Published in final edited form as:
Computer (Long Beach Calif). 2015 December ; 48(12): 80–87. doi:10.1109/MC.2015.372.

Third-Party Software's Trust Quagmire

J. Voas and **G. Hurlburt**

Abstract

Current software development has trended toward the idea of integrating independent software sub-functions to create more complete software systems. Software sub-functions are often not homegrown – instead they are developed by unknown 3rd party organizations and reside in software marketplaces owned or controlled by others. Such software sub-functions carry plausible concern in terms of quality, origins, functionality, security, interoperability, to name a few. This article surveys key technical difficulties in confidently building systems from acquired software sub-functions by calling out the principle software supply chain actors.

Introduction: The Software Conundrum

Software is ubiquitous, pervasive, and ethereal. It is physical in terms of written lines of code that can be reproduced or even printed, but non-physical in terms of behavior when code is executed. Partly art and partly science [1], software is difficult to measure and assess. Unchecked, however, software can induce harmful consequences, particularly in safety-critical operations. Thus, the matter of software assessment is non-trivial. Nonetheless, in order to do a truly meaningful software assessment, a number of considerations must be brought to bear. Unfortunately, many of the true factors underlying software assessment remain unaddressed. This situation is further complicated by the ascent of third party assessors and evaluators in the software marketplace.

By its nature software defies physical analysis. As in the case of well-written literature, well-composed software can be aesthetic. Here, as in literature, syntax and context make a difference, not only perhaps from an aesthetic viewpoint, but also from the dynamics during execution. Software as an art point of view speaks to the creative human outlet that software often provides its most innovative purveyors. Coding artists structure exquisite mathematical algorithms as a painter might create a masterpiece. Another view, however, looks at software development as an engineering endeavor. In true engineering fashion, many believe software can be decomposed to piece parts and these components may be rapidly reassembled to create useful functions on demand.

Unfortunately, both points of view are valid and further cloud the software assessment picture, adding ambiguity to abstract but important questions such as: (1) What is "good"

jeff.voas@nist.gov.

software?, (2) What is “good enough” software?, (3) What makes software comparable in such a way that comparisons of which product is “better” are plausible?, and (4) Who creates criteria that answer these questions?

The “pieces” and “parts” theory

Concepts like Component Based Software Engineering (CBSE)[1]Software of Unknown Pedigree (SOUP)[2]and Commercial Off-the-Shelf (COTS) software [4] were topics that hit their prime in popularity in the late 1990s and early 2000s. There was an excitement and a belief that a maintenance paradigm used in swapping in new hardware components for decommissioned parts was ready for use in software development. If true, these best practices could also alter earlier life-cycle phases such as architecture, design, and implementation. The visual idea was that smaller “chunks” of software could be treated as software Lego ™ units, i.e., as commodities of logic. From a commerce standpoint, this would open up new marketplaces of catalogues of the software components that would bring software engineering practice to the utopian goal of faster, better, and cheaper software development [5]. Moreover, it would eliminate the need for systems to be bespoke (tailor made), and would foster greater competition.

Taking a hardware perspective and moving it to software, however, requires understanding of what makes the hardware component viable in the marketplace and moving that appreciation to the software realm. In this case, it is access to information about component tolerances, quality, and possibly information about how the components were tested. Information is also needed about a component’s functionality, interfaces and interoperability, and expectations from all other subsystems it will interact with (e.g., human, software, hardware) as this information will begin to evolve a understanding of potential *emergent behaviors* forged after composition, some good, and some not so good (e.g., malicious). As a simple example, consider a mobile app that controls the camera but has no permissions to connect to the Internet. If that app is able to collude with another app that does have access, the camera app now does.

So from the commerce standpoint, faster, better, and cheaper is mainly achieved after components are integrated; there will still be an expense in time and resources to produce and tag components with this component-specific information. This added effort, however, may take a bite out of the arguments for faster and cheaper in favor of better.

In 1998 the software assurance community got a wake-up call on this subject in a report by the US National Academy of Science: “*A consumer [patient] may not be able to assess accurately whether a particular drug is safe, but [they] can be reasonably confident that drugs obtained from approved sources have the endorsement of the U.S. Food and Drug Administration (FDA) which confers important safety information. Computer system trustworthiness has nothing comparable to the FDA. The problem is both the absence of*

[1]“*Component-based software engineering* (CBSE) is concerned with the rapid assembly and maintenance of component-based systems, where components and platforms have certified properties, and these certified properties provide the basis for predicting properties of systems built from components.” [2]

[2]“*Software* of unknown (or uncertain) pedigree (or provenance) is defined as software for which full and complete access to the source code, documentation, and/or development history is unavailable.” [3]

standard metrics and a generally accepted organization that could conduct such assessments. There is no Consumer Reports for Trustworthiness" [6].

Are we still there today? Unfortunately "yes." Software is: reused, re-purposed, often used in operational contexts for which it was never intended, becomes bloated with new functionality over time when no one understands its original logic, is exploited for its vast malleability benefits, and the list goes on. But except for issues related to supply chains, why should we care about the software's pedigree or origins? To most, it is simply a 3rd party product – time-to-market and near instant availability are often key factors in the acquisition decision. But, can we keep accepting this condition when it comes to software?

1st and 2nd Parties

Before discussing key technical difficulties in confidently building systems from 3rd party software, we introduce the 1st and 2nd parties. The *end-user* or *consumer* is the 1st party. They are likely devoid of knowledge concerning the component's development and production. The 2nd party is a *system integrator*. This party is the *composer* of the end product from the parts. Note that the 1st and 2nd parties may or may not communicate. If the end product is a mass marketed commercial offering, likely not. If the end product is based on a specification supplied by the 1st party, interaction and communications throughout development and testing is more likely.

The 3rd Parties

Acquired software parts will need some type of a quality check before being integrated. We suggest three distinct human roles here.

First, there is a component *vendor*. The vendor is responsible for producing the 3rd party software components and descriptions of their functionality.

The second 3rd party, the *assessor*, will be responsible for creating the necessary quantitative and qualitative information about the vendor's product. Put simply, assessors are responsible for determining if a yet untrusted component is actually trustworthy.

The assessor should offer a value proposition centered around: easier composition, quicker integration, reduced liability and risks of defective software, and access to someone or an organization that *independently* publicizes the limits of claims made by the vendor when warranted. This is the traditional role attributed to the independent software tester. Note that for this to work the vendor and the assessor cannot be in *collusion*. The problem with this, though, is it places extreme authority in the assessor's hands, therefore creating a need for yet another 3rd party.

Since software quality data is the assessor's primary product, an assessor must first decide what qualities to collect (and how) that will sell. This decision ultimately builds a foundation for an assessor's "why us" sales pitch. But what if an assessor's results are essentially noise to most integrators, that is, although the assessor produces volumes of measures or other

discoveries about a component, the measures are irrelevant to most integrators' needs? This "loss-loss" scenario cannot be ignored - it benefits few except possibly a competing assessor.

The third 3rd party, the *evaluator*, is responsible for assessing the quality and accuracy of the information produced by the assessor. The evaluator reviews the assessor's work, and not the vendor's product *per se*. The evaluator is, in essence, the judge of the assessor and serves as the independent "evaluator" in the traditional test and evaluation scenario.

It often seems that the assessor is in a precarious position far graver than that of vendor and evaluator in terms of the number of inappropriate certifications made and associated liability. Stated simply, software is non-physical; direct quality measurements are harder to take and believe than if the software had measurable physical characteristics. Therefore indirect measurements of quality are often all that are possible to fill the void of few or no direct measures. Given this, can we still expect reduced liability, indemnification, and reduced risks from 3rd party code since we have no physics (e.g., a periodic table) for non-physical systems and we need a defined, timely, repeatable, and fair process from the assessor?

The question of how much liability do the assessors actually incur from their assessments, and how do they indemnify themselves is important. For example, what if they make incorrect assessments that harm the reputation of the vendor or the integrator? The answer is many-fold, however the key to their success and reduced liability is a function of *what* they are assessing for *and how accurate* their results are. Several key factors for potential assessors to consider are: the *timeliness* of results, the *repeatability* of results, *fairness* between competing or similar products (this is where standards are needed), the *ease* of results interpretation, and *bounds* on how much data is collected. For example, bounds should foster timeliness. Bounds should also avoid the results looking like heavy, amorphous, unreadable baggage. Engineering precision is the goal.

While this hierarchy sounds complicated and somewhat duplicative in workload, there is precedent in automotive repair and safety-critical regulated industries. Here a regulatory agency often plays the evaluator role of judging the evidence provided by one or more assessors before authorizing a product to go to market. The Defense Test and Evaluation structure[3] tends to mirror this model in both developmental and operational testing. A secondary benefit in the commercial sector is the creation of a competitive marketplace of assessors from which to choose. This offers the opportunity for promoting the quality and accuracy of their services versus that offered by the competition. That is to say, an assessor is cleared to check for component properties x, y, and z because an assessor has been deemed competent by an evaluator to assess for x, y, and z (and only those three). A good example of this in practice today is the National Institute of Standards and Technology's (NIST) Cryptographic Module Validation Program (CMVP)[4].

[3]Office of the Secretary of Defense "Test and Evaluation Management Guide, 6th Edition", Defense Acquisition University, U.S. Department of Defense, Dec 2012. URL: http://www.dau.mil/publications/publicationsDocs/Test%20and%20Evaluation%20Management%20Guide,%20December%202012,%206th%20Edition%20-v1.pdf

[4]"*Vendors of cryptographic modules use independent, accredited Cryptographic and Security Testing (CST) laboratories to test their modules. The CST laboratories use the Derived Test Requirements (DTR), Implementation Guidance (IG) and applicable CMVP programmatic guidance to test cryptographic modules against the applicable standards. NIST's Computer Security Division (CSD) and*

Multiple risks from faulty assessments by the assessor and evaluator are possible. This begs the question of whether we still need a yet another actor to oversee the evaluator, and if so, when does this recursion end, i.e., a fifth actor to oversee the fourth? The question then is how many eyes are enough? The formal processes engaged by proprietary systems test and evaluation are often streamlined by rapid crowdsourcing on the open source side. While both methods have distinct advantages from a security perspective, they both suffer from risks as well.

At the highest level of abstraction, there are three messages software assessments can convey: (1) the software was developed and tested according to prescribed life-cycle rules or possibly best practices, (2) the software complies with its functional requirements or specification, and (3) the software is *fit for purpose*. "Fit for purpose" is where *context*[5] comes in, particularly when it represents operational inputs (malicious and non-malicious) and associated probabilities of selection. Fit for purpose is more of a 1st and 2nd party issue than a 3rd party concern.

So what have we got? 3rd parties, direct measurement problems, a non-physical entity to evaluate, three key messages a certificate can convey (not all of which are equal in claims), accepting that we are in a quality deficit and no better off than we were in the 1990s, and a feeling that somehow access to *certified* software parts make better building blocks for architecting and maintaining fit for purpose systems or components. Clearly we're missing more than a periodic-like table chronicling ethereal software attributes!

The App Store as a Special Case

Large repositories of 3rd party software apps, termed app-stores, play the role of a somewhat unique 3rd party. This party is not an assessor precisely, but plays the role of an assessor in the interest of licensing-out the vendor's offerings; they are the *marketplace*. In essence they validate a vendor's product prior to public consumption. While they may regard the quality assessment data they collect as gathered for internal use only, the quality or lack thereof of the products in their repositories can tarnish a marketplace's reputation, e.g., the apps in AppStore X are buyer beware for malware because AppStore X did nothing to check for such.

A Software and System Assurance Model

Figure 1 shows three spaces (sets) with unique members. **A** and **B** comprise what we will label as the *system*: **A** represents one or more versions $\{v_1, v_2, v_3, \ldots\}$ of a component, and **B** represents one or more hardware[6] configurations $\{p_1, p_2, p_3, \ldots\}$ that some v_i will execute on. **B**and **C** comprises an implementation's *universe*: **C** represents one or more input environments[7] $\{e_1, e_2, e_3, \ldots\}$ that v_i will execute on as well as a special category of malicious internal computations[8] that create corrupted internal data states[9] or malicious

CSEC [Communications Security Establishment (Canada)] jointly serve as the Validation Authorities for the program, validating the test results and issuing certificates." [7]

[5]For now, consider *context* to simply be all external stimuli that the software experiences at run-time.

[6]Hardware is the computing platform. **B** affects software quality issues such as performance and security.

external inputs $\{m_1, m_2, m_3, \ldots\}$. By selecting one member from **AB**and one or more members from **C**we create *one instantiation*. So an instantiation could look something like $\{v_1, p_6, e_2, m_{10}, m_{44}\}$ representing implementation version 1 running on platform 6 executing input profile 2 while facing security threats 10 and 44. From instantiations assurance arguments are built.

While this paper's title uses 'trust', we use the terms trust and assurance interchangeably. We did so because the distinctions are subtle, not to mention political. For example, there are those in the cybersecurity community that argue that they own the term 'trust' from the early computer security days where trusted operating systems and trusted platforms were being developed and evaluated. Even today there are security-specific research topics that still do, e.g., roots of trust. Here, we have opted to use assurance more in the spirit of what is discussed in the software assurance community [12]. Note that system assurance is at least a function f(**ABC**), and for software assurance, the context of implementation v_i at run-time is a function f(**BC**).

Policy is another assurance concern. Policy can define what is demanded of a system's functionality and quality, or 'that which is technically feasible' can define policy. For example, when it was discovered years ago that a probability of failure estimate (a reliability measure) of 10^{-9} was not feasible, budding safety-critical projects were blacklisted until such fine-grained levels of fidelity were deemed believable. (Determining what is feasible before writing a policy is wiser than writing a policy for the impossible.)

Assurance's Qualities

Software is traditionally viewed *statically* or *dynamically*. Assessors may use both views to create their assessments. For example a static view can offer insights into integration and what is needed from the interfaces during composition. Dynamic views can offer insights into how the software will interact with a universe when executing. Repeated executions according to a universe produces evidence referred to as the software's *behavior*.

Often lost in this community are the invisible behaviors that ultimately determine confidence in messages such as 'complies with requirements' and 'is fit for purpose'. For example: (1) Reliability[10](2) Security[11]and (3) Performance[12]. These "ilities", when directly measured (quantified) or qualified, play an evidential role [8, 9]. Reliability and performance are quantifiable using repeated trials with selected data from a specific universe[13]. Security is unlikely to be quantified since threat spaces are typically *unknown*. Therefore security is more likely qualified, possibly relying on documented evidence concerning which security

[7]Environment includes operational input signals including the probability density function (pdf) of signals. Operational input signals are received from humans, other executing software functions that communicate with v_i at run-time, the hardware **B**and other physical entities, i.e., sensors.
[8]Malicious internal computations are caused by malware that forces v_i into insecure internal states at run-time.
[9]Note that corrupted internal data states are a function of the e_i selected.
[10]*Software reliability* is the probability that software will work properly in a specified environment and for a given amount of time. Using the following formula, the probability of failure is calculated by testing a sample of all available input states. Probability = Number of failing cases / Total number of cases under consideration.
[11]"*Application security* is the use of software, hardware and procedural methods to protect applications from external threats" [10]
[12]"Software performance analysis assesses the quantitative behavior of a software system by comprehensively analyzing its structure and its behavior, from design to code." [11]

risks were mitigated using generally accepted development standards, tools, or methodologies. The question is whether the vendor applied these correctly. Because these behaviors are assessed at specific times, the results are time-stamped.

Equation 2 illustrates that system assurance is more than a function of **AB**and **C**and instead is also a function of the "ility" measurements that emerge at time t_0. Here $\hat{J}$ represents a reliability estimate, $\hat{H}$ represents security evidence, and $\hat{W}$ represents a performance estimate. Equation 1 states that the software assurance of some software implementation v_i is similarly a function of the "ility" assessments at time t_0.

$$\text{Assurance}(v_i \in \mathbf{A})_{t0} = f(p_k \in \mathbf{B}_{t0}, e_l \in \mathbf{C}_{t0}, m_n \in \mathbf{C}_{t0}, \hat{J}_{t0}, \hat{H}_{t0}, \hat{W}_{t0}) \quad \text{(Eq. 1)}$$

$$\text{Assurance}(\text{System})_{t0} = f(v_i \in \mathbf{A}_{t0}, p_k \in \mathbf{B}_{t0}, e_l \in \mathbf{C}_{t0}, m_n \in \mathbf{C}_{t0}, \hat{J}_{t0}, \hat{H}_{t0}, \hat{W}_{t0}) \quad \text{(Eq. 2)}$$

Note for brevity we are only illustrating assurance using three "ilities"; in practice, the "ilities" considered will be related to the level and type of assurance desired.

To close this section, consider the interplay between different components' "ilities." Consider the plight of an integrator asked to sequentially integrate two components, ξ and ψ (Figure 2). Not only must the integrator be confident that ξ and ψ composed will yield the expected emergent functionality, but also that $\xi\,\hat{J}_{ty}$ composes satisfactorily with $\psi\,\hat{J}_{tz}$, i.e., the composite reliabilities are compatible, e.g., two highly reliable components do not have an emergent behavior of low reliability. And this is true for security and performance. Note that this is further complicated when we consider composing heterogeneous attributes, e.g., $\xi\,\hat{W}_{ty}$ composed with that $\psi\hat{H}_{tz}$. In layman's terms, performance built into ξ could negatively affect the security built into ψ and vice versa. Recognize that all we have illustrated involved two components and three "ilities" -- imagine this for hundreds of components additional "ilities". The key point is that the interactions between "ilities" from standalone software parts at run-time create yet another factor affecting claims of assurance, similar to those of medical drug interactions.

Conclusions

Given that the 3rd party model appears to be gaining strength for business and legal reasons, to say nothing of improved assurance, it suggests some necessary actions. First, vendors will continue to innovate with sophisticated algorithms that would be the pride of any quant. This, in turn, challenges the assessors to get out ahead of the technology they are challenged to assess. Otherwise, they will be relegated to the sidelines and unable to make valid contributions to the industry and consumers, much less assure levels of security. Finally, the evaluators need to get out ahead of the assessors to effectively pass judgment on relative assessment acumen. In essence, it becomes everyone's responsibility to be on top of the

[13]Note that quantifiable measures are no panacea. For illustration, how do you quantify a basket of apples? Weight? Number? Number per variety? Number per color? Etc. Reliability metrics are no different. Different reliability models will give different estimates, particularly when distinct models input different parameters or weigh the parameters differently. This will be problematic for assessors.

game. Despite current perceptions, fueled by a burgeoning entrepreneurial movement, innovation is not the sole province of the vendor – there is much left for the other third parties.

REFERENCES

1. McConnell S. The Art, Science, and Engineering of Software Development. IEEE Software. 1998 Jan-Feb;15(1)
2. Bachmann, Felix; Bass, Len; Buhman, Charles; Comella-Dorda, Santiago; Long, Fred; Robert, John; Seacord, Robert; Wallnau, Kurt. Technical Report CMU/SEI-2000-TR-008/ESC-TR-2000-007. Software Engineering Institute, Carnegie Mellon University; 2000 May. Technical Concepts of Component-based Software Engineering; p. 7
3. Rowlands, Gereth. Software, Safety and SOUP. Standards in Defense News. 2003; (187)
4. Perry WJ. Secretary of Defense Memo: Specifications and Standards – A New Way of Doing Business. http://sw-eng.falls-church.va.us/perry94.html.
5. Menzies, T.; El-Rawas, O.; Hihn, J.; Boehm, B. Can we build software faster and better and cheaper?. Proc. of the 5th Int'l Conf. on Predictor Models in Software Eng. (PROMISE'09); ACM, New York. 2009.
6. National Academy of Sciences Report. National Academy Press; 1998. Trust in Cyberspace.
7. http://nist.gov/cmvp
8. Voas J. "Software's Secret Sauce: the ìlities," Quality Time Column. IEEE Software. 2004 Nov; 21(6):2–3.
9. Voas J. Software Quality Unpeeled. Crosstalk. 2008 Jun.:27–30. www.stsc.hill.af.mil.
10. Ram Mohan Rao K, Pant Durgesh. A Threat Risk Modeling Framework for Geospatial Weather Information System: A DREAD based Study. International Journal of Advanced Computer Science and Applications. 2010 Sep; 1(3):20–28. , (Section B: Security Assessment on Page 21).
11. Cortellessa, Vittorio; Di Marco, Antinisca; Inverardi, Paola. Model-based Software Performance Analysis. Berlin Heidelberg: Springer: 2011. Chapter 1 on Page 1.
12. https://buildsecurityin.us-cert.gov/swa

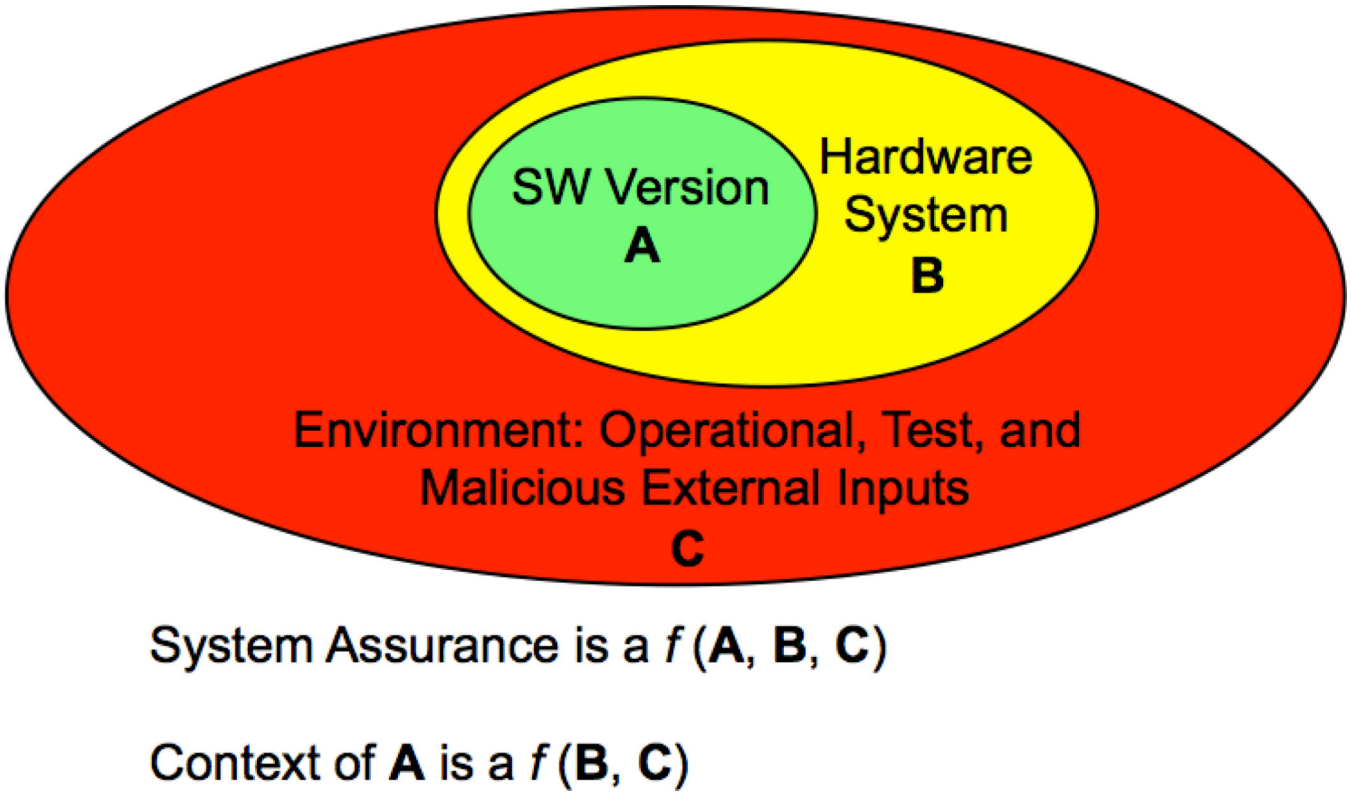

Figure 1.
A Systems Assurance Model

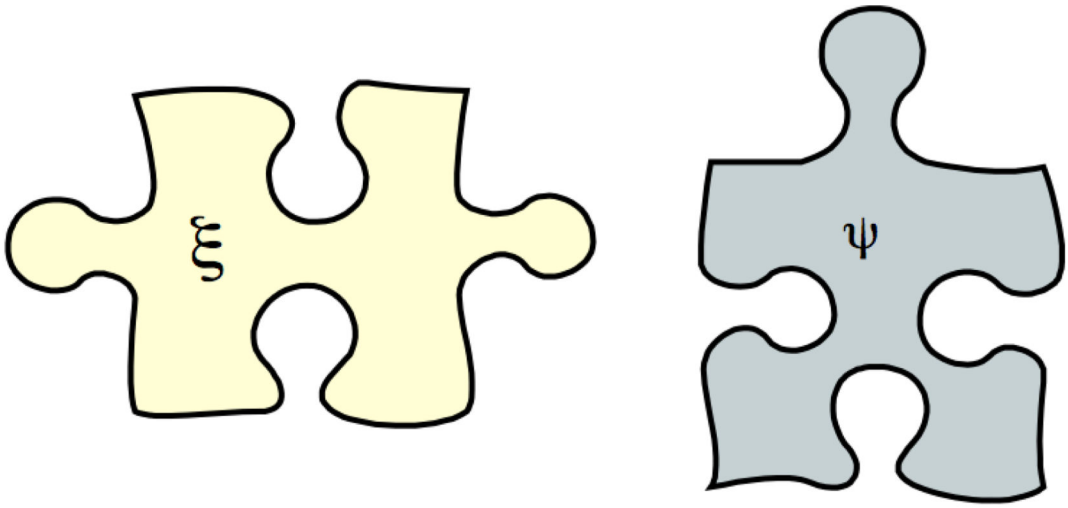

Figure 2.
Component Composition Concerns