Organic Programming - Archetypes for Robust Software Engineering

Author: Nitin Uchil, Mindware, Inc.

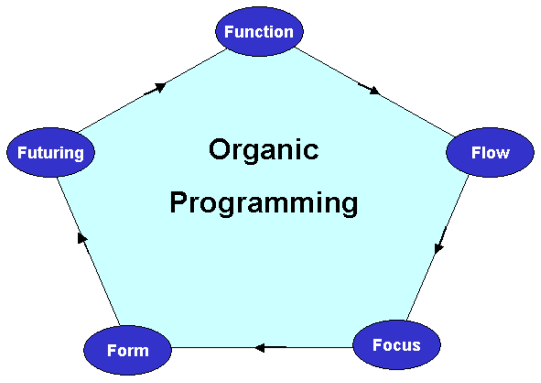
About

In 1968 Edsgar Dijkstra wrote a letter to CACM entitled GOTO Statement Considered Harmful. The central ideas of software engineering were being born.......

There is a fundamental problem facing enterprise developers today. Writing distributed business applications is difficult. Writing any large application is difficult. This is compounded if the application is distributed, or lives in multiple pieces on a network. This is further compounded if the application must execute its business logic in a guaranteed, reliable fashion. This paper is about software engineering - the tenets to be followed in writing robust, reliable and reusable code that stands the test of time and changes in architectures.

"So here I stand before you preaching organic architecture: declaring organic architecture to be the modern ideal and the teaching so much needed if we are to see the whole of life, and to now serve the whole of life, holding no `traditions' essential to the great TRADITION. Nor cherishing any preconceived form fixing upon us either past, present or future, but-instead-exalting the simple laws of common sense-or of super-sense if you prefer-determining form by way of the nature of materials..." - Frank Lloyd Wright, An Organic Architecture, 1939

Organic programming is a new software methodology designed for constructing complex yet flexible structures of program modules. It is about a new philosophy of software engineering - where form and function are one, where documentation and code are written in unison. Much like Organic Architecture, it is an attempt to integrate modules into a coherent whole: a marriage between the programming construct and its functionality and a union between the abstraction and its implementation.



It is organized about the concepts of **5Fs (Function, Flow, Focus, Form and Futuring)** to build robust, scalable and resilient software architectures. It reinforces the need that software development should be iterative rather than following the waterfall approach of current practices.

Background

All mature engineering disciplines draw from a collective compendium of time-honored, battle-tested "best practices" and "lessons learned" for solving known engineering problems. Great engineers don't just design their products strictly according to the principles of math and science. They must adapt their solutions to make optimal trade-offs and compromises between known solutions, principles, and constraints to meet the ever-increasing and ever-changing demands of cost, schedule, quality, and customer needs. Patterns help bring order out of chaos by identifying what is constant and recognizable in the midst of such incessant change. In this sense, patterns appear to resemble strange attractors, the convergence of dynamically interacting components into stable configurations, that recur all throughout successful systems.

In software development, architectures are like ideas - everyone has them, they're all valid, and beauty is in the eye of the beholder. Software architecture provides a continuous opportunity to explore different ways of achieving the same goals, as well as a chance to absorb new concepts.

Software projects are quantified by four variables:

1. **scope,**
2. **resources,**
3. **time and**
4. **quality**

Scope is how much is to be done. Resources are how many people are available. Time is when the project or release will be done. And quality is how good the software will be and how well tested it will be.

A software methodology is the set of rules and practices used to create computer programs. A heavyweight methodology has many rules, practices, and documents. It requires discipline and time to follow correctly. A lightweight methodology has only a few rules and practices or ones which are easy to follow.

The process of software development tends to be rather chaotic. Team members come and go, requirements change, schedules change, entire projects get canceled, entire companies go out of business, and so on. The programmer's job is to successfully navigate this chaos and in the end produce a "quality" product in a "timely" manner.

Besides being chaotic, the software development process also tends to be rather iterative. As a software product is developed, it continuously evolves based on feedback from many parties. This iterative process works from release to release (each release is one iteration) and within the development cycle of a single release. From release to release, for example, the feedback of customers with the current version indicates which bug-fixes and enhancements are most important to make in the next version. Within the development cycle of a single release, the vision of the end target is continuously adjusted by forces within the company as development progresses.

Software development process of a single release cycle is typically categorized by the following phases:

1. Specification
2. Design
3. Impelementation
4. Integration
5. Test
6. Deployment
7. Live Administration

Often architecture and infrastructure are used interchangeably. There is a subtle difference. Infrastructure consists of applications, hardware, and software. Architecture on the other hand, refers to the overlying principles and processes that lead the organization's infrastructure deployment. Infrastructure is a snapshot, while arcitecture is a continuously evolving set of ideas and philosophies. Architecture includes infrastructure, but the two aren't synonymous.

In the past 12–18 months, a wide range of publications—*Software Development, IEEE Software, Cutter IT Journal, Software Testing and Quality Engineering,* and even *The Economist*—has published articles on what Martin Fowler calls the New Methodology (see www.martinfowler.com/articles/newMethodology.html), reflecting a growing interest in these new approaches to software development (Extreme Programming, Crystal Methodologies, SCRUM, Adaptive Software Development, Feature-Driven Development and Dynamic Systems Development Methodology among them). In addition to these "named" methodologies, scores of organizations have developed their own "lighter" approach to building software.

Organic Programming is about defining architecture that is flexible, scalable and designed to embrace new standards and technologies as they emerge. It is designed to help IT teams implement and support robust, change-resilient systems that meet the requirements of a continuously changing business environment.

OO Primer

The solution to the problem should resemble the problem, and observers of the solution should be able to recognize the problem without necessarily knowing about it in advance.

OO is divided into the following parts:

- **Object Oriented Analysis (OOA):** A method of analysis in which requirements are examined from the perspective of the classes and objects found in the vocabulary of the problem domain.
- **Object Oriented Decomposition:** The process of breaking a system into parts, each of which represents some class or object from the problem domain. The application of object-oriented design methods leads to an object-oriented decomposition, in which we view the world as a collection of objects that cooperate with one another to achieve some desired functionality.
- **Object Oriented Design (OOD):** A method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design; specifically, this notation includes class diagrams, object diagrams, module diagrams, and process diagrams.
- **Object Oriented Programming (OOP):** A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships. In such programs, classes are generally viewed as static, whereas objects typically have a much more dynamic nature, which is encouraged by the existence of dynamic binding and polymorphism.

The world is object-oriented, and the object-oriented programming paradigm attempts to express computer programs in ways that model how people perceive the world.

While traditional structured languages were using functions and procedures that operate on (global) data, object oriented languages are using classes and objects that consist of methods, which operate only on their own internal data.

An object-oriented program has three explicit characteristics and one implicit characteristic. The three explicit characteristics are encapsulation, inheritance, and polymorphism. The implicit characteristic is abstraction.

- **Encapsulation** - hide the implementation & expose the interface
- **Inheritence** - extending existing, or subclass existing class - reuse don't reinvent.
- **Polymorphism** - one name, many forms - same method name different input arguments.
- **Abstraction** - is used to specify new abstract data types (ADT).

OOP involves a whole new vocabulary (or jargon) which is different from or supplemental to the vocabulary of procedural programming. For example the object-oriented programmer defines an abstract data type by encapsulating its implementation and its interface into a class. One or more instances of the class can then be instantiated. An instance of a class is known as an object. Every object has state and behavior where the state is determined by the current values stored in the instance variables and the behavior is determined by the instance methods of the class from which the object was instantiated. Inherited abstract data types are derived classes or subclasses of base classes or superclasses. We extend superclasses to create subclasses. Within the program the programmer instantiates objects (creates instances of classes) and sends messages to the objects by invoking the class's methods (or member functions).

If a program is "object oriented", it uses encapsulation, inheritance, and polymorphism. It defines abstract data types, encapsulates those abstract data types into classes, instantiates objects, and sends messages to the objects.

To make things even more confusing, almost every item or action used in the OOP jargon has evolved to be described by several different terms. For example, we can cause an object to change its state by sending it a message, invoking its methods, or calling its member functions.

Design Patterns

Patterns for software development are one of the latest "hot topics" to emerge from the object-oriented community. They are a literary form of software engineering problem-solving discipline that has its roots in a design movement of the same name in contemporary architecture, literate programming, and the documentation of best practices and lessons learned in all vocations.

Fundamental to any science or engineering discipline is a common vocabulary for expressing its concepts, and a language for relating them together. The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships lets us successfully capture the body of knowledge which defines our understanding of good architectures that meet the needs of their users. Forming a common pattern language for conveying the structures and mechanisms of our architectures allows us to intelligibly reason about them. The primary focus is not so much on technology as it is on creating a culture to document and support sound engineering architecture and design.

A pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces. Each pattern is a three-part rule,

1. the quality - which expresses a relation between a certain context,
2. the gate - a certain system of forces which occurs repeatedly in that context, and
3. the way - a certain software configuration which allows these forces to resolve themselves.

Patterns expose knowledge about software construction that has been gained by many experts over many years. All work on patterns should therefore focus on making this precious resource widely available. Every software developer should be able to use patterns effectively when building software systems. When this is achieved, we will be able to celebrate the human intelligence that patterns reflect, both in each individual pattern and in all patterns in their entirety.

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not in can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample ... code to illustrate an implementation. Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages ....

Patterns represent distilled experience which, through their assimilation, convey expert insight and knowledge to inexpert developers. They help forge the foundation of a shared architectural vision, and collective of styles. If we want software development to evolve into a mature engineering discipline, then these proven "best practices" and "lessons learned" must be aggressively and formally documented, compiled, scrutinized, and widely disseminated as patterns (and anti-patterns). Once a solution has been expressed in pattern form, it may then be applied and reapplied to other contexts, and facilitate widespread reuse across the entire spectrum of software engineering artifacts such as: analyses, architectures, designs, implementations, algorithms and data structures, tests, plans, and organization structures.

Component Architecture

Components are self-contained, reusable software units that can be visually composed into composite components, applets, applications, and servlets using visual application builder tools. Components expose their features (for example, public methods and events) to builder tools for visual manipulation. A Bean's features are exposed because feature names adhere to specific design patterns. A "JavaBeans-enabled" builder tool can then examine the Bean's patterns, discern its features, and expose those features for visual manipulation. A builder tool maintains Beans in a palette or toolbox. You can select a Bean from the toolbox, drop it into a form, modify it's appearance and behavior, define its interaction with other Beans, and compose it and other Beans into an applet, application, or new Bean. All this can be done without writing a line of code.

Unified Modeling Language

he Unified Modeling Language (UML) has gained broad industry acceptance as the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It simplifies the complex process of software design, making a "blueprint" for construction. The Unified Modeling Language™ (UML) was developed jointly by Grady Booch, Ivar Jacobson, and Jim Rumbaugh at Rational Software Corporation, with contributions from other leading methodologists, software vendors, and many users. Based on extensive use of the Booch, OMT, and Jacobson methods, the UML is the evolution of these and other approaches to business process, object, and component modeling. The UML provides the application modeling language for:

- Business process modeling with use cases.
- Class and object modeling.
- Component modeling.
- Distribution and deployment modeling.

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

Extreme Programming

Author: Kent Beck

Extreme Programming, or XP, is a lightweight discipline of software development based on principles of simplicity, communication, feedback, and courage. XP is designed for use with small teams who need to develop software quickly in an environment of rapidly-changing requirements. XP has a few rules and a modest number of practices, all of which are easy to follow. XP is a clean and concise environment developed by observing what makes software development go faster and what makes it move slower. It is an environment in which programmers feel free to be creative and productive but remain organized and focused

XP is successful because it stresses customer satisfaction. The methodology is designed to deliver the software your customer needs when it is needed. XP empowers your developers to confidently respond to changing customer requirements, even late in the life cycle.

XP improves a software project in four essential ways:

1. **communication,**
2. **simplicity,**
3. **feedback, and**
4. **courage**

Whereas UML is highly structured, XP is different. It is a lot like a jig saw puzzle. There are many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen. This is a significant departure from traditional software development methods and ushers in a change in the way we program.

The twelve XP practices are:

- ***The Planning Process,*** sometimes called the Planning Game.
  he XP planning process allows the XP "customer" to define the business value of desired features, and uses cost estimates provided by the programmers, to choose what needs to be done and what needs to be deferred. The effect of XP's planning process is that it is easy to steer the project to success.

- ***Small Releases.***
  XP teams put a simple system into production early, and update it frequently on a very short cycle.

- ***Metaphor.***
  XP teams use a common "system of names" and a common system description that guides development and communication.

- ***Simple Design.***
  A program built with XP should be the simplest program that meets the current requirements. There is not much building "for the future". Instead, the focus is on providing business value. Of course it is necessary to ensure that you have a good design, and in XP this is brought about through "refactoring", discussed below./dd>

- ***Testing.***
  XP teams focus on validation of the software at all times. Programmers develop software by writing tests first, then software that fulfills the requirements reflected in the tests. Customers provide acceptance tests that enable them to be certain that the features they need are provided./dd>

- ***Refactoring.***
  XP teams improve the design of the system throughout the entire development. This is done by keeping the software clean: without duplication, with high communication, simple, yet complete.

- ***Pair Programming.***
  XP programmers write all production code in pairs, two programmers working together at one machine. Pair programming has been shown by many experiments to produce better software at similar or lower cost than programmers working alone.

- ***Collective Ownership.***
  All the code belongs to all the programmers. This lets the team go at full speed, because when something needs changing, it can be changed without delay.

- ***Continuous Integration.***
  XP teams integrate and build the software system multiple times per day. This keeps all the programmers on the same page, and enables very rapid progress. Perhaps surprisingly, integrating more frequently tends to eliminate integration problems that plague teams who integrate less often.

- ***40-hour Week.***
  Tired programmers make more mistakes. XP teams do not work excessive overtime, keeping themselves fresh, healthy, and effective.

- ***On-site Customer.***
  An XP project is steered by a dedicated individual who is empowered to determine requirements, set priorities, and answer questions as the programmers have them. The effect of being there is that communication improves, with less hard-copy documentation - often one of the most expensive parts of a software project.

- ***Coding Standard.***
  For a team to work effectively in pairs, and to share ownership of all the code, all the programmers need to write the code in the same way, with rules that make sure the code communicates clearly.

Extreme Programming (XP) was created in response to problem domains whose requirements change. Your customers may not have a firm idea of what the system should do. You may have a system whose functionality is expected to change every few months. In many software environments dynamically changing requirements is the only constant. This is when XP will succeed while other methodologies do not.

Much of what went into XP was a re-evaluation of the way software was created. The quality of the source code is much more important than one might realize. Just because our customers can't see our source code doesn't mean we shouldn't put the effort into creating something we can be proud of.

Feature Driven Development

Feature Driven Development (FDD), pioneered by Jeff de Luca (http://www.nebulon.com/) and Peter Coad, is another process with rapidly growing interest. Superficial similarities between FDD and XP hide a number of very important differences between the two processes. Feature Driven Development is introduced in Chapter 6 of *Java Modeling in Color with UML* book [Coad]. The chapter is also available in electronic form at http://www.togethersoft.com/jmcu/ and in Together's online help (*Users Guide - Part 1: Modeling with Together - 2. Introductions to Modeling*)

Agile Programming

Author: Martin Fowler

Facilitating change is more effective than attempting to prevent it. Learn to trust in your ability to respond to unpredictable events; it's more important than trusting in your ability to plan for disaster.

Most software development is a chaotic activity, often characterized by the phrase "code and fix". The software is written without much of an underlying plan, and the design of the system is cobbled together from many short term decisions. This actually works pretty well as the system is small, but as the system grows it becomes increasingly difficult to add new features to the system. Furthermore bugs become increasingly prevalent and increasingly difficult to fix. A typical sign of such a system is a long test phase after the system is "feature complete". Such a long test phase plays havoc with schedules as testing and debugging is impossible to schedule.

We've lived with this style of development for a long time, but we've also had an alternative for a long time: methodology. Methodologies impose a disciplined process upon software development with the aim of making software development more predictable and more efficient. They do this by developing a detailed process with a strong emphasis on planning inspired by other engineering disciplines.

These methodologies have been around for a long time. They've not been noticeable for being terribly successful. They are even less noted for being popular. The most frequent criticism of these methodologies is that they are bureaucratic. There's so much stuff to do to follow the methodology that the whole pace of development slows down. Hence they are often referred to as heavy methodologies, or to use Jim Highsmith's term: monumental methodologies.

As a reaction to these methodologies, a new group of methodologies have appeared in the last few years. For a while these were known a lightweight methodologies, but now the accepted term is agile methodologies. For many people the appeal of these agile methodologies is their reaction to the bureaucracy of the monumental methodologies. These new methods attempt a useful compromise between no process and too much process, providing just enough process to gain a reasonable payoff.

The result of all of this is that agile methods have some significant changes in emphasis from heavyweight methods. The most immediate difference is that they are less document-oriented, usually emphasizing a smaller amount of documentation for a given task. In many ways they are rather code-oriented: following a route that says that the key part of documentation is source code.

However I don't think this is the key point about agile methods. Lack of documentation is a symptom of two much deeper differences:

- *Agile methods are adaptive rather than predictive.* Heavy methods tend to try to plan out a large part of the software process in great detail for a long span of time, this works well until things change. So their nature is to resist change. The agile methods, however, welcome change. They try to be processes that adapt and thrive on change, even to the point of changing themselves.
- *Agile methods are people-oriented rather than process-oriented.* They explicitly make a point of trying to work with peoples' nature rather than against them and to emphasize that software development should be an enjoyable activity.

In the past few years there's been a rapidly growing interest in agile (aka "lightweight") methodologies. Alternatively characterized as an antidote to bureaucracy or a license to hack they've stirred up interest all over the software landscape.

The Manifesto for Agile Software Development

Seventeen anarchists agree:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions* over processes and tools.
- *Working software* over comprehensive documentation.
- *Customer collaboration* over contract negotiation.
- *Responding to change* over following a plan.

That is, while we value the items on the right, we value the items on the left more.

We follow the following principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

—Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas
http://www.agilealliance.org/

The SCRUM Development Process

Scrum is an agile, lightweight process that can be used to manage and control software and product development. Wrapping existing engineering practices, including Extreme Programming, Scrum generates the benefits of agile development with the advantages of a simple implementation. Scrum significantly increases productivity while facilitating adaptive, empirical systems development.

What is Scrum ?

- Scrum is an agile, lightweight process to manage and control development work.
- Scrum is a wrapper for existing engineering practices.
- Scrum is a team-based approach to iteratively, incrementally develop systems and products when requirements are rapidly changing
- Scrum is a process that controls the chaos of conflicting interests and needs.
- Scrum is a way to improve communications and maximize co-operation.
- Scrum is a way to detect and cause the removal of anything that gets in the way of developing and delivering products.
- Scrum is a way to maximize productivity.
- Scrum is scalable from single projects to entire organizations. Scrum has controlled and organized development and implementation for multiple interrelated products and projects with over a thousand developers and implementers.
- Scrum is a way for everyone to feel good about their job, their contributions, and that they have done the very best they possibly could.
- Scrum is a pattern.

Scrum naturally focuses an entire organization on building successful products. Without major changes - often within thirty days - teams are building useful, demonstrable product functionality. Scrum can be implemented at the beginning of a project or in the middle of a project or product development effort that is in trouble.

Scrum is a set of interrelated practices and rules that optimize the development environment, reduce organizational overhead, and closely synchronize market requirements with iterative prototyes. Based in modern process control theory, Scrum causes the best possible software to be constructed given the available resources, acceptable quality, and required release dates. Useful product functionality is delivered every thirty days as requirements, architecture, and design emerge, even when using unstable technologies.

Over fifty organizations have successfully used Scrum in thousands of projects to manage and control work, always with significant productivity improvements. Scrum wraps an organization's existing engineering practices; they are improved as necessary while product increments are delivered to the user or marketplace. As heard about Scrum, "oh, that's just my idea X by another name". Except, Scrum is spelled out as values, practices, and rules in a development framework that can be quickly implemented and repeated.

Sustainable Engineering

By using the practices adopted by mature engineering organizations, product development groups using Scrum have built solid, sustainable, maintainable products.

Use of these sustainable engineering practices is critical to the product's success. We have seen customers refuse to take new releases of products because they have experienced poor quality initial releases. This poor quality is almost always trackable to non-existent or poor engineering practices.

These practices are often referred to as "engineering standards". Some are techniques used by the actual product development teams; other practices are supplied and supported by infrastructure engineering, or sustainable engineering, groups - whose job is to create and maintain an engineering environment within which the new product teams can operate.

Engineering Practice	**Why**
Daily product build	Ensures that entire product can be built, that all code compiles, and that all components are present. Not doing this daily usually results in a fragmented, corrupted product that must be fixed before the end of the Sprint. Also results in non-compilable code being used by multiple team members.
Daily product quick test	Tests main product functionality after the daily product build to flush out any major defects since the last build; stops engineering teams from continuing to build on a product with major flaws. Usually performed using automated testing tools.
Defect reporting and work	Provides central place for identifying, describing, prioritizing, and tracking the correction of defects. Used by customers as well as internal groups.
Code Review or Paired Programming of defect co	Lowers probability that new defects are introduced while fixing a defect.
Source code control	To ensure that coding isn't lost or inadvertently muddled, that only authorized parties are able to create or sustain code, and that product engineering groups can be distributed by time and geography.
Version control	Freezes a copy of code that represents a version of the product (e.g. version 8.0.15) that is released to customers (alpha, beta, full release). This code set version number is used for defect tracking and fixing.
Release management	Ensures that every release can be completely identified, including source, defects fixed, new functionality, components, database structure, test materials, and third party components.
Regression testing	To fully test that all functionality works and all results are accurate before releasing a product - so that a customer doesn't find product defects.
Backup/Restore	To ensure checkpoints of production and development databases,data, software, and code that can be restored when uncorrectable corruption occurs.

Performance monitoring	To ensure that the user experience is within the target range. Trend tracking facilitates timely capacity upgrades prior to outages.
Common components	To ensure common look and feel of reused functionality within a product, such as security, time/date, etc.
Product simulation	"What-if" capability to drive capacity changes to improve product performance.

Implementing these engineering practices involves:

1. Review of existing practices, review with engineers, and modification of practice to best suit development environment
2. Organizational changes or additions to support practices
3. Introduction of automated tools to support practice
4. Development of supporting material (such as tests, inquiries, announcements, etc.)
5. Training
6. Initial, selected use of practice to assess impact
7. Full implementation
8. Review and modification

Crytal-Adaptive Methodologies

Authors: Alistair Cockburn & Jim Highsmith
Crystal collects together self-adapting family of "shrink-to-fit," human-powered software development methodologies based on these understandings:

1. Every project needs a slightly different set of policies and conventions, or methodology.
2. The workings of the project are sensitive to people issues, and improve as the people issues improve, individuals get better, and their teamwork gets better.
3. Better communications and frequent deliveries communication reduce the need for intermediate work products.

Whitepaper on Adaptive Programming

Literate Programming

Literate programming is a phrase coined by Donald Knuth to describe the approach of developing computer programs from the perspective of a report or prose. The focus, then, is on description (and documentation) of the approach in human-readable form. This is in contrast to the normal approach of focusing on the code. For more information Click here.

Literate programming is the combination of documentation and source together in a fashion suited for reading by human beings. In fact, literate programs should be enjoyable reading, even inviting! (Sorry Bob, I couldn't resist!) In general, literate programs combine source and documentation in a single file. Literate programming tools then parse the file to produce either readable documentation or compilable source. The WEB style of literate programming was created by D.E. Knuth during the development of his TeX typsetting software.

All the original work revolves around a particular literate programming tool called WEB. Knuth says:

The philosophy behind WEB is that an experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like TeX for formatting, and a language like C for programming. Neither type of language can provide the best documentation by itself; but when both are appropriately combined, we obtain a system that is much more useful than either language separately.
The structure of a software program may be thought of as a web that is made up of many interconnected pieces. To document such a program we want to explain each individual part of the web and how it relates to its neighbours. The typographic tools provided by TeX give us an opportunity to explain the local structure of each part by making that structure visible, and the programming tools provided by languages such as C or Fortran make it possible for us to specify the algorithms formally and unambigously. By combining the two, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into a working software system that matches the documentation.

Organic Programming

"Beautiful buildings are more than scientific. They are true organisms, spiritually conceived; works of art using the best technology."

- Frank Lloyd Wright on architecture

Frank Lloyd Wright espoused "organic architecture", a type of architecture which achieved an elemental harmony between the man-made environment within the home and the natural environment outside. Wright defined his version of organic architecture when he said; "Organic architecture sees actuality as the intrinsic romance of human creation or sees romance as actual in creation... In the realm of organic architecture human imagination must render the harsh language of structure into becomingly humane expressions of form instead of devising inanimate facades or rattling the bones of construction. Poetry of form is as necessary to great architecture as foliage is to the tree, blossoms to the plant, or flesh to the body." What Wright was basically getting at here was that the form of a structure should enter into a poetic dialogue between its environment and it inhabitants. Another quote from Wright gets to the heart of this idea. "My prescription for a modern house: first pick a good site. Pick that one at the most difficult site- pick a site no one wants- but one that has features making for character: Trees, individuality, a fault of some kind in the realtor mind." The sites he's talking about are sites which are, in a word, rugged. Places where the terrain and surrounding natural environment impose their tacit will upon the form of a structure and thus require the architect to accommodate nature as much as the desires of the inhabitants. A location that forces you to enter into that dialogue with nature if you are to use the site at all as opposed to the conventional bull-dozed flat suburban housing lot where nature is obliterated and the environment simply manufactured and compartmentalized.

"*Code should be a piece of art, as beautiful as nature itself.....*"

Author: Nitin Uchil

Organic Programming (OP) is actually a deliberate and disciplined approach to software development. It combines the coding structures of UML and the best practices of XP to bring about object-oriented code that is component based and bug free. It follows the philosophy of Gestalt - "*The whole is greater than the sum of its parts*". It stresses on a "Zero Administration Client" environment once the application is production deployed. It is a culmination and re-organization of best practices followed by Mindware in its projects of building knowledge management frameworks at Ford Motor Company.

Allocating Resources: Typically projects spend twenty times as much on people than hardware. So creating optimized code to reduce cost in hardware creates software that is difficult to maintain in the long run. Like XP, OP emphasizes on reducing software costs by creating clean, simple code. Like UML, OP emphasizes on highly constructed documentation so that projects can be "re-lived" at moments notice.

Managing Work Load: In an Organic Environment, programmers work on 2-3 projects simultaneously in pairs (note that these pairs are preferably different in each of the projects). Migrating from one project to another helps developers come back with "fresh" eyes rather than develop stale or bad code if they persist on one project.

Appropriate Programming: Just like organic architecure, OP is programming that is appropriate to time, appropriate to place and appropriate to person (developer/client/user). Most applications have both expert and novice modes for navigation and are hyperlinked to an extensive glossary.

Code Reuse: Metaphors (Archetypes) captured in previous projects as Design Patterns will be reused.

Intuitive Design: The goal of intuitive design is to create for the user the experience of *flow*. According to Csikszenthmihalyi "flow tends to occur when a person faces a clear set of goals that require appropriate responses. Another characteristic of flow is that they provide immediate feedback. They make clear how well you are doing...flow tends to occur when a person's skills are fully involved in overcoming a challenge that is just about manageable".

Focus: Focus follows the earlier concept of flow (intuitive design). It is a way to quickly do what is set for a particular task, a concept or a module.

Stereo Coding: Coding and documenting the code should go hand in hand in any development environment. Documentation is different from line commenting, block commenting or api commenting and entails writing a whitepaper consisting of the technical details of the application with appropriate snap shots of the utility pages.

Coding to Interfaces: By coding to interfaces, developers can quickly publish the methods by which their objects are to be used. This technique results in decreased design time and offers developers great flexibility in their implementations.

Incoporating Redundancies: In typical data connectivities, we use SQL*NET listeners. In our years of consulting we have noticed that sometimes some listeners go down. Redundances are included to cycle thru the ports that work so that the application is not affected.

Testing Code: As in case of XP, tests are created before the code is written, while the code is written, and after the code is written and are classified as:

1. Unit Tests
2. Class Tests
3. Connectivity Tests
4. Functional Tests
5. Stress or Bashing Tests

6. Cron or Automated Tests

Error Containment: If after all redundancies are cycled thru and there is still a problem, the application should indicate to the user that there is a problem, catalog them and automatically notify them when the problem is fixed.

Change Management: Typical consultants following the UML methodology hate change. Only recently with the introduction of automated code generation schemes from UML is there a probability of managing changes in specifications dictated by the clients. What we have realized in the course of our consulting is that changes in specifications are the norm - a feature that consultants have to adopt to maintain lasting relationships with their clients. Borrowing a page from XP:

Another thing your customers will notice is the attitude XP programmers have towards changing requirements. XP enables us to embrace change. Too often a customer will see a real opportunity for making a system useful after it has been delivered. XP short cuts this by getting customer feed back early while there is still time to change functionality or improve user acceptance. Your customers are definitely going to notice this.

In XP, there is a process where we remove redundancy, eliminate unused functionality, and rejuvenating obsolete designs we are refactoring. Refactoring throughout the entire project life cycle saves time and increases quality. In OP we go one step ahead of this - we cycle thru our applications every so often, incorporating into it all the latest features that we have developed for "newer" projects free of charge to our customers. Of course as the code is being modified, we manage it in a staging area before deployment as the next version of the code. This helps us maintain a lasting relationship with our clients.

References

Patterns Home Page

Extreme Programming Org

Extreme Programming Site

Testing Resources for Extreme Programmers

The King's Dinner - A Once upon a Time tale...

Java Modeling in Color with UML by Peter Coad, Eric LeFebvre, Jeff De Luca

The Timeless Way of Building & A Pattern Language by Christopher Alexander

A Series of Whitepapers on Design Techniques

Nitin Uchil is founder and President & CEO of Mindware, Inc. For the past three years he has been consulting at Ford Motor Company building a knowledge management framework to depict design verification, competitive intelligence, web-based training and content management. He has architectured web-enabled enterprise knowledge bases and expert system tools to house and intelligently mine competitive vehicle information, schedule/archive/correlate Test and CAE analysis, build prediction tools using statistical analysis, rule based systems and artificial intelligence to automatically store and sign-off deliverables using historical data via the Verification Portal. He is also responsible for administering the Computer Based Enhancement (CBE) curriculum at the Ford Design Institute that provides web-based training and testing courses that deliver interactive content using Java Applets & Servlets, Livewire, multi-media based CD-ROM using a back-end Oracle database. He is a graduate of the University of Oklahoma (1987) and has over ten years of experience working with engineering simulations related to aerospace and automotive applications in the fields of structural analysis and computational fluid dynamics. He has held engineering positions at Universal Analytics (1988-1989), the developers of UAI/NASTRAN, eBASE and ASTROS (structural analysis and optimization packages employing the finite element methodology) now part of MSC Software, Analysis and Design Application Co. (adapco) (1989-1996), consultants in aerospace and automotive engineering and the developers of the pre- and post- processor for STAR-CD (a computational fluid dynamics code), and TechneGroup Design (1996-1997), consultants in CAE Simulations now part of ICEM-CFD, a subsidiary of ANSYS Inc. He is currently in the process of defining BDRIVE.COM, a company created to integrate middleware to accelerate eBusiness.

Mindware is a Michigan based IT company dedicated to providing enterprise solutions in engineering and technology using intelligent mining schemes, client-server techniques and the object oriented methodology to deliver content with dynamic and interactive capabilities. More information can be found at http://www.mindware-inc.com/.