Software Construction

with

Object Oriented Approach.

role of Interface Oriented Development (IOD),

Service Oriented Architecture(SOA) & Extreme programming(XP).

Abstract:

Software Construction is the main aspect of today's computer world. There are number of known procedures followed to build a flawless Software. For various reasons Object Oriented Approach has been widely accepted and is indispensable from Software Construction. Apart from just building a flawless software product the community demands for reuse of the built code, easy maintainability and easy modifiability. With these requirements to be satisfied, various approaches have come up. Some of them are Interface – Oriented Approach (IOD), Service Oriented Architecture (SOA) & Extreme Programming (XP); This paper discusses some of these main approaches which have been widely accepted and successful almost everywhere.

The 21st century Software Construction is lifted on the arms of Object Orientation. The Object Oriented Approach has been an long waited approach for software construction. It has given answers for many questions like software reusability, interoperability, easily modified, extended and maintained software.

Software reusability is not a topic which is well-understood by the masses. For example, many software reusability discussions incorrectly limit the definition of software to source code and object code. Even within the object-oriented programming community, people seem to focus on the inheritance mechanisms of various programming languages as a mechanism for reuse. (In many OOPLs, one of the steps in creating a subclass is the setting up a "backwards pointer" to one, or more, superclasses. This allows the subclass to "reuse" the code from the superclass(es).) Although reuse via inheritance is not to be dismissed, there are more powerful reuse mechanisms.

Consider a computer network with different computer hardware and software at each node. Next, instead of viewing each node as a monolithic entity, consider each node to be a collection of (hardware and software) resources. Interoperability is the degree to which an application running on one node in the network can make use of a (hardware or software) resource at a different node on the same network. It seems that there should be a more direct way to connect object-oriented approaches and interoperability. Polymorphism is a measure of the degree of difference in how each item in a specified collection of items must be treated at a given level of abstraction. Polymorphism is increased when any unnecessary differences, at any level of abstraction, within a collection of items are eliminated.

When conventional engineers design systems they follow some basic guidelines:

- They may start with the intention of designing an object (e.g., an embedded computer system), or with the intention of accomplishing some function. Even if they begin with the idea of accomplishing a function, they quickly begin to quantify their intentions by specifying objects (potentially at a high level of abstraction) which will enable them to provide the desired functionality. In short order, they find themselves doing object-oriented decomposition, i.e., breaking the potential product into objects.
- They assign functionality to each of the parts (object-oriented components). Looking ahead to reusing the parts, the engineers may modify and extend the functionality of one, or more, of the parts.
- Realizing that each of the objects in their final product must interface with one, or more, other objects, they take care to create well-defined interfaces. Again, focusing on reusability, the interfaces may be modified or extended to deal with a wider range of applications.
- Once the functionality and well-defined interfaces are set in place, each of the parts may be either purchased off-the-shelf, or designed independently. In the case of complex, independently-designed parts, the engineers may repeat the above process.

Without explicitly mentioning it, we have described the information hiding which is a normal part of conventional engineering. By describing the functionality of each part as an abstraction, and by providing well-defined interfaces, we foster information hiding. However, there is also often a more powerful concept at work here. Each component not only encapsulates functionality, but also knowledge of state even if that state is constant.

This state, or the effects of this state, is accessible via the interface of the component. By carefully examining the functionality of each part, and by ensuring well-thought-out and well-defined interfaces, the engineers greatly enhance the reusability of each part. However, they also make it easier to modify and extend their original designs. New components can be swapped in for old components -- provided they adhere to the previously-defined interfaces and that the functionality of the new component is harmonious with the rest of the system.

Conventional engineers also employ the concept of specialization. Specialization is the process of taking a concept and modifying (enhancing) it so that it applies to a more specific set of circumstances, i.e., it is less general. By maintaining a high degree of consistency in both the interfaces and functionality of the components, engineers can allow for specialization while still maintaining a high degree of modifiability. By identifying both the original concepts, and allowable (and worthwhile) forms of specialization, engineers can construct useful "families of components." Further, systems can be designed to readily accommodate different family members.

Object-oriented software engineering stresses such points as:
- Encapsulation (packaging) of functionality with knowledge of state information,
- Well-defined functionality and interfaces for all objects within a system,
- Information hiding -- particularly hiding the details of the underlying implementations of both the functionality and the state information, and
- Specialization using, for example, the concept of inheritance.

The concepts of encapsulation, well-defined functionality and interfaces, information hiding, and specialization are key to the modification and extension of most non-software systems. It should come as no surprise that, if used well, they can allow for software systems which are easily modified and extended.

In a comparison between object-orientation and component-orientation, Objects and components are comparable, in the following sense; they are supposed to encapsulate their own state completely, and to provide an interface to the outside world. In the context of object-orientation however, this is true during design, but very often not anymore during the implementation. It is rather common to haggle with the principles of

OO-design in the implementation, for example for reasons of performance. In a component world, interface oriented design and also implementation is crucial, and to some extent enforced by the middleware. Since it is impossible to predict the possible uses of a component, so in a component, interface design is more important than it usually is in object-oriented design, and encapsulation of the state is enforced.
In both component and service development, the design of the interfaces is done such that a software entity implements and exposes a key part of its definition. Therefore, the notion and concept of "interface" is key to successful design in both component-based and service-oriented systems.

The following are some key interface-related definitions:

- **Interface** — defines a set of public method signatures, logically grouped but providing no implementation. An interface defines a contract between the requestor and provider of a service. Any implementation of an interface must provide all methods.

- **Published interface** — An interface that is uniquely identifiable and made available through a registry for clients to dynamically discover.

- **Public interface** — An interface that is available for clients to use but is not published, thus requiring static knowledge on the part of the client.

- **Dual interface** — Frequently interfaces are developed as pairs such that one interface depends on another; for example, a client must implement an interface to call a requestor because the client interface provides some callback mechanism. This concept was introduced by Web services.

A service is behavior that is provided by a component for use by any other component based only on the interface contract. A service stresses interoperability and a service may be dynamically discovered and used. A Service Oriented Architecture is a design and a way of thinking about building software components. Contract design between components is a critical activity in a service-oriented architecture. The difference between a public interface and a published interface comes into play. A public interface is an interface that can be used by components within a system. The public interfaces of a component are easier to change, because they are only used by known clients. A published interface is one that is exposed to the network and may not be changed so

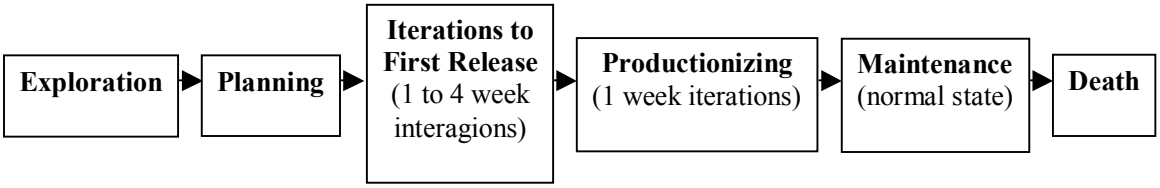
easily, because the clients of the published interface are not known. The difference is analogous to an intranet-based site only accessible by employees of the company and an Internet site accessible by anyone. A service-oriented architecture, first and foremost, stresses interoperability. That means that each component must provide an interface that can be invoked through a payload format and protocol that is understood by all of the potential clients of the service. A service must be dynamically discovered. This means that a third party mechanism must be used to find the service. Hard coding of a machine location is not consistent with a service-oriented approach. There are a lot of benefits to using a service-oriented architecture approach, but there are also some risks involved.

Patterns are a recent software engineering problem-solving discipline that emerged from the object-oriented community. Patterns have roots in many disciplines. It is a written document that describes a general solution to a design problem that recurs repeatedly in projects. Patterns use a formal approach to describing a problem, its solution, and any factors that might effect the solution. The goal of the pattern community is to build a body of literature to support design and development in general. There is less focus on technology than on a culture to document and support sound design. Software patterns first became popular with the object-oriented Design Patterns book. But patterns have been used for domains as diverse as development organization and process, exposition and teaching, and software architecture. At this writing, the software community is using patterns largely for software architecture and design.

Service oriented architectures take the level of coupling to another degree of detail. Instead of combining the properties and methods of a given domain model entity, we separate the data from the processing, and turn those processes (public interfaces, compiled of various methods and properties) into services.

Extreme programming plays a very different but one of the best available role in the field of software construction. XP takes commonsense principles and practices to extreme levels.

It reviews code all the time (pair programming). Everybody will test all the time (unit testing), even the customers (functional testing). Coming to the design phase, design is part of everybody's business (refactoring). XP says always leave the system with the simplest design that supports its current functionality. With XP everybody works defining and refining the architecture all the time (metaphor). Integrate and test several times a day (continuous integration). Iterations really, really short – seconds and minutes and hours, not weeks and months and years (the Planning Game).



Basic Life Cycle of eXtreme Programming.

The Rules and Practices of Extreme Programming:

Planning:

- User stories are written.
- Release planning creates the schedule.
- Make frequent small releases.
- The Project Velocity is measured.
- The project is divided into iterations.
- Iteration planning starts each iteration.
- Move people around.
- A stand-up meeting starts each day.
- Fix XP when it breaks.

Designing:

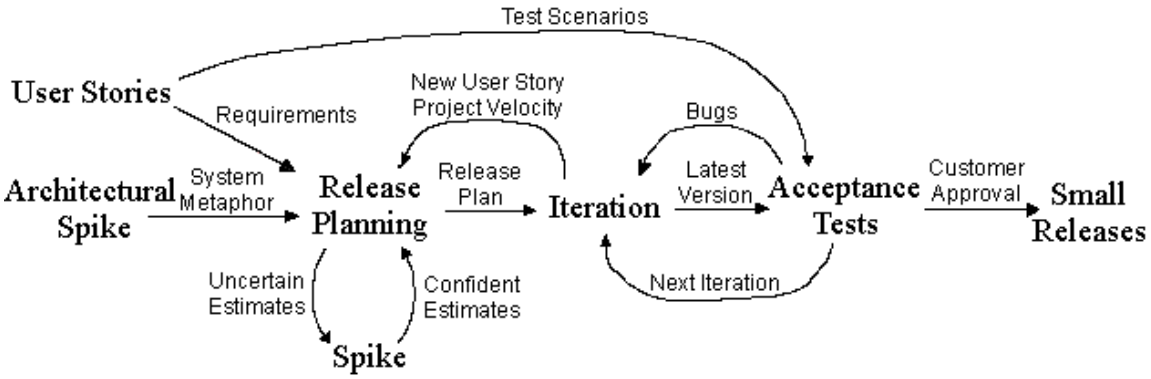
- Simplicity.
- Choose a system metaphor.
- Use CRC cards for design sessions.
- Create spike solutions to reduce risk.
- No functionality is added early.
- Refactor whenever and wherever possible.

Coding:

- The customer is always available.
- Code must be written to agreed standards.
- Code the unit test first.
- All production code is pair programmed.
- Only one pair integrates code at a time.
- Integrate often.
- Use collective code ownership.
- Leave optimization till last.
- No overtime.

Testing:

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.



Detailed Life Cycle of eXtreme Programming

A short scenario of XP:

The most obvious way to start extreme programming (XP) is with a new project. Start out collecting *user stories* and conducting *spike solutions* for things that seem risky. Spend only a few weeks doing this. Then schedule a release planning meeting. Invite customers, developers, and managers to create a schedule that everyone agrees on. Begin your iterative development with an *iteration planning* meeting. Usually projects

come looking for a new methodology like XP only after the project is in trouble. In this case the best way to start XP is to take a good long look at your current software methodology and figure out what is slowing you down. Add XP to this problem first. For example, if you find that 25% of the way through your development process your requirements specification becomes completely useless, then get together with your customers and write *user stories instead.* If you are having a chronic problem with changing requirements causing you to frequently recreate your schedule, then try a simpler and easier release planning meeting every few iterations. Try an iterative style of development and the just in time style of planning of programming tasks. If your biggest problem is the number of bugs in production, then try automated acceptance tests. Use this test suite for regression and validation testing. If your biggest problem is integration bugs then try automated unit tests. Require all unit tests to pass (100%) before any new code is released into the code repository.

If one or two developers have become bottlenecks because they own the core classes in the system and must make all the changes, then try *collective code ownership*. Let everyone make changes to the core classes whenever they need to. You could continue this way until no problems are left. Then just add the remaining practices as you can. The first practice you add will seem easy. You are solving a large problem with a little extra effort. The second might seem easy too. But at some point between having a few XP rules and all of the XP rules it will take some persistence to make it work. Your problems will have been solved and your project is under control. It might seem good to abandon the new methodology and go back to what is familiar and comfortable, but continuing does pay off in the end. Your development team will become much more efficient than you thought possible. At some point you will find that the XP rules no longer seem like rules at all.

There is a synergy between the rules that is hard to understand until you have been fully immersed. This up hill climb is especially true with pair programming, but the pay off of this technique is very large. Also, unit tests will take time to collect, but unit tests are the foundation for many of the other XP practices so the pay off is very great. XP projects are not quiet; there always seems to be someone talking about problems and solutions. People move about, asking each other questions and trading partners for programming. People spontaneously meet to solve tough problems, then disperse again. Encourage

this interaction, provide a meeting area and set up workspaces such that two people can easily work together. The entire work area should be open space to encourage team communication. Detailed explanation on XP is too huge for this paper. For further information please do go through the references.

Conclusion:

As we have seen, the Object Oriented Approach has been an long waited approach for software construction. It has given answers for many questions like software reusability, interoperability, easily modified, extended and maintained software. If the components were not considered, software construction might have been a night mare. Components have simplified and solved many problems of software construction. It is impossible to predict the possible uses of a component, so in a component, interface design is more important than it usually is in object-oriented design. Interface Oriented Design is magnified at this point. Along with Interface Oriented Design, Service Oriented Architecture goes hand in hand. A Service Oriented Architecture is a design and a way of thinking about building software components. A service-oriented architecture, first and foremost, stresses interoperability. That means that each component must provide an interface that can be invoked through a payload format and protocol that is understood by all of the potential clients of the service. At this point of view, agile methodologies are brought into light. eXtreme Programming is one such methodology which is with in the scope of this paper. eXtreme Programming is completely a revolutionary methodology, it can be concluded saying that “XP is not rules”. So we have seen the major concepts like, Interface Oriented Development, Service Oriented Architecture and eXtreme Programming, for Software Construction with Object Oriented Approach.

References:

1. Service-Oriented Architecture Introduction, By Michael Stevens
2. http://www.itmweb.com/
3. http://webservices.xml.com/
4. www.rational.com/media/whitepapers
5. www.doc.ic.ac.uk/~nfur/iceni/AHM2003/soa.pdf
6. www.nwfusion.com/links/Encyclopedia/S/6187.html
7. java.sun.com/developer/Books/j2ee/jwsa/JWSA_CH02.pdf
8. oopsla.acm.org/oopsla2003/files/track-des.html
9. www.extremeprogramming.org/rules.html
10. http://www.softwarereality.com/ExtremeProgramming.jsp
11. hillside.net/patterns/DPBook/DPBook.html
12. www.nplus1.net/nplus1/static/patterns/ soa_patterns/patterns1.html
13. www-106.ibm.com/developerworks/rational/library/ content/03July/2000/2169/2169.pdf
14. dave.cornelson.net/categoryview.aspx/