Abstract

Software applications are dynamic and constantly changing in several ways. New features and new requirements occur at measured rates of more than 1% per month during development and about 8% per year after release. In operation software applications are among the most dynamic constructs ever built, with thousands of transactions occurring every second and often millions of data items passing through the application as it executes. At longer intervals software applications may be attacked by external cyber threats such as hacking, worms, viruses, and denial of service attacks.

Almost all current methods of sizing, estimating, planning, and designing software projects are static and use combinations of text and rigid diagrams such as flowcharts or the unified modeling language (UML). This paper proposes methods to augment text and static diagrams with full-color, animated, and three-dimensional representations that can show software development progress, execution performance, and also highlight critical topics such as the impact of various kinds of cyber attacks on operating software applications. The animated diagrams would be based in part on the principles of fluid dynamics.

The paper also proposes a new method of software development based on pattern matching, reuse of certified materials, and using intelligent agents to find data and facts about similar applications already constructed and operational. This method in theory can top 100 function points per staff month for projects of 1000 function points in size.
Introduction

If you have a goal of producing a software application of 1,000 function points in size with a productivity rate faster than 100 function points per staff month, how could you do it? Let’s assume you also need close to zero defects and you need zero security flaws. Today’s rates for 1000 function points are between about 6 and 13 function points per staff month, even for agile projects. Today’s applications have far too many bugs at delivery and far too many security flaws even for agile projects.

Custom designs for software applications and manual coding are intrinsically expensive, error-prone, and slow regardless of which programming languages are used and which development methodologies are used. Agile may be a bit faster than waterfall, but it is still slow compared to actual business needs.

User stories and the unified modeling language (UML) are useful, but they are also static representations and fail to capture the dynamic features of software such as requirements growth and performance problems. In the modern world deployed software is at great risk from external cyber attacks of various kinds: hacking, data theft, viruses, worms, and denial of service.

The only effective solution for software engineering is to move towards construction of applications using standard reusable materials rather than custom design and development. The idea is to build software more like Ford builds automobiles on an assembly line rather than like the custom design and manual construction of a Formula 1 race car.

An ordinary passenger car and a Formula 1 race car have about the same number of mechanical parts, but the race car costs at least 10 times more to build due to the large volumes of skilled manual labor involved. The schedule would be more than 10 times longer as well. Custom designs and manual construction are intrinsically slow and expensive in every industry.

If you compare the costs and schedules of building an 80-story office building, an 80,000 on cruise ship, and an 80,000 function point software system the software is much more expensive and also much slower than the other two. When deployed the software is much less reliable and has many more defects that interfere with use than the other two. Worse, the software is much more likely to be attacked by external criminals seeking to steal data or interfere with software operation.

These problems are endemic but not impossible to cure. It is technically possible today in 2014 to build some software applications from standard reusable components. It is also possible to raise the immunity of software to external cyber attack.
In the future more and more standard components will expand the set of applications that can be assembled from certified standard parts free from security vulnerabilities rather than needing custom design and laborious manual coding that tend to introduce security flaws. Assembly from certified components can be more than 10 times faster and cheaper than the best manual methods such as agile, and also much more secure than today’s norms where security vulnerabilities are rampant.

**Dynamic Software Sizing, Modeling, and Planning**

When visiting the Ringling circus museum in Sarasota, Florida I was struck by the extreme efficiency of circus logistics. A full circus crew including entertainers, construction personnel, cooks, and others total to about 1,300 people. They perform shows every day in cities as far as 150 miles away from the city of the day before. The entire circus has to be unloaded from trains, set up, do one or more performances, taken down, put back onto a train, and then travel to the next location. The entire crew needs to be fed, and they also need rest periods. Both animals and humans may need medical treatment from time to time. The travelling circuses do these things every day for several months.

The logistical efficiency of circuses made me consider the logistics of software development. It soon became obvious that the sequence of software development work is not really optimized. This paper suggests some fairly major changes in software logistics based in part on the logistics of circus transport.

For many years the first stage of conventional software development whether agile, waterfall, the Rational Unified Process (RUP) or something else is to gather user requirements. This is the wrong starting place and only leads to later problems.

The first stage of development should be “pattern matching”, or using a formal taxonomy to identify exactly what kind of software application will be needed. Pattern matching is possible because in 2014 the majority of applications are not new and novel, but either replacements for existing applications or variations based on existing applications.

The elements of the formal taxonomy used with the author’s Software Risk Master (SRM) tool include:
Software Risk Master (SRM) Application Taxonomy

1. Nature: New or enhancement
2. Platform: Cloud, smart phone, tablet, personal computer, mainframe, etc.
3. Scope: Program, departmental system, enterprise system, etc.
4. Class: Internal, commercial, open-source, military, government, etc.
5. Type: Telecom, medical device, avionics, utility, tool, social, etc.
6. Problem complexity: Very low to very high
7. Code complexity: Very low to very high
8. Data complexity: Very low to very high
9. Country of origin: Any country or combination of countries (telephone codes)
10. Geographic region: State or province (using telephone codes)
11. Industry: North American Industry Classification (NAIC) codes
12. Methodology: Agile, RUP, TSP, Waterfall, Prince2, etc.
13. Languages: C, C#, Java, Objective C, combinations, etc.
14. Reuse volumes: 0% to 100%
15. CMMI level: 0 for no use of CMMI; 1 – 5 for actual levels
16. SEMAT use: Using or not using Software Engineering Methods and Practices
17. Start date: Planned start date for application construction
18. Deployment date: Planned date of deployment of the application to intended clients

Once the application is placed on the taxonomy the next step is to dispatch intelligent agents to find out exactly how many applications exist that have the same patterns and what their results have been. Assuming that the intelligent agents find 50 similar existing applications, using data from benchmark sources and web searches, it is then possible to aggregate useful information derived from the set of similar projects.

As of 2014 there are about 30 major sources of software benchmarks with an aggregate data collection of almost 100,000 software projects. Of course some of the data is commercially marketed and not available for free, but even so there is data available on hundreds of types of software and thousands of individual applications. Some current sources of software benchmarks include in alphabetical order 4SUM, Aestimat GmbH, CAST software, CERT, COSMIC consortium, Congressional Cyber Security Caucus, Cyber Security and Information Systems (CSIASSC), David Consulting Group, Galorath Inc., Forrester, Gartner Group, German Computer Society Interest Group, IBM, Industrial Information & Control Systems, InfoQ, Information Technology Metrics and Productivity Institute (ITMPI), International Function Point Users Group (IFPUG), International Software Benchmark Standards Group (ISBSG), Namcook Analytics LLC, Price Systems LLC, Process Fusion, Project Management Institute (PMI), Quantimetrics, Quantitative Software Management (QSM), Q/P Management Group, RCBS Inc., Reifer Consultants LLC, Software Benchmark Organization, Software Engineering Institute
(SEI), Software Improvement Group (SIG), Software Productivity Research (SPR), and Test Maturity Model Integrated (TMMI).

Starting with results is a normal practice in other kinds of work outside of software. If you want to hire a contractor to build a 3,000 square foot home it is normal to check references and consider the results of previous homes built by the same contractor. If the contractor has done well in the past the odds are that this will continue for your house. If the contractor has been sued or denied certificates of occupancy, then some other contractor is urgently needed. When commissioning an architect it is also normal to check references and examine past building designed by the architect being considered.

Note that for software projects, subsets of the SRM taxonomy can also be used, such as seeking out embedded software from any industry or any country. The taxonomy also supports interesting statistical studies, such as showing the results of all projects at CMMI level 3, showing the results of all projects that used the Objective C language, showing the results of all projects from the banking industry, or showing the results of all projects using the Software Engineering Methods and Practices (SEMAT) approach.

(This kind of search using a standard taxonomy is somewhat equivalent to using the Zillow data base for searching out similar homes to check assessed value, or using the Kelley Blue Book for searching out the costs for specific automobile models with specific sets of accessories.)

The useful information collected from similar projects will include rates of change, development schedules, staffing, costs, languages, methodologies, and other relevant factors. Further, post-deployment information would show maintenance and also the probability and kinds of cyber attacks that might have occurred for similar projects, and the security precautions used to recover or prevent such attacks.

The data would not be displayed merely in text, spreadsheets, or static diagrams but rather a using a moving, dynamic picture of the application. This animated plan would show rates of requirements change, staff coming and going by occupation group, bugs or defects from various origins entering the software, defect removal methods and effectiveness, and other key factors.

Additional useful information would be architecture patterns, design patterns, code patterns, and defect removal patterns derived from the most successful applications that share the same taxonomy as the application under development.

The metaphor that comes to mind is that software development would perhaps resemble the flow of a river from its origin point to its destination, with changes occurring like tributary streams entering the main flow. After release various kinds of cyber attacks might be visualized by attempts to dam the stream, by diverting some of the water from its normal channel, by
introducing pollutants, or by seine netting to extract valuable fish from the flow of the river; i.e. data theft and identify theft.

Another possible metaphor is the unloading, setup, and reloading of a major circus that travels by train from one city to another giving daily shows. A full circus crew can top 1,300 people and the logistics of circuses are so good that both the U.S. Army and the German Army sent officers to travel with the Ringling Brothers to find out how they could move so much equipment so rapidly and so efficiently. Part of the secret of both software and circuses is that of knowing where to start and the optimal sequence of activities from beginning to end.

Once the project taxonomy is firm and empirical results have been analyzed from similar projects, another set of patterns come into play:

**Software Application Patterns**

1. Architectural patterns for the overall application structure
2. Design patterns for key application features
3. Data patterns for the information created and used by the application
4. Occupation group patterns (designers, coders, testers, QA, etc.)
5. Development activity patterns (requirements, architecture, design, code, etc.)
6. Growth patterns of new features during development and after release
7. Reuse patterns for standard features from standard components
8. Source patterns for the mix of legacy, COTS, reuse, open-source, and custom features
9. Code patterns for custom code to avoid security flaws
10. Risk patterns based on similar applications
11. Security patterns (kinds of attacks noted on similar applications)
12. Governance patterns for software dealing with financial data
13. Defect removal patterns for the sequence of inspections, static analysis, and test stages
14. Marketing patterns for distribution of the software to clients
15. Usage patterns for typical usage scenarios
16. Maintenance patterns for defect repairs after release
17. Support patterns for contacts between customers and support teams
18. Enhancement patterns of future changes after initial deployment
19. Cost and schedule patterns for development and maintenance
20. Value and ROI patterns to compare project costs to long-range value

The basic taxonomy identifies the nature of the specific application to be built and all similar projects that have been built for the past five years. The second set of patterns identifies the technical aspects of the project and provides stakeholders and management with details of an optimal development scenario based on the combined results of all similar projects sharing the same taxonomy.
The author’s Software Risk Master (SRM) tool predicts the patterns of staffing, activities, maintenance, and many others. Due to a patent-pending early sizing method it can so before requirements during the pattern-matching phase. In fact due to being able to predict results before requirements even begin, SRM also predicts the number of pages, words, and diagrams in the requirements themselves and the probable number of requirements bugs or defects. This is done via pattern matching from older applications with the same taxonomy.

One major step after the taxonomy, pattern matching, and data collection from similar projects is to carry out a thorough risk analysis based on the empirical results of all similar projects that share the same taxonomy.

There are a total of 210 risks that face software projects, and every project needs a formal risk analysis and risk abatement plan before serious development money is allotted and spent. Some of the more common risks that would be included in the formal risk analysis include these risks as predicted by the author’s Software Risk Master (SRM) tool:

**Software Risk Master Application Risk Assessment**

<table>
<thead>
<tr>
<th>Risk Analysis from similar projects</th>
<th>Normal Odds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimistic cost estimates</td>
<td>35.00%</td>
</tr>
<tr>
<td>Inadequate quality control using only testing</td>
<td>30.00%</td>
</tr>
<tr>
<td>Excessive schedule pressure from clients, executives</td>
<td>29.50%</td>
</tr>
<tr>
<td>Technical problems hidden from clients, executives</td>
<td>28.60%</td>
</tr>
<tr>
<td>Executive dissatisfaction with progress</td>
<td>28.50%</td>
</tr>
<tr>
<td>Client dissatisfaction with progress</td>
<td>28.50%</td>
</tr>
<tr>
<td>Poor quality and defect measures (omits &gt; 10% of bugs)</td>
<td>28.00%</td>
</tr>
<tr>
<td>Poor status tracking</td>
<td>27.80%</td>
</tr>
<tr>
<td>Significant requirements creep (&gt; 10%)</td>
<td>26.30%</td>
</tr>
<tr>
<td>Poor cost accounting (omits &gt; 10% of actual costs)</td>
<td>24.91%</td>
</tr>
<tr>
<td>Schedule slip (&gt; 10% later than plan)</td>
<td>22.44%</td>
</tr>
<tr>
<td>Feature bloat and useless features (&gt;10% not used)</td>
<td>22.00%</td>
</tr>
<tr>
<td>Unhappy customers (&gt; 10% dissatisfied)</td>
<td>20.00%</td>
</tr>
<tr>
<td>Cost Overrun (&gt;10% of planned budget)</td>
<td>18.52%</td>
</tr>
<tr>
<td>High warranty and maintenance costs</td>
<td>15.80%</td>
</tr>
<tr>
<td>Cancellation of project due to poor performance</td>
<td>14.50%</td>
</tr>
<tr>
<td>Low reliability after deployment</td>
<td>12.50%</td>
</tr>
<tr>
<td>Negative ROI due to poor performance</td>
<td>11.00%</td>
</tr>
<tr>
<td>Litigation (patent violation)</td>
<td>9.63%</td>
</tr>
<tr>
<td>Security vulnerabilites in software</td>
<td>9.60%</td>
</tr>
<tr>
<td>Theft of intellectual property</td>
<td>8.45%</td>
</tr>
<tr>
<td>Litigation (breach of contract)</td>
<td>7.41%</td>
</tr>
</tbody>
</table>
Toxic requirements that should be avoided 5.60%
Low team morale 4.65%
**Average Risks for this size and type of project** 18.44%
**Financial Risk: (cancel; cost overrun; negative ROI)** 44.02%

These risk patterns are derived from statistical aggregation of the results of all projects that share the same taxonomy with the project now under development.

At this point with a solid taxonomy and with empirical data from similar projects it is time to examine specific user requirements for the project to be built. However user requirements are no longer a blank tablet as they are with so many software projects.

The aggregate set of user requirements from all of the similar historical projects would be available for the users to examine and select for the new application. In other words the process of exploring user requirements would be somewhat similar to buying a new car, and selecting color, engine size, leather seats, navigation, sound system, etc. from standard option lists. Only a very few unique features would require custom design and hand coding and hence have custom requirements.

The dynamic planning model would show gradual increase in requirements which add to the total size of the overall application, and also add new potential defects. In addition the dynamic model would be superimposed on a timeline so that progress is instantly visible every day from the initialization of the project through delivery to clients or customers.

The dynamic model would also show the available certified reusable components and also the features that would still require custom design and hand coding. Some components that require custom development might be candidates for future reuse, and if so these features would require certification to near zero-defect status. Features that need custom development would probably use the SEMAT approach and be built by experts at CMMI level 3 or higher.

SEMAT is an interesting approach to identifying the essence of software engineering and thereby improving quality and productivity. SEMAT is a new approach just beginning to have field trials. Because custom design and manual coding are intrinsically error-prone and expensive, probably the best use of the new SEMAT approach would be in developing families of certified reusable components.

Continuing with the metaphor of fluid dynamics and rivers, the main sequence of development using standard reusable components would move at the speed of rapids and quickly be completed. However custom designs and manual coding would greatly slow down progress and be equivalent to part of a river being diverted to a swamp with very slow current speed.
Dynamic Software Development from Certified Reusable Components

Because custom designs and manual coding are slow, error prone, and inefficient it is useful to show the future impacts of varying levels of reusable components. The phrase “reusable components” refers to much more than just reusable source code. The phrase also includes:

Reusable Software Components

1. Reusable requirements
2. Reusable architecture
3. Reusable design
4. Reusable project plans
5. Reusable estimates
6. Reusable source code
7. Reusable test plans
8. Reusable test scripts
9. Reusable test cases
10. Reusable marketing plans
11. Reusable user manuals
12. Reusable training materials
13. Reusable HELP screens and help text
14. Reusable customer support plans
15. Reusable maintenance plans

Figure 1 illustrates why software reuse is the ultimate methodology that is needed to achieve high levels of productivity, quality, and schedule adherence at the same time. Figure 1 illustrates a generic application of 1,000 function points coded in the Java language:
Today in 2014 applications of 1000 function points in size are normally created at rates of between about 6 and 13 function points per staff month using custom designs and manual coding. Waterfall would be at the low end of the spectrum while agile, RUP, TSP, and other advanced methods would be at the high end of the spectrum.

Here are a few samples of various development methods and associated productivity rates using function points per staff month for applications of a nominal 1000 function points in size:

**Productivity Ranges with and without Software Reuse**

<table>
<thead>
<tr>
<th>Methodologies</th>
<th>Function Points Per Staff Month</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Mashup with 85% reuse</td>
<td>37.41</td>
</tr>
<tr>
<td>2. Hybrid with 50% reuse</td>
<td>14.71</td>
</tr>
<tr>
<td>3. Hybrid with 25% reuse</td>
<td>13.52</td>
</tr>
<tr>
<td>4. Agile/scrum</td>
<td>12.08</td>
</tr>
<tr>
<td>5. Spiral at CMMI Level 5</td>
<td>12.05</td>
</tr>
<tr>
<td>6. Extreme programming (XP)</td>
<td>11.89</td>
</tr>
<tr>
<td>7. TSP at CMMI level 5</td>
<td>11.54</td>
</tr>
<tr>
<td>8. Rational Unified Process (RUP)</td>
<td>9.92</td>
</tr>
</tbody>
</table>
As can be seen the examples with reuse are at the top of the list. Below the top two examples
zero reuse is assumed. It should be noted than none of the reuse shown above was “certified” as
discussed in this article. Primarily the reusable components came from similar applications
within the same company. No doubt some of these reusable materials contained bugs, security
flaws, or both. Uncertified reuse is cheaper than custom development, but also somewhat
hazardous.

However without reuse no method would top about 15 function points per staff month for
applications of a nominal 1000 function points in size. This is much slower than needed for
rapidly changing business situations. It is somewhat analogous to having a national automobile
speed limit of only 25 miles per hour.

Figure 1 illustrates reuse from 0% to 90% which is likely to be the upper limit for many
applications. If it were possible to create new applications from 100% reusable components
productivity could top 150 function points per staff month, or about 15 times faster than today’s
averages in 2014 even for agile projects.

Reuse also benefits quality and security. Table 1 shows the approximate impact of reuse on
delivered software defects for an application of 1000 function points in size. Defect potentials
are shown in terms of defects per function point because that metric allows all defect origins to
be included (requirements defects, design defects, code defects, document defects, and bad fixes
or secondary defects):

<table>
<thead>
<tr>
<th>Percent of Reuse</th>
<th>Defects per Function Pt.</th>
<th>Defect Removal Percent</th>
<th>Delivered Defects per FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%</td>
<td>1.00</td>
<td>99.50%</td>
<td>0.01</td>
</tr>
<tr>
<td>80%</td>
<td>1.25</td>
<td>98.00%</td>
<td>0.03</td>
</tr>
<tr>
<td>70%</td>
<td>1.50</td>
<td>95.00%</td>
<td>0.08</td>
</tr>
</tbody>
</table>
Table 2 shows the same sequence only for the prevention and removal of security flaws, also for an application of 1000 function points in size. In general there are fewer security flaws than defects, but they are harder to find and to eliminate so the defect removal efficiency is lower against security flaws than against ordinary bugs:

Table 2: Reuse and Software Security Flaws at Delivery

<table>
<thead>
<tr>
<th>Percent of Reuse</th>
<th>Security Flaws per Function Pt.</th>
<th>Flaw Removal Percent</th>
<th>Delivered Flaws per FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%</td>
<td>0.40</td>
<td>94.53%</td>
<td>0.02</td>
</tr>
<tr>
<td>80%</td>
<td>0.50</td>
<td>93.10%</td>
<td>0.03</td>
</tr>
<tr>
<td>70%</td>
<td>0.60</td>
<td>90.25%</td>
<td>0.06</td>
</tr>
<tr>
<td>60%</td>
<td>0.80</td>
<td>87.40%</td>
<td>0.10</td>
</tr>
<tr>
<td>50%</td>
<td>1.00</td>
<td>85.50%</td>
<td>0.15</td>
</tr>
<tr>
<td>40%</td>
<td>1.20</td>
<td>83.60%</td>
<td>0.20</td>
</tr>
<tr>
<td>30%</td>
<td>1.50</td>
<td>80.75%</td>
<td>0.29</td>
</tr>
<tr>
<td>20%</td>
<td>1.91</td>
<td>78.85%</td>
<td>0.40</td>
</tr>
<tr>
<td>10%</td>
<td>2.50</td>
<td>76.95%</td>
<td>0.58</td>
</tr>
<tr>
<td>0%</td>
<td>3.16</td>
<td>75.05%</td>
<td>0.79</td>
</tr>
</tbody>
</table>

The bottom line is that certified reusable components would be substantially free from both latent defects and also from latent security flaws.

Reuse potential volumes vary by industry and application type. Reuse potential is the percentage of overall application features that are provided by certified reusable components rather than being custom designed and manually coded. Table 3 shows approximate reuse potentials for the current year of 2014, and then future reuse potentials for 2023 or ten years from now:
Table 3: Software Reuse Potentials in Selected Industry Segments

<table>
<thead>
<tr>
<th>Industry Segment</th>
<th>2014 Reuse Potential</th>
<th>Future Reuse Potential</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Electric power applications</td>
<td>35%</td>
<td>95%</td>
</tr>
<tr>
<td>2 Insurance applications - property</td>
<td>45%</td>
<td>90%</td>
</tr>
<tr>
<td>3 Insurance applications - life</td>
<td>50%</td>
<td>90%</td>
</tr>
<tr>
<td>4 Banking applications</td>
<td>60%</td>
<td>85%</td>
</tr>
<tr>
<td>5 State government applications</td>
<td>35%</td>
<td>85%</td>
</tr>
<tr>
<td>6 Education applications - primary/secondary</td>
<td>30%</td>
<td>85%</td>
</tr>
<tr>
<td>7 Wholesale applications</td>
<td>60%</td>
<td>85%</td>
</tr>
<tr>
<td>8 Municipal government applications</td>
<td>40%</td>
<td>80%</td>
</tr>
<tr>
<td>9 Retail applications</td>
<td>40%</td>
<td>80%</td>
</tr>
<tr>
<td>10 Manufacturing applications</td>
<td>45%</td>
<td>75%</td>
</tr>
<tr>
<td>11 Federal civilian government applications</td>
<td>30%</td>
<td>75%</td>
</tr>
<tr>
<td>12 Insurance applications - health</td>
<td>25%</td>
<td>70%</td>
</tr>
<tr>
<td>13 Education applications - university</td>
<td>35%</td>
<td>70%</td>
</tr>
<tr>
<td>14 Weapons systems</td>
<td>20%</td>
<td>55%</td>
</tr>
<tr>
<td>15 Medical applications</td>
<td>15%</td>
<td>45%</td>
</tr>
<tr>
<td><strong>Average Reuse Potential</strong></td>
<td><strong>38%</strong></td>
<td><strong>78%</strong></td>
</tr>
</tbody>
</table>

For many industries most corporate software applications do pretty much the same thing as every other company in the industry. The concept of reusable components is to identify the specific sets of features that are potentially reusable for every company in specific industries. For some industries such as banking and stock trading, there are Federal laws and mandates that make reuse mandatory for at least some critical features.

Some examples of common reusable features circa 2014 include but are not limited to: accounting rate of return, automotive GPS software, bar code reading, browser add-ins, compound interest, Crystal reports, cryptographic key processing, currency conversion, Excel functions, facial recognition, inflation rates, internal rate of return, metrics conversion, real estate depreciation, state sales tax calculations, traffic light controls, and Word templates. As of 2014 reusable components approximate roughly 15% of the features in many common applications, and sometimes top 30%. As of 2014 reuse is not always certified, but the major commercial reusable components are fairly reliable.

Unfortunately there are several gaps in the reuse domain that need to be filled: 1) There is no effective taxonomy of reusable features; 2) There are no available catalogs of reusable features that might be acquired from commercial sources; 3) Software measurements tend to ignore or
omit reusable features, which distorts productivity and quality data; 4) Some software estimating tools do not include reuse (although this is a standard feature in Software Risk Master (SRM)).

One major barrier to expanding reuse at the level of specific functions is the fact that there are no effective taxonomies for individual features used in software applications. Current taxonomies work on entire software applications, but are not yet applied to the specific feature sets of these applications. For example the widely used Excel spreadsheet application has dozens of built-in reusable functions, but there is no good taxonomy for identifying what all of these functions do.

Obviously the commercial software industry and the open-source software industry are providing reuse merely by selling software applications that are used by millions of people. For example Microsoft Windows is probably the single most widely used application on the planet with more than a billion users in over 200 countries. The commercial and open-source software markets provide an existence proof that software reuse is an economically viable business.

Commercial reuse is fairly large and growing industry circa 2014. For example hundreds of applications use Crystal Reports. Thousands use commercial and reusable static analysis tools, firewalls, anti-virus packages, and the like. Hundreds of major companies deploy Enterprise Resource Planning (ERP) tools which attempt reuse at the corporate portfolio level. Reuse is not a new technology, but neither is it yet an industry with proper certification to eliminate bugs and security flaws prior to deployment.

**Development of Suites of Reusable Materials**

Although reusable materials have major benefits when deployed, the construction of reusable materials and their certification normally is more expensive than ordinary custom development due to the need for rigorous quality control and security flaw prevention (and removal). In general developing a reusable component takes about 50% longer and is about 75% more expensive than developing the same feature using normal development practices. The sequence for developing standard sets of reusable components includes but is not limited to the following:

**Development Stages for Certified Reusable Components**

1. Sizing of each feature using function points, SNAP, and logical code size
2. Planning and estimating schedules and costs of feature construction
3. Planning for potential risks and security attacks for each feature
4. Market analysis of reuse potential for each feature (from 10 to 1,000,000 uses)
5. Make or buy analysis for certified reusable materials
6. Tax and government restriction analysis for all reusable materials
7. Mining legacy applications for potential reusable components
8. Patent analysis for unique or valuable intellectual property
9. Formal and reusable requirements creation for each feature
10. Formal and reusable design creation for each feature
11. Formal and reusable code creation for each feature
12. Formal and reusable test plan creation for each feature
13. Formal and reusable test script creation for each feature
14. Formal and reusable test case creation for each feature
15. Formal and reusable user training material for each feature
16. Formal and reusable user documentation for each feature
17. Formal translation into foreign languages, if needed for global sales
18. Formal and reusable HELP text for each feature
19. Formal security inspection for each feature
20. Formal usability inspection for each feature
21. Formal quality inspection for each feature
22. Formal text static analysis
23. Formal running of FOG or FLESCH readability tools on all text
24. Formal code static analysis for all source code
25. Formal mathematical test case construction for each feature’s test cases
26. Formal measurement of cyclomatic complexity for all code
27. Formal measurement of test coverage for reusable test cases
28. Formal testing of each reusable code segment for errors and security flaws
29. Formal collection of historical data (quality, costs, etc.) for each feature
30. Formal certification of each feature prior to integration in component library

As can be seen almost half of the development stages for constructing reusable materials would probably not be performed for ordinary components that are not specifically aimed at being entered into libraries of reusable materials. This is why development of certified reusable components is slower and more expensive than ordinary development. The high initial costs are of course offset by the value and much lower costs of each subsequent reuse of the component.

Because the costs and schedules of creating and certifying reusable software components are much larger than for ordinary development, there is no economic value from creating these components unless they are likely to be reused many times. This brings up a point that in the United States some reusable materials may be taxable assets by the Internal Revenue Service (IRS). Companies should check with tax accountants and tax attorneys prior to beginning a formal reuse program.

It is likely that some kind of brokerage business might be needed to handle the aggregation and distribution of reusable materials. Specific companies are probably not set up or qualified to market their own reusable materials.

A reuse clearing house might be created that could purchase reusable software materials at wholesale prices and then remarket the materials at retail prices. For example reusable banking
modules might be purchased at a cost of $1,500 per function point from individual banks, and then sold to other banks for $150 per function point.

Since it would probably cost each bank more than $1000 per function point for custom development, being able to acquire key banking features for only $150 per function point would be a significant cost saving. Of course the reuse clearing house would need to sell more than a dozen copies of each reusable component in order to make a profit. Of course since banking is a major industry each reusable component might be sold to hundreds or even thousands of banks.

**Summary and Conclusions**

For historical reasons software development methods assume that all applications are unique. This is not true in 2014 although it was true in the 1960’s and 1970’s. The vast majority of software applications within specific industries such as banking, insurance, stocks, energy, etc. are as much alike as peas in a pod. These similarities could lead to much faster development with better quality and higher security levels if certified reusable components began to move into the main stream.

Custom designs and manual coding are intrinsically slow, expensive, error prone, and vulnerable to security flaws no matter what development methods are used and what programming languages are utilized for coding.

A synergistic combination of a formal taxonomy, dynamic visual design methods, and ever-growing libraries of certified reusable components have the theoretical potential for increasing software development productivity rates from today’s norms of less than 10 function points per staff month to more than 100 function points per staff month.

The labor content of software is much higher than it should be. Quality and reliability are much worse than they should be. Security flaws and cyber attacks represent one of the major business threats in all history and improvements are urgently needed. Certified reuse has the potential for improving speed, costs, quality, and security at the same time.
References and Readings by the author Capers Jones

Jones, Capers; *The Technical and Social History of Software Engineering*; Addison Wesley 2014.

Jones, Capers and Bonsignour, Olivier; *The Economics of Software Quality*; Addison Wesley Longman, Boston, MA; ISBN 10: 0-13-258220—1; 2011; 585 pages.


Jones, Capers; *Conflict and Litigation Between Software Clients and Developers*; Software Productivity Research, Inc.; Burlington, MA; September 2007; 53 pages; (SPR technical report).


Jones, Capers; “*A Short History of Lines of Code Metrics*”; Namcook Analytics LLC, Narragansett, RI 2014.
Jones, Capers; “A Short History of the Cost per Defect Metric”; Namcook Analytics LLC, Narragansett RI 2014.

Jones, Capers; “Early Sizing and Early Risk Analysis”; Capers Jones & Associates LLC; Narragansett, RI; July 2011.

Jones, Capers; “The Economics of Object-Oriented Software”; Namcook Analytics; Narragansett, RI; 2014.


18
General References and Readings by other authors


Bundschuh, Manfred and Dekkers, Carol; The IT Metrics Compendium; Springer, 2005.


Crosby, Philip B.; Quality is Free; New American Library, Mentor Books, New York, NY; 1979; 270 pages.


Ebert, Christof; Dumke, Reinder, and Bundschuh, Manfred; Best Practices in Software Measurement; Springer; 2004.


Fowler, Martin; Analysis Patterns; Addison Wesley Professional, 1996.

Fowler, Martin; Patterns of Enterprise Architecture; Addison Wesley Professional, 2002.

Freeman, Elizabeth; Head-First Design Patterns; O’Reilly Media; 2004.


Garmus, David; Russac Janet, and Edwards, Royce; Certified Function Point Counters Examination Guide; CRC Press; 2010.

Gilb, Tom and Graham, Dorothy; Software Inspections; Addison Wesley, Reading, MA; 1993; ISBN 10: 0201631814.


Harris, Michael; Herron David, and Iwanici, Stasia; The Business Value of IT; CRC Press, 2008.

IFPUG (52 authors); The IFPUG Guide to IT and Software Measurement; Auerbach publishers; 2012.


Jacobsen, Ivar; Ng Pan-Wei; McMahon, Paul; Spence, Ian; Lidman, Svente; The Essence of Software Engineering: Applying the SEMAT Kernel; Addison Wesley, 2013.


Popp, Karl: Advances in Software Economics; Books on Demand; 2011.

