eliminated is also shown.

Using the matrix format of Generatrix, developers can easily maintain it and see where the data they need is changing in the next generation of the system. A financial system may need to be re-run for past years for purposes of audit, retroactive corrections, legal reasons, or analysis and decision-making. The tabular representation of the past form of the system should be useful to all stakeholders.

3 DESIGN ABSTRACTION DESIGN PATTERN

Also may be known as: Place-holder. Conceptual Design. Form-Agnostic. Algorithm specification.

3.1 Intent

During the design process, we need various techniques for specifying structure and functionality without committing to final forms. We may use specific concrete example, knowing they are place-holders for an abstraction that we have yet to understand. We may use constructs available in the design language which are specifically geared toward representing abstractions. We intend that design to not abandon these abstractions as the details come about. Rather, we intend for the abstractions to retain knowledge of the reasoning behind design decisions. We need for the framework in which these artifacts evolve will provide connection into the past generations of a system that are appropriate for the system's long-term viability.

During system maintenance, engineers who are newly assigned to the product line need ways into the design: its specific and abstract structures and processes. They need to be able to find their way through the system by first getting a handle on the abstractions. They benefit from abstractions that provide paths into the derivative detail design artifacts.

This design pattern seeks to counter the problem of design intentions getting lost as the components become well defined. It is different from an "`abstract class`" or "`interface`" in that it does not assert the particular form or method signatures. In that sense it is *form-independent* or "form agnostic" while being specific about purpose. When a detail design is developed to implement a Design Abstraction, or derived from a previous version, it can take various forms. The component design can consist of one or more classifiers, design patterns involving a collaboration of objects, and even resources such as files, configuration variables, and functionality provided by frameworks such as dependency injection and database operations. This pattern expects that in some way these artifacts will capture the fact that their design predecessor is that particular *design abstraction*. Whether this connection is accomplished in-place through annotations or through a separate indexing artifact is not discussed here. The form of this information is not constrained by this pattern except that it is intended to be well maintained through multiple generations of the system.

Because of the availability of the explicit Design Abstraction, the software design provenance of all components of the system can be traced to their ultimate purpose if the pattern is followed judiciously.

3.2 Motivation

Software engineers encounter several problems in the effort to design for long-term viability of the system beyond what source-control systems alone can address.

- Designers like to use standardized diagrams for constructs. UML lacks a symbol for a *design abstraction*.
- The concept of "predecessor" of a class may take the form of inheritance, reference to another class, or replacement of the class within a separate "copy" of the containing system.
- Code changes may obscure what was essential and what was intended about a previous version of a component.
- The programming language may not have a construct supporting certain techniques that would be used to evolve a component. As an example, the "trick" of invalidating a constructor or override method for the purpose of taking it out of the "type" is not directly supported by Java. Doing so could be accomplished by adding a feature such as supervenience (J. R. Reza, Java supervenience).
- The need for explicit design artifacts representing the abstractions behind the implementation design has been recognized in the engineering discipline of circuit design (Altium), resulting in productivity CASE tools. The concept was described early in the evolution of object-orientation as a mechanism important to capturing the derivation of artifacts (Koopman and Siewiorek). Designing abstractions is a bit different from designing classes or even abstract classes and takes deliberation (Ousterhout). If we were to simply declare that developers shall create documentation about abstractions, it is greeted with either claims that we already do that or we don't need to do that extra work. If presented as a design pattern instead, it may be accepted enthusiastically.

3.3 Summary of solution

Create an explicit artifact as part of the software design to represent components purpose, functionality, and some of the constraints. The artifact is intended to retain knowledge of the reasoning behind subsequent component designs and implementations. The Design Abstraction may serve as a Place-holder until specific design are created. But unlike the usual temporary nature of "place holders" it is not discarded but maintained as a permanent part of the design. As such it may be part of a complete Conceptual Design of the system. It may contain a complete algorithm with everything but the actual names signatures and format. Finally, The Design Abstraction artifact is itself a Component. As such, in this pattern language, it may be modified for purposes in the next generation of the system in which case the original remains unchanged and the new "version" of it will identify it as a Predecessor, thus supporting Design Provenance.

3.4 Structure

The six configurations in Figure 6 demonstrate uses of the Design Abstraction pattern. They show that either Components and Design Abstractions may be in the role of the Predecessor of the other. The configuration at the bottom shows that two components have been combined in some way. Perhaps it is discovered that they are the same or can be gently modified for one component to take on the responsibility of both. Components also split (not shown).

3.5 Applicability

Use a Design Abstraction software element when:

It is desirable to capture the fact that a design can be implemented in other ways. This is to retain the knowledge of the reasons behind a particular (concrete) group of software elements that might get lost or fall out-of-date unless the abstraction is somehow tied to the concrete design. This is thinking in terms of UML classes.

Redesign is expected to involve more than simple refactoring.

When reverse-engineering a legacy system you discover the underlying design intention behind a component especially if that discovery process was harder than it seems it should have been.

When you realize that your new design has changed the underlying design intention of an artifact, you can point out where the new design came from (your new/modified Design Abstraction) and the artifact that your new abstraction was derived from whether it was an abstraction of specific design.

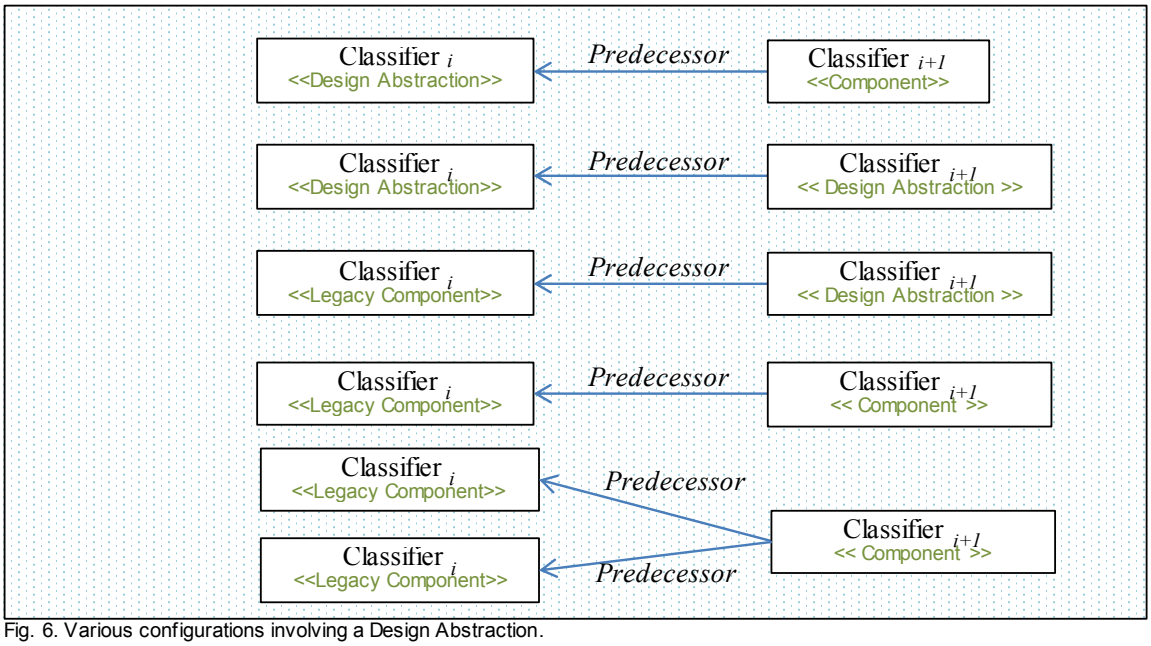

Fig. 6. Various configurations involving a Design Abstraction.

3.5.1 Illustration from another discipline.

This fanciful example is not about software development. It gives us an more visual idea of the archetypal use of design abstractions.

The concept of an abstract design element can be illustrated by analogy to architecture. Suppose a professional office complex is being designed. The architect defines an entrance component that can be detailed at construction time as a sliding door, a revolving door, a hinged pair, or other product meeting specified constraints. The overall design is agnostic about the exact kind of door at some point. The architectural design "does not care", in a sense, and the customer decides separately what door product with be installed.

After the building has been constructed and is in use for several years, new requirements lead a renovation architectural firm to plan to replace the revolving door with a heavy security door. Fortunately, the blueprints retain the abstract door design which specified dimensions, operational constraints, heat-conductivity, and local building codes that must be satisfied. Now, the renovation architect does not have to try to infer the purposes that were intended from the actual door installed because they are documented with the final blueprints.

Current blueprints include or reference the design abstraction. Those intentions remain when the renovation is done, and a subsequent repair or renovation again does not have to rely solely on the door then in place to understand the design provenance of the entrance.

This fictional example may help to visualize how the software Design Abstraction component may play out.

3.6 Implementation

An example of an implementation of Design Abstractions is found in all of the UML-like diagrams in this paper. Developers can adopt a convention to use existing UML classifier symbols to represent Design Abstractions.

There are two UML elements that can be used to indicate that a class or object is considered to represent a form-agnostic abstraction. The classifier annotated with `<<Design Abstraction>>` represents a class or collaboration of software elements and resources.

3.7 Consequences

The availability of an artifact for representing Design Abstractions invites the possibility of requiring and measuring their use by developers. The patterns of this paper can be used uniformly or selectively where deemed useful. By including Design Abstraction elements in the actual code base, with linkage to external documentation where necessary, we can retain the knowledge behind design choices in the next generation of the system as well as in the present.

4 DESIGN PROVENANCE DESIGN PATTERN

Also may be known as: Design Traceability. Design for Reproducibility. Knowledge Management.

4.1 Intent

We need to ensure that the reasoning behind a design is not lost. We need to ensure that the requirements trace to design elements remains accessible and valid. We need to ensure that the previous implementations of the design will be able to operate concurrent with new generations of the product line. As staff members transition in and out, they need to know how to access all of the foregoing. These things are collectively the *design provenance* of a software system.

4.2 Motivation

- Loss of knowledge of previous versions of components contributes to difficulty in understanding the original intent of a component.
- This loss of knowledge increases the tendency to create a new version by starting from scratch.
- Lack of understanding of an existing component's design can lead to making a whole copy of it as a starting point rather than trying to understand where a sensible segmentation might be accomplished. This leads to duplication and hence future duplicate maintenance effort and defects. This problem is very pervasive in in-house code shops.
- Requirements traceability alone does not create a record of the derivation of a given component from its actual or conceptual predecessor(s).
- New components, whether derived from legacy or created in support of fresh requirements, collaborate with legacy components in the previous generations. Those components often have dependencies on collaborating components that must be of the same generation. With the loss of this design knowledge, the new functionality may create subtle defects, especially when the behavior appears to work but gives different output.

Doing some of these "right" things may be punished in the form of performance reviews because it reads like extra work that is not specifically required. Such work is easy to omit when we do not even have the words to call for it and common acceptance of its value.

4.3 Summary of solution

Data provenance and software provenance are not the same as (and do not cover) design provenance (K. Fowler). Software provenance addresses concerns about the authenticity of a product, as in "is this download safe? Is it the real thing?" Data provenance addresses the concern that data must be handled in a way that does not allow intentional or accidental changes (IEEE). Finally, the well-known "requirements traceability" does not encompass the design decisions and analysis underlying a resulting design instance. This design pattern provides for explicitly capturing a trace to the source of knowledge behind a design.

4.4 Structure

The Form-Agnostic predecessor reference represents various ways that a component may be a derivation of a component in a previous generation of the system. The most basic form is an actual (ordinary) reference, and instance member variable.

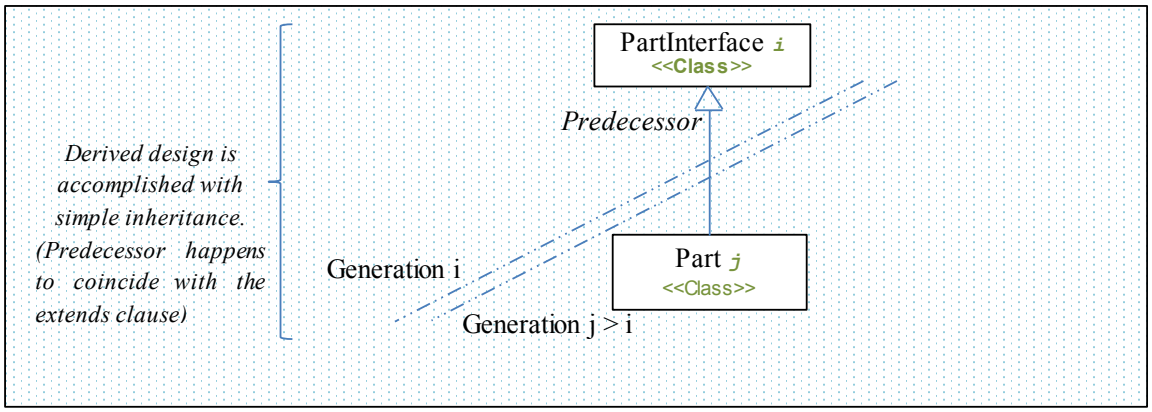

Fig. 7. Predecessor relationship accomplished via inheritance.

In Figure 7 several configurations are shown. The inheritance construct is illustrated as one way to directly implement the Predecessor concept. The interface of a Chain of Responsibility is shown unchanged across several generations of the system. However, since the interface is also a Component, it can also be revised in a new generation and maintain predecessor relationships.

The Predecessor relationship may or may not be realized as a single artifact. For example, if inheritance is used in implementing a new component, then there is a simple physical artifact corresponding to the relationship that connects the new component to its predecessor. The notation is typically the "extends" syntax. (Note: this does not suggest that all inheritance implies "previous"). In other situations however, the new component or components may have no connection in code with its predecessor; no reference, inheritance, or mention of a predecessor. Here, it is the knowledge of the predecessor relationship that is considered valuable and needing to be preserved in the pan-generational design.

The illustration of Figure 8 suggests a situation in which a component's functionality is enhanced by letting it delegate to its predecessor component where possible so as to avoid duplicating code. This design may be an application of the Chain of Responsibility pattern.

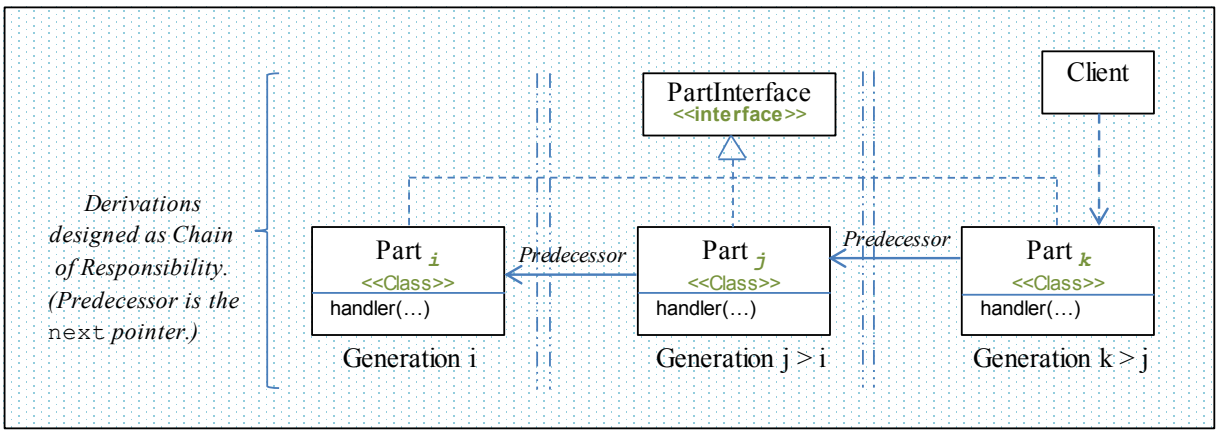

Fig. 8. A class family revised across generation with unchanging interface.

5 CREATOR DESIGN PATTERN

May also be known as: Creational design pattern, Specification-Driven provider, Abstract Creator.

5.1 Intent

A higher level abstraction is needed to represent the creational patterns that create a software component. This creator abstraction needs not specify how to construct, initialize, retrieve, populate, initialize, and/or build it. It may specify functional requirements that imply some constraints on the type of creational design patterns that can be used. We need to decouple the development of the creator component(s) from the development of the component. We ultimately want the design provenance of all creators in the system to be traceable from one generation to its predecessors and successors.

5.2 Motivation

Programming languages have constructs for abstraction of classes and constructs for creational methods. But these do not directly support the design of abstract creational components. Software engineers must provide for capturing the abstract reasoning behind concrete constructors, factories, initializers themselves. This allows the opportunities to lose the knowledge and know-how that went into a given design.

- Developers hard-code the way to create instances, perform initializations and accomplish cumulative construction or populating of object. The separation of the creating code and the objective "work" code can be obscured. Maintenance programmers must later reverse engineer this knowledge out of the code.
- Some components are appropriately hard-coded as to the way they must be created because that is intrinsic to their design concept. Other components are hard-coded with their own creation, using a constructor, or partner class builder, or other specific creational technique. Now when these components are present in the design of the system, later developers often have no direct way of knowing which hard-coded creational approach was incidental to the design of the component and which were intended as central.
- As a component evolves, the creation portion of it may become complicated and distributed to several other components. Worse yet, it might accumulate in a single excessively large class.
- The knowledge of the reasoning behind the separation of creation and work concerns is often not captured or findable when needed.

5.3 Summary of solution

The Creator design pattern is a Design Abstraction for representing "what" a software component does with little or no specification the form, structure, and perhaps how it is implemented. However, it is more than just a specification.

Each of the creational patterns from GoF and other sources are named in a way that strongly suggests that the pattern is all about "how" it creates objects. The other patterns leave out the creator of objects, unless a specific kind of creator is intrinsic to the pattern, as in the Abstract Factory. In the Generatrix, however, the creator of components is an intrinsic feature of the pattern and being an abstraction is the key feature of the Creator to emphasize the point that its purpose is specific but the particular design pattern and the "how" is separate from it.

5.4 Applicability

Use the Creator design pattern when . . .

Design of software is being done at an architectural level where the particular way to instantiate (and complete) certain objects is not pertinent to design decisions.

An explicit generational design is created. The abstraction of a Creator should be used even if the designer "knows" up front which kind of creational pattern is intended at the time, for the present generation.

5.5 Structure

A representative few of the applicable configurations of this pattern is presented here.

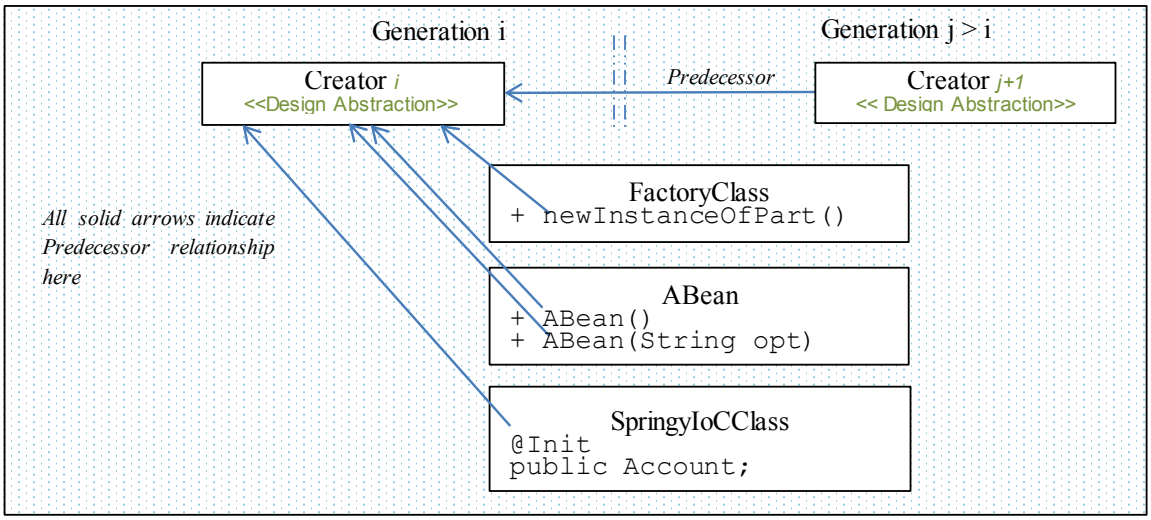

Fig. 9. Creator Design Abstraction used four different ways.

The diagram Figure 9 illustrates four of many possible configurations involving a Creator. A Creator is shown as being updated in the next generation (top). There, it has a specific Creator as its Predecessor. A Creator may also be design when introducing Coherent Generational Design into a legacy system. In that case the Predecessor may be an ordinary Constructor, Factory, Builder or other familiar pattern.

Next in the diagram is a Factory method that is derived from its Creator predecessor. It is the method that has a predecessor, not the class containing it. Here the Creator and created component may be in the same generation, or across generations. But there should not be a succession of predecessors beyond this within the same generation. Upon normal coding iteration, only the latest creator or component is the one that matters to the design.

Next is Abean which demonstrates that more than one element, constructors in this case, can come from the same Predecessor.

Lastly in the diagram is an example of a kind of object creator that is beyond the core language. In this case the annotation that causes a framework to inject an object is the design element that implements the Creator abstraction.

5.6 Consequences

The design retains more of the knowledge of the reasoning that goes into specific design artifacts. The explicit representation of artifacts in the same language supports maintenance of that knowledge.

The explicit representation of the Creator might appear to developers as an extra maintenance burden with delayed rewards that will accrue to others.

5.7 Related patterns

The Creator represents a creational design. Any of the GoF patterns (E. Gamma, R Helm, R. Johnson, J. Vlissides) as well as some of the other constructs are suited to be the design instance represented by the Creator.

6 GENERATION-SENSITIVITY DESIGN PATTERN

Also may be known as: Robust maintainable design, Configurable design, Evolvable, Upward-compatibility.

6.1 Intent

We have two issues in designing in anticipation of a next generation. First is the simple matter of establishing a convention for identifying the generation that a component as part of its conceptual interface so that it will

only work if used in the generation(s) which it was designed. Second, we need to recommend a simple architecture-level organization that will support the development of subsequent generations and their making best use of former generations design base.

The client of a system has a specific requirement as to which generation of the system it uses. There is a range of possible generation-compatibility requirements that apply to different systems. One application may require that the latest release or current database be automatically provided when a client requests service from that system. In another system, the client is an application designed to request only the specific generation of the service system it was designed for. An upgrade involves deliberation and may involve a team effort. It may or may not provide an automatic invitation to upgrade. In other situations, an application may provide the user with the capability to select a "year" or set of past and present database states. A system that can run "as" more than one generation of itself for a certain purpose is called pan-generational.

6.2 Motivation

In this context, more than one generation of the software and its database are maintained and in-use concurrently. The version expected by a client, and by components that use the system, take one of several possible forms.

- The client module is expected to contact what the user thinks of "the system" and determine for the user which generation of the system should respond. The user effectively "does not care" which system responds, delegating the "care" decision to the client application which possesses generation-specific logic. (Or…)
- The client application has no concern about which version of the system responds. The client can be from an any generation and the system will handle its request by redirection as necessary. (Or…)
- The choice of generation may be made by examining the compatibility of request and service or methods available, rather than explicit identification of generation "number", say. (Or…)
- The user is aware that the latest generation is required and the client application performs to that expectation.

6.3 Summary of solution

- The most fundamental design decision is to never render the previous generation code base and database state unusable as a result of replacing components of the system.
- Establish a clear generation-awareness capability that can be maintained across multiple generations of the system without struggle. The system may be highly *generation agnostic*, or very specific about client-system matching.
- Establish a structure for source, resources, and executables that is capable of supporting multiple generations without creating new maintenance problems or overhead frameworks.

6.4 Applicability

Use the Generation-Sensitivity pattern when . . .

Awareness of the generation of a system is an identified concern.

The software designer is aware of using, reusing, extending, deriving, or refactoring a component of the previous generation to create a new one. This knowledge can be captured as the predecessor relationship without imposing a particular form if it abstract, or identifying the actual implementation element as predecessor if possible.

This pattern would *not* be applicable when a system never develops beyond a single release: it would not have a concern about generations as discussed in this paper. Maintainability is narrowly interested in keeping the system running with no big changes.

6.5 Implementation

The two issues involved with designing for multiple generations are considered here.

The most direct approach to associate classes with a given generation of the system is to establish a project (as in the Eclipse or NetBeans environment) or a set of class paths to host all artifacts developed for a new generation of the project. The Java class loaders can be configured to benefit from the correspondence of packages so that classes from multiple generations reside in separate directories but may appear in the same packages. This approach allows multiple generations of same-named classes from different generations to be loaded in the same JVM if necessary. It is flexible and simple for simple needs, and only grows in complexity with complex need.

Aside from this approach that should be familiar to most java developers, annotations or constants can be used by convention. A simple identifier is sufficient to associate a design artifact with the generation of the system that it was developed for. In Java, certain classes are designed with a built-in version ID requirement. This represents a conscious design decision to anticipate upgrades in the future. Projects decide on different mechanisms to populate the ID field, or to ignore it.

The issue of how to organize the architecture to support multiple generations may later be subject to one or more design patterns. For now, an approach that has been used successfully is presented. It is not as general as will be needed to fully realize the intentions of Generation-Sensitivity. Here the context is a Java-centric code base in a NetBeans or Eclipse environment.

The approach is based on creating a number of "projects." This project construct is the handy mechanism that the IDE uses to organize source and classes into a directory hierarchy with class path (project root) at their top. A development "project" may be implemented using multiple IDE "projects".

The concept is to use a feature of the Java language in which the loader can find classes in multiple places provided that their "package" paths correspond with respect to their project root. Given that software developers are well-versed in using this feature the solution to organizing multiple-generations is simple.

1. The legacy generation remains in its own project. It is assumed to be unchanged; no more maintenance fixes.

2. The next generation now under development resides in its own project. The particulars of naming conventions are not significant to this approach.

3. A complete copy of the legacy project is made so that its code can be shared by the next generation's project. This will be called the legacy-copy project.

4. As development of the next generation progresses we will have three situations in which a component is new, a replacement, or an "inheriting" update of its legacy predecessor.

   A) New: no complications. The new module resides only in the new project.

   B) Replacement: The same-named module is deleted from the legacy-copy project completely and the new component resides in the new project. The class loader will find just one class of the intended qualified name among the two projects (the new and legacy-copy projects).

   C) Inheriting Upgrade: Here is a situation that often drives coders to come up with ad hoc version-naming conventions or bring in a 3rd party dependency-injection framework. The present approach does not require those actions. First, the new component is created and resides in the new project. It needs to delegate to the version of itself that resides in the legacy project. The copy in the legacy-copy must be deleted. In order to load the original version of the component now residing in the (unchangeable) legacy project the reflection mechanism of the class loader is used. This is the part of this approach that might seem tricky to developers. It can introduce complexity if used inappropriately. Developers should not see this as a reason to start loading everything through reflection.

This architectural construct lends itself to development generations which are infrequent. This approach is probably inappropriate for numerous frequent iterations as in an Agile environment. The management of too many "project" containers is no great burden for relatively few and occasional new generations of the system.

6.6 Related patterns

- Database anti-patterns (Karwin). For example, Foreign Keys creates a rigid relationship between tables. When the structure, use, or range of values of one table is drastically changed in a new

generation of the system, the use of the foreign key column may have to be constrained, or queries using it may need additional logic to produce different effects depending on whether the table is being used for the previous generation or the next generation of the pan-generational system.

- Canonical Versioning design pattern of (Erl). It deals with Service Oriented Architectures using different versioning schemes. It views the solution to problems as being taken care of by conventional versioning tools.

7 ACCORD DESIGN PATTERN

Also may be known as: Coupling to version (version–specific). Family of objects in Accord with each other (mutually version-specific).

This design pattern is included without a great deal of elaboration in order to show how the pattern language can promote the discovery of additional design patterns. This pattern is also an example of a second-order pattern in that it uses certain other design patterns as some of its basic components, rather than just conventional representations of class-level items.

7.1 Intent

Two or more components have a conceptual dependency or coupling to each other based on the generation of the system. When they are upgraded in the next generation, they are not interchangeable with their respective predecessors. The coupling is designed intentionally and has been called an Accord relationship (J. R. Reza).

Unlike the coupling created by a single reference from one class to another, this coupling need not have an explicit code that a maintenance programmer could readily discover. Instead, the functionality and dependencies through other objects makes for a generation-specific co-dependency.

This design pattern provides an explicit design construct for the *accord*, a strategic agreement, between two or more families of classes or method families. A family in this context means that each generation results in a new version of the latest component in the family. That revision should be done to both families in order to ensure coherent design through the next generation.

7.2 Motivation

- The knowledge of when components must work only with collaborators of a certain generation is lost or becomes subtle and obscured. There is no specific notation in UML or typical languages to express this abstraction directly.
- Even though components will not throw exceptions if used together improperly across generations of the system, their output will be wrong or unreliable and the cause of the problem difficult to discover by outward behavior.

When a new generation of the system introduces a feature that must work with a specific generation of the product, or transparently with a previous generation, the components that are dependent on generation may be difficult to isolate unless they are identified as such when they are being designed.

7.3 Summary of solution

The corresponding families of classes or methods should be annotated so that each family names the other as interdependent by generation. The Accord design pattern provides for an explicit linkage between families of components indicating their compatibility by generation. The linkage may be in the form of a design abstraction (artifact) or it can take the form of a reference, annotation, or other element that is directly supported by the language.

7.4 Illustration

A classic paper by the Liskov substitution principle (Drossopoulou and Yang) describes a simulation of a Car and a Driver which are extended to a Race Car and RacingDriver class, respectively. It defines the problem that could arise from a kind-of upward compatibility substitution. The RaceCar must be passed an argument value that is an instance of a RacingDriver. This dependency is an artifact of the discovery of *racing* in the

second generation of the product. The predecessors, Car and Driver, are compatible in their generation. The next generation extends Car to design RaceCar which takes a RaceCarDriver as a parameter type. The RaceCarDriver is an extension of a Driver. The implied constraint on the arguments to RaceCar is that the object must be a RaceCarDriver, not just any Driver. But if coded incorrectly a runtime Driver could be passed to the RaceCar.

The problem is not with the original design. But future extension of the system in a next generation could be coded to allow the unintended Driver types to find their way into the driver seats. An explicit Accord between the families of classes would give the originator's knowledge directly to the future developer to understand and comply with.

7.5 Applicability

Use the Accord pattern when …

1. Two "partner" components are designed to work together fully and properly in the same generation.
2. An upgrade of functionality to either partner in the next generation warrants a corresponding upgrade in the other in that generation.
3. It is required that the design and implementation of the previous generation of the system will be available for use or reference by the next generation and absolutely unaltered for any purpose of the next generation.

7.6 Implementation

In the first generation of a fast conventional car, there is a fuel pump family of components and a flow controller family of components. In the next generation of the same product line, there is a new over-drive feature for sudden acceleration. Here, the flow controller can send a new signal to the fuel pump which the new fuel pump understands. The fuel pump could work in the previous generation of the vehicle, but its new feature of faster flow would never be exercised there.

If Java inheritance were used to implement the next generation of components, the new Pump and FuelController classes would each extend their respective predecessors. The methods for sending and receiving flow control signals would be explicit `@Overrides`. In the game simulation, some cars are of the previous generation and some are of the new. There will be runtime object instances of both generations of Pump and both generations of FuelControllers. It is up to the programmer to make sure that pump and controller objects *only talk to their partner components of the same generation*. There are coding hazards in which the program can appear to be correct, but because of subtle effects of the types of argument variables and the types of the declared method parameters, the runtime dispatcher can route a call to the wrong generation (an easy mistake to miss in coding).

The Accord relationship insists on a consistent means of ensuring that the coding hazards are avoided. It requires that when inheritance or other derivation mechanism is used to connect a new generation of a component to its predecessor, the predecessor relationship must be explicitly represented.

Any form that the Accord relationship takes will have qualities resembling Java inheritance. The default dispatch function is based on parametric contravariance. But to achieve the intention of Accord, parametric covariance would be the most direct mechanism. Since the dispatcher is generally not a component available to developers to readily change, other "work-around" coding techniques are generally used.

The specific means of representing "predecessor" and the generational partnering relationship are not defined in this paper. It will vary with languages. In Java, however, a simple @annotation mechanism can be adopted by the developers. For the purposes of this pattern language it is sufficient to describe the need for these representations in support of the Generatrix pattern. The Pump family of classes may have an annotation such as `@Accord(Partner="FlowController")`, and the `FlowController` annotated `@Accord(Partner="Pump")`, along with appropriate path qualifiers or imports.

8 CONCLUSION

The various concepts comprising the coherent generational design pattern are often discussed during the design process. However, these concepts have not been thought of as parts of an overall higher-level pattern. The vocabulary of the concepts are not universally recognized or used consistently. It is hoped that understanding the concerns and solutions for multi-generation software systems will lead to the benefits suggested here.

8.1 Other work

The *Pattern System for Tracing Architectural Concerns* (Mirakhorli and Cleland-Huang) gives us a possible solution to software maintenance problems based on adding traceability from design elements to underlying design decisions. The intent of their pattern system identifies the problem of degradation of the architecture as it evolves. As small coding iterations are performed during maintenance, the changes depart from the architecture's requirements and reduce maintainability. The problem is compounded when design knowledge is lost in the development of a new generation of a system that derives from one or more predecessors.

The approach described in this paper differs from Mirakhorli and Cleland-Huang in a few ways. The Coherent Generational Design pattern language does not involve requirements traceability. Also, the Design Provenance is not limited to pointers from specific design artifacts to their architecture-level items. It is not limited to having (present generation) architecture to which a design refers, but instead provides for generations of the architecture and design artifacts as a unified lineage. In addition to associating design artifacts with their underlying abstractions, it associates design artifacts with their predecessor designs which are not abstractions. Finally, this pattern language does not impose a 3rd party support system for maintaining pointers.

Several studies of programmer performance, summarized by Yamashita, found that "*On unplan-like programs … experts' performance deteriorated, as they seemed confused by the rule violations, and indeed their performance levels dropped to near the level of the novice programmers*." (Yamashita). Experienced programmers are naturally concerned for the next generation and can feel thwarted in projects with immature planning habits. The Coherent Generational Design pattern helps the experienced originators of a system, the less experienced maintenance programmers, and then the future experienced programmers who wish to reach across the time that has passed and access the knowledge of the originators. The shift in software development paradigm implied by this paper may promote a more ***timeless way of thinking*** in general.

8.2 Future work

Additional real-world application of the coherent generational design will progress slowly, as the systems that it is applicable to move on the scale of calendar years. On small applications it can be applied with deliberation. But the benefit of saving abstract design knowledge would be smaller and may be hard to justify until the application grows and personnel turn-over exposes the loss.

One obstacle to acceptance of the Generatrix design pattern is that different languages should be annotated using their own mechanisms. The matrix format (e.g. a spreadsheet) is nice but developers would immediately point out the possibility of automatic updating of the matrix. That would make a nice graduate project.

The concept of organizing projects and using class loader features should be described in the form of a proper design pattern. It should be presented with several examples. Its advantages and disadvantages should be studied and compared to the solution offered by other approaches such as dependency injection frameworks.

8.3 Summary

The real-world experiences that have lead up to the development of this pattern language have been summarized in this paper. Research on the application of Generatrix concepts has continued in the context of a large financial institution with a multi-generation MIS system. Each of the supporting design patterns has been exercised on different systems and for various objectives. A degree of stability has lead the author to feel justified in exhibiting the pattern language for the community to consider at this time.

REFERENCES

admin. "A Tour of Scala." 06 March 2009. *The Scala Programming Language.* École Polytechnique Fédérale de Lausanne (EPFL). <http://www.scala-lang.org/node/117>.

Altium. "Design Abstraction - a Practical View." 2011. *www.altium.com.* 2013. <www.altium.com/files/pdfs/DesignAbstraction-aPracticalView.pdf>.

Apache. "Byte Code Engineering Library." 17 Oct 2011. *Apache Commons.* 02 June 2013. <http://commons.apache.org/proper/commons-bcel/index.html>.

Apple, Inc. *Programming With Objective-C*. 2012. <http://developer.apple.com/library/ios/documentation/cocoa/conceptual/ProgrammingWithObjectiveC/ProgrammingWithObjectiveC.pdf>.

Apple, Inc. (Runtime). *Objective-C Runtime Programming Guide*. 2012. <http://developer.apple.com/library/ios/#documentation/cocoa/conceptual/ObjCRuntimeGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008048>.

Apple, Inc. [Clusters]. "Class Clusters." 09 January 2012. *Concepts in Objective-C Programming, in iOS Developer Library.* <http://developer.apple.com/library/ios/#documentation/general/conceptual/CocoaEncyclopedia/ClassClusters/ClassClusters.html>.

Apple, Inc. [Customizing]. "Customizing Existing Classes." 13 December 2012. *iOS Developer Library, in Programming with Objective-C.* <http://developer.apple.com/library/ios/#documentation/cocoa/conceptual/ProgrammingWithObjectiveC/CustomizingExistingClasses/CustomizingExistingClasses.html>.

Apple, Inc. "Object Initialization." *Concepts in Objective-C Programming-* <http://developer.apple.com/library/ios/#documentation/general/conceptual/CocoaEncyclopedia/Initialization/Initialization.html>.

Begemann, Ole. "Faking instance variables in Objective-C categories with Associative References." 12 2011. *iOS Development.* May 2013. <http://oleb.net/blog/2011/05/faking-ivars-in-objc-categories-with-associative-references/>.

Conference of State Bank Supervisors. "Executive Summary of the Sarbanes-Oxley Act of 2002 P.L.107." 2002. *CSBS.* 20 05 2012. <http://www.csbs.org/legislative/leg-updates/Documents/ExecSummary-SarbanesOxley-2002.pdf>.

Defense Logistics Agency. "What is Sustaining Engineering." n.d. *Defense Logistics Agency - Aviation .* 09 05 2012. <http://www.aviation.dla.mil/ExternalWeb/UserWeb/AviationEngineering/Engineering/Sustainment/whatissustainingengineering.asp>.

Doric Loon, et al. "Synchronic Analysis." 15 May 2013. *Wikipedia.* 02 June 2013. <http://en.wikipedia.org/wiki/Synchronic_analysis>.

Drossopoulou, Sophia and Dan Yang. "Perm(co) – A Permissive Approach to Covariant Overriding of Subclass Members." *Formal Techniques for Java-like Programs - Proceedings* 21 July 2003.

E. Gamma, R Helm, R. Johnson, J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

Eclipse Foundation. "Aspectj crosscutting objects for better modularity." n.d. 02 Jun1 2013. <http://www.eclipse.org/aspectj/>.

Erl, T. "Canonical Versioning." *SOA design patterns*. Prentice Hall, 2008. <http://www.infoq.com/articles/3-SOA-Design-Patterns-Thomas-Erl>.

Fowler, Kim. *Mission-Critical and Safety-Critical Systems Handbook*. Newnes, 2009.

Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. <http://books.google.com/>.

Fowler, Martin, et al. *Refactoring: Improving the Design of Existing Code*. 2002. <http://martinfowler.com/books.html#refactoring>.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

Haire, Meaghan. "A Brief History of The Walkman." *Time Magazine* (2009). <http://www.time.com/time/nation/article/0,8599,1907884,00.html>.

Hirschfeld, Robert, Pascal Costanza and Oscar Nierstrasz. "Context-oriented Programming." *Journal of Object Technology* March-April 2008: 125-151. <http://www.jot.fm/issues/issue_2008_03/article4/>.

IEEE. "SWEBOK Guide, V3." 2013. *www.swebok.org.* <www.computer.org/portal/web/swebok>.
informatique. "Software Engineering Terminology." n.d. *Umons Université de Mons.* 2013. <http://informatique.umons.ac.be/genlog/SE/SE-contents.html>.
Karwin, Bill. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf, 2010.
Koopman, Philip J. Jr and Daniel P. Siewiorek. "DESIGN ABSTRACTION AND DESIGN FOR CHANGE." Draft, revised August 16, 1994. 1994. <http://www.ece.cmu.edu/~koopman/abs_chng/abstract.html>.
Liskov, Barbara. "Data Abstraction and Hierarchy." *SIGPLAN Notices* May 1988.
Microsoft. "Partial Classes and Methods ." 2013. *Visual Studio 2012 or maybe C# Programming Guide, it says both.* June 2013. <http://msdn.microsoft.com/en-us/library/vstudio/wa80x488.aspx>.
Mirakhorli, Mehdi and Jane Cleland-Huang. "A Pattern System for Tracing Architectural Concerns." *PLoP* 2011.
Odersky, Martin. *The Scala Language Specification, Version 2.9*. Switzerland: PROGRAMMING METHODS LABORATORY (EPFL), 2011. <http://www.scala-lang.org/docu/files/ScalaReference.pdf>.
Oracle, Inc. "Default Methods." 2014. *Java Tutorials.* <http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html>.
Orchard, Riley. "Compatible Change." 2009. *SOA Patterns - A Community Site for SOA Design Patterns.* Ed. http://www.soapatterns.org/. Arcitura Education. <http://www.soapatterns.org/compatible_change.php>.
Ousterhout, John. "Designing Abstractions, Lecture Notes for CS349W, Fall Quarter 2008." 2008. *www.stanford.edu*. December 2012. <http://www.stanford.edu/~ouster/CS349W/lectures/abstraction.html>.
PBS staff. Credits to Scott Freeman and Jon C. Herron. "Punctuated Equilibrium." 2001. *PBS.* WGBH Educational Foundation and Clear Blue Sky Productions, Inc.. 07 2012. <http://www.pbs.org/wgbh/evolution/library/03/5/l_035_01.html>.
Pomeroy-Huff, Marsha, et al. "http://www.sei.cmu.edu." 02 2010. *The Personal Software ProcessSM (PSPSM) Body of Knowledge, Version 2.0.* Software Engineering Institute.
Reza, J. R. "Java supervenience." *Computer Languages, Systems & Structures 38* 2012: 73–97.
—. "Java Supervenience." *Computer Languages, Systems & Structures* 38 (2012) 73–97 (2011).
—. "The Generatrix Design Pattern." *IEEE SoutheastCon, March 18*. Orlando, Florida. USA: IEEE, 2012. <http://www.southeastcon2012.org/2012/Mar/BROCHURE%203%205%2012%20A.pdf>.
Reza, Juan R. "Accord: Family Polymorphism on Core Java." *http://www.codetown.us/profiles/blogs/class-family-accord* (2011).
Steve Smith. "Don't Repeat Yourself." 24 11 2009. *O'Reilly Commons.* O'Reilly. 09 07 2012. <http://programmer.97things.oreilly.com/wiki/index.php/Don%27t_Repeat_Yourself>.
Sun Developer Network (SDN). "Intercepting Filter." 2002. *Core J2EE Pattern Catalog.* Sun Microsystems. 2012. <http://java.sun.com/blueprints/corej2eepatterns/Patterns/InterceptingFilter.html>.
Ural, Ural. *Oracle automatically keeps the old, historical table statistics*. 04 07 2011. 07 07 2012. <http://uralural.blogspot.com/2011/07/oracle-automatically-keeps-old.html>.
Wikipedia, with contributions from Sony Inc. "Walkman." 2012. 30 06 2012. <http://en.wikipedia.org/wiki/Walkman>.
Wilson, Eduard Osborne. *Consilience: the Unity of Knowledge*. Vintage Books, imprint of Random House, 1998. <http://wtf.tw/ref/wilson.pdf>.
Yamashita, Aiko Fallas. *A multi-method approach for evaluating software maintainability and comprehensibility*. Box 134, 1325 Lysaker, Norway: Simula Research Laboratory, 2008. 09 07 2012. <http://simula.no/research/se/publications/Simula.SE.677/simula_pdf_file>.

Received May 2012; revised September 2012; accepted October 2012