Pattern Language: Coherent Generational Design

Author: Juan Rolando Reza

Keywords: Generatrix, agnosticity, form-agnostic, abstract design, generational awareness, constructor, builder, provenance, maintainability, strategic design pattern, accord, source control, versioning, software evolution.

A. Abstract

A set of design patterns for dealing with multi-generation software systems is presented. This pattern language focuses on issues that arise when a software system goes through a series of quantum leaps, major new releases, or technologies. A central design pattern called Generatrix seeks to promote a kind of cohesion from one generation of a system to the next.

The key objectives are:

1. Maintain knowledge underlying the design of previous generations of the system so as to prevent re-invention, facilitate re-use, design drift, and support different implementation forms allowed by design intentions.
2. Establish a development practice that will support these objective in the next generation.
3. In recognition of experiences that find this concern to be a pervasive source of problems and very amenable to redress:
   - 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.
5. Provide a simple metric that can help organizations recognize the effort-saving from the techniques suggested here.

A novel kind of abstraction is introduced to support the patterns of this paper called agnosticity. Simply stated, it is a way of defining function without committing to a particular form or design, as contrasted with conventional abstract classes which define forms (e.g. signatures) and leave semantic details to implementations.

The same kinds of design re-use and maintainability issues arise in all SDLCs from Waterfall to Agile. They take on a different character in large-step generations of a software product. This pattern language attempts to bring out these qualities and provide an approach to the stated objectives.

The world of business information systems and that of analytic software for scientific and engineering purposes have quite different interests, languages and practices. Both business and technical fields, however, have in recent years experienced the consequences of failing to maintain the capability to reproduce evidence and results from past generations of their information-processing software. This calls for technical provenance. Laws are enacted to regulate financial institutions, leading to large, abrupt demands on their MIS systems [1]. In a comparable way, science journals have withdrawn research papers after discovering that research results 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* [2]. Generally, continued development with legacy systems leads to ever increasing cost [3].

Motivated by these interests, the pattern language presented in this paper seeks to illustrate some experiences and workable approaches to toward coherent design.

B. Overview

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. As an overview, the definition and elaboration of terms is necessarily given in the upcoming sections. Briefly,

- Form-Agnostic patterns serve two objectives of this pattern language. First, they allow diagrams to express a pattern more generally than would be possible with conventional class-like diagrams. These abstractions capture purpose without committing to a particular form. Second, by virtue of enabling us to create form-agnostic designs explicitly, the design implementations need not become disassociated from the purpose expressed by their abstractions.

- Form-Agnostic Software Element is the abstraction for a design that specifies functionality, qualities, and purpose but intentionally keeps the form of implementation flexible. The design implementation artifacts maintain attached to their underlying abstractions.

- Form-Agnostic Builder represents any creational design pattern; the non-specific means of creating and populating software artifacts and elements, in the broadest sense, called Parts for convenience here.
- The generational design has Part-Builder pairs, shown with small boxes labeled B and P. modified part.
- A key feature of this pattern language is the concept of a predecessor relationship between a part and an earlier design addressing the same or earlier set of requirements.
- Accord (design pattern): 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.

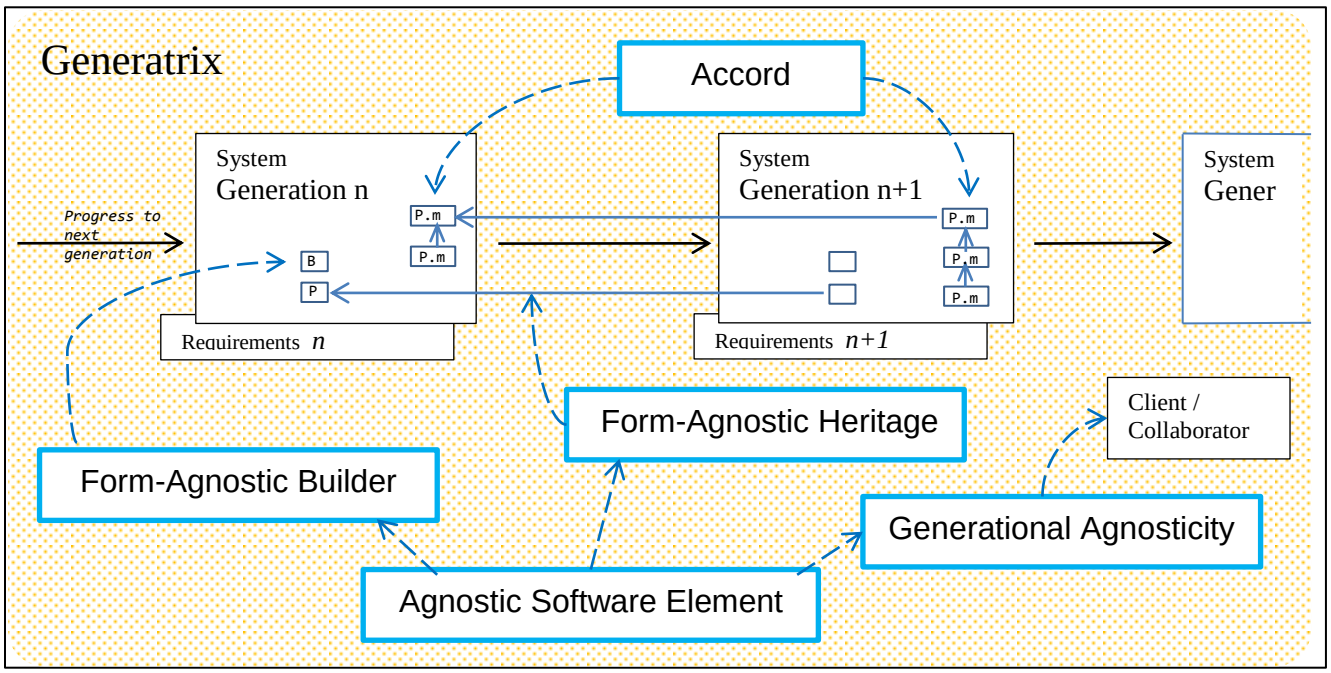
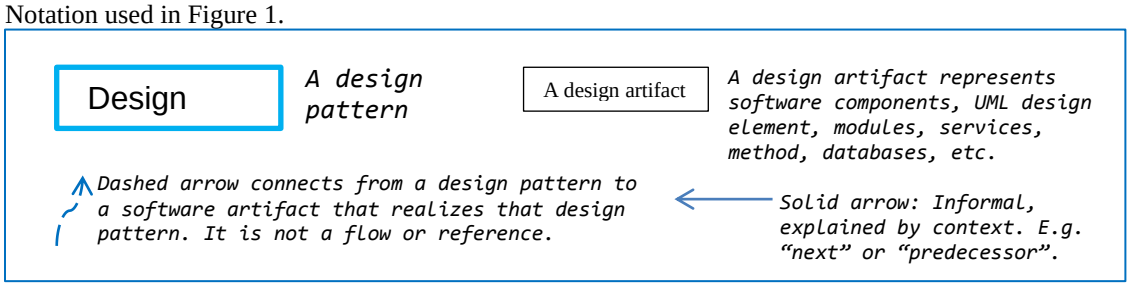


Figure 1. Informal view of Generatrix as a collaboration of patterns, applied to several generations of a software system.

Notation used in Figure 1.



C. Terminology

- **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 from one generation to the next.
- **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 smooth continuous pattern of small changes. A series of generations is more like *punctuated equilibrium* [4], to borrow a term from evolutionary biology.
- **Software maintenance** - Maintenance updates do not warrant a new generation. Maintenance programming, strictly speaking, maintains the set of functions under the same set of requirements. Its objective is largely to fix errors, improve qualities such as performance, and accommodate changes in its environment through configurable elements,

and sometimes to complete the implementation of requirements that were deferred to let the product be released "good enough" on schedule.

- **Pan-generation operations** – typically in some large MIS[1] systems, statistical analysis and reporting requires several consecutive generations of the system to inter-communicate and run at the same time (perhaps in the same process, address space, JVM, say). This architecture gives rise to the need for some technique for making components loadable together which creates a maintainability issue. Contrast this with "multi-generation" product line which may include several generations of the system without inter-communication.
- **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 may be between any software artifact including a method, a class, interface, package or group of artifacts (and not necessarily between artifacts of the same kind).
- **Accord** – one or more components with a dependency on a component of the same generation. This relationship is easily explained with an example. A classic paper on the Liskov substitution principle [5] describes a simulation of a Car and a Driver which are extended to a RaceCar and RacingDriver, respectively. 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, say. The predecessors, Car and Driver, are compatible in their generation. In Accord, this same-generation dependency is explicitly represented and is not limited to the inheritance construct. It is can be thought of as components with parallel heritage.
- **Part-Builder relationship** - A builder in this paper represents any creational pattern that creates, initializes, and/or populates a Part. A Part is any software artifact within a component or system. A Part may have a non-null predecessor in the previous generation(s) of the system or component. A builder is usually a Part of the system as well. A component may be designated Part or Builder to indicate its role for the point being made although it may be both.
- **Resource** - a component that is designed to be stable across generations. Example: the programming language, database, interfaces. However, it can be a Part at another perspective and undergo generational change.
- **Resource** - A software component that is referenced by components of more than one generation. Just as Builder may be a Part, and Part may be a Builder, the designation of Resource is used to point out its role in a particular context while the resource may also play the role of Part and Builder from another point of view. A resource obtains no generational awareness by its role as a resource. It can be an ordinary component.
- **Agnosticity** – the quality of a component whereby it is intentionally insensitive to a form, generation, or compatibility in a way that allows more flexibility in the design.

D. Audience

Before stating the intent of this pattern language, based on early reviews of this paper by practitioners from different backgrounds, it seemed useful to state who the pattern would be most applicable to.

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[2], 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.

(However,)

- Developers and researchers whose audience consists of other programmers or their own academic research might *not* have the problems addressed by this pattern language.

E. Intent

We recognize the need to improve productivity and quality of software systems when a quantum leap of technology or requirements takes place. The need to make the future requirements part of the present set of requirements is considered a viable way to ensure design for re-use, prevent duplicative and incoherent code, and preserve the knowledge of abstraction (unwritten) relationships underlying previous generations of the system.

---

[1] MIS – Management Information System, typically enterprise scale, massive database-centric software systems.

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

The Generatrix design pattern was presented in stand-alone form recently [6] as an attempt to address the (above) stated intent. Based on discussion of that pattern the areas that are difficult to explain came to light. It was decided to create supporting design patterns around the Generatrix. The vocabulary and illustrations in this paper now use the supporting patterns. This separates the presentation of the central concept of coherent generational design from the explanation of the supporting ideas of agnosticity, accord, part-builder abstraction and the informal notations.

The previous work also revealed an interest in showing how we might predict and measure the benefits of applying the Generatrix pattern in a progression of software system designs. If the requirement does not call for work that plans for the future of the system, we can't spend the time. A simple metric has been introduced in this paper for that purpose.

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. There is no formula for deciding when a system is "a new generation", being an intuitive concept that depends on the kind of system. The section What is a Generation? illustrates what is meant by a generation in this paper.

F. Motivation

The need to deal with degradation of architectural intent has been recognized as a problem in the software development lifecycle in general. The work of Mirakhorli and Cleland-Huang [7] gives us a possible solution based on adding traceability from design elements to underlying design decisions. The intent of the pattern system for architectural concerns identifies the problem of degradation of the architecture. As small coding iterations are performed during maintenance, the changes depart from the architecture 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 problem area is the loss of quality and knowledge that takes place from one generation of a system to the next (a "generation" loosely defined as a "quantum leap" of scope, technology and/or architecture). Maintainability is not just for maintaining the requirements of the generation being supported but to maintain the qualities that will help reuse and derivation in the next generation of the system for new requirements.

The Coherent Generational Design pattern language does not conceive of solving problems by creating requirements traceability as in [8]. Instead, the approach is based on design patterns that strive to capture knowledge of design intent that will survive into the next generation of the system (through a series of generations).

> Abstraction. Knowledge that is captured in documentation or other forms becomes detached from design and software artifacts. The form-agnostic design patterns seek to remedy this be supporting more design knowledge in the same kinds of artifacts as the conventional design and implementation.
>
> Predecessor reference. Knowledge is often lost when upgrades and replacement components become disassociated from the items they came from. Also, when components delegate or inherit from other components that existed in the predecessor generation of a system, the use of this relationship for purposes of past reproducibility is lost or reduced.
>
> Part-Builder. When the design of a component is intermixed with its building process, the potential for maintaining and deriving from them is reduced.

This pattern language is motivated by another interest which is separate from design and code techniques. To promote planning for the next generation, it addresses a human factor in the software development process. This pattern language helps a project to be more "plan-like". 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*." [9]. Experienced programmers are naturally concerned for the next generation and can feel thwarted in projects with immature planning habits.

Summary of the areas of interest for this pattern language…

- Organizational "Software Development Lifecycle" (SDLC):
  - Lack of an effective (operational) way to build into the software requirements, schedule and budget the mandate for programmers to write maintainable, understandable, code with anticipation of future use.
  - Programmers who are team-players are punished for their labors if they take the time to bring together incoherent code sections which will later benefit the product. This deals with human organization factors in development [10].

- Coding issues:
  - Repeated chunks of code that are not readily maintainable are hard to prevent and eliminate.
  - Ad-hoc naming and classpath tactics are used when multiple generations of a system must run together. This practice creates extra copies and various naming conventions for versioning.
  - Coupling between large components that could be made into cohesive dependence at lower granularity.
  - Solutions to capture generational dependencies and traceability are available for specific languages that do not work with a system having multiple languages. And solutions are available that are so general that they create maintenance overhead and too easily fall by the way side.
- Architecture issues:
  - Loss of knowledge of which modules belong to which generation; loss of opportunities to use delegation due to lack of an explicit reference to the predecessor components.
  - Loss of knowledge of which modules depend on other modules that must be of the same generation.
  - Tendency to start a new generation either completely from scratch or continue with the legacy base.

G. What is a Generation?

Very succinctly, a new generation of a software system is characterized as a quantum leap in features, requirements addressed, or underlying technology. The concept is intuitive and has no general formula for distinguishing a generation from some smaller iteration or release. It is different for different kinds of applications. One ingredient of the concept is whether the system has been published [11]. But any minor iteration can be published as well.

This section describes two very different kinds of application going through multiple generations. This extended explanation of the concept is a response to earlier presentations and reviews in which the audience tended to understand generation as equal to major release or such. Generations of Sony Walkman, and Generations of an Insurance product are used as the examples of multi-generation systems.

1. Multi-generation product line; Sony Walkman

The Sony Walkman product line had 3 or 4 generations of products called Walkman [12]. These generations could be called, in summary fashion, (1) the cassette-based Walkman, (2) the disk-based Walkman with several codecs, and (3) the flash memory-based MP3 player. A 4th generation would be the ear-bud format, as a matter of opinion that it is still a Walkman in that form.

Figure 2. Three generations of the Sony Walkman

The three generations of the Sony Walkman are shown in Figure 2 [12], [13]. Just by looking, 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. The next (center) is clearly a new generation of the product. The visible change is due to 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.
The Walkman illustrates 3 kinds of **quantum leap**. Each kind is embodied by a new generation.

1) Creation. A new set of technologies and requirements are established (from no predecessor).
2) Technology is the quantum leap. Replace tape with flash memory. Requirements basically follow.
3) 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. Although these were technologies to the product, they were not the driver but the means to the quantum leap.

During the life of each generation, there were undoubtedly new iterations of internal software, stable or tagged releases, some hardware changes, and fixes. Internal to the development team, some of those iterations might have embodied a new generation of software architecture such as change of language.

Software systems and components progress in quantum leaps from time to time: a "total redesign", introduction of a new language or platform, or new tools may be the defining basis of a new generation of the system based on Technology.

Software systems can have large game-changing requirements placed on them such as requirements beyond configuration and refactoring to implement [14]. This fosters a new generation of the third kind as illustrated in the next example. Often, such requirements are implemented without appropriate reengineering and the development process struggles.

2. Pan-generation insurance product line system

This example of a real-world insurance product line has no gadgets to illustrate.

1) Each year, and sometimes more often, 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 court decisions, which may involve completely rerunning the system as it was in a year past. The requirements also come from new compliance mandates of Federal and consortium sources. Unexpected requirements also arrive from the occasional tectonic shift of an acquisition or merger.

2) Important DSS[3] functions run 3 consecutive "years" of the system in order to determine charges, trends, and various statistics about groups of clients. In the example 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". Consequently, each year hundreds of lines of code across the whole system are tweaked with conditional logic (e.g. if year = '2009'…). 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.

3) Database structures are designed to reference basis data in the past year, thus making it impossible to separate the data of 4 years and older from the current set.

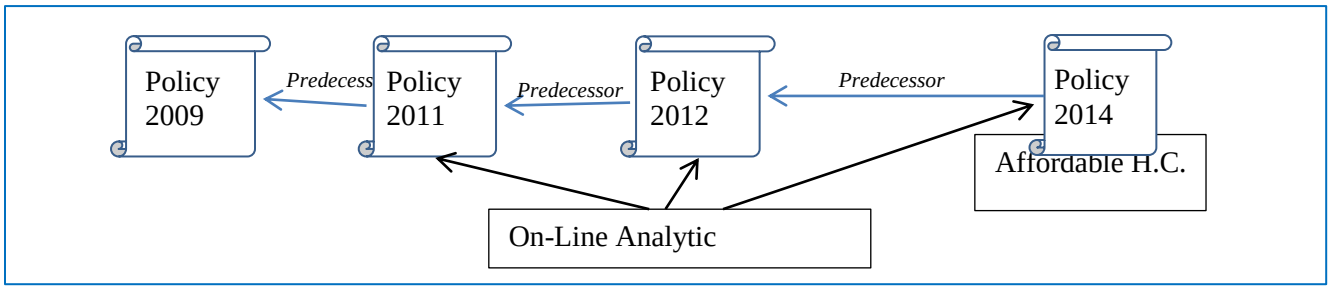


Figure 3. Pan-generational design: insurance system example.

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 short projects. There is no time to assess the global situation. As such, the system can build up toxic code over time and periodically experiences crises.

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 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.

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.

[3] DSS: decision-support system

H. Structure

The core of this pattern language is the Generatrix design pattern. The supporting patterns are:

- Accord Design Pattern
- FASE: Form-Agnostic Software Element
- Form-Agnostic Heritage
- Form-Agnostic Builder
- Generational Agnosticity

A Form-Agnostic Builder may also be a Part with a predecessor.

Example 1. A constructor serving as the design instance underlying an Agnostic Builder is the predecessor of a Template Method.

Example 2. An Abstract Factory, which builds a number of parts perhaps with a number of part-specific builders, has a simple Singleton as its predecessor.

These examples would be illustrative of a situation where the predecessor had fewer concerns and requirements than the new generation. The predecessor can be a different implementation of the same design pattern.

The Form-Agnostic and Generational Agnosticity patterns exist essentially to define features of the Generatrix design pattern. They provide vocabulary and diagramming tools for explaining and using the Generatrix pattern. Their role in a software system design is comparable to a blueprint for making the product blueprints.

The Accord pattern can stand on its own as well as playing the support role like the others. It can be useful without the context of Generatrix as a metaphor for a generational system.

I. Future work:

- The appendix contains the first generation attempt to provide in-language support for the concepts of the Generatrix and its support patterns. It has two deficiencies. It may clutter the actual design with artifacts that are pan-generational concerns or abstractions. It also depends on the language to support the `@annotations` feature. The next generation should provide language independence to eliminate these obstacles.
- The possibility of an IDE with the capability to understand the form-agnostic classifier should be explored. It would be useful to allow refactored elements to retain knowledge of their form-agnostic origin so that they can be re-refactored again and again.
- Form-Agnosticity compliments form-explicit abstractions. It seems worth exploring further. As a concept it can work stand-alone. It may have broader applicability than the pattern language of this paper.

J. Conclusion

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 is continuing in the context of a large financial institution with a multi-generation MIS system. The concept of Form-Agnosticity should help to untie design from the single generation at hand.

The pattern language described in this paper presents a set of design patterns supporting multi-generation software development. It suggests specific design patterns and suggests changes in software development paradigm could lead to a more **timeless way of thinking**.

II. Generatrix Design Pattern

The conventional practice of software maintenance may be the nemesis of multi-generation software systems. In the interest of changing and adding features to a system, 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. The cause of irreproducible results becomes an untraceable mystery. Conventional source control systems alone do not effectively capture the evolving knowledge and

certainly do not magically discover the point at which a system or its data have made a quantum leap. Simple design rules such as "checkstyle" [15, p. 1.2.3. Forms] do not adequately address architecture-level design change issues.

A. Intent

Software development practices often neglect opportunities to design for the next generation of the system. The new generation often involves a lot of reverse-engineering. Interdependencies between components that were intended to work together in the same generation may break or lead to reinventing the components. 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.

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.

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.

B. Motivation

- Software developers and non-technical managers need a common vocabulary for discussing the design to support requirements that are expected to, or have already been changing.
- As a software manager (any stakeholder other than developer) *you have experienced difficulty getting what you need from programmers, on time, on budget, and when software changes you experience a loss of control; you ran an analytic program that now behaves differently for unknown reasons. You said "design for maintainability" but you can't tell if it ever has an effect except that developers say they have done so. They complain about how previous designs are unreadable. They complain if you mandate documentation overhead work.*
- As software engineer or similar participant in development, *you have sometimes had difficulty taking over projects or legacy systems from predecessors; it seems they left a maintenance headache. You and your stakeholders want to design for maintainability but that is not in the spec and schedule and you didn't give then a way to do so.*
- Software engineers are facing an installed base of legacy software that was not designed with any particular strategy to ensure its maintainability. Every company has its own approach, if any.
- Ad hoc naming conventions, duplication of modules, and ad hoc conditional logic sensitive to system version introduce complexity and cumulatively make maintenance and evolution more difficult and costly.
- The maintenance of concurrent generations of a system becomes more difficult and creates more opportunities for defects.
- Developers often make complete copies of previous generation components in order to make small changes, rather than creating small new components that share the functionality of their stable predecessors.

C. Forces

- It is difficult to write actionable requirements that cause programmers to write code with (1) no duplication, (2) no diffuse creational activities, (3) attention to separation of concerns, and (4) appropriate flexibility for future design reuse, extensibility, and the lifecycle of a next generation.
- Programmers are not given a design pattern that would provide a conceptual and actionable framework for designing-in the features supporting cohesive generational development.
- An application began as a small single-purpose program. Over months and years, it grew to support more functionality. Programmers used an ad-hoc approach. Requirements were informal. The program is nearly impossible for new staff to understand, maintain, and develop new functionality.

- 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." [16]

D. Structure

Throughout this paper, the UML-like class diagrams take some liberties with the notation. The method and property compartments are not shown. The Predecessor reference usually has plurality of 0..1 but can have 0..n.

The first example of structure, Figure 4, shows a combination of Parts and Builders with their relationships across generations. The separation of builder and part 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. Part T_j was derived from, and might even delegate some function to Part T_i in a previous generation of the system. In generations *j* and *k*, the Part and Builder are shown both evolving. This pair is a candidate for the Accord pattern. As shown, the Parts only work with the partner of the same generation (although that is not the main point of this diagram). The PartT in generation *i* may be a legacy component, denoted with the informal archetype `<<part ?>>`.

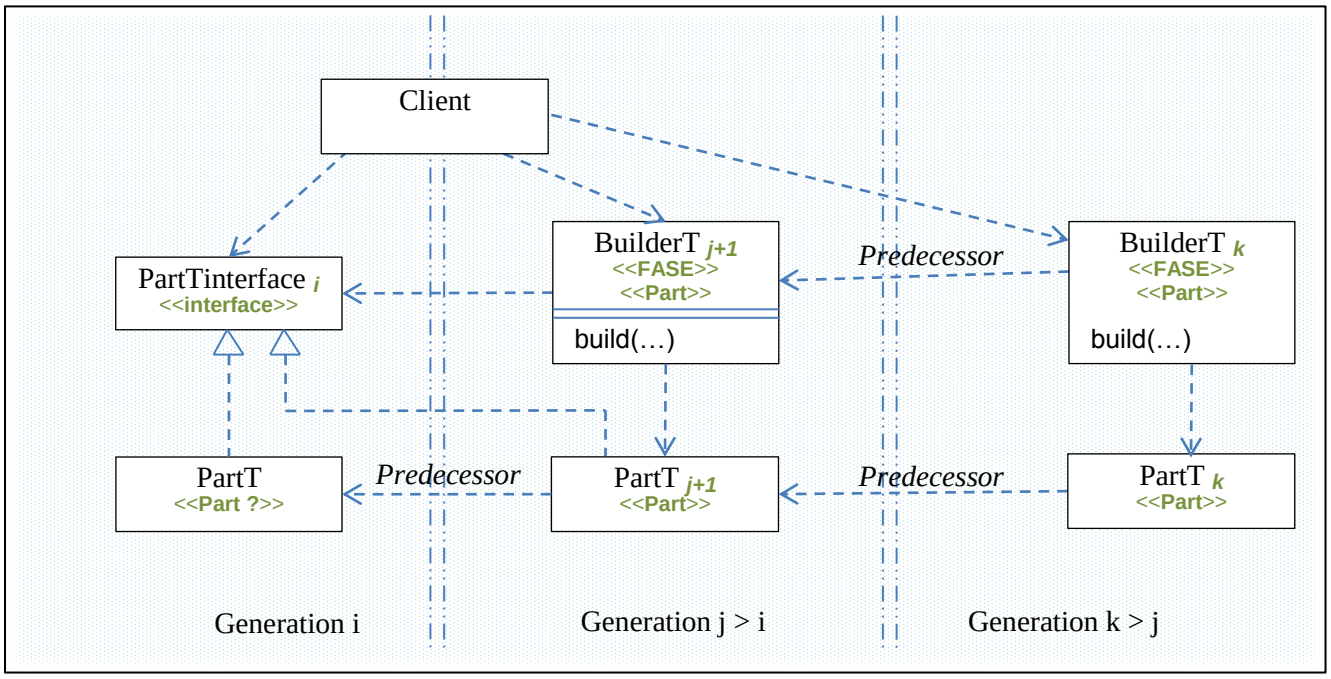


Figure 4. Illustration: a Client invokes the generational system.

The configuration in Figure 4 illustrates a complex of Part components. The Client component is shown disregarding the generation borders to indicate that the client is completely unaware of the generation nature or version.

The scenarios in Figure 5 illustrate the basic configurations that may be recombined to create the previous example.

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

Scenario B. A Part is modified in the next generation, but the same Builder is able to build it. Presumably, the Builder can use some information to build the Part that makes the part different. That difference could be something as simple as the release version of the system.

> 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 $_j$ Builder that causes it to build a Generation $_{j+1}$ Part. That could simply be the part to the implementation of the Generation $_{j+1}$ Part.

Scenario C. A Part is built differently, but its design did not change in the next generation. The Builder was redesigned for reasons that could include defect correction, completing requirements, or improved qualities. The notation <<FASE>> is an "archetype" designating the classifier (the box) as a Form-Agnostic Software Element.

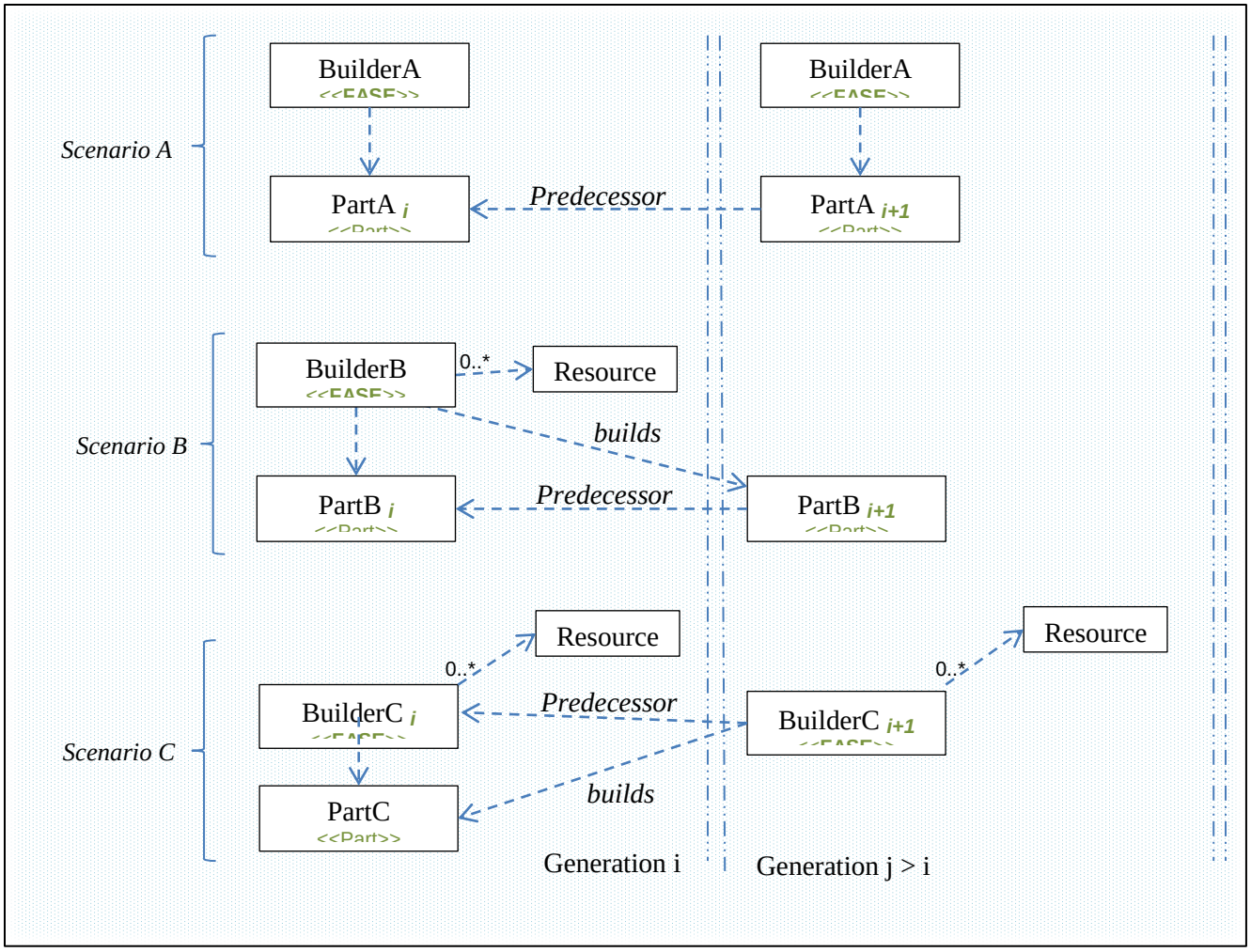


Figure 5. Part-Builder configurations scenarios.

A tangible example of each Scenario will help understanding. Referring to the race car example mentioned earlier, we can envision a design in which the complex objects Car and RaceCar are built at runtime in a simulation model or game.

Example application of the scenarios:

Scenario A. The Car design in generation *i* is built by BuilderA. The RaceCar is so different that it has to be built by a completely new kind of Builder. However, significant parts of the RaceCar simply delegate to the original Car design. (Although it might seem that any builder of a part with a predecessor must be considered to have a predecessor builder, the focus is on whether the new builder design has benefitted from the builder of the previous part.)

Scenario B. The RaceCar in this scenario is not much different from an ordinary Car. As such, the Builder from the previous generation of the system can build the RaceCar in the same way. Presumably, the Builder simply retrieves a new powerful Engine from the Resource and plugs it into the same design.

Scenario C. The race car example doesn't exactly support this scenario because the new generation does not define a race car. Bending the example a bit, we'll say that a conventional Car will be used in a race only if its Builder satisfies some regulations in the building process such as checking the Driver's license type at the time of delivery to the race track. The new Builder delegates much of the component-wise building of the Car itself, but makes a careful record of the safety-tests and manufacture sources of each part during the build.

E. Applicability

Use the Generatrix pattern when . . .

1. Multiple generations of a system must run simultaneously to perform an analysis or reporting result and it is not practical to create a single system capable of running as each separate system does.
2. It is desirable to maintain and develop the system with the next generation's development in mind. We need an actionable and measurable way of expressing requirements and designs to this end that a range of stakeholders understand.
3. Each generation becomes more complicated because attention only to immediate requirements has caused a lot of code that is duplicated or can be consolidated.
4. The architectural knowledge underlying the predecessor tends to get lost. This includes the loss of knowledge about components that must be of the same generation.
5. 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.

F. Measurability

When developing new features based on similar code from the past generation, source code is often duplicated and varied. There is a cost to discovering how to create a slightly more general design with parameters and configuration settings to achieve the new variant functionality. Duplicative code is just one, though important area that can be addressed to improve generational re-use and knowledge maintainability.[4]

In this section, the problem of duplicate code is contrasted with parallel programming. In a sense, making a single programmer (processor) maintain multiple (parallel) code segments that should be consolidated into a single place is like decomposing an algorithm into multiple duplicate pieces but running it on a single-processor system anyway.

If we had the luxury of supplying multiple programmers to act on each code snippet in parallel, we would be all set. Instead, we have the opposite of the cost-saving availed by parallel algorithms. With this metaphor in mind, we can use a formula for the cost of *duplicative* code based on Gustafson's Law [17], applied in reverse. Gustafson tells how duplication leads to speed-ups on multi-processor systems. We take it as obvious that extravagantly duplicate (or nearly duplicate) code can reduce programmer productivity. The result is a "loss function" [18] that will help decide which features of the application to reengineer. Given a reasonable first-approximation of a loss function, a control chart [19] based on the loss function can assist QA in evaluating how well the redesign matches the generation pattern. The subject of using control charts and loss functions is complex and further detail about they're potential application is beyond the scope of this paper.

Gustafson's Law is stated very simply as…

(1) $S(P) = P - \alpha (P-1)$, where
P is the number of processors,
S is the speedup, and
α is the non-parallelizable fraction of the parallel process.

By analogy, multiple near-replicate code snippets are like having multiple work-nodes. It saves time if we have multiple hardware processors. But it represents losing time when each of the codes snippets must be found, reverse-engineered, updated, unit-tested and designed-updated by a single process: the programmer.

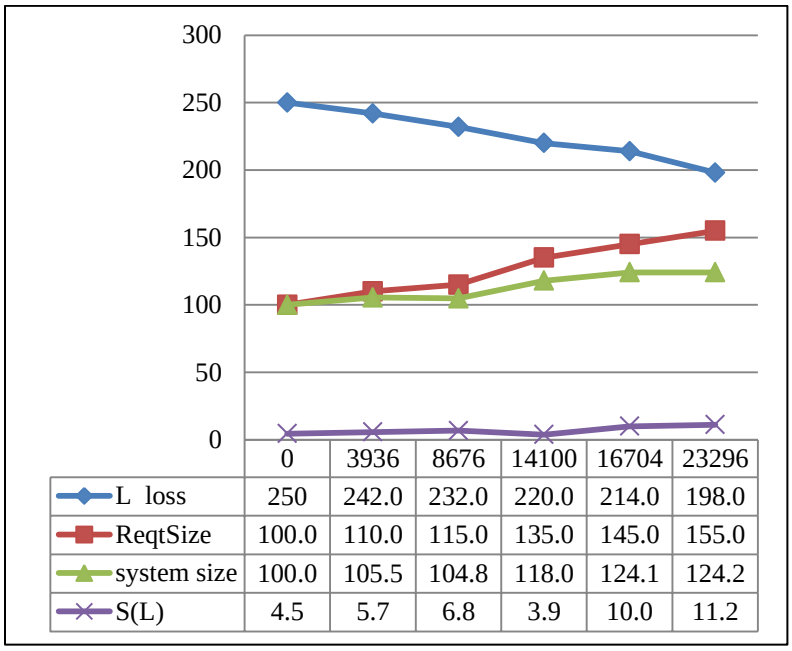
A small computational experiment was performed to create an illustration of this concept. A hypothetical development project with multiple generations was created in which each generation allowed a certain percentage of the duplicative code to be coalesced. Each generation involved more new requirements. So there is a tension between the forces that increase the complexity (hence maintenance load) of the system and the drive to simplify.

A loss function based on this scenario is defined as:

[4] Other potentially measurable improvement areas include: Architecture becomes deeply inappropriate to the new set of requirements. Code base cannot be changed without breaking many things. Trace from requirements or behavior to code is lost.

(2) S(L) = L' - **α** (L -1), where

L = number of code snippets that could be consolidated (using part-builder refactoring).
L' = number < L which we actually decide to change in developing a new generation of the product
α = number of code elements that cannot be improved by consolidation (using part-builder refactoring).
S(L) = the relative number of opportunities for developers reduction of effort.



	0	3936	8676	14100	16704	23296
L loss	250	242.0	232.0	220.0	214.0	198.0
ReqtSize	100.0	110.0	115.0	135.0	145.0	155.0
system size	100.0	105.5	104.8	118.0	124.1	124.2
S(L)	4.5	5.7	6.8	3.9	10.0	11.2

Figure 6. Loss function showing declining relative effort or cost.

The chart, Figure 6 illustrates the loss function (1) for a hypothetical software system with 6 generations in which each generation increases its scope (more required components). In this example, some components are coalesced during the work of each successive generation. The duplicative components are refactored using the Part-Builder scenarios. Some identical duplication is eliminated and other cases of hard-coded differences may be factored by parameterizing Builders and Parts. This avoids or reduces the **system size** (proportional to the number of requirements) that *would* have resulted from a headlong reaction to immediate requirements (without regard to using common code). The loss of potential productivity due to the targeted set of duplicative components is expected to decline at each generation.

Although we cannot predict or measure the actual amount of effort associated with maintaining or eliminating a given element, we can count the elements involved. By identifying duplicative code elements and making the extra effort for maintaining them obvious, this pattern could help to expose a design as embarrassingly duplicative in a way that is analogous to algorithms that are "embarrassingly parallel." [5] The outcome should be seen as an opportunity to justify the up-front work needed to reduce the avoidable work for the benefit of the next generation of the design.

G. Known Uses [6]

- Legacy tax analysis program
    - A database-centric legacy program had become very difficult to maintain and change. Portions of the code base were redesigned to simultaneously function for more than just the current. Each year, new

[5] In Parallel algorithms [28] , Aaron Harwood 2003-10-22, "*A problem of size is embarrassingly parallel if it is quite easy to achieve a computational speedup without any inter-process communication….*"

[6] The known uses described here are from the author's experiences at several organizations. The literature is full of examples of software redesign to improve reuse and maintenance with other techniques but they would not demonstrate a known use of *this* pattern.

triennial processing requires massive manual tweaking and duplication of code. No one knows if the analytics and reports actually meet regulatory mandates.

- The effort began to identify the most numerous duplicative code chunks and then attempted to coalesce them under the part-builder pattern.

H. Related Patterns

- Abstract Factory, [20].
  - 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 Parts. All components of each alternative are different and specific to the whole variant under the Abstract Factory.
- Chain of Responsibility, [20].
  - 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. See Figure 9 which illustrates different predecessor designs.
- Intercepting Filter, [21].
  - Filter chains in SOA, can modify messages along the way. Just as a generational component was distinguished from a Chain of Responsibility, here the component can consume the request and respond without delegation.
- Façade, [20].
  - A generational component can be designed to accept requests that can be 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.

I. Summary

In practice, no technological fix can overcome sub-optimal work by people. The generational pattern described here can provide one tool for encouraging, verifying, and enforcing certain qualities in a design. This pattern does not create ever more complex regulations against the next software crisis. Instead of trying to impose rules and guidelines, this pattern provides a vocabulary to include generational awareness in requirements. It suggests elements to promote inter-generation cohesion and knowledge transfer: the predecessor and the accord.

III. Accord Design Pattern

A. Intent

Two or more components have a dependency coupling to each other. Their coupling is also specific to each generation of the system. Consequently when they are upgraded in the next generation, they are not interchangeable with their respective predecessors. This is an accord relationship [22].

In an object-orient language such as Java, functionality is can be added by extending a class with additional methods or method overrides. The override may only work with a parameter of the new generation. When inheritance is used to add functionality in a new generation of the system, the "extends" is also a "predecessor" relationship.

The dispatch mechanism of the language determines which override or overload is called according to runtime types of parameters, after compile-time constraints of method families according to compile-time types.

Inheritance is only one form of predecessor relationship. The same-generation coupling of components can take other forms such as Chain of Responsibility, Decorator, and compatibility mechanisms in service-oriented architectures.

B. Motivation

The knowledge of when components must work only with collaborators of a certain generation is easily lost. There is no notation in UML or typical languages to express this abstraction directly.

C. Applicability

Use the Accord pattern when …

Two "partner" components are designed to work together fully and properly in the same generation.

An upgrade of functionality to either partner in the next generation warrants a corresponding upgrade in the other in that generation.

It is a solid assumption 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.

D. Structure

Returning to the example of the race car simulation mentioned earlier...

In the first generation of a fast conventional car, there is a fuel pump and flow controller. 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.

If Java inheritance were used, 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.

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

E. Implementation

Any form that the Accord relationship takes will have qualities exhibited by 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.

IV. FASE: Form-Agnostic Software Element

Also Known As: Abstraction of form.

This design pattern introduces a design concept which is the converse of the usual "abstract class" or "interface".

A. Intent

This pattern language needs to have an abstraction that defines semantics without form, which we can call form-agnostic software elements. As an example, rather than presenting a UML Class symbol for a particular creational pattern, present an abstraction for the builder of a part.

To support the expression of the Generatrix pattern, we need an abstraction that defines the semantic purpose without committing to a particular form

Conventional object-oriented languages and UML class diagrams offer `interface` and `abstract class`. These define "form" and leave purpose and semantics to separate definition (in a concrete class). In contrast, an element that defines purpose and semantics, but leaves the form unspecified is an abstraction we call Form-Agnostic.

The form-agnostic design should be carried throughout the life of the system as a means of supporting the maintenance and redesign of the components in the next generations. Without this "blueprint" for a specific design, the work of understanding the intent of a connected set of components is left to the designer to remember, or mentally reverse-engineer, figure-out from documentation that falls out-of-date.

The higher abstraction should cover not only the classifiers of the language, but other software artifacts.

B. Motivation

- A design pattern dealing with generations of a system needs a graphical and conceptual means of expressing design elements purpose without specifying their form. The form of the implementation is be detailed separately.
- 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.
- A similar concept applies to "reference" and "method call" in that the function of a "reference" or call can be accomplished with various different mechanisms.
- The concept of a creational element may take the form of a constructor, retrieval from a database with implied instantiation, factory methods, static instantiation, and other mechanisms.
- Another scenario: the parent constructor is still accessible in the derivative class, which might not be appropriate for the derivative class and, strictly testing, should be shadowed with a matching constructor that would issue an exception. The "trick" of invalidating a construction or override method for the purpose of taking it out of the "type" is not directly supported by Java. Doing so would require a supervenience feature [23].

C. Example

The concept of a form agnostic design element can be illustrated by analogy to architecture (in the sense of cities, bridges, houses, transportation systems). Suppose a professional office complex is being designed. The architect defines a front entrance 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 design "does not care", in a sense, and the customer "does not know" what will be built.

After the building has been constructed and is in use for several years, new requirements lead the maintenance crew to plan to replace the revolving door with a heavy security hinged door. Blueprints retain the information that the particular door is considered unimportant to the overall design,.

D. Applicability

Use a Form-Agnostic software element when:

- Designing at a level of abstraction in which particular design decisions are not necessary.
- 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.
- Redesign involves more than simple refactoring. (Refactored code is not so involved that its concept will be lost. Simple name changes or collecting lines of code into a method don't need the abstraction).

E. Implementation

The form-agnostic elements are represented here with class diagrams deviating slightly from plain UML. Architype annotations express the abstraction intended. Unlike the normal use of class diagrams, no compilable code should be generated by an IDE supporting UML with conventional round-trip engineering.

A class diagram can be used for sketching an application in the abstract. The class elements (boxes with compartments) can be annotated as described in the section in section II.D. Structure.

In normal class elements, a constructor is depicted within the class of objects it constructs. Form-agnostic abstractions require that the conceptual builder of a class is depicted as a separate component. It may be realized in the form of a constructor or it can take the form of one of the many creational patterns. The form-agnostic Part which it builds may itself take the form of one or many classes and other design artifacts.

The Predecessor is a kind-of abstract reference. There could be other uses for this abstraction of a reference, but the Generatrix pattern requires this notational mechanism identifying the component's predecessor. The annotation `<<Part?>>` in Figure 7 is intended as an informal reminder that a component in a legacy "generation" was not under the Generatrix design pattern. This notation helps a legacy system to be brought into this approach.

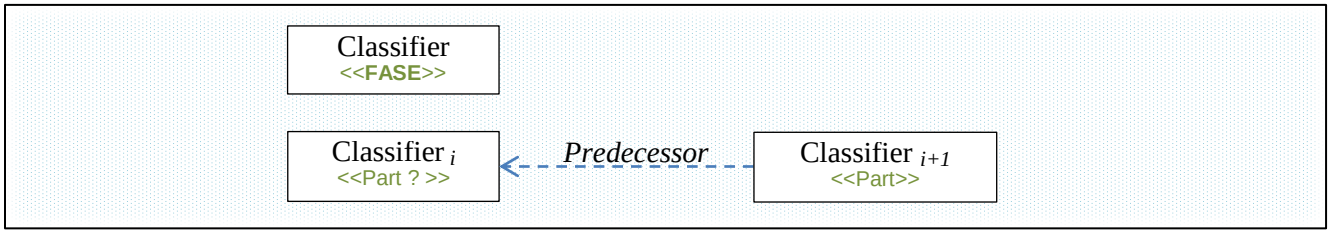


Figure 7. Notation for the Form-Agnostic Software Element and the Form-Agnostic reference

F. Consequences

- Developers are freed from having to select particular mechanisms for construction.
- The SDLC culture should treat the abstract design artifacts as part of the current design.
- It is hard for anyone to avoid committing to a particular form or shape when expressing an abstraction. Any effort to "standardize" a formless way of doing so would seem to be a self-contradiction. Nevertheless, capturing the intention of a design in some form can be done, making it clear which aspects are allowed to take different forms.

V. Form-Agnostic Heritage

Also Known As: the predecessor relationship

A. Intent

A higher level abstraction is needed to represent the relationship between the design of a software component and an earlier design of the component in a previous generation of the system; its "heritage".

Software development productivity is reduced when the trace to earlier implementations of a given component are lost or confused. Merging and splitting as well as renaming and moving to different parts of a hierarchy contribute to this loss of trace to the predecessor(s) of a given component. An explicit representation of "predecessor" is believed to be a practical way to reduce this loss. The word "heritage" informally alludes to its cousin "inheritance".

B. 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.

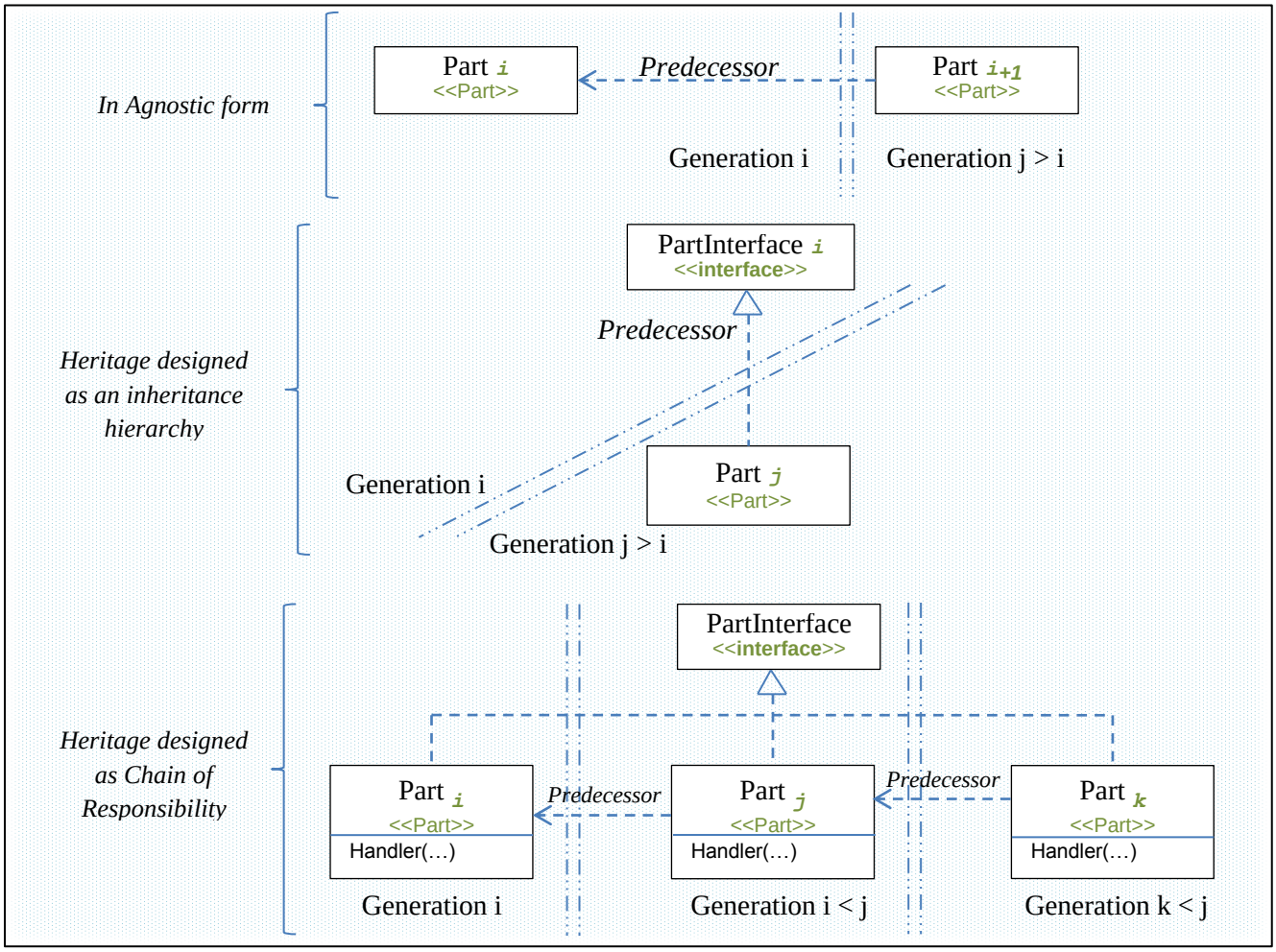


Figure 8. Form-Agnostic Reference used to represent Predecessor: 3 configurations.

The 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.

In Figure 8 the interface of a Chain of Responsibility is shown unchanged across several generations of the system. However, since the interface is also a Part, it can also be revised in a new generation and maintain predecessor relationships.

VI. Form-Agnostic Builder

Also Known As: Creational Design Pattern as a design element.

A. Intent

A higher level abstraction is needed to represent the relationship between the design of a software component and how to construct, initialize, and build it. In order for the Generatrix design pattern to express the structure of the Part-Builder relationship independent of the particular ways in which the "building" is done, a notation for the functionality without form is required.

Often, a component part contains its own methods for building its instances. Consequently code may be duplicated, or nearly duplicate, when either the building or the work parts of the component change in the next generation. A regular practice of separating the builder design from the part can reduce duplication. With this separation, the form of the builder can be designed abstractly and the abstraction carried along with future generations of the builder.

It is not enough for informal documentation or institutional knowledge to be a record of how a component objects are intended to be constructed and iteratively populated with values and references. The design itself must have a corresponding realization that is part of the executing system.

B. Motivation

- The way a given component is constructed and initialized may be simple at one stage of the evolution of an application and become more complicated in a later generation of the code base.
- Developers hard-code the way to create instances, perform initializations and accomplish cumulative construction or populating of object. Their approach is suited to the visible requirements at hand. They usually don't document what aspects of the design are incidental or essential to their purpose. Maintenance programmers later on have to figure out (reverse engineer) this knowledge out of the code.

C. Applicability

Use the form Agnostic Builder 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 form Agnostic Builder should be used even if the designer "knows" up front which kind of creational pattern is intended at the time, for the present generation.

D. Structure

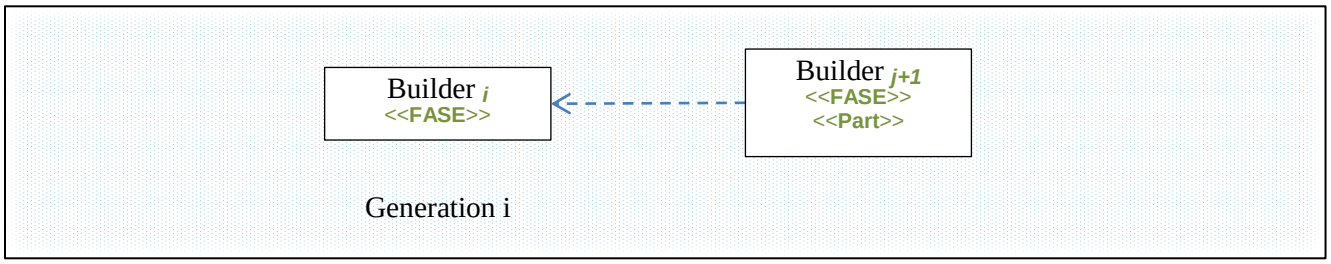


Figure 9. A Builder as a <<part>> and as a simple Form-Agnostic component.

The abstraction in Figure 9 represents the simplest view of a builder component and its (optional) predecessor. This should not be interpreted as meaning that any arbitrary creational pattern can be realized and be interchangeable with any other creational pattern. It also should not be interpreted as a single class or even a separate class.

E. Consequences

- The design that realizes a Form-Agnostic Builder might be more general than would otherwise be justified solely on the basis of the requirements at a particular generation of the system.
- The anticipatory design decisions are expected to return benefit as the system evolves.
- The explicit representation of the Builder might appear to developers as an extra maintenance burden.

F. Related Patterns

The Agnostic Builder represents a creational design. Any of the GoF patterns [20] as well as some of the other constructs listed below are suited to be the design instance represented by the Agnostic Builder.

GoF patterns

- Abstract Factory
- Builder
- Template Method
- Factory Method
- Prototype
- Singleton

Other patterns and idioms, examples from Fowler [24]

- Query Object (related to/ aka DAO)
- Active Record
- Plugin
- Data Mapper
- Serialized LOB
- Lazy Static initialization

An Agnostic Builder may also be a Part with a predecessor. That implies that any of the components can be the predecessor design instance of any of the others.

Example 1. A constructor serving as the design instance underlying an Agnostic Builder is the predecessor of a Template Method.

Example 2. An Abstract Factory, which builds a number of parts perhaps with a number of part-specific builders, has a simple Singleton as its predecessor. These examples would be illustrative of a situation where the predecessor had fewer concerns and requirements than the new generation.

VII. Generational Agnosticity

A. Intent

A higher level notational abstraction is needed to represent groups of components and software elements without specifying the form or details of how the compatibility is determined.

Knowledge of the abstraction needs to be preserved in the next generation of the system so that software engineers and other stakeholders do not have to reverse-engineer the intended flexibility from the particular embodiment of the implementing design.

Some components are designed to work only with a certain generation while others are able to work across generations or accommodate the generational character of a collaborating part. A means of designating generation-coupling (or not) is needed in order to keep this knowledge as part of the design. This designation is the quality of generational agnosticity.

B. Motivation

A pan-generation system requires a mechanism for expressing combinations of the following functions. Each one represents a form of agnosticity. The client and system have a particular set of constraints regarding the behavior and information the client will accept. Here the terminology of client and system can apply to any level of the system: an object that sends a request or calls a method is the “client” and the object hosting the message end-point or method implementation is the “system”. The following each represent a different “level” that a client and system can be “agnostic” about which generation is required or acceptable.

- The client 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.
- 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 it's request by redirection as necessary.
- The user is aware that the latest generation is required and the client application performs to that expectation. Here, the quality of agnosticity is a kind of zero case: the client is completely generation specific.
- 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".

C. Applicability

Use the Generational Agnosticity pattern when . . .

1. Awareness of the generation of a system is an identified concern.
2. 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.

This pattern is an abstraction and does not have a structure of its own. It provides a vocabulary for designing with a certain kind of intentional indeterminacy.

D. Example

This section gives a simplified example based on real experiences of how generational agnosticity may be. In this case, there are software artifacts that embody the predecessor relationship explicitly. The example is presented as a learning experience like the example in the Forces section as a way of suggesting how learning the lesson the hard way can be avoided. The alternative style of presentation might be to give a design rule, ready and thought-out by authority. But that approach would fail to preserve the knowledge of how the rule was derived!

A development team initially set all `serialVersionUID` values to 1L or applied `@suppress('serial')` . This code neglected the generational agnosticity of the system which was planned to be distributed and be upgraded with additional requirements in future. (This design element is used with Java `Serializable` and `Externalizable` classes [25].)

Upon realizing the mistake, the team went to the other extreme of automatically generating large random numbers every time a class was updated, even if just to edit javadoc comments. This thwarted the natural versioning capability of the java environment to be sensitive to an incompatible change but be "agnostic" about minor edits and recompilations of classes with compatible serial representation. The consequence was excessive re-downloading the whole application.

Then, a new design concept was set up. Each generation of the application would have a Singleton that provides a `serialVersionUID`, constant for the life of that generation. Each class that may require serialization then included a statement like:

```
public static final serialVersionUID = MyGeneration.generationUID;
```

Finally, the components that actually needed only "the latest" could access a pan-generation resource to ensure their matching `serialVersionUID` value, and only if not matched perform a fresh download.

This approach was resisted at first under the belief that different classes should have the same serial version UID. Prototyping demonstrated the safety of the approach by showing that different classes could not accidentally be serialized with the state of another class.

```
/**
 * Component-encapsulated awareness of its generation.
 */
interface GenerationIdentification {
    Class<GenerationIdentification> id();
}

/**
 * Marker interface for components driving the mediation of client and component generations.
 */
interface ConstraintSpec {

}

/**
 * Encapsulated mechanism for generation-aware matching of client and component.
 */
abstract public class GenerationalAgnosticity {
    abstract GenerationIdentification getGenerationIdentification();
    abstract ConstraintSpec getObjectConstraints();
}
```

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. Either way, they are designing-in a particular case of generational agnosticity.

E. Known Uses

- A legacy application[7] involving annual financial cycles was originally built on the concept that the year associated with stored data would drive dispersed ad-hoc business logic that varied by year. Over time, data arrived or updated in years other than the year to which they were intended, resulting in massive untraceable errors in billing and valuation processes. Some of the dispersed components responsible for determining the year to which a given record applied were changed, comparable to what generation-awareness would require.

F. Related Patterns

- Database anti-patterns [26]. 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 [27]. 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.

References

[1] Conference of State Bank Supervisors, "Executive Summary of the Sarbanes-Oxley Act of 2002 P.L.107," [Online]. Available: http://www.csbs.org/legislative/leg-updates/Documents/ExecSummary-SarbanesOxley-2002.pdf. [Accessed 20 05 2012].

[2] U. Ural, "Oracle automatically keeps the old, historical table statistics," 04 07 2011. [Online]. Available: file:///C:/edu/_Communications/_Application%20Prep/_My%20Writings/_PLoP2012%20-%20Generatrix%20design%20pattern%20-%20due%20May20/_resources/PLoP2011/Ural%20URAL%27s%20Oracle%20Blog%20Oracle%20automatically%20keeps%20the%20old,%20historical%20table%20stat. [Accessed 07 07 2012].

[7] This legacy system refers to an actual MIS encountered recently. Confidentiality and non-disclosure prevents naming the institution. The case was an environment for experimenting with the concepts that have evolved into the Generatrix pattern. The scenario is not so unique that a research citation is warranted.

[3] Defense Logistics Agency, "What is Sustaining Engineering," [Online]. Available: http://www.aviation.dla.mil/ExternalWeb/UserWeb/AviationEngineering/Engineering/Sustainment/whatissustainingengineering.asp. [Accessed 09 05 2012].

[4] PBS staff. Credits to Scott Freeman and Jon C. Herron, "Punctuated Equilibrium," WGBH Educational Foundation and Clear Blue Sky Productions, Inc. , 2001. [Online]. Available: http://www.pbs.org/wgbh/evolution/library/03/5/l_035_01.html. [Accessed 07 2012].

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

[6] J. R. Reza, "The Generatrix Design Pattern," in *IEEE SoutheastCon, March 18*, Orlando, Florida. USA, 2012.

[7] M. Mirakhorli and J. Cleland-Huang, "A Pattern System for Tracing Architectural Concerns," *PLoP*, 2011.

[8] M. Mirakhorli and J. Cleland-Huang, "A Pattern System for Tracing Architectural Concerns," *PLoP*, 2011.

[9] A. F. Yamashita, "A multi-method approach for evaluating software maintainability and comprehensibility," Simula Research Laboratory, Box 134, 1325 Lysaker, Norway.

[10] J. Rost and R. L. Glass, The Dark Side of Software Engineering: Evil on Computing Projects, Wiley-IEEE Computer Society Press, 2011, p. 305.

[11] R. Orchard, "Compatible Change," [Online]. Available: http://www.soapatterns.org/compatible_change.php .

[12] Wikipedia, with contributions from Sony Inc., "Walkman".

[13] M. Haire, "A Brief History of The Walkman," *Time Magazine*, 01 07 2009.

[14] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: Improving the Design of Existing Code, 2002.

[15] M. Pomeroy-Huff, R. Cannon, T. A. Chick, J. Mullaney and W. Nichols, "http://www.sei.cmu.edu," Software Engineering Process Management Program, 02 2010. [Online].

[16] Steve Smith, "Don't Repeat Yourself," 24 11 2009. [Online]. Available: http://programmer.97things.oreilly.com/wiki/index.php/Don%27t_Repeat_Yourself. [Accessed 09 07 2012].

[17] J. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, no. 5, p. 532.533, May 1988.

[18] P. J. Ross, Taguchi Techniques for Quality Engineering, 2nd Ed., McGraw-Hill Professional, 1995.

[19] S. D. Boer, Six Sigma for It Management, Van Haren Publishing, 2006.

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

[21] Sun Developer Network (SDN), "Intercepting Filter," 2002. [Online]. Available: http://java.sun.com/blueprints/corej2eepatterns/Patterns/InterceptingFilter.html. [Accessed 2012].

[22] J. R. Reza, "Accord: Family Polymorphism on Core Java," *To Appear*, 2012.

[23] J. R. Reza, "Java supervenience," *Computer Languages, Systems & Structures 38*, p. 73–97, 2012.

[24] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002.

[25] Sun Microsystems, "Interface Externalizable," [Online]. Available: http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Externalizable.html. [Accessed 10 07 2012].

[26] B. Karwin, SQL Antipatterns: Avoiding the Pitfalls of Database Programming, TODO, TODO.

[27] T. Erl, "Canonical Versioning," in *SOA design patterns*, TODO, 2001.

[28] A. Harwood, "Parallel Algorithms," Department of Computer Science and Software Engineering, 22 10

2003. [Online]. Available: http://ww2.cs.mu.oz.au/498/notes/node40.html. [Accessed 09 07 2004].

Appendix – Supplementary Material

```
package com.agnos.framework;
import static com.agnos.framework.DerivabilityConstraint.ALWAYS;
. . .
@Target(value = { METHOD, CONSTRUCTOR, ANNOTATION_TYPE, FIELD, PARAMETER })
@Retention(value = RUNTIME)
@Inherited
public @interface Predecessor {

          /** Predecessor syntax form: <i>Project</i>/dottedTypeName */
          String[] predecessor() default "";

          /** {@link com.agnos.framework.DerivabilityConstraint} */
          DerivabilityConstraint derivability() default ALWAYS;
}
package com.agnos.framework;
. . .
@Target(value = { PACKAGE, TYPE, ANNOTATION_TYPE })
@Retention(value = RUNTIME)
@Inherited
/**
 * Designates a class, interface, or enum as a Generational object type.
 * Allows predecessor to be specified and DerivabilityState constraints specified.
 * <br> <i>The list of predecessors is a list of Strings rather than classes since
 * (a) the predecessor relationship can be applied to methods and since
 * (b) the predecessor may be in a location not in the current element's classpath</i>
 */
public @interface Generational {

          /** Predecessor syntax form: <i>Project</i>/dottedTypeName */
          String[] predecessor() default "";

          /** {@link com.agnos.framework.DerivabilityConstraint} */
          DerivabilityConstraint derivability() default ALWAYS;
}

package com.agnos.framework;
. . .
/**
 * Constraint options governing a Generational type
 */
public enum DerivabilityConstraint {
/** can have predecessor and be referenced as a predecessor */
ALWAYS,
/** can be referenced as predecessor but cannot have predecessor of its own. syn: Top, Head */
ROOT,
/**is derived from a predecessor but cannot be a predecessor to another element derived from it */
FINAL,
/** cannot have a predecessor and cannot be the predecessor of another Generational type
 * Example: a class and one of its methods are marked ALWAYS and another method of the same  class is marked NEVER, indicating that it is
not intended to participate in the generationality of the class.
 */
NEVER,;
          private final static DerivabilityConstraint defaultValue = ROOT;

          private boolean isEmpty(Collection<GenerationalType> predecessors) {
                    return predecessors == null || predecessors.size() == 0;
          }

          public String value() {
                    return this.name();
          }

          /**
           * validates a Generational type to enforce the Derivability constraints.
           */
          public void validate(GenerationalType gt) {
            if (gt.isGenerational()) {
              assert gt.derivabilityConstraint() != ROOT || isEmpty(gt.getPredecessors());

              /** derived types may check that they do not have this as their pred if this is a NEVER */
              assert gt.derivabilityConstraint() != NEVER || isEmpty(gt.getPredecessors());

                              for (GenerationalType pred : gt.getPredecessors()) {
                                        assert pred.derivabilityConstraint() != NEVER;
                                        assert pred.derivabilityConstraint() != FINAL;
          }         }         }

          public static DerivabilityConstraint getDefault() {
                    return defaultValue;
}         }
package com.agnos.framework;
. . .
public interface GenerationalType {
          /** returns the derivability constraint or the default
           *  {@link com.agnos.framework.DerivabilityConstraint#defaultValue}
           */
          DerivabilityConstraint derivabilityConstraint();

          /**
           * @return true if this file is a Generational type
           */
          boolean isGenerational();
}
```