YAZILIM TESTLERİNE GÖRE GELİŞTİRME SÜRECİ VE MİMARİ AÇIKLAMA

Yazılım Mühendisliği Ana Bilim Dalı

Tezsiz Yüksek Lisans Bitirme Projesi

Egemen İsa TARİH

Proje Danışmanı: Prof. Dr. Femin YALÇIN KÜÇÜKBAYRAK

Ocak 2023

İzmir Kâtip Çelebi Üniversitesi Fen Bilimleri Enstitüsü Yazılım Mühendisliği A.B.D. öğrencisi **Egemen İsa TARİH** tarafından hazırlanan **Yazılım Testlerine Göre Geliştirme Süreci ve Mimari Açıklama** başlıklı bu çalışma tarafımca okunmuş olup, yapılan inceleme sonucunda kapsam ve nitelik açısından başarılı bulunarak tarafımdan YÜKSEK LİSANS BİTİRME PROJESİ olarak kabul edilmiştir.

ONAYLAYAN:

Proje Danışmanı: **Prof. Dr. Femin YALÇIN KÜÇÜKBAYRAK**
İzmir Kâtip Çelebi Üniversitesi

Proje çalışmasına katkılarından dolayı danışmanım Sayın Profesör Doktor Femin YALÇIN KÜÇÜKBAYRAK'a teşekkürlerimi sunarım.

YAZILIM GELİŞTİRME SÜRECİ VE MİMARİ GÖSTERİME DAYALI YAZILIM TESTİ

Öz

Bu çalışma yazılım geliştirme sürecinin farklı aşamalarında kullanılabilen ve kullanılması önerilen birçok test etme tekniğini açıklamaktadır. Söz konusu süreç yaygınlıkla kullanılan (örneğin Tümleşik Süreç, Atik Yazılım Geliştirme Süreci) herhangi bir süreç olabilir fakat test tanımları için V-modeli temel alınmıştır. Birim ve entegrasyon testleri gibi testler, tamamlanmış yazılım seviyesine göre tanımlanmıştır.

Geleneksel yazılım test etme teknikleri açıklanmış ve sistem seviyesinde uygulanabilecek biçimsel bir teknik tanıtılmıştır. Bu teknik geliştirilmekte olan yazılımın resmi mimari gösterimine bağlı olarak testlerin tanımlanması için kullanılmaktadır. Mimari gösterim, sistemin yüksel seviyedeki görünümünü resmi, matematiksel bir şekilde verdiği için testleri tanımlayacak otomatik bir programın yapımına da uygundur.

Tanıtılan mimari gösterime bağlı test tanımlama tekniği, testlerin yeni sürüm aşamasında yeniden kullanılabilmesi için bir ileri noktaya götürülmüş. Yeni sürüm testleri yeni sürümler yayımlanmadan önce koşulan testler kümesidir. Tüm sistem ve Kabul testlerinin her yeni sürümde koşulması genellikle uygun değildir. Tanıtılan tekniği kullanarak yeni sürüm testlerinde koşulması gereken minimal test kümesini seçmeye yönelik metotlar tanıtılmıştır.

Anahtar Sözcükler: Yazılım Testi, Birim Testi, Cam Kutu Testi, Kara Kutu Testi, Bütünleştirme Testi, Sistem Testi, Kod Kapsaması

1.GİRİŞ

1.1 Test Nedir?

Borland'dan James Bach, yazılım testini "düşünülemez olandan kaçınmak için görünmez olanı belirsiz olanla karşılaştırma süreci" olarak tanımlar. Test aslında bir sistemdeki kusurları bulmayı amaçlayan bir süreçtir. Test, hata ayıklama ve kabul amacıyla da kullanılabilir. R. Vaderwall'un belirttiği gibi "Yazılım testing, bir ürünün davranışını tahmin etme ve bu tahmini gerçek sonuçlarla karşılaştırma sürecidir".

Test sürecinin biraz belirsiz olduğu ve geliştiricilerin sesine çok bağlı olduğu "Hata Ayıklama Odaklı Test" (Gelperin & Hetzel) ile başlayarak, yazılım testi son 50 yılda birçok gelişme kaydetti. Testin önemi ve yüksek maliyeti maalesef çok kötü deneyimlerle anlaşılmıştı; Altı cana mal olan Therac25, 500 milyon dolara mal olan ARIANE 5 ve Mars görevleri, yörünge kaşifi ve kutup inişi 300 milyon dolara mal oldu. Yazılımın artan boyutu ve karmaşıklığı ile test, maliyetini azaltmak için yazılım geliştirme sürecinin çok erken aşamalarında entegre edildi. Tasarım gibi projenin ilk aşamalarında bir hata bulmak , geliştiricinin sadece birkaç günlük çalışmasına mal olacak, ancak daha sonraki aşamalarda maliyet artışları, üretimin durmasına, sahadaki ürünler üzerinde yeniden çalışmaya, hatta bir kullanıcının ömrüne neden olabilir.

Yazılım, kullanılan geliştirme tekniğinden bağımsız olarak asla doğru değildir. Test süreci, yazılımdaki hataları bulmayı ve böylece düzeltilmelerini amaçlamaktadır. Test sürecinin projeye erken aşamalarda entegre edilmesi maliyeti düşürecek ve defe cts'yi bulma şansını artıracaktır.

1.2 Geliştirme Süreci Nedir?

Yazılımın boyutlarının artması, geliştirme ekiplerinin büyüklüğünün artması ve başarılı projelerin yürütülememesi ile birlikte projelerin başarısını artırmak için geliştirme süreçleri tanımlanmaya başlanmıştır. Havaalanı kontrol ve uçuş rezervasyon sistemleri gibi projelerin artan karmaşıklığı ile nesne yönelimli programlama ve gerçek zamanlı işletme sistemleri gibi yeni teknikler tanımlanmıştır. Bu teknikler taktiğe dayanıyordu, böl ve yok et. Sisteme tanıtılan bileşenler ve iş parçacıkları ve bu bileşenler genellikle projeye harcanan zamanı azaltmak için bireysel geliştiriciler tarafından geliştirilmiştir. Bu geliştiricilerin organizasyonu, şelale, aşırı süreç ve birleşik süreç gibi bazı geliştirme süreçleriyle yapılmaya başlandı.

Yazılımın parçalanmış yapısı ve bir üründeki öneminin artması içerisinde güvenilirlik (düzgün çalışma şansı) kilit konulardan biri olmaya başlamıştır. Örneğin, 0,9 güvenilirliğe sahip iki bileşenden oluşan bir ürün güvenilirliği 0,81'dir. Yazılım insan tarafından geliştirildiğinden ve güvenilirlik kabaca 0,7 olduğundan, başarı hedefine ulaşmak için her bileşenin güvenilirliği ciddi testlerle arttırılmalıdır. Bu nedenle test, geliştirme sürecinin her adımında göz önünde bulundurulmalıdır.

Yazılımın gözlemlenebilirliği, birim düzeyinde test, hata, kabul testi tanımı gibi terimlerin tanımı, "V" model testine geçilecek ve testin geliştirme sürecindeki yeri aşağıdaki çalışmalarda tanıtılacaktır, ancak test teknikleri ve yöntemlerinin süreçlerin soyut tutulması için belirli bir süreç düşünülmeyecektir.

Bir geliştirme süreci temel olarak amacı yazılımın geliştirilmesi veya evrimi olan bir dizi faaliyettir ve bir yazılım sürecindeki genel faaliyetler spesifikasyon, geliştirme, doğrulama ve evrimdir.

1.3 Mimari Açıklama Nedir?

Yazılım mimarisi ve tanımı, yazılım ve geliştirme ekiplerinin artan büyüklüğü ile önem kazanmaktadır. Bileşenlerin nasıl entegre edileceği , sistemin süreçlerinin bileşenlere nasıl dağıtılacağı ve sistemin düzgün çalışacağından nasıl emin olunacağı gibi teknik konular. "Yazılım mimarileri, sistemlerin temel üst düzey yapısal ve davranışsal özelliklerini tanımlamayı amaçlamaktadır. Birrchitecture Description Dilleri, bu özellikleri algoritmik olarak analiz edilebilecek ve manipüle edilebilecek şekillerde tanımlar. " (Jin ve Offutt)

Yazılımın mimari açıklamaları özellikle entegrasyon, sistem ve entegrasyon testi için yararlı olabilir. Yazılım mimarilerinin resmi tanımlarını analiz etmek, aşağıdaki çalışmanın endişesi değildir. Çok sayıda yazılım mimarisi dili mevcuttur. Bu çalışmada, 'bulunması zor hataları' bulmak ve test çabasını en aza indirmek temel birimdir ve yazılım mimarileri bu amaca ulaşmada ana araçlardır.

Aşağıdaki çalışmayı yararlı kılmak için, mimari tanımlama için en iyi dili bulmak endişe verici değildir, ancak test metodolojilerini tanımlamak ve devlet diyagramları gibi pazar ve süreç onaylı, popüler yazılım mimarisi tanımlama dilleri için yararlılığını göstermek bu çalışmanın odak noktasıdır.

1.4 Yaşam Döngüsü Olarak Test Etme

Test, projenin tanımından başlayarak kabulüne kadar yazılım geliştirme yaşam döngüsünün bir parçası olarak düşünülmelidir. Kabul temel olarak kabul testini geçmektir. Kabul testinin oluşturulması projenin en başında yapılması gereken bir

görevdir. Kabul kriterleri oluşturulmalı ve kabul kriterleri uygulanmalıdır. Doğrulama ve doğrulama, bir yazılım ürününü onaylamanın iki ana anahtar tanımıdır.

Doğrulama, temel olarak sistemin gereksinimlere, müşteri isteklerine (varsa) uyup uymadığını ve hedeflenen kalite düzeyinde amaçlandığı işlevleri yerine getirip getirmediğini belirlemektir. Başka bir deyişle, 'doğru şeyi inşa ediyor muyuz?' sorusunun cevabıdır.

Doğrulama temel olarak sistemin güvenilir teknikler kullanıp kullanmadığını ve seçilen bileşenleri ve işlevleri doğru şekilde yerine getirip getirmediğini belirlemektir. Başka bir deyişle, 'doğru inşa ediyor muyuz?' sorusunun cevabıdır.

Doğrulama, doğrulamanın düşük düzeyli bir etkinlik olduğu yüksek düzeyli bir etkinliktir. Bu nedenle, sistemin mimari bir de scription'unu göz önünde bulundurarak,onu doğrulamak için, ne yapacağını, yapması gereken şey olup olmadığını görmeye çalışmalıyız. Doğrulama, sürecin spesifikasyon düzeyinde dikkate alınmazsa çok maliyetli olabilir. Şirket, müşterinin sipariş etmediği bir ürünle sonuçlanabilir.

Doğrulama, geliştirilmekte olan tasarımın ve yazılımın gerçekten bir şey yapıp yapmayacağını bulmaya çalışan düşük seviyeli bir etkinliktir.

Ürünün hazır olduğunu onaylamak için testler alışsa da, amaç ürünün güvenilir olmadığını ve doğru çalıştığını kanıtlamak olmalıdır. Yazılımın özellikleri bu amaç için ana belgedir.

Yazılımın çalışmadığını kanıtlamak için, geliştirme sürecinin farklı seviyelerinde farklı teknikler kullanılabilir. Bu çalışma boyunca düşük ila yüksek seviyeli testler dikkate alınacaktır. Anahtar terimlerin tanımları verilecek ve yazılımın en küçük barışlarını test etme teknikleri, birimleri, verilecektir. Bu teknikler temel olarak statik ve dinamik olmak üzere iki gruba ayrılacaktır. Avantajları ve dezavantajları olan bir grup teknik tanıtılacaktır, çünkü ilk önce entegre edeceğimiz tüm birimlerin düzgün çalıştığından emin olmalıyız. Yazılım mimari tanımlama dillerine giriş verilecek ve entegrasyon testi tanımlama ve indirgeme teknikleri tanıtılacaktır. Yazılım test sürecine odaklanarak en çok kabul gören 'V' modeli olacak.

Test çabalarını geliştirme süreciyle son derece entegre tutmak için tartışıldı. Gelecekteki çalışmalar ve test sürecinin otomasyonu hakkında birkaç söz de harcanacaktır.

Geliştirme süreci göz önünde bulundurularak, ünitelerin böylece birim testinin hazırlanması, entegre edilerek entegrasyon testi, sistemin oluşturulması böylece kök testi, ürünün kabul edilmesi ve değişikliklerin yapılması ve yeni versiyonların

hazırlanması böylece regresyon testi buna göre yürütülür. Bu çalışma, gerekli arka plan bilgilerini veren bu öğelere odaklanmaya çalışacaktır.

2. BİRİM TESTİ

2.1 Birim Test Türleri

Bir yazılım ürünü hazırlanırken, kullanılan her yazılım modülünün veya biriminin düzgün çalıştığı kanıtlanmalıdır. Genellikle yazılım ürünleri bir geliştirici ekibi tarafından hazırlanır vefarklı geliştiriciler tarafından hazırlanan ünitelerin entegrasyonu sırasında birçok pr oblemi ortaya çıkar. Sorunları en aza indirmek ve aynı zamanda nihai ürünü hazırlamak için harcanan çaba ve zamanı en aza indirmek için, her ünite ayrı ayrı test edilmeli, düzeltilmeli ve doğrulanmalıdır. Re, uygulanması önerilen tekniklerden oluşan bir gruptur. Bu teknikleri geliştirme maliyetlerine ve hata bulma konusundaki ardıllık oranlarına göre tartışacağız. Ayrıca, projenin sağlamlığına ve sıfatlarına göre test etmek için harcanacak zaman miktarını da tartışacağız.

Birim testleri iki ana gruba ayrılabilir; statik ve dinamik testler. Bu testler, bazı kesişme noktaları olmasına rağmen farklı alanlarda güçlüdür. Statik testler, sisteme düşük maliyeti ile yazılım grupları için özellikle kullanışlı ve güçlüdür. Bu tür bir test, test edilecek kaynaktan hazırlanmış herhangi bir yürütülebilir dosya gerektirmez.

Dinamik test metodolojileri, test edilecek yazılım bloğu için derlenmiş kaynak ve test taslakları ve vektörleri veya prepakırmızısı olan test yazılımı gerektirir. Her modülün sürücüleri ve taslakları olmalıdır. Sürücüler test edilen yazılım bloğuna mesaj gönderir ve saplamalar bu bloktan gelen mesajları kabul eder ve yanıtları döndürür. Her modülün sürücüleri ve saplamaları olması gerekse de, pratikte, bir modül test edildikten sonra, genellikle diğer modüller için bir sürücü / saplama olarak kullanılır. Bu, birim ve entegrasyon testi arasındaki ayrımı gerçekten bulanıklaştırır ve önerilmez.

Gözlemlenebilirlik, özellikle dinamik birim testi için yazılım testinin temel konularından biridir. Gözlemlenebilirlik, operasyonlardan önce ve sonra durumu gözlemleme yeteneğidir, "gizli" devletler dahil her kod satırına, işlevin arayüzü olan yöntemler aracılığıyla erişilebilmelidir.

2.2 Statik Birim Testi

Özellikle statik test stratejileri için bazı çok temel test kuralları ve disiplinleri akılda tutulmalıdır. Statik testler, geliştiriciler, programcılar veya test ediciler grubu tarafından yapılır, bir kod bloğu olarak, incelemelerinin konusu olarak, hepsi tarafından amacın programcıyı aşağılamak değil, hataları bulmak olduğu kabul edilmelidir.

Programcı hazırlamış olduğu yazılımı test etmemelidir. Katılımcılar, sadece programın yapması gerekeni yaptığını değil, aynı zamanda yapmaması gerekeni yapmadığını da test etmelidir. Her türlü testin amacı, yazılımın hatasız olduğunu kanıtlamak değil, hataları bulmaktır. Herkes tarafından bilindiği gibi, hatasız program test ile kanıtlanamayabilir yapılan test miktarından gard alınmayabilir. Hiçbir test miktarı böyle bir şeyi garanti edemez. Genel olarak, yazılımın çok sayıda hatanın bulunduğu bölümleri, daha fazlasını aramak için çok iyi bir yerdir.

Statik birim testi temel olarak programcıların kesinleştirilmiş, derlenmiş bir program hakkındaki bir grou p'sinin gözden geçirilmesidir. Programcı, yazılım bloğunda bazı testler yapmakta özgürdür, belki de sadece yapması gerekeni yapıp yapmadığını görmek için. Ancak beklenen şey, yazılım bloğunun sonlandırılması (işlevselliğin doğru olması beklenmemektedir) ve derlenebilir olmasıdır. Derleyicinin verdiği tüm hatalar ve uyarılar, geliştiriciler grubu tarafından kullanılması kararlaştırılan uyarı düzeyine bağlı olarak temizlenmelidir. Statik analizler (birim testleri),yukarıdaki koşulları yerine getiren programının kaynak kodu üzerinde yapılır; eksiksiz, derlenebilir ve derleyici uyarısı içermez.

2.2.1 Kod İncelemesi

Kod denetimi 3-5 programcıdan oluşan bir ekip tarafından yapılır. Bunlardan biri materyalleri dağıtan, toplantı çağrısı yapan ve toplantı sırasındaki hataları kaydeden moderatör olur.

Kod inceleme toplantısının katılımcılarına gözden geçirilecek yazılımı tanıtmak için kod incelemesinden birkaç gün önce bir ön toplantı yapmak genellikle çok yararlıdır. Programcı hazırlamış olduğu yazılımı çok kısa bir süre tanıtmaktadır. Bu toplantı sırasında kod düzeyinde girişten kaçınılmalıdır. Modüldeki fonksiyonlar, gelişim amaçlarına ve görevlerine göre tanıtılmalıdır. Yazılım bloğunun n mimari açıklaması, kullanım durumu veya yazılıma referans olan standart tanıtılabilir. Bu toplantının amacı, programcılara yazılım birimini kısa sürede tanıtmak, böylece yazılımı daha etkin bir şekilde inceleyebilmelerini sağlamaktır.

Tanıtım toplantısı sırasında ayrıntılı bilgi vermek, gözden geçirenlerin yazılımın bir bölümüne odaklanmasına ve bazı hataları gizlemesine neden olabilir. Giriş toplantısı kısa olmalı ve yazılım bloğunda yer alan işlevsellik, algoritma ve tasarım kalıpları hakkında kısa bilgi vermelidir. Bu toplantı yaklaşık yarım saat içinde yapılmamalıdır. Bu toplantı sadece süreci hızlandırmak içindir ve müfettişlerin soru sormalarına izin verilmez, kod gözden geçirirken kendi kendine bir sapma yapmalıdır.

Bu toplantıdan sonra moderatör tarafından katılımcılara kaynak kodu dağıtılmalı ve kod incelemesi için toplantı planlanmalı ve katılımcılarasatır satır bakmaları ve not almaları için gerekli zaman verilmelidir.

Bu seviyede iki farklı yöntem kullanılabilir. Bunlardan biri, her katılımcıya tüm kaynağı inceleme görevi vermektir, çünkü farklı programcılar kodun tüm bölümlerine odaklanacaktır. bir hata bulma olasılığı çok yüksektir. Diğer yöntem, yazılımı daha küçük parçalara bölmek ve bu parçaları katılımcılara atamaktır. Bir kod satırı sınırı verilebilir, örneğin 250 satır kod. "Şaşırtıcı bir şekilde, bu 'taşa oyulmuş' bir sınır! Hardcop y, gösterilen kaynak kodu satırlarının sayısını içermelidir." (Baldwin, 1992) İkinci yöntem, programcılar üzerinde daha az zaman harcayacakları için daha az maliyetli olacaktır, ancak büyük olasılıkla bulacaktır.

İlk yöntemden daha az hata. İkinci yöntem için, özellikle bölünmüş parçalar arasındaki arayüzler biraz odak dışıdır.

Daha önce hazırlanmış bir kontrol listesi, bu denetimler için çok yararlı olabilir. Bu kontrol listesi birçok temel soru içermelidir, çünkü basic soruları normalde çok maliyetli olabilecek temel hataları önleyebilir. "Bildirilen tüm değişkenler?", "Varsayılan değerler anlaşıldı mı?", "Diziler ve dizeler başlatıldı mı?", "Her döngü sona eriyor mu?" gibi sorular bu kontrol listesine eklenebilir.

İnceleme yapmak için, okuduklarınızı tam olarak anlamaya çalışarak kod satırını satır satır gözden geçirin. Her kod satırında veya bloğunda, geçerli soruları arayarak denetim kontrol listesine göz atın. Uygulanabilir her soru için, bir sapmanın "evet" olup olmadığını bulun. Evet cevabı, olası bir kusur anlamına gelir. Bir yere yazın. Bazı soruların çok düşük seviyeli olduğunu ve sözdizimsel ayrıntılarla ilgilendiğini, diğerlerinin ise üst düzey olduğunu ve bir c ode bloğunun ne işe yaradığının anlaşılmasını gerektirdiğini fark edeceksiniz. Zihinsel odağınızı değiştirmeye hazır olun. (Baldwin, 1992)

Kod inceleme toplantısı sırasında, katılımcılardan biri tanıtıcı olur. Kaynak tüm programcılar tarafından görüntülenecektir, bu nedenle bir reflektör veya büyük ekran monitörü çok zor olabilir. Tanıtıcı, satır satır içinden geçerek kaynağın ne yaptığını söyleyecektir. Bu süre zarfında, incelenen yazılımın programcısı dışındaki tüm katılımcılar, riskli veya sorunlu parçaları yorumlayabilir ve bildirebilir. Softw'un yazarının yorum yapmasına izin verilmez ve sadece olası soruları cevaplamak için oradadır. Moderatör, bu toplantı sırasında bildirilen hataların notlarını alacaktır.

Program, ortaya atılan sorulara, katılımcılar tarafından yapılan yorumlara ve bildirilen hatalarla varsa kontrol listesine göre analiz edilecektir.

Bu beyin fırtınası toplantısının iki saatten uzun sürmesi önerilmez. Gözden geçirmenin daha fazla zamana ihtiyacı varsa, toplantıyı sonlandırmanız ve kaynağın geri kalanı için başka bir toplantı zamanlamanız önerilir . İnceleme tamamlandığında, moderatör hataları yazılımın programcısına bildirecek ve değişiklikleri takip etmekten sorumlu olacaktır. Moderatörün, kod üzerinde çok sayıda değişiklik yapılması durumunda başka bir incelemeyi geri çağırmasına izin verilir. İncelemeye göre modülde bir tasarım değişikliği varsa bunu yapmanız önerilir.

Kod incelemesi için harcanan iki saat, geliştirmenin erken bir aşamasında hataların% 60-70'ini bulacaktır. Hataları erken aşamalarda bulmak, çözümü sistem için daha az maliyetli hale getirecektir. Yazılımdaki hatayı bulmak için en erken seviye muhtemelen yazılımın tasarım seviyesindedir.

2.2.1.1 Tasarım İncelemesi

Test için kaynak kodu veya yazılım bulunmamasına rağmen tasarım incelemeleri yapmanız şiddetle tavsiye edilir. Yazılım uygulandıktan sonra tasarımdaki bir hatayı düzeltmek, yazılımı tekrar yazmak anlamına gelebileceği ve programcı için birkaç haftaya mal olabileceği için maliyetli olacaktır.

Bir tasarım gözden geçirme toplantısı, incelenen modülün sorumlusu tarafından planlanabilir. Tasarım tamamlanmalıdır. Toplantı için herhangi bir dokümantasyon veya giriş gerekmez. Geliştirmenin bu düzeyinde belgeler mevcut olmayabilir.

Toplantıda tasarımcı, tasarımın ne yapması gerektiğini açıklar ve akış şemaları, durum diyagramları, sınıf diyagramları gibi mimari tanımlama dillerini kullanarak bunu nasıl yapacağını açıklamaya çalışır. Toplantının katılımcıları tasarım hakkında yorum yapar ve sorular sorar, önerilerde bulunur ve sistem bunlara göre yeniden tasarlanır.

2.2.2 İzlenecek yol

İzlenecek yol, 3-5 programcıdan oluşan bir ekip tarafından yapılır. Bunlardan biri, işlemci tarafından yapılması amaçlanan işlemleri yapan test cihazı olur. Varsa kağıt veya karton. Moderatör, kod incelemesinde olduğu gibi aynı role sahiptir, materyalleri dağıtır, toplantı çağrısı yapar ve toplantı sırasındaki hataları kaydeder.

O'Neill (1999), "Software izlenecek yollar, üreticinin anlayışını doğrulamak ve alınan yaklaşımı doğrulamak için kullanılan gayri resmi bir incelemedir" demektedir.

Uyarılardan derlenen ve temizlenen kod toplantıda incelenir. Test cihazı işlemcinin rolünü oynar ve gerekli işlemleri uygular. İşlev/yöntem için ayarlanmış bir giriş katılımcılar tarafından seçilir. Test cihazı, yazılımda yapılan işlemler giriş kümesine uygulanırken gerekli tüm değerleri bir tahtaya veya kağıda yazar. Tüm loops ve kararlar teorik olarak, fonksiyon / yöntem dönene kadar satır satır yürütülür. Döngüler geri dönene kadar döndürülür. Toplantının katılımcıları bulunan hataların miktarına ikna olana kadar yeni başlangıç değeri kümeleri uygulanmalıdır.

Yazılım kılavuzu, yazılım yapıtının tüm yönleri hakkında üstün bilgi edinmede yazılım yapıtının üreticisinin veya yazarının ihtiyaçlarına hizmet etmek üzere düzenlenmiştir. Bu bir öğrenme deneyimidir. Walkthrou gh yazılımının arzu edilen bir yan etkisi, hakemler arasında ortak bir vizyonun oluşturulması ve katılımcılar arasında alınan yaklaşımlar, uygulanan ürün ve mühendislik uygulamaları, yeteneklerin ve özelliklerin bütünlüğü ve doğruluğu ve domai n ürünü için yapım kuralları konusunda fikir birliğine varılmasıdır. (O'Neill, 1999)

Toplantı sona erdiğinde, moderatör hataları yazılımın programcısına bildirecek ve değişiklikleri takip etmekten sorumlu olacaktır. Moderatörün, kod üzerinde çok fazla değişiklik yapılması durumunda başka bir toplantıyı yeniden çağırmasına izin verilir . İncelemeye göre modülde bir tasarım değişikliği varsa bunu yapmanız önerilir.

Metodoloji olarak, bulunan hataların sonuçları ve türleri kod incelemesine çok benzer olacaktır; genellikle tek bir yöntem olarak kullanılırlar. Grup gerektiriyorsa, muhtemelen kullanılan karmaşık algoritmalar varsa, gerekli program segmentine izlenecek yollar uygulanabilir.

2.3 Dinamik Birim Testi

Yazılım süreçlerinin birçoğuna ve test süreçlerine göre, hem geliştirme süreci ile entegre hem de statik testlerden sonra bireysel, dinamik birim testi yapılmalıdır. Bir grup geliştirici, programcı veya test kullanıcısı tarafından incelenen yazılım modülü, bu incelemelere göre yapılan düzeltmelerden sonra dinamik test sürecine girecektir.

Dinamik test, yazılım modülünün yazılımla test edilmesidir. Test yazılımı, test taslakları ve sürücülerden oluşacaktır. Sürücüler, test altındaki yazılım bloğuna mesaj gönderir vebu bloktan gelen ac cept mesajlarını saplar ve yanıtları döndürür. Her modülün sürücüleri ve saplamaları olması gerekse de, pratikte, bir modül test edildikten sonra, genellikle diğer modüller için bir sürücü / saplama olarak kullanılır. Bu, birim ve iç grasyon testi arasındaki ayrımı gerçekten bulanıklaştırır ve önerilmez.

Birim testi, yazılım geliştirme sırasında gerçekleştirilen en düşük test seviyesidir; burada bireysel yazılım birimleri bir programın diğer bölümlerinden ayrı olarak test edilir ve dinamik testler, test sürecinde yeniden kullanılabilir anahtar unsurdur. Birim testleri, yazılım her değiştirildiğinde veya yeni bir ortamda kullanıldığında tekrarlanmalıdır. Tüm testler, yazılımın yaşam döngüsü boyunca sürdürülmelidir.

Yapılandırılmış ve tekrarlanabilir birim testini gerçekleştirmek için yazılımın geliştirilmesinden önce veya buna paralel olarak yürütülebilir un it-test yazılım programları geliştirilmelidir. Bu test programları, kodu gereksinimler tarafından belirtilen işlevselliği sağladığını ve tasarlandığı gibi kaldığını doğrulamak için gerekli şekillerde uygular. Birim testi programları, geliştirme projesinin bir parçası olarak kabul edilir ve proje geliştikçe gereksinimler ve kaynak koduyla birlikte güncelleştirilir. (Dustin, 2002)

Dinamik test bir yazılım testidirve programcılar tarafından sürdürülmelidir. "Bazı durumlarda, bir test sürücüsünün kodu, test edilen birimdeki koddan önemli ölçüde daha büyük olabilir. Bu, test sürücüsünün kendisinin birim testi sorununu ortaya koyuyor." (McGregor, 2001). Yazılım test cihazı,bir yazılım geliştiricisinin alabileceği bir roldür ve bu rolün test edilecek yazılım modülünün programcısına verilmemesi önerilir, ancak durum çok önemlidir.

Dinamik yazılım birimi testi için çok popüler, bilinen ve yaygın olarak kullanılan iki metodoloji vardır; kara kutu ve beyaz (cam) kutu testi. Bu metodolojiler ve hazırlanan test eserleri tekrar tekrar kullanılacağı için güvenleri önemlidir ve test eserlerini incelemek için temel test yöntemleri de tanıtılmaktadır.

Yazılımın dinamik testleri iki ana bölümden oluşmaktadır. Saplama, test edilen modülün ihtiyaç duyduğu uygulama programı arayüzlerini temsil eder ve simüle eder. Diğer modüllerden çağrılan fonksiyonlar veya yöntemler, nee ds testlerine göre burada simüleedilir. Çağrıldığını göstermek için işlevin adını yazdıran bir saplama olabilir veya örneğin yazılımın veritabanı bölümünü simüle eden bazı değerler veya veriler döndürebilir. Bu bölümün belirli bir donanım için bir sürücüyü simüle etmesi gerekebileceğinden, bazen simüle ettiği yazılım bloğundan bile daha büyük olabilir.

Sürücü, tüm test çalışmalarının veya vektörlerin depolandığı bölümdür. Bu durumlar temel olarak fonksiyona girilecek verileri hazırlayan vehazırlanan veriler ile gerekli function'ı çağıran fonksiyonlar olacaktır. Bazen test edilen işleve doğru verileri sağlamasını sağlamak için saplamayı denetleyebilir.


```
int method1 (int& a; int& b) {
    c = getCvalue(a, b);
     if(a <= 7)
          b++;
    diğer{
          if(c >= 0)
             a--;
          b--;
          theJobToDo();
    }
    if(a b < c || b) < b)
          dönüş b;
    dönüş 0;
}
```

Yukarıdaki fonksiyona sahip olan bir yazılım bloğunu kendi başına test edebilmek için, normalde başka bir modülden veya yazılım bloğundan çağrılan getCvalue yöntemi, saplama yazılımında simüle edilmelidir. Bu işlev, orijinalin özelliklerine bağlı olacak ve en azından testlerde kullanılan değerler için belirtilen değeri döndürecektir. theJobToDo, saplama yazılımında simüle edilecek diğer işlevdir. Sadece bu işlevin ne zaman çağrıldığını görmemiz gerektiğinden ve bu özel işlev için çıktısını umursamadığımızdan, yazılım testini yapabilmek için bazı bilgileri yazdırabilir veya bazı flag benzeri verileri saklayabiliriz otomatik olarak. Geliştirilmekte olan

yazılım, yazılım geliştirme sürecinde hazırlanan saplama ve sürücü ile birden çok kez test edileceğinden çıktıların analiz edilebilir hale getirilmesi esastır.

Verilen işlev için sürücü seviyesi yazılımı, farklı 'a' ve 'b' değerlerine sahip "method1" fonksiyon çağrılarından oluşacaktır.

2.3.1 Kara Kutu Testi

Bir modüldeki fonksiyonlar veya yöntemler,terface'leri aracılığıyla ve beklenen davranışlarına ve çıktılarına göre kara kutu testi olarak test edilir. "Kara kutu test tasarım teknikleri, uygulama detayları hakkında açık bilgi sahibi olmadan, sistemlerin işlevsel davranışına dayanmaktadır." (Broekman ve Notenboom, 200 3). Kara kutu testi bazen opak kutu testi, fonksiyonel test veya davranış testi olarak bilinir. Bu testler içyapıya değil, beklenen davranışa dayanır.

Siyah testi için, yazılımın kalitesine göre, olası tüm değerler, başka bir deyişle olası tüm işlev çağrıları yapılabilir. Ancak genellikle durum böyle değildir, fonksiyonlar genellikle olası değerlerin sınırlarına yakın değerlerle ve bazı iç değerlerle test edilir. Örneğin, test edilen fonksiyonun giriş parametresi olarak 8 bitlik imzasız bir tamsayısı varsa, kullanılacak test değerleri 0, 1, 127, 128, 254 ve 255 olabilir. Test cihazı, aşırı değerlerdeki ve kenar değerlerindeki hataları aramalıdır.

Sistemin sağlamlık testi için beklenmeyen değerler be hata kontrol mekanizmasının düzgün çalışıp çalışmadığını görmek için kullanılabilir. Kara kutu testi, iç yapıya bağlı olmayan, veri odaklı bir giriş / çıkış testidir, bu nedenle sınır değerleri, fonksiyonun iç sınırları hakkında endişelenmez. Bu tür bir test , yazılım bloğunun tüm yollarını test etmeyebilir, ancak bu yöntem, belirtimine bağlı olarak eksik yolları bulmada çok etkilidir.

Verilen örnek fonksiyon "method1" in test edilmesi için, tanımı, spesifikasyonu ve arayüzü kara kutu testi için endişe verici olacaktır. Belirtimin, 'a' ve 'b' her ikisi de 'n' ye eşitse, getCvalue'nun girişlerin toplamını döndürdüğü yerde işlevin 'n' değerini döndürmesi gerektiğini söylediğini varsayalım. Bunu test edebilmek için 'n' için 0, 1, 128, 254 ve 255 değerlerini kullanabiliriz. Böylece sahip olacağız;

Sonuç = yöntem1 (0,0);

Sonuç = yöntem1 (1,1);

İki vektör veya durum olarak. Sonuç değerlerini kontrol ederek, fonksiyonun 0 ve 1 değerleri için yapması gerekeni yapmadığını göreceğiz.

Tamsayılar için kullanılacak sınır değerleri minimum, 0 ve maksimum olabilir; booleanlar için doğru ve yanlış; null, blank ve maksimum uzunluktaki dizeler için. Kullanılacak değerler, işlevin ayrıntılı açıklamasına ve hizmet hedeflerinin kalitesine bağlıdır. Programcı ve test cihazı yakından çalışmalıdır.

2.3.2 Beyaz Kutu Testi

Kara kutu testinden farklı olarak, beyaz kutu testi, çıktıları incelemek için belirli programlama kodu bilgisini kullanır. Cam kutu, şeffaf kutu ve açık kutu testing olarak da bilinir ve yapısaldır. Tüm spesifikasyonun uygulandığını garanti edemez.

Bu tür testler, yazılımın iç yapısıyla ilgili hataları bulmaya çalışır. Yazılımın tüm olası kalıpları, seçilen değerlerle tanımı yoluyla işlevlere ulaşan oluşturulan test durumları tarafından kapsanmalıdır. İşlevin içindeki tüm kalıplara arayüzü üzerinden ulaşabilmek için, fonksiyon içindeki tüm kararlara interface'inden ulaşılabilir olmalıdır. Bu nedenle Gözlemlenebilirlik kilit konudur. İşlev gözlemlenebilir değilse, yeniden uygulanmalıdır.

Eksik yollar beyaz kutu testi ile belirlenemeyebilir ve fonksiyonun cevabı olmayan bazı değerler varsa bunlar beyaz tarafından caught olmayabilir.

Kutu test metodolojileri. Sistem düzeyi için beyaz kutu testi olan bir test çalışması, alt sistem düzeyi için bir kara kutu test çalışması olabilir.

Bir grup beyaz kutu test tekniği vardır. En popüler ve yaygın olarak kullanılanlar yol kapsamı ve ifade kapsamıdır. En azından incelenen fonksiyonun ekstre kapsamı tamamlanmalıdır; bu aynı zamanda sistemin 'Gözlemlenebilirliği' için bir kanıt olacaktır.

Yol kapsamı değerlendirilecek nesnede gerekli bir minimum yol sayısını yürütüyor. Araştırdığımız örnek işlev için beyaz kutu test kümesi, işlevin tüm yollarından geçen vaka grubu olabilir.

Örneğin, "if(a <= 7)" kararı ve "diğer" yolu için iki durum tanımlanabilir;

yöntem1(6, X);

yöntem1(7, X);

Bu iki çağrı, ikinci parametrenin değerine bağlı kalmadan, sistemi iki farklı yoldan geçirir.

Benzer şekilde if (c >= 0) ve if(a < b || c için de durumlar < b) uygulanmalıdır. Belirtim dikkate alınarak, yürütülecek yollar ve ilgili test çalışmaları tanımlanabilir.

Deyim kapsamı tekniği, değerlendirilecek nesnede gerekli minimum sayıda ifadeyi yürüten test durumlarını tanımlar. "Gerekli kapsama derecesine bağlı olarak, test çalışması tanımı için ya tüm ya da yalnızca belirli sayıda ifade kullanılmalıdır." (Myers, 1991) Herhangi bir test çalışması, işlevin ilk satırındaki "c = get C değeri(a, b);" ifadesi kapsayacaktır. Test durumlarını yol kapsama tekniğine göre seçerek fonksiyonun her ifadesini yürütmek iyi bir test yaklaşımıdır. İfadelerle ilgili kapsam raporları derleyicilerden alınabilir veya bazı yazılım araçları kullanılabilir. Hizmet kalitesine ve işlevin önemine bağlı olarak, yazılımın piyasaya sürülmesi için kapsama yüzdesi hedeflenebilir.

Yazılımdaki karar hem doğru hem de yanlış değerlerle yürütülmelidir. Farklı olası değerlerin test edilmesi önerilir. Örneğin örnek olarak aldığımız fonksiyon için "if(a < b || c < b)" aşağıdaki doğruluk tablosuna sahiptir;

a < b	c < b	a b < c || b < b
Doğru	Doğru	Doğru
Doğru	Yanlış	Doğru
Yanlış	Doğru	Doğru
Yanlış	Yanlış	Yanlış

'A', 'b' ve 'c' değerlerine bağlı olarak dört koşulu da test edecek test durumlarına sahip olmak akıllıca olacaktır.

Döngü kapsamı, beyaz kutu testinin bir başka önemli eseridir. Bu ölçü, her döngü gövdesini sıfır kez, tam olarak bir kez ve birden fazla kez (ardışık olarak) yürütüp yürütmediğinizi bildirir. Devam eden döngüler için döngü kapsamı, gövdeyi tam olarak bir kez ve birden fazla kez çalıştırıp çalıştırmadığınızı bildirir. Bazı algoritmaların test edilmesi için, bazı kritik değerler ve döngünün içindeki olası dallar için döngülerin birden fazla kez çalıştırılması gerekebilir.

Beyaz kutu testi tüm yolları kapsayabilir, her ifadeyi en az bir kez yürütebilir, her kararı doğru ve yanlış ile test edebilir ve her karardaki tüm olası koşul kombinasyonlarını kontrol edebilir. Yol kapsama tekniği ve ifade kapsamı artefaktının beyaz kutu test durumlarının baskın endişeleri olması önerilir.

Test çalışmaları, beyaz ve siyah testler için hazırlanan vektörler, test edilecek modül için test yazılımının sürücü kısmını oluşturacaktır. Bu testlerin otomatikleştirilmesi, geliştirmenin sonraki aşamaları için yapılan çalışmaları en aza

indirecek ve bunları entegrasyon ve regresyon testleri gibi gelecekteki testler için yeniden kullanmamıza izin verecektir.

2.3.3 Hata Tohumlama ve Mutasyon

Yazılım projelerinin boyutları büyüyor ve hazırlanan test yazılımlarının boyutları da büyüyor. Genellikle hazırlanacak çok sayıda sürüm ve yapılacak regresyon testleri olduğundan, test paketine duyulan güven yüksek düzeyde olmalıdır. Test kaynağını test yazılımı ile test etmek bizi bir kısır döngüye sokacaktır. Çok fazla çaba sarf etmeden süite olan güveni yüksek seviyede tutmak için hata tohumlama ve mutasyon teknikleri tanımlanmıştır.

Siyah beyaz kutu test sürücülerinin ve saplamalarının bir bileşimi olan test paketinin yazılımdaki hataları yakalaması beklenir, bu nedenle zaman zaman yazılıma bazı hataların eklenmesi önerilir. "Hata tespiti ile ilgili çalışmalar yapmak, hataları otomatik ve sistematik olarak bir programa ekleyen hata tohumlama tekniklerinin eksikliği nedeniyle zordur. Bir hata tohumlama yöntemi, mutasyon testi, bir programa küçük sözdizimsel değişiklikler veya hatalar ekler. " (Harrold, Offutt ve Tewary, 1997)

Hatalar sisteme dahil edilmelidir. Bunlar aritmatik bir işlemin değiştirilmesiyle yapılabilir. Örneğin, 'yöntem1'in üçüncü satırını 'b++;' yerine 'b--;' olarak değiştirmek. İfade kaldırma, yazılıma bir hata eklemek için başka bir tekniktir. 'theJobToDo();' olan sof tware'in 8. satırını kaldırmak, yazılıma başka bir hata getirecektir. Bu tür modifikasyonlara mutasyon denir ve yazılım mutant olarak adlandırılır.

Hazırlanan test paketi, bir dizi değişiklikten sonra çalıştırılmalı, arızalar sisteme tanıtılmalıdır. Ana endişe, arıza tespit oranıdır ve bu, test setinin kalitesi hakkında bir fikir verir.

Bir sonraki adım, mutant yazılımdaki yakalanmamış hatalar için yeni test durumları uygulamak olmalıdır. Bu yeni durumlar test paketine dâhil edilmeli ve genişletilmiş paket orijinal yazılım üzerinde çalıştırılmalıdır. Yeni test paketi ile daha önce yakalanmamış bazı hataları yakalamak olasıdır.

3.ÜST SEVİYE TESTİ

3.1 Giriş

"Bir kişinin tek başına eksiksiz bir sistem geliştirebileceği zaman geçti." (Broekman & Notenboom, 2003). İhtiyaç duyulan yazılım sistemleri gittikçe büyüyor ve çok sayıda geliştirici tarafından geliştiriliyor. Bazen bu geliştiricilerdaha küçük ekiplere bölünür ve ayrı çalışırlar. Günümüzde bazı yazılım ürünleri için farklı kıtalarda bulunan ekipler birlikte çalışmaktadır.

Rekabetçi pazarın bir sonucu olarak, daha kaliteli ürünlere olan talep artmaktadır. Yazılım, güvenlik açısından kritik durumlarda ve tıbbi cihazlarda kullanılmaya başlanmıştır. Özellikle içinde astronotların bulunduğu bir uzay mekiği gibi hayati önem taşıyan durumlar ya da Mars'a gönderilen pathfinder gibi yüksek kıyı uygulamaları düşünülmeli ve gerekli bir test montajı yapılmalıdır.

Günümüzde yazılım boyutları çok büyüktür ve pazara sunma süresi birinci önceliktir, bu nedenle yazılımın yeniden kullanılabilirliği ve diğer yazılım ekipleri tarafından geliştirilen modüllerin kullanılabilirliği, birbirini izleyen yazılım ürünleri için kilit konular haline gelmektedir. Açık kaynaklı yazılım modülleri bu büyük sistemlerin parçaları olarak kullanılmaya başlanıyor.

Elbette, tam yol kapsamı tüm hataları tespit edecektir, ancak bu mümkün olmayabilir ve mümkün olduğunda bile, bunu yapmak sonsuz zaman alabilir. "Birim ve entegrasyon testleri doğası gereği sonludur, ancak tamamen yürütülse bile tüm kusurları tespit edemez." (Broekman & Notenboom, 2003). Önemli kusurları tespit etmek, test gruplarının ve yazılım geliştiricilerin odak noktası haline geliyor.

80/20 Kuralı herhangi bir şeyde birkaçının (yüzde 20) hayati önem taşıdığını ve birçoğunun (yüzde 80) önemsiz olduğunu belirtir. Pareto'nun durumunda bu, halkın yüzde 20'sinin servetin yüzde 80'ine sahip olduğu anlamına geliyordu. Juran'ın ilk çalışmasında kusurların yüzde 20'sini tespit etti.

Sorunların yüzde 80'ine neden oluyor. Proje Yöneticileri, işin yüzde 20'sinin (ilk yüzde 10 ve son yüzde 10) zamanınızın ve kaynaklarınızın yüzde 80'ini tükettiğini bilir. 80/20 Kuralını yönetim biliminden fiziksel dünyaya kadar hemen hemen her şeye uygulayabilirsiniz. (Reh, 2005)

Pareto Kuralı'nın yazılım testine uygulanması, yani kusurların yüzde 20'sinin sorunların yüzde 80'ine neden olması, ürün için test stratejisinin belirlenmesi gerekir. Daha önce de belirttiğimiz gibi, erken tespit edilen kusurların düzeltilmesi kolaydır,

çünkü geliştirme sürecinde zaman geçtikçe hataları düzeltmenin maliyeti artar. Bu, erken ve kapsamlı testlere duyulan ihtiyacı açıklar. Projenin sonunda ürüne kalite katmak kolay değildir ; en başından itibaren inşa etmelisiniz.

Birim ve entegrasyon testleri, geliştiricilerin kendileri tarafından yapılmalıdır ve genellikle yapılır. Birim testleri muhtemelen modülün geliştiricisi tarafından ele alınacaktır, sistemin geri kalanıyla entegre edilmesi gerektiğinde bir entegrasyon sistemi seçilmelidir.

3.2 Entegrasyon Testi

Entegrasyon, test edilen birimlerden daha büyük varlıklar oluşturma işlemidir. Entegrasyon testi, entegre işletmeyi oluşturan birimleri yeniden test eder ve işletmenin tasarım gereksinimlerinin karşılandığından emin olmak için birimler arasındaki iletişimi ve etkileşimi daha fazla test eder. Entegrasyon testi, bileşen işlevselliğinin ve arayüz uyumluluğunun doğrulanmasına odaklanır. Bununla birlikte, hataların işlenmesini ve hatalardan kurtarılmasını doğrular (Sistem Geliştirme Merkezi, 2002).

Birim düzeyinde test sırasında, bazı hataları gözden kaçırmak mümkündür. Bu tür hataları iki kategoriye ayırmak mümkündür; programla ilgili olan bileşenler arası hatalar ve programlama ile ilgili olmayan birlikte çalışabilirlik hataları. Tabii ki, birim testi sırasında yakalanmayan hatalar olacaktır, ancak olması gerekirdi.

Bileşenler arası arızalar, birden fazla modül, bileşen ve iş parçacığı ile ilgili arızalar olarak kabul edilir. "Birden fazla bileşenle ilişkili programlamayla ilgili hatalar, bileşenler arası hatalar olarak kabul edilir" (Gao, Tsao & Wu, 2003). Kilitlenmeler,bu tür hataların tipik sınavlarıdır. Örneğin, bir sınıfın üye değişkeni ortak bir işlevle sıfırlanabilirse ve aynı değişkenin değeri başka bir işlevle döndürülebilirse. Başka bir modül bu yöntemleri kullanırsa vesıfır olacak şekilde döndürülen v alue'yu bölücü olarak kullanırsa, sorunlara neden olur. Birim testi ile bu tür bir hatayı yakalamak mümkün olmayacaktır.

Birlikte çalışabilirlik hataları aslında üç türe ayrılır; sistem seviyesi, programlama seviyesi ve spesifikasyon seviyesi. Öncelikle farklı makinelerde, altyapılarda farklı modüller derlenebilmekte, dolayısıyla uyumlu olmayabilmektedir ve buna sistem düzeyinde birlikte çalışabilirlik hatası denir. Örneğin, modüllerin bayt hizalaması aynı olmayabilir.

Programseviyesi birlikte çalışabilirlik hataları, modüllerin C ve C ++ gibi farklı dillerde uygulanabileceği ve diller arasındaki uyumsuzluğun sorunlara neden

olabileceği anlamına gelir. Örneğin, kayan nokta işlemleri ve değişkenleri bu durumlarda özellikle tehlikelidir.

Spesifikasyon seviyesi birlikte çalışabilirlik hataları temel olarak spesifikasyonların yanlış yorumlanmasından kaynaklanır. Yapılacak en iyi şey, geliştiricileri geliştirme sürecinde birbirlerinin ne yaptıkları hakkında bilgilendirmektir, ancak bu mümkün olmayabilir. Genellikle yanlış yorumlamalardan kaynaklanan hatalar, bileşen etkileşimlerinin veya arabirimden geçen verilerin kalıplarıdır.

Entegrasyon, tüm yazılım ürününü üretmek için art arda birkaç daha büyük oluşturma ve test aşamasını içerebilir. Entegrasyon testi, yazılım varlıkları tasarım spesifikasyonlarına ve test edilen işletmeye tahsis edilen işlevsel gereksinimlere uygun olarak tek bir sistem olarak çalıştığında tamamlanır. (Sistem Geliştirme Merkezi, 2002).

Entegrasyon testi sırasında yardımcı olabilecek odak noktaları ve tekniklerin yanı sıra, bir stratejiye ihtiyaç vardır.

Entegrasyon testinin bir grup bileşeni bir araya getirmesi gerekir. Bu süreci yeterince yürütmek içiniki soruyu cevaplamamız gerekiyor: bileşenleri kademeli olarak entegre etmek için takip ettiğimiz siparişler nelerdir ve yeni entegre edilen bileşenleri nasıl test ederiz? (Gao, Tsao & Wu, 2003).

Entegrasyon stratejisi, farklı modüllerin eksiksiz bir sisteme nasıl entegreedildiğine dair bir karardır. Entegrasyon donanım ve yazılımı kapsar. Farklı yazılım modülleri, farklı donanım parçaları ve yazılım ile donanım arasındaki tüm bağımlılıklar nedeniyle hangi stratejinin kullanılacağına karar vermek önemlidir. (Br oekman & Notenboom, 2003).

Kullanılan stratejinin projenin zamanlaması üzerinde bir etkisi olduğundan, çok erken aşamalarda karar verilmelidir. İzlenecek temel olarak üç strateji vardır; Büyük patlama, aşağıdan yukarıya ve yukarıdan aşağıya, ancak entegre edilecek bileşenlerin mevcudiyetine ve sistemin mimarisine ve boyutuna bağlı olarak, bu üçünün birleşimiyle çeşitli stratejiler ortaya çıkar.

3.2.1 Big Bang Entegrasyonu

Bu metodoloji, yazılımın tüm modüllerini bir kerede entegre eder. Çok basittir, tüm modüller entegre edilmiştir ve sistem bir kez test edilir. Bunun en büyük avantajı, taslakların veya sürücülerin kullanılması gerekmeyecek olmasıdır. Ve dezavantajları; entegrasyon ancak tüm modüller mevcutsa başlayabilir ve bir arıza varsa, buna kimin sebep olduğunu anlamak çok zordur. Tüm bileşenler hataları barındırma olasılığına tabidir, bu nedenle arıza tespiti maliyetlidir.

Bu strateji yalnızca aşağıdaki durumlarda başarılı olabilir:

- Sistemin büyük bir kısmı kararlıdır ve sadece birkaç yeni modül eklenir;
- Sistem oldukça küçüktür;
- Modüller sıkıca bağlanmıştır ve farklı modülleri kademeli olarak entegre etmek çok zordur. (Broekman & Notenboom, 2003)

Sistem yukarıdaki koşullardan birine uymuyorsa, aşağıdaki artımlı stratejilerden birinin kullanılması şiddetle tavsiye edilir.

3.2.2 Aşağıdan Yukarıya Entegrasyon

Aşağıdan yukarıya strateji, tabiri caizse, sistemin işlevsel yapraklarından başlar ve en üst seviyeye (muhtemelen kullanıcı arayüzü) gider. Bu hemen hemen her sistem için uygun bir stratejidir.

Düşük seviyeli modüller (donanıma en yakın) muhtemelen en az sayıda bağımlılığa sahip olanlardır. Bu nedenle, bu modüller, bu modülleri test etmek için sürücüleri kullanarak entegrasyon için başlatılacak yerlerdir. Yani bir alt sistem paralel olarak birikiyor olabilir ve tüm sisteme entegre edilebilir. Proje stratejisi modülleri aşağıdan yukarıya doğru bir şekilde oluşturmaksa, entegrasyon sürecin başlarında başlayabilir.

Bu stratejinin avantajı,eğer yazılım süreci ile de entegre edilmişse, hataların, arızaların ve arayüz sorunlarının proje geliştirme sürecinin erken aşamalarında tespit edilecek olması ve büyük olasılıkla en geç entegre edilen modülden kaynaklandığı için arızanın tespit edilmesinin kolay olmasıdır . Başlıca dezavantajları; Bu stratejiyi uygularken birçok sürücü kullanılacaktır (bunun için gereken çabayı hesaplamanız ve dikkate almanız tavsiye edilmekle birlikte); ve testlerin yinelemesi zaman alıcı olacaktır.

3.2.3 Yukarıdan Aşağıya Entegrasyon

Yukarıdan aşağıya strateji, deyim yerindeyse, fonksiyonel ayrışma ağacının en üst seviyesinden başlar ve sistemin yapraklarına gider. Sistemin kontrol yapısı bu stratejide başı çeker.

Kontrol yapısının yukarıdan aşağıya bir şekilde inşa edildiği d zarflama süreçleri, yukarıdan aşağıya entegrasyon, entegrasyona girme ve böylece problemlerin gelişimin erken bir seviyesinde bulunma imkanı verecektir. Test, sistemin kontrol mekanizmasında başlarve her yeni seviyede bağlı modüller entegre edilir.

Bu stratejinin temel avantajı, sistemin ana parçaları henüz uygulanmamasına rağmen, tüm sistem hakkında erken bir fikre sahip olmanın mümkün olmasıdır. Başlıca dezavantajları; düşük seviyeli modüllerde değişikliklere neden olabilecek gereksinimler üzerinde yapılan değişiklikler, üst düzey modüllerde değişikliklere neden olabilir ve bu da entegrasyonun yeniden başlatılmasına neden olabilir. Diğer bir dezavantaj, bileşenlerin kısmi bir grubunu test ederken henüz uygulanmayan her bileşen için saplamaların ayrı ayrı geliştirilmesi gerektiğidir.

3.2.4 Merkezi Entegrasyon

Bu stratejinin güçlü noktası, stratejilerin birleşimidir, bu nedenle sistemin bir parçası için en etkili olanı seçme yeteneğidir. Bu tür bir entegrasyon, sistemin merkezi kısmı sistemin geri kalanından daha önemli olduğunda veya sistemin geri kalanının bir işletim sisteminin çekirdeği gibi düzgün çalışması gerektiğinde özellikle yararlıdır.

Orta kısım testleri çalıştırmak için çok gerekliyse ve bir saplama ile değiştirmek çok zorsa, yukarıdan aşağıya ve aşağıdan yukarıya entegrasyon metodolojileri de çok yardımcı olmayacaktır. Benzer şekilde sistemin merkezi kısmı geliştirilip üretime hazır hale getirilirse ve piyasaya sürüldükten sonra yeni modüller veya alt sistemler gibi yükseltmeler hazırlanırsa bu yöntem de avantajlıdır.

Merkezi entegrasyon ilk önce merkezi parçayı test eder ve bir sonraki adım kontrol yapısının entegrasyonudur. Alt sistemlerin yukarıdan aşağıya veya aşağıdan yukarıya stratejiye göre paralel olarak test edilebilmesinin başlıca avantajıdır (veya hatta alt sistem yeterince büyükse merkezi entegrasyon yaklaşımı). Ana dezavantaj, test edilen merkezin ve test edilen alt sistemlerin entegrasyonudur. Bazen orta kısmın modülleri çok sıkı bir şekilde birleştirileceğinden, son adım olarak büyük patlama metodolojisine ihtiyaç duyulabilir.

3.2.5 Katman Entegrasyonu

Bu entegrasyon metodolojisi,daha önce de belirttiğimiz gibi ana üç strateji olan yukarıdan aşağıya, aşağıdan yukarıya ve büyük patlama entegrasyonları için özel bir koşul olarak olabilir.

Bu strateji, arayüzlemenin yalnızca doğrudanaltındaki veya doğrudan üstündeki katmanlar arasında gerçekleştiği katmanlı mimari sistemler için kullanışlıdır. Sistemin her katmanı yalıtılmış olarak test edilmeli, entegrasyonun üç ana stratejisi uygulanabilir ve entegrasyon sistemin keyfi bir katmanında başlayabilir.

Katman bağımsız olarak test edildikten sonra, bir sonraki adım onu yukarıda veya aşağıda test edilen katmanla entegre etmek olmalıdır. Orada ana stratejilerin tüm avantajları ve dezavantajları, kullanıldıkları sürece bu metodolojiye taşınır, ancak arayüzlerin izolasyonu ve entegrasyonu katmanlı bir mimaride çok açık ve kolay olduğundan, kusurların nedenlerini keşfetmek, tanımlamak, problemler geleneksel üç metodolojiye kıyasla çok daha kolaydır.

3.2.6 İstemci/Sunucu Entegrasyonu

Katman entegrasyonuna benzer şekilde, istemci/sunucu entegrasyonu sistemin mimarisine bağlı olarak özel bir durumdur. Sunucuyu bir saplama ve bir sürücü ile değiştirmeniz ve istemciyi üç ana metodolojiden birini kullanarak entegre etmeniz önerilir; yukarıdan aşağıya, aşağıdan yukarıya ve büyük patlama veya sunucu aynı yaklaşım izlenmeli ve testlerden sonra sunucu ve istemci entegre edilmelidir. Katman entegrasyonuna benzer şekilde, avantajlar ve ana stratejilerin dezavantajları kullanıldıkları sürece bu zincire uygulanır.

3.2.7 İşbirliği Entegrasyonu

Bu entegrasyon stratejisi, yalnızca nesne yönelimli sistemler için uygulanabilecek büyük patlama entegrasyonunun çok özel bir durumudur. UML'de açıklandığı gibi, birleşik süreç ve diğer birçok prosedür, bir kullanım örnekleri sisteminin gerçekleştirilmesi için birçok işbirliğini destekleyebilir.

Bir kullanım örneği modeli, bir sistemin gerekli yeteneklerini ve davranışını tanımlamaktır ve 1992 yılında nesne yönelimli yazılım geliştirme ile kullanılmak üzere Ivar Jacobson tarafından tanıtılmıştır. Kullanıcı perspektifinden bir sistemin yeteneğini veya işlevselliğini temsil eder.

İşbirliği diyagramları, bir kullanım örneğindeki nesnelerin arabirimlerine odaklanır ve bir sistem, birçok kullanım örneğini gerçekleştireceği için birçok işbirliğini destekler. Bu gerçekleşmelerden sonra birçok nesnenin birden fazla işbirliğine ait olacağı aşikar olacaktır. Dolayısıyla, akıllıca bir işbirliği seçimi ile tüm sistem kapsanabilir. Bu metodolojiyi kullanabilmek için tüm işbirlikleri açıkça tanımlanmalı ve tüm bileşenleri ve ara yüzleri kapsamalıdır.

Tüm işbirlikleri tamamlandığında, entegrasyon, büyük patlama yaklaşımında olduğu gibi bir işbirliğinin tüm bileşenlerini bir araya getirerek başlayabilir. Bu yaklaşımın temel avantajı odak noktası uçtan uca işlevsellikte olduğundan, yalnızca

birkaç teste ihtiyaç duyulmasıdır. Bileşenlerin ve işbirliklerinin örtüşmesi nedeniyle, büyük olasılıkla her bir vakayı test etmek gerekli değildir. Testler bir grup bileşen üzerinde yapıldığından, saplamalara ve sürücülere nadiren ihtiyaç duyulacaktır. Ana dezavantaj, farklı işbirlikleri arasındaki ince bağımlılıkların işbirliği modelinin dışında bırakılabilmesidir. Elbette bu yaklaşımın en büyük dezavantajlarından biri, nesne yönelimli bir de işaret olmadıkça ve varsa kullanılmasının mümkün olmamasıdır çok fazla kullanım durumu varsa, sistemi kullanım durumlarına göre uygulamak çok akıllıca olmayacaktır.

3.2.8 Sandviç Entegrasyonu

Belirttiğimiz gibi, yukarıdan aşağıya yaklaşım entegrasyon sürecinin erken bir aşamasında mimari ve yüksek h seviyesi kusurların tespit edilmesine yardımcı olabilir, ancak bazen saplamaları uygulamak çok maliyetli olabilir. Aksine, aşağıdan yukarıya yaklaşım, entegrasyon process'in erken bir aşamasında yüksek seviyeli ve mimari kusurları tespit etmek olmayacaktır, ancak sürücüleri uygulamak, saplamaları uygulamaktan daha kolaydır.

Sandviç yaklaşımı, bu yaklaşımların her ikisinin de dezavantajlarını aşmaya çalışır ve her ikisinin de avantajlarını en üst düzeye çıkarmaya çalışır. Bu nedenle, özellikle sürücüleri ve saplamaları uygulamak için gereken çabaya odaklanarak ve mimari ve daha üst düzey kusurları erken bir aşamada yakalamaya çalışarak, bu yaklaşımların her ikisinin de aynı anda kullanılması ve daha sonraki entegrasyon seviyelerine entegre edilmesi önerilir. Geliştirme süreci bu metodolojide başarılı olabilmek için entegrasyon yaklaşımına odaklanmalıdır.

3.3 Sistem ve Kabul Testleri

Farklı olmalarına rağmen, sistem ve kabul testleri genellikle birlikte ele alınır ve entegrasyon tamamlandıktan sonra uygulanır. Bu testler için gerekli olan ana doküman, sistemin gereklilikleri, özellikleri olacaktır. Daha önce de belirtildiği gibi, bir hata, sistem gereksinimlerine uymayan sistemin çalışmasıdır. Bu nedenle, sistemin belirli bir durumda çökmesi veya yeniden başlatılması bekleniyorsa , kararlı olduğunu belirtirse ve yapılması daha mantıklı bir şey olsa bile durumu beklenmedik bir şekilde ele alırsa bir hatadır.

Sistem kabul testi, nihai ürünün müşterinin onaylanmış ve onaylanmış gereksinimlerine karşı sistem düzeyinde fonksiyonel bir denetimidir. Sistem kabul testi,

kullanıcının bakış açısından aşağıdaki gibi bir ortamda gerçekleştirilir neredeyse mümkün olduğunca kullanıcının ortamı gibi. (Sistem Geliştirme Merkezi, 2002).

Özellikle veri güdümlü sistemler için, test komut dosyaları kabul ve sistem testi sırasında yürütülmesi çok yararlı olabilir.

Kabul ve sistem testleri bir test planına göre yönlendirilmelidir. Kullanılacak test plan ve prosedürleri, geliştirme sürecinin çok erken aşamalarında hazırlanmalı, gerektiğinde modüllere ve sisteme paralel olarak veri ve otomasyon hazırlanmalıdır. Sistemin güvence kalitesi, hangi testlerin yürütüleceğini ve ne kadar süreyle yürütüleceğini belirlemede kilit konudur. "Bir projenin çok sayıda gereksinimi olduğunda, test prosedürleri geliştirmeye nereden başlayacağınıza karar vermek çok zor görünebilir." (Dustin, 2002) Devide ve conquer yaklaşımı, testleri uygulamak için iyi bir yoldur.

Test görevleri sorulara, neyin test edileceğine, test verilerinin ve komut dosyalarının ne zaman geliştirilmesi gerektiğine, sistemden bağımsız testin hangi bölümlerinin tanımlanabileceğine ve son olarak bu testleri kimin tasarlayıp uygulayabileceğine göre değişebilir. Sıkı programlar ve maliyet sınırlamaları nedeniyle , test çabasını kimin yürüteceği tüm test döngüsünde çözülmesi en zor sorun olabilir. Genellikle test grupları, şirketler için ekstra ve çok yararlı olmayan bir maliyet gibi görünüyor ve geliştiriciler yazılım testinin önemini ve iyi test sistemleri tasarlamanın ne kadar zor olduğunu henüz anlamamışlardı. Test geliştiricileri atamanız ve yalnızca teste odaklanmalarına izin vermeniz önemle tavsiye edilir.

Sistemi çalıştırabilmek ve kabul testlerini yapabilmek için farklı teknikler uygulanabilir. Musa'nın sistem geliştirme merkezi, sistem ve kabul testleri için aşağıdaki teknikleri önermektedir.

Denetim: Bir gereksinimin karşılanmasının, donanımın, kaynak kodun ve / veya yazılımın diğer fiziksel tezahürlerinin incelenmesiyle doğrulandığı bir tekniktir.

Test: Bir gereksinimin karşılanmasının, yazılımı çalıştırarak ve ardından uyarana verilen yanıtı kaydedip analiz ederek doğrulandığı bir tekniktir.

Gösterim: Bir gereksinimin karşılanmasının, yazılımın performansını gözlemleyerek doğrulandığı bir tekniktir.

Analiz: Bir gereksinimin karşılanmasının, yazılımın ve bileşenlerinin sonsuz yapısının incelenmesine ve analizine dayanan çıkarımlarla doğrulandığı bir tekniktir. Bu, gereksinimler doğrudan test edilemediğinde ve gözlemlenemediğinde gerekli olabilir. Doğrulama açısından bakıldığında, error için daha büyük bir potansiyel nedeniyle analiz gerektiren gereksinimleri en aza indirmek en iyisidir.

Simülasyon: Bir gereksinimin karşılanmasının, eksik bir bileşenin temsili kullanılarak doğrulandığı bir teknik (örneğin, bir donanım bileşenini temsil etmek için özel olarak geliştirilmiş bir kod modülü). Bu, bir requirement doğrudan test edilemediğinde ve gözlemlenemediğinde gerekli olabilir. Doğrulama açısından bakıldığında, daha büyük bir hata potansiyeli nedeniyle simülasyon gerektiren gereksinimleri en aza indirmek en iyisidir. (Sistem Geliştirme Merkezi, 2002)

Bu bölümün başında da belirtildiği gibi sistem ve entegrasyon testleri birbirinden farklıdır. Sistem testi, yazılımı doğrulamak içindir ve yazılım gereksinim belirtimleriyle karşılaştırılır. Geliştirme ekibi tarafından yapılır. Kabul testi, yazılımın doğrulanmasıdır ve yazılım son kullanıcı gereksinimleri ile karşılaştırılır. Müşteri tarafından yapılır, çoğu durumda müşteri, program lideri, proje lideri veya yönetici gibi şirketten biridir.

Uygulamada, doğrulama ve doğrulama terimleri, test etmeye ve ürünün gerekli şeyi yapacağından emin olmaya yol açan tüm faaliyetler için kullanılır. Ancak temelde farklı amaçlara hizmet ederler. Doğrulama, "sistemi doğru inşa ediyor muyuz?" sorusunun cevabıdır, eğer sürecin her adımına odaklanıyorsa,

Doğrulama, "doğru sistemi kuruyor muyuz?" sorusunun cevabı, ürünün gereksinimleri karşılayıp karşılamayacağına odaklanıyor, başka bir deyişle müşterimizin istediği bu mu?

Böylece sistem testleri, ayrıntılı gereksinimlere, sınır koşullarına, ürünün sınırlarına ve performansa daha fazla odaklanacaktır. Yazılımı onaylayabilmek için, sistemin performansıyla ilgili testlerdikkate alınmalıdır. Yük ve stres testleri en çok bilinen ve kullanılan testlerdir.

3.3.1 Yük ve Stres Testi

Bazen performans veya güvenilirlik testi olarak adlandırılan yük ve stres testi, "sistemin ihtiyaç duyduğu kaynakları tüketmeye yaklaşan koşullar altında bir sistemi çalıştırmak" tır (McGregor & Sykes, 2001). Bazen yük ve stres testleri, uygulamanın yük testi olarak büyük bir görevi nasıl yerine getireceği ve uygulamanın en yüksek etkinlik koşulları altında ne yapacağı gibi farklı şekilde kabul edilir. Bu teknikleri tek bir olarak tanıtmaya çalışacağız.

Bahsedildiği gibi yük ve stres testleri zor şartlar altında performansı görmeye çalışacaktır. Bu nedenle, RAM'i nesnelerle doldurmak, sabit sürücüyü doldurmak, fareyi hızla hareket ettirmek, büyük bir metin ve boş bir metin kullanan bir metin girişi

varsa, sistem aynı anda on isteği işlemek üzere tasarlanmışsa, on ve on bir istek göndermek iyi örnekler olabilir. Gerçekleştirilecek test, sistemlerin işlevselliğine, giriş ve çıkış mekanizmalarına ve istenen working koşullarına bağlı olmalıdır. Limitte ve limitin üzerinde koşullar test edilmelidir.

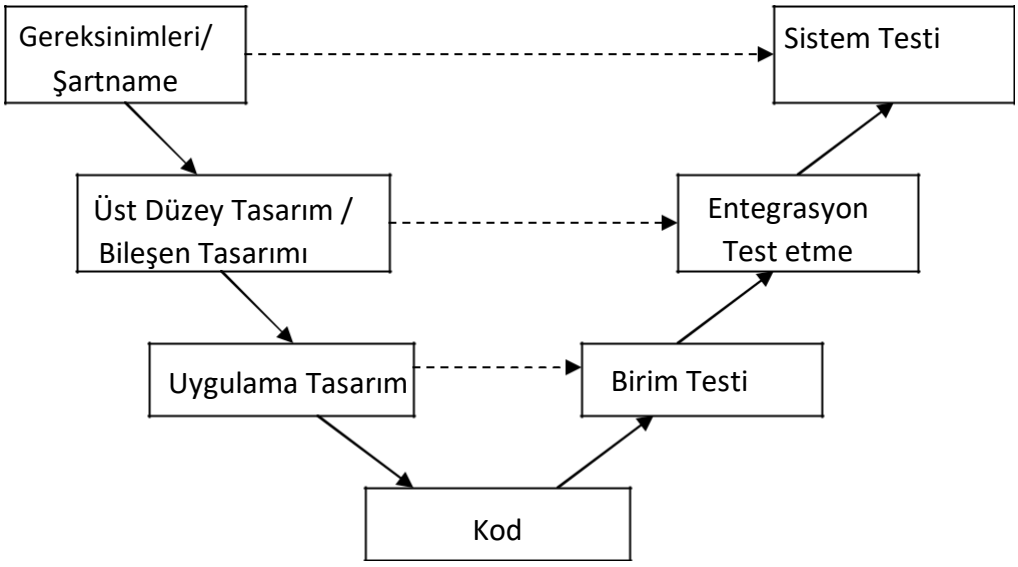
Nesne yönelimli sistemler genellikle çok sayıda sınıf örneği oluşturularak vurgulanır. Normal çalışmada gerçekten çok sayıda insansa sahip olması muhtemel sınıfları seçin. Parametrelerin değerlerini değiştirmek için rastgele sayı üreteçleri veya diğer cihazları kullanın, çünkü bu, oluşturucuları çeşitli sınıflarda test etmek için iyi bir fırsattır.

Nesneler genellikle düşündüğünüzden daha büyüktür. 30 saniyelik tam hareketli video klibe başvuru içeren bir nesnenin nesne başvurusu vardır (genellikle 4 bayt), ancak bu nesnenin örneğini oluşturmak için gereken toplam bellek, video klibin bir bölümünü tutmak için gereken belleği içerir. Object-oriented sistemler genellikle normal çalışmada belleği strese sokar, çünkü geliştiriciler nesnelerin gerçek boyutlarına yeterince dikkat etmezler. Geliştirme yaşam döngüsü boyunca test, birim testleri sırasında az sayıda nesnenin kullanılmasıyla, tümleştirme ve sistem testleri sırasında n ormal operasyonel sayıda nesneyle ve daha sonra sistem operasyonel sınırlar altında kararlı hale geldiğinde bir sistem testinde olağanüstü sayıda nesneyle başlar. (McGregor & Sykes, 2001)

Yük ve stres testi, özellikle istemci/sunucu tarzında çalışan veri tabanı ve internet uygulamaları nedeniyle yazılım testinin en iyi anlaşılan ve kabul edilen kısmıdır. Aşırı koşulları test etmek için bir grup test yazılımı ürünü mevcuttur ve bu tür testlerin dış kaynaklı olma olasılığı yüksektir.

3.4 V-Modeli

Yazılım geliştirme yaşam döngüsünde hem geliştirme faaliyeti hem de test faaliyetleri neredeyse aynı anda ellerinde aynı bilgilerle başlar. Geliştirme ekibi hedeflere ulaşmak için "do-procedures" uygulayacak ve test ekibi bunu doğrulamak için "kontrol prosedürleri" uygulayacaktır, ancak bu ekipler büyük olasılıkla aynı kişilerden oluşacaktır. Yedilerin şelale yazılım süreç modeli, yaşam döngüsünün sonunda test etmek, ortak olan büyük sistemler için çok maliyetli olduğundan, birçok profesyonel için pratik değildir. V-modeli, paralel bir süreçtir ve nihayet umarım kabul edilebilir düzeyde hata veya hatalarla ürüne ulaşır ve 2000 yılında Spillner tarafından sunulur.



Yukarıdaki şekil , V-Model test türünün breif tanımını göstermektedir. "V-modelinin, test faaliyetinin kodlamadan sonra resmi olarak başlaması gerektiğini söylemediğini ve bunun sadece çeşitli test seviyelerine göre geliştirme aşamasının bir haritalandırılması olduğunu not ediyoruz." (Vijay, 2001) Şimdiye kadarv modelinde yer alan tüm testleri tanıtmaya çalıştık ve neyi temsil etmeleri ve neyi temsil etmeleri gerektiğinden bahsettik. V-modeli, bir geliştirme sürecinde bahsettiklerimizin bir özetidir. Birim testi modüllere, bileşene dayanmalıdır, bu yüzden sistemin ayrıntılı design'si. Entegrasyon testleri, nesne yönelimli bir durumda işbirliği gibi sistemin üst düzey tasarımına dayanmalıdır ve sistem testleri sistemin özelliklerine dayanır. Bahsettiğimiz gibi, kabul testleri customer tarafından yapılmalı ve onaylandığında geliştirmenin sonu anlamına gelmelidir, bu nedenle V modeline dahil değildir.

Yinelemeli sistemler kullanılmaya başlandıkça ve geliştirilen sistemler karmaşıklaştıkça, daha gelişmiş teknikler ve süreçler tanıtılmaktadır. Küçük şelaleler ve yinelemeler içeren modeller, spiral ve evrimsel şelaleler gibi şelale modelinin kendisinden kısa bir süre sonra tanıtıldı. Benzer şekilde, prototiplemeye ihtiyaç duyulduğunda veya bir gömme ded sisteminde olduğu gibi donanım tasarımının dahil edilmesi gibi daha karmaşık sistemler için, V modelinin türevleri birden fazla V modeli ve iç içe V modelleri gibi tanıtılmaktadır. Ancak, belirli uygulamalar için gerekli yaşam döngüsünü kullanmak için aplikatöre bırakılmalıdır.

4.MİMARIÇE TABANLI TEST

4.1 Giriş

Geleneksel test teknikleri, metodolojileri ve süreçleri, hizmet kalitesi beklentilerinin yüksek olduğu durumlar için hala yeterli görünmemektedir. Testin nerede durdurulacağına ve en azından önemli bugs'lerin gerçekten bulunup bulunmadığına karar vermek zordur. Önceki bölümlerde açıklanan, birim seviyesi ve üstü birim seviyesi (entegrasyon, sistem) testleri için geçerli olan teknikler, yazılımın birimleri arasındaki etkileşimleri ve softwar e'nin yapısını kullanır ve kontrol eder. Ve bu teknikler, güvence göstergelerinin kalitesi yardımıyla testler için yeterlilik kriterlerini tanımlamaya çalışır. Bu strateji mimari test için geçerli olmayabilir, çünkü mimari test, bileşenlerin ve konektörlerin birbirleriyle nasıl etkileşime girdiğine odaklanmalıdır.

Geleneksel, birim testi, sistem ve tümleştirmenin üstünde, genel sistemin sistem gereksinimlerini karşılayıp karşılamadığı ile ilgili sorunları ele alır, ancak mimari test, birim düzeyinin üzerinde bir test olmasına rağmen, mimari açıklama düzeyinde etkileşimleri test etmeye çalışır. "... mimari tabanlı analiz, geliştirme sürecinde normalden çok daha erken başlayabilir (uygulamadan sonra ve entegrasyon sırasında), böylece yazılım yaşam döngüsünün başlarında,düzeltilmesi daha az maliyetli ve hatasız bir şekilde düzeltilme olasılıkları daha yüksek olduğunda kusurları tespit edebilir. " (Dias, Vieira & Richardson, 2000). Kodlamaya, uygulamaya başlamadan önce yazılımın mimarisinin hazırlanması ve gözden geçirilmesi gerektiğinden, mimari test için tanımlanan testleruygulama düzeyinde kullanılabilir. Bu nedenle mimari tabanlı testin iki olası avantajı vardır;

- Geleneksel test metodolojilerini kullanarak çok maliyetli veya bazen bulunması mümkün olmayan hataları bulmaya yardımcı olabilir ve,
- Önemli hataları, gelişimin ilk aşamalarında, bunları çözmenin daha az maliyetli olduğu durumlarda bulabilir.

Aşağıdaki bölümlerde, bir mimari tanımlama dili (ADL) olan Wright tanıtılacaktır. Mimariye bağlı olarak testleri tanımlamak için kullanılacak teknik ve Arayüz Bağlantı Grafiği (ICG) ve Davranış Grafiği (BG) üzerinde tanımlanan test yolları tanıtılacaktır. Wright mimarisi softwa re'ninarayüz bağlantı grafiği ve davranış grafiğine eşlenmesi tanıtılacaktır. Genel, genel mimari açıklamasıyla başlayacağız ve genel yazılım mimarisi (SA) tanımlama dilinin özniteliklerini tanıtacağız.

Odak noktası, mimari açıklamaya bağlı olarak testleri tanımlamak olacaktır. "Endişe, SA'nın (yazılım mimarisi) tutarlılığının ve doğruluğunun analizinde değil, implementasyonun test tekniğini yönlendirmek için SA düzeyinde açıklanan bilgilerden yararlanmaktır. " (Bertolino, Corradini, Inverardi

Muccini, 2000). Mimari ve ürün arasındaki bağlantı, ürünün henüz uygulanmadığı için spesifikasyonu konudur. Mimarinin doğrulanması, dolayısıyla eğer right sistemini inşa ediyorsak, bu belgenin odak noktası değildir. Mimarinin gereksinimleri yerine getirmek için doğru olduğu varsayılır ve buna bağlı olarak yazılımı test etmek için teknikler tanıtılır. Bu, geleneksel sistemin veya integration testlerinin yerine geçmez.

4.2 Yazılım Mimarisi

Yazılım mimarileri, geliştiricilerin gereksiz uygulama ayrıntılarını gizleyerek büyük resme odaklanmalarını sağlar. Bileşenler arasındaki üst düzey iletişim protokolleri, sistem yapısı, geliştirme süreci,yazılım bileşenlerinin donanım parçalarına eşek ateşlenmesi vb. geliştiricilerin odak noktası haline gelir. Mimari Açıklama Dilleri (ADL'ler) bu özellikleri algoritmik olarak analiz edilebilen ve manipüle edilebilen şekillerde tanımlar. Bu,sistem düzeyinde testler türetmek için benzersiz bir fırsat sağlar. Bu testlerin geleneksel sistem testlerinin yerini alması gerekmez, ancak onlar için bir eklenti olmaları amaçlanmıştır.

"Mimarlık, mimari elemanların seçimi, etkileşimleri ve bu elemanlar üzerindeki kısıtlamalar ve gerekli etkileşimleri ile ilgilidir. Gereksinimleri karşılamak ve tasarım için bir temel oluşturmak için bir çerçeve sağlamak." (Perry ve Wolf, 1992)

Yazılım mimarisi için farklı tanımlar olmasına rağmen, genellikle dört unsur bunun yapı taşları olarak tanımlanır; bileşenler, konektörler, arayüzler ve konfigürasyon.

Bileşen: Bağımsız varlığı olan bir nesne, örneğin bir modül, işlem, prosedür veya değişken.

Arabirim: Bir bileşen ile ortamı arasındaki mantıksal etkileşim noktası olan yazılı bir nesne.

Bağlayıcı: Arabirim noktaları, bileşenleri veya her ikisini birden ilişkilendiren yazılı bir nesne.

Yapılandırma: Nesneleri belirli bir mimariye bağlayan kısıtlamalar topluluğu. (Moriconi ve Qian, 1994)

Bileşenler, arabirimler ve bağlayıcılar ana öğelerdir, genellikle adları vardır ve bileşenlere, bağlayıcılara ve arabirimlere ayrıştırılarak rafine edilebilirler. Configuration, bu yazılım mimarisi öğelerinin bir sistem oluşturmak için nasıl birleştirilebileceğini açıklar. Aynı ana eleman kümesiyle, farklı konfigürasyon anlamına gelecek ve bize iki ayrı yazılım mimarisi verecek bir halka veya yıldız ağı uygulamak mümkündür.

Yazılım mimarileri, geliştiricilerin bir uygulamanın tek tek bileşenlerinin ayrıntılarını soyutlamalarına olanak tanıyarak, bu bileşenler arasındaki etkileşimleri açıklayan ilişkili bağlayıcılara sahip bileşen kümeleri olarak görülmelerini sağlar. Dahası, sistemin matematiksel bir tanımını verdiği için otomatik araçlarla kolayca manipüle edilebilen bir formdadır. Bunlar bunun iki ana avantajıdır.

Şimdiye kadar,bazıları belirli bir alan için önerilen birçok mimari tanım langua ges tanımlandı. Birçoğu mimarlık temsilleri için biçimsel yaklaşımlar kullanır. Rapide, Aesop, UniCon, MetaH, LILEANNA, C2, Darwin, ACME ve Wright bazı örneklerdir. Bir sonraki bölümde Wri ght mimari tanımlama diline odaklanacağız.

4.3 Wright

Wright, CMU'da Dr. David Garlan tarafından geliştirilmiştir ve iletişim protokollerinin analizine vurgu yaparak tasarlanmış genel amaçlı bir mimari tanımlama dilidir. "Wright, bileşenlerin, bağlayıcıların ve konfigürasyonların temel mimari soyutlamaları etrafında inşa edilmiştir. WRIGHT, bu öğelerin her biri için açık gösterimler sunarak, bileşenin genel kavramlarını hesaplama ve bağlayıcıyı i nteraction modeli olarak resmileştirir. " (Allen, 1994)

4.3.1 Bileşenler

Bileşen yerel, bağımsız bir hesaplamayı tanımlar. Bileşenin tanımı, hesaplama ve arayüz olmak üzere iki önemli bölüme sahiptir. "Arayüz bir dizi bağlantı noktasından

oluşuyor. Her bağlantı noktası , bileşenin katılabileceği bir etkileşimi temsil eder." (Allen, 1994) Hesaplama, bileşenin gerçekte ne yapacağını açıklar. Hesaplama, arayüz veya bağlantı noktaları tarafından açıklanan etkileşimleri gerçekleştirir ve bağlantı noktalarının bileşeni oluşturmak için nasıl bağlandığını gösterir. Bağlantı noktalarından oluşan bir bileşenin arabirimi, bir bileşenin iki yönünü gösterir; sistemden davranış ve beklentiler.

Bağlantı noktası belirtimi, bileşenin söz konusu bağlantı noktası üzerinden görüntülenmesi durumunda sahip olması gereken özellikleri gösterir . Bu nedenle bir bağlantı noktası, bileşenin kısmi bir belirtimidir; her bağlantı noktası belirtimi bize hesaplama hakkında bir şeyler söyler. Hesaplama, ne yapıldığının daha eksiksiz bir açıklamasını sağlar. Ancak, bağlantı noktası belirtimleri, hesaplamanın tam bir belirtimini oluşturmak için birleştirilmez.

Bileşenin analizi hesaplamaya dayanmalıdır; bağlantı noktaları ek bir soyutlama düzeyi içindir ve bileşenin etkileşimini analiz etmeye yardımcı olur. Yaniport sayısı, bir programlama dilindeki soyut veri türü bildirimlerine benzer. Belirli bir bileşen türündeki her bağlantı noktasının adı, o bileşen içinde benzersiz olmalıdır. Wright'taki bir bileşen tanımının yapısı şu şekilde verilebilir;

"Komponent SplitFilter

Bağlantı Noktası Girişi [veri sonuna ulaşılana kadar verileri okuyun]

Sol Bağlantı Noktası [çıkış verileri tekrar tekrar]

Port Right [çıkış verileri tekrar tekrar]

Hesaplama [Giriş'ten tekrar tekrar okunur, sonra çıkış, Sol ve Sağ bağlantı noktaları arasında dönüşümlü olarak.]" (Allen, 1994)

4.3.2 Bağlayıcılar

Bağlayıcı, bileşenler arasındaki etkileşimi temsil eder. Wright mimari tanımlama dili, bağlayıcı türlerini açık hale getirerek iki amaca ulaşmayı amaçlamaktadır: analizin uygulanabilirliğini genişletmek ve kompozisyonların bağımsızlığınıartırmak. Wright, normalde yazılım mimarisinde gerçekleşen ortaklığı, bir bağlayıcı türünde bir etkileşim deseni çizerek ve ardından deseni bağlayıcı örneklerinde tekrar tekrar kullanarak yapar.

Açık konektörlerin kullanılmasının bir avantajı vardır, bir bileşenin diğerleriyle etkileşime girme şeklini oluşturarak bağımsızlığını arttırır. Bir bileşenin ortamından ne beklediğini açıkça ortaya koyan ve birden fazla bağlamda kullanılabilmesini sağlayan

bir sınırdır. Bileşen ne yapacağını belirtecek ve konektör sistemin geri kalanıyla nasıl entegre edileceğini gösterecektir.

Wright, bir bileşeni tutkal ve roller olmak üzere iki parçada tanımlar. Roller etkileşimdeki katılımcıları belirtir ve yapıştırıcı katılımcıların birlikte nasıl çalışacağını açıklar. Rol, bir bileşenin bağlantı noktasına benzer şekilde, kendisiyle etkileşime girecek bileşenden ne beklendiğini açıklar. Konektörün tutkalı şunları gösterir:

Konektörün gerçekte d o olacağı, bir bileşenin hesaplanmasına benzer şekilde. Daha büyük bir hesaplama oluşturabilmek için bileşenleri koordine eder. Bu nedenle, bileşenler rol tarafından açıklandığı gibi etkileşime girdiğinde, konektörün yapıştırıcısına göre daha büyük hesaplama gerçekleşecektir. Wright'taki bir konnektör tanımının yapısı şu şekilde verilebilir;

"Konnektör Boru

Rol Kaynağı [verileri tekrar tekrar teslim etme, kapanışla sonlandırma sinyali verme]

Rol Havuzu [verileri tekrar tekrar okuma, verilerin bitiminde veya öncesinde kapanma]

Tutkal [Sink,Kaynak tarafından teslim edilen aynı o rder'de veri alır]" (Allen, 1994)

4.3.3 Yapılandırma

Bağlayıcıların ve bileşenlerin birleşimi, Wright mimarisindeki yapılandırmayı oluşturur. Bir sistemin bileşenleri ve bağlayıcıları birden fazla kez kullanılabileceğinden, her kullanımı onun bir ' örneği' olacaktır. Her örneğin benzersiz bir adı olmalıdır. Bu nedenle, C++'daki yazılım sınıflarına ve nesnelerine benzer. Böylece bileşenler ve bağlayıcılar, örneklerinin özelliklerini temsil eder.

Mimarinin yapılandırması, bu örnekler arasındaki bağlantılar tanımlandıkça tamamlanacaktır. Bu bağlantılara 'ekler' denir. Ekler, bir bağlantıya hangi bileşenin veya konektörün katılacağını gösterir. Bu nedenle ekler, bir bileşenin bağlantı noktası ile bağlayıcının rolü arasındaki bağlantılardır. Örneğin, aşağıdaki örnek "Split. Left" ek bildirimini P1 olarak gösterir. "Kaynak", Split bileşeninin P1 etkileşiminde Kaynak rolünü oynayacağını gösterir. Bu rolü Sol liman aracılığıyla dolduracak." (Allen, 1994) Which, Split'in 'Sol' bağlantı noktasına çıkış yaptığı verilerin, teslim edilecek P1'in 'Kaynağı' tarafından yakalanacağı anlamına gelir. Eşleşen bildirim "Upper.Input as P1. Sink", "Upper" bileşeninin "Input" portuna teslim edileceği anlamına gelir.

4.4 Revised Petri Net

Aşağıdaki bölümlerde, test amaçlı arayüz bağlantısı ve davranış grafiklerini tanıtacağız, bu gösterimlere bağlı olarak testlerin nasıl tanımlanacağı hakkında bilgi vereceğiz veyazılım mimarisinin Wr ight açıklamasını bu grafiklerle eşlemek için yöntemler sunacağız. Davranış grafiği Petri Net'i yapı taşı olarak kullandığından, burada Petri Net'in ne olduğunu tanıtıyoruz ve ihtiyaçlarımıza uyacak üç değişiklik tanımlıyoruz. Değişikliklerle ortaya çıkan grafiğe Revize Petri Ağı denir.

Petri Net, dağıtılmış sistemleri modellemek için tanıtıldı ve 1962'de Carl Adam Petri tarafından tanımlandı. Sistemi görselleştirmeye yardımcı olan grafiksel bir yapıya ve otomatik bir sistem üzerinde analiz yapmak ve uygulamak için yararlı olabilecek matematiksel bir temsile sahiptir. Petri Net, dörtlü PN = (P, T, I, O) ile temsil edilen iki parçalı yönlendirilmiş bir grafiktir:

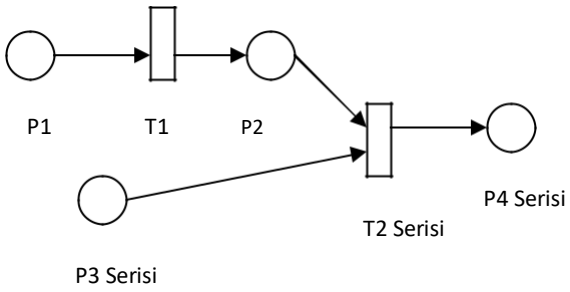
P = {p1,... ,pn} sonlu bir yer kümesidir.

T = {t1,... ,tm} sonlu bir geçiş kümesidir.

I (p,t), yerlerden geçişlere yönlendirilmiş yaylar kümesine karşılık gelen bir eşleme P x T -> {0,1}'dir.

O(t,p), geçişlerden yerlere yönlendirilmiş yaylar kümesine karşılık gelen bir eşleme T x P -> {0,1}'dir.

Petri Net tanımındaki her yer bir arabellek, kaynak veya koşul anlamına gelir ve grafikte bir daire düğümü ile temsil edilir. Her geçiş bir algoritma, bir süreç veya bir olay anlamına gelir ve grafikte bir çubuk düğümle temsil edilir. Hem içeri hem de dışarıdaki yaylar yönlendirilir ve sabitlenir. Eğer I(p,t) = 1 ise, p yerinden t geçişine, dolayısıyla t geçişine giriş yapan bir ar c olduğu anlamına gelir ve eğer O(t,p) = 1 ise, geçiş t'den p yerine bir yay olduğu anlamına gelir, böylece t geçişinden çıktı.



Şekilde örnek Petri Net sistemi gösterilmektedir. Bu sistemin matematiksel temsili;

P = {P1, P2, P3, P4}

T = {T1, T2}

I(P1, T1) = 1	I(P1, T2) = 0
I(P2, T1) = 0	I(P2, T2) = 1
I(P3, T1) = 0	I(P3, T2) = 1
I(P4, T1) = 0	I(P4, T2) = 0

O(T1, P1) = 0	O(T2, P1) = 0
O(T1, P2) = 1	O(T2, P2) = 0
O(T1, P3) = 0	O(T2, P3) = 0
O(T1, P4) = 0	O(T2, P4) = 1

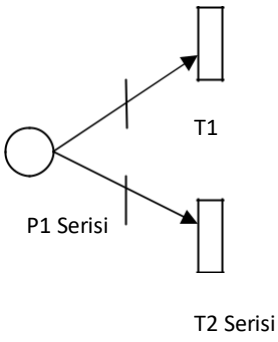
Görüldüğü gibi sistem matris olarak temsil edilebilir. Bir Petri Net, yerlerde bulunan noktalarla (●) temsil edilen belirteçler içerebilir. Belirteçler, kaynağın kullanılabilirliğini, arabellekteki öğe sayısını veya kullanıma bağlı olarak koşulun yerine getirilmesini temsileder. Petri Net sistemindeki bir geçiş, yalnızca ve yalnızca tüm giriş yerleri en az bir düğüm içeriyorsa tetiklenir. "Petri Net'in işaretlenmesi, her yere negatif olmayan bir tamsayı (belirteç sayısı) atayan bir haritalama M'dir." (Jin, 200 0) Bu nedenle, M(p) sıfırdan büyük olmalıdır.

Geçiş etkinleştirildiğinde, tetiklenebilir, böylece algoritmayı, işlemi veya temsil ettiği olayı yürütebilir. Sistemdeki işaretler bir geçişin ateşlenmesiyle değişecektir . Bu, her girişten bir belirteci kaldırır ve her çıktıya bir belirteç bulur. Dolayısıyla, şekil 4.1'deki T2, hem P2 hem de P3'ün belirteçleri olmadığı sürece ateş etmeyebilir. T2 geçişi tetiklendiğinde, P2 ve P3'ün her birinden bir belirteç kaldırır ve P4'e bir belirteç ekler.

Revize Edilmiş bir Petri Net, yukarıda tanıttığımız Petri Net sisteminden üç farklılığa sahip olacaktır. Sürecin sonu temsil edilecek, iç ve dış tercihler sisteme kazandırılacaktır.

İşlemin sonu, yeni bir P uç yeridir. It, grafikte kalın bir daire düğümü ile temsil edilir. Bu, Petri Nets'in herhangi bir teorik özelliğini değiştirmez, sadece bir temsil değişikliği olarak değişir.

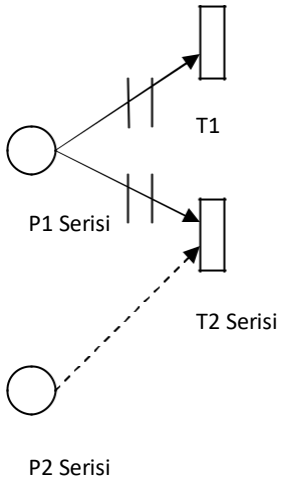
Dâhili seçenekler mevcut olacaktır. Ben içsel veya o içsel bu seçimleri temsil edecektir. Bu seçimler sürecin kendisi tarafından yürütüldüğü için, deterministik değildirler. İç seçimler, üzerlerinde küçük tek dikey çizgiler bulunan yaylarla temsil edilir. Bu durum Şekilde görülebilir.



Dış seçimleri de temsil etmek mümkün olacaktır. Dış seçimler sürecin kendisinden ziyade çevre tarafından yapılır ve deterministiktir.

Dış seçimler, üzerlerinde küçük çift dikey çizgiler bulunan yaylarla temsil edilir.

Ben dışsal ya da O dışsal bu seçimleri temsil edecektir. Bu durum Şekilde görülebilir.



P1'de belirteç olduğunda, T1 veya T2 ateşlenir ve bu karar P2'de bir belirtecin kullanılabilirliği olarak verilir. P2 geçişinde bir belirteç varsa T2 ateşlenir. Ateşleme kararını veren Place P2, yazılımdaki başka bir bileşenin parçası olacaktır.

4.5 Arayüz Bağlantı Grafiği (ICG)

Grafik repkızgınlıkları, test edicilerin testin yeterliliğini ve dolayısıyla testi ne zaman durduracaklarını tanımlamalarına yardımcı olur. Tüm mimari tanımlama dillerine benzer şekilde, arabirim bağlantı grafiğinin bileşenleri ve bağlayıcıları vardır ve bunlar arasındaki bağlantı ilişkilerini temsil eder. Konektörler ve bileşenler arasındaki dış bağlantıların yanı sıra, bunların iç ilişkileri de arayüz bağlantı grafiğinde temsil edilir.

Bir arabirim bağlantı grafiğinin yapı taşları, bileşenler, konektörler, interface'lerdeki bileşenler, konektör arabirimleri, bileşenler ve konektörler arasındaki bağlantılar ve bileşenlerin veya konektörlerin içindeki bağlantılardır. Bileşenler dikdörtgen kutularla, arayüzleri bu kutuların kenarlarında net dairelerle temsil edilir. Konektörler köşeleri dikdörtgen kutular ile temsil edilir.

Yuvarlatılmış ve arayüzleri bu kutuların kenarlarında gölgeli dairelerle. Dış bağlantılar katı oklarla ve kesikli çizgili astarlı oklarla iç bağlantılarla temsil edilir.

Arayüz bağlantı grafiği şu şekilde tanımlanır; ICG = (N, C, N_Interf, C_Interf, N_Ex_arc, C_Ex_arc, N_In_arc, C_In_arc), burada

N = {N1, ... Nn} sonlu bir bileşen kümesidir.

C = {C1, ... Cm} sonlu bir konektör kümesidir.

N_Interf = {N1.interf1, ... N1.interfs, ... Nn.interf1, ... Nn.interfx} sonlu bir bileşen arabirimleri kümesidir.

C_Interf = {C1.interf1, ... Cm.interft), sonlu bir konektör arayüzleri kümesidir.

N_Ex_arc(n, c), bileşenlerden konektörlere yönlendirilmiş yaylar kümesine karşılık gelen bir eşleme N x C -> {0,1}'dir.

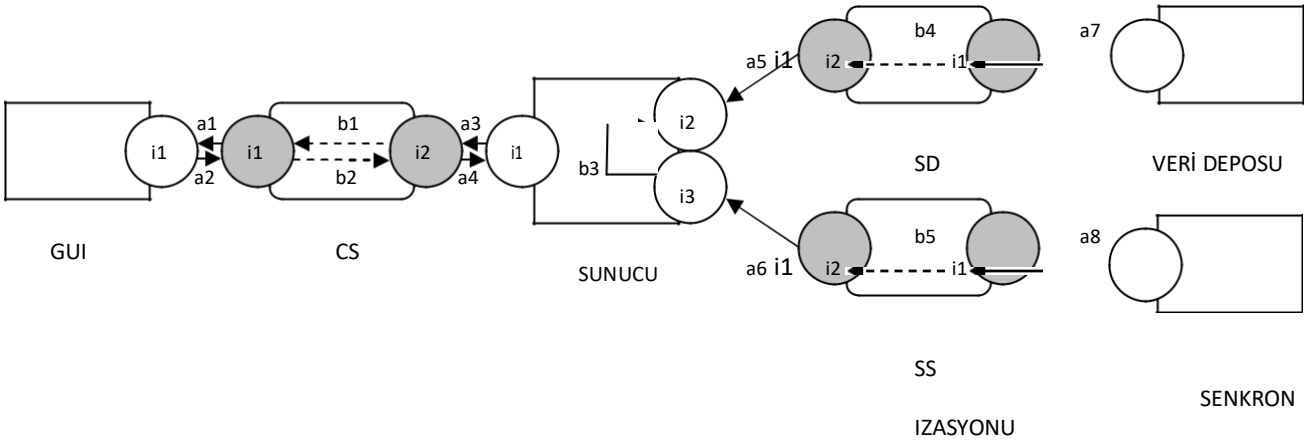
C_Ex_arc(c, n), konektörlerden bileşenlere yönlendirilmiş yaylar kümesine karşılık gelen bir C x N -> {0,1} eşlemesidir.

N_Inter_arc(n.interf1, n.interf2) x N_Interf -N_Interf {0,1} > bir eşlemedir.

bir bileşenin arayüzleri arasındaki yönlendirilmiş yaylar kümesine karşılık gelir.

C_Inter_arc(c.interf1, c.interf2) x C_Interf -C_Interf {0,1} > bir eşlemedir bir konektörün arayüzleri arasındaki yönlendirilmiş yaylar kümesine karşılık gelir.

Daha önce de belirttiğimiz gibi, grafikler matrisler olarak temsil edilebilir ve bu nedenle bunlara bağlı olarak otomatik bir test üretimi tanımlamanın bir yolu vardır. Bir arabirim bağlantı grafiğinin insidans matrisi, n'nin bileşen arabirimleri s ve m'nin konektör arabirimleri anlamına geldiği bir (n+m) x (n+m) matrisi olacaktır. M ij'nin hücre değeri, i'den j'ye yönlendirilmiş yay varsa 1, bu arayüzler arasında bağlantı yoksa 0 ve bağlantı j'den i'ye ise -1 olacaktır.



4.6 Davranış Grafiği (BG)

Bir davranış grafiği, Gözden Geçirilmiş Petri Ağıdır ve iki ilgili bileşenin ilişkisi ve davranışıdır. Her biri bunlardan birinin davranışını temsil eden davranış grafiği iki bileşen alt ağından oluşur. Bu bileşen alt ağları arasındaki bağlantıları gösteren yerler ve geçişler vardır. "Her bileşen alt ağı, her bağlantı noktasının beklenen davranışını açıklayan bir veya daha fazla bağlantı noktası alt ağı içerir ." (Jin, 2000) Bu ilişkiler transfer veya sipariş ilişkileri olabilir.

Davranış grafiği resmi olarak şu şekilde tanımlanabilir; BG = (Comp1(Pn1, Tn1, In1, On1), Comp2(Pn2, Tn2, In2, On2), Pc, Tc, Ic, Oc), burada

Comp1, C1 Pn1 = {p 11 bileşenini tanımlayan grafiktir,p1n} sonlu bir yer kümesidir. Tn1 = {t11,t1m} sonlu bir geçiş kümesidir.

In1(p1n,t1n), yerlerden geçişlere yönlendirilmiş yaylar kümesine karşılık gelen bir Pn1 x Tn1 -> {0,1} eşlemesidir.

On1(t1n,p1n), geçişlerden yerlere yönlendirilmiş yaylar kümesine karşılık gelen Tn1 x Pn1 -> {0,1} eşlemesidir.

Comp2, C2 Pn2 = {p21 bileşenini tanımlayan grafiktir, p2n} sonlu bir yer kümesidir. Tn2 = {t21, t2m} geçişlerin sonlu setidir.

In2(p2n,t2n), yerlerden geçişlere yönlendirilmiş yaylar kümesine karşılık gelen bir Pn2 x Tn2 -> {0,1} eşlemesidir.

On2(t2n,p2n), geçişlerden yerlere yönlendirilmiş yaylar kümesine karşılık gelen Tn2 x Pn2 -> {0,1} eşlemesidir.

PC = {pc1,pcn}, C1 ve C2 bileşenlerini bağlamak için kullanılan sonlu bir yer kümesidir. Tc = {tc1,tcm}, C1 ve C2 bileşenlerini bağlamak için kullanılan sonlu bir geçiş kümesidir

Ic : ({Pn1, Pn2, Pnc} x {Tn1, Tn2, Tnc}) -> {0,1} Eğer value 1 ise, o zaman yay vardır, aksi takdirde yay yoktur.

Oc : ({Tn1, Tn2, Tnc} x {Pn1, Pn2, Pnc}) -> {0,1} Değer 1 ise, yay vardır, aksi takdirde yay yoktur.

Ve $Pn1 \cap Pn2 \cap PC = \emptyset$, $Tn1 \cap Tn2 \cap Tc = \emptyset$.

Davranış grafiği matris olarak da gösterilebilir.

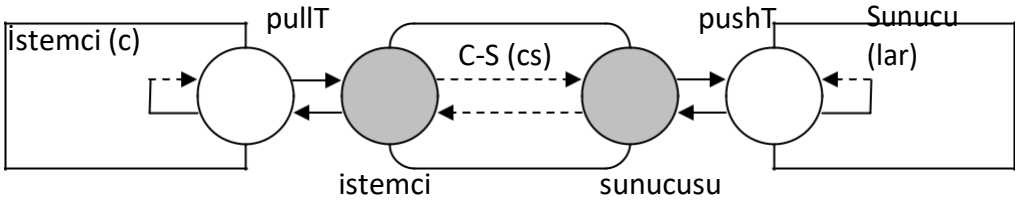
4.7 Wright'ı ICG'ye Eşleme

Bir Wright açıklamasını bir arayüz bağlantı grafiğiyle eşlemek aslında basittir. Arabirim bağlantı grafiği, her bileşeni, bağlayıcıyı, bileşenlerin bağlantı noktalarını ve konektörlerin rollerini,bileşenler ve konektörler arasındaki c onneksiyonlarını ve bunların içindeki olası bağlantıları görsel olarak içerecektir.

Wright açıklamasının her bileşeni, arabirim bağlantı grafiğindeki bir bileşen kutusuyla eşleşir. Bağlantı noktaları arabirimlere karşılık gelir. Benzer şekilde, her bağlayıcı bir bağlayıcıyla ve bunların rolleri arabirimlerle eşleşir. Arayüz bağlantı grafiğinin bileşen iç yayları, ilgili bileşenin hesaplanmasıyla belirlenecektir. Wright'taki connectors'un tutkalı, konektörler için de aynı şeyi yapacaktır. Wright mimari açıklamasının ekleri, arabirim bağlantı grafiğindeki bağlayıcılar ve bileşenler arasındaki bağlantıları yansıtır. Adlar Wright mimariçözümlemesi ile arabirim bağlantı grafiği arasında eşlenmelidir.

Arayüz bağlantı grafiği, Wright açıklamasının yapısal ilişkilerini temsil eder. Temel olarak bileşenlerin hesaplamalarıyla gösterilen davranışsal bilgileri temsil edebilmek için, davranış grafiğini kullanacağız.

Şekilde, bölüm 4.3.4'te tanıttığımız istemci-sunucu mimarisinin arabirim bağlantı grafiğini görüntüler.



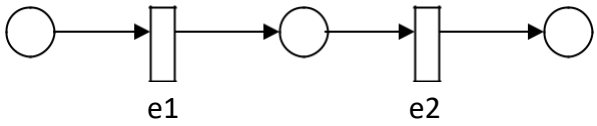
4.8 Wright'ı BG ile Eşleme

Wright Açıklamasını bir Davranış Grafiği ile eşlemek, onu bir Arabirim Bağlantı Grafiği ile eşlemekten daha karmaşıktır. Bileşen alt ağları dikkate alınmalı ve bileşenlerin bağlantı noktası açıklamaları bu bileşen alt ağları içindeki bağlantı noktası alt ağları haline gelmelidir. Bileşenin hesaplanması, bağlantı noktaları arasındaki aktarım linklerini tanımlar, sıralama kuralları açıklanıyorsa, bir hesaplama alt ağı olarak temsil edilebilir. Bağlayıcılar için bir alt ağ oluşturmaya gerek yoktur, ancak bu bağlayıcıların yapıştırıcıları iki bileşen arasındaki bağlantıyı temsil eder. Bir bağlayıcı hesaplamasına benzer şekilde, sıralama kuralları varsa, bir tutkal alt ağı oluşturmak düşünülebilir.

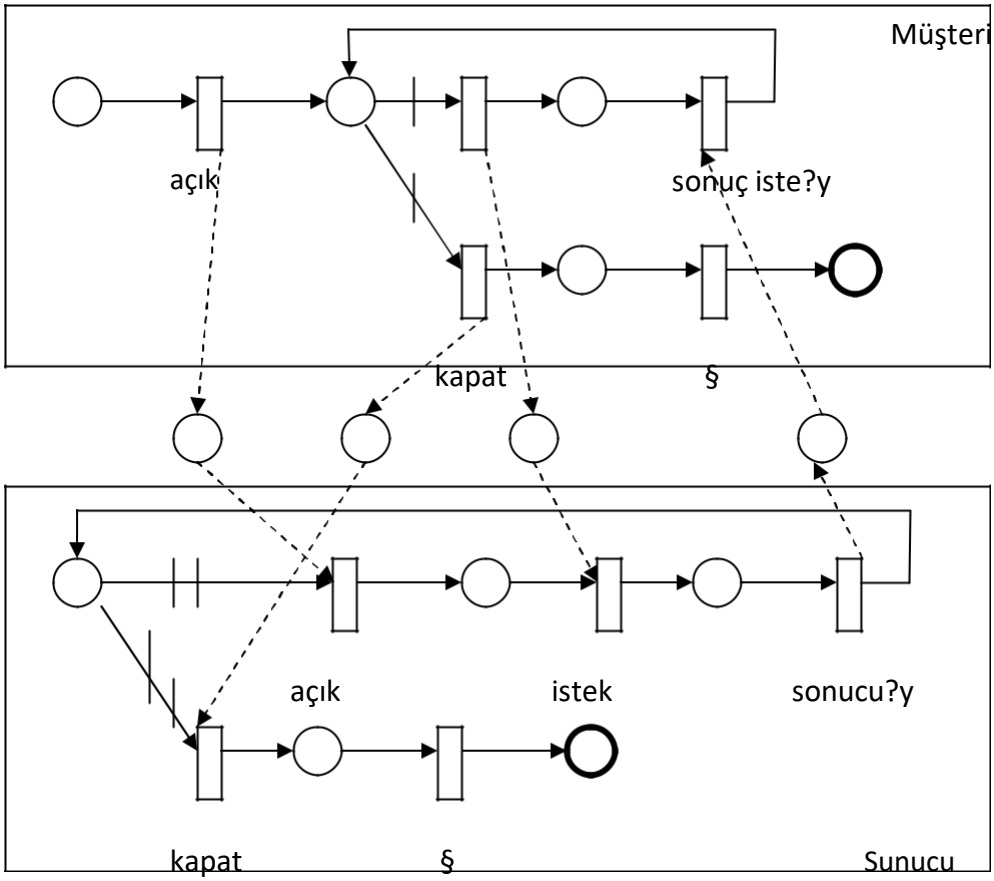
Olaylar geçişler olarak temsil edilir. Mekanlar yüksek öneme sahip olmadığı için onlara isim vermek zorunda değiliz ama daha az önemleri yok. Şekil 4.7.a, Wright'taki P=e->P açıklamasının Davranış Grafiğinde nasıl temsil edildiğini gösterir. 4.7.b, özel etkinliğin temsilini gösterir.

a: P=e->P b: §

Wright tanımında ";" ve "->" temsilleriyle iki tür sıralı kompozisyon vardır ve bunların her ikisi de Şekil 4.8'deki ile aynı şekilde temsil edilir. Sırasıyla ∏ ve □ ile temsil edilen iç ve dış seçimler, bölüm 4.4'teki açıklamalarımıza benzer şekilde görüntülenir. Sırasıyla şekil 4.2 ve 4.3'te gösterildiği gibi tek ve çift çizgiler kullanılacaktır.



Bir Wright açıklamasını Davranış grafiği biçiminde temsil edebilmek için, önce bileşenler bileşen alt ağlarıyla eşlenmeli, ardından yukarıdaki dönüştürme kuralları kullanılarak bileşenin hesaplanması eşlenmeli ve ardından benzer şekilde bağlayıcıların yapıştırıcısı aktarılmalıdır.



İstemci-sunucu modelinin Davranış Grafiği gösterimi

Bu ödevin 4.3.4 numaralı bölümünde verilen sunucu-istemci modelinin BG'si açıklanan yöntemle, yukarıdaki Şekilde oluşturulmuş ve sunulmuştur. Gördüğünüz gibi bir Davranış Grafiği, Arabirim Bağlantı Grafiğinden daha fazla ayrıntı verir.

4.9 Test Tekniği

Yol kapsamı, her test cihazı tarafından bilindiği gibi, yazılımdaki tüm hataları ortaya çıkaracak gerçek test tekniğidir, ancak uygulanan yazılımın boyutu son yıllarda çok arttığı için küçük yazılım sistemlerinde bile kullanmak mümkün değildir. Anlatılacak tekniği kullanarak en azından mimarlık düzeyindeki yolları kapsayabileceğiz. Bu, her biri gerekli birim testlerini geçen mimari bileşenler arasındaki ilişkilerin test edilmesine odaklanacaktır. Seilişkilerine, yürütme emri kuralları, veri aktarımı ve kontrol aktarımı gibi olası bağlamalara dayanan mimari ilişkiler denir. Detaylı tasarım veya uygulama seviyesi hataları bu tekniğin odak noktası değildir, bileşenler ve konektörler arasındaki mimari ilişkiler odak noktalarıdır, bu nedenle tüm mimari ilişkiler testlerde ele alınmalıdır.

Mimari düzeyde test aşağıdaki hususlara odaklanmalıdır;

- Bileşenden konektöre bağlantı
- Konektörden bileşene bağlantı
- Komponent iç arayüzleri
- Konektör iç arayüzleri
- Doğrudan ve dolaylı bileşenden bileşene bağlantı

Tüm yapı bağlantısı

Test edilecek yollar yukarıdaki bağlantılara ve interf aslarına göre tanımlanmalıdır. Tüm yapı bağlantısı, sistemin tüm iç ve dış bağlantılarını test etmektir. Mimari düzeyde mevcut tüm yolları bularak ve hangilerinin işlendiğini bularak, yaptığımız mimari testlerin yol kapsama yüzdesini anlayabiliriz.

4.10 ICG için Yol Tanımları

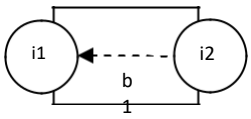
Bir önceki bölümde, testlerin kapsayacağı yollar gayri resmi olarak tanıtılmış, bu bölümde; Kapsanacak Arayüz Bağlantı Grafiği'nin iki arayüzü arasındaki yollar resmi olarak tanıtılacaktır.

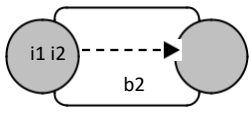
4.10.1 İç Transfer Yolları

Dahili transfer yolları hem bileşenler hem de konektörler için tanımlanır. İç aktarım ilişkisi,bir bileşen veya konektör içindeki in terfaces arasındaki veri veya kontrol ilişkilerini tanımlar. Bu ilişki geçişli, simetrik veya refleksif değildir. Bir konektör veya bileşenin iki arabirimi olduğunda, örneğin i1 ve i2, aralarında bir kontrol ilişkisi olabilir. i1 d kapalıyken i 2 kapanabilirve bu bir kontrol ilişkisidir. Benzer şekilde, i1 veri alırsa ve i2 bu verileri kullanırsa, bu arayüzler arasında bir veri ilişkisi vardır. İki arayüz arasında bir iç transfer ilişkisi varsa bir İç Transfer Yolu vardır ve bu path test edilmelidir. Bu yollar şöyle tanımlanabilir;

ComponentInternalTransferPath = {(i, j)} | i ∈ N_Interf, j ∈ N_Interf ve N_InternalTransferRelation(i, j) = 1}

ConnectorInternalTransferPath = {(i, j)} | i ∈ X_Interf, j ∈ C_Interf ve C_InternalTransferRelation(i, j) = 1}





ComponentInternalTransferPath ve

ConnectorInternalTransferPath

4.10.2 Dahili Sipariş Kuralları

Bileşenlerin veya bağlayıcıların arabirimleri bazı yürütme sırası kurallarına göre davranabilir ve paralel veya sıralı olarak hareket edebilir. Bu durumda, bir bağlayıcının veya bileşenin bu arabirimleri arasında bir sıralama ilişkisi vardır. Bu ilişki simetrik ve geçişlidir, ancak refleksif değildir. Üç tür sıralama tanımlanmıştır.

(i1 | i2) : arayüz i1, arayüz i2'ye veri veya kontrol aktarımına sahiptir.

(i1 => i2) : arayüz i1 çalışır ve işlemeyi bitirir, ardından arayüz i2 çalışır.

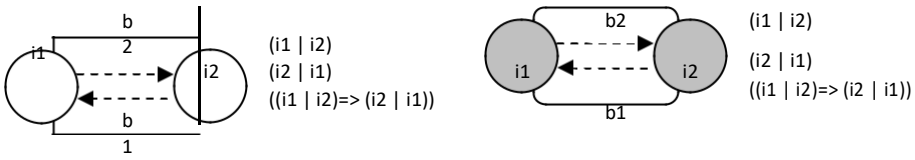
(i 1 // i2) : i1 ve i2 arayüzleri paralel olarak çalışır.

Bir sıralama ilişkisi tanımlanmışsa, bu sıralama ilişkisini test edebilmek için bir yol olmalıdır. Bu yollar şöyle tanımlanabilir;

ComponentInternalOrderingRelation = {OrderRule(i,j)} | i ∈ N_Interf, jN_Interf ve N_InternalOrderingRelation(i, j) = 1}

ConnectorInternalOrderingRelation = {OrderRule(i,j)} | i ∈ C_Interf,j

C_Interf ve C_InternalOrderingRelation(i, j) = 1}



Şekil 4.11 ComponentInternalOrderingRelation ve Connector-InternalOrderingRelation

Şekil 4.11'de verilen örnekler, i1'in i2'ye veri veya kontrol aktarımına sahip olduğunu ve benzer şekilde i2'nin i1'e veri veya kontrol aktarımına sahip olduğunu açıklamaktadır. Üçüncü kural, i1'den i2'ye olan b2 aktarımının, b2 aktarımının çalıştırdığından daha fazla çalıştığını ve işlemeyi tamamladığını belirtir.

4.10.3 Bileşenden Bağlayıcı Yoluna

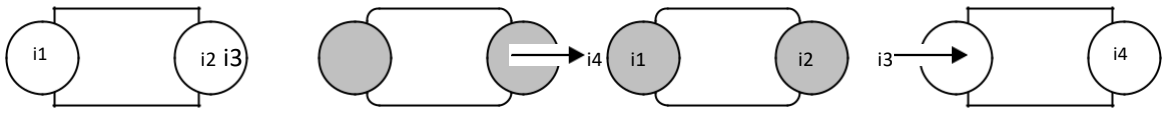
Bir bileşenin arayüzü bir konektörün arayüzüne bağlıysa, aralarında refleksif, geçişli veya simetrik olmayan bir ilişki vardır. Bu ilişki test edilmeli ve yol şöyle tanımlanabilir;

N_C_Path = { (i, j)} | i ∈ N_Interf, j ∈ C_Interf ve N_C_Relation(i, j) = 1}

4.10.4 Bileşen Yoluna Konektör

Bir konektörün arayüzü bir bileşenin arayüzüne bağlıysa, aralarında refleksif, geçişli veya simetrik olmayan bir ilişki vardır. Bu ilişki test edilmeli ve yol şu şekilde savunulabilir;

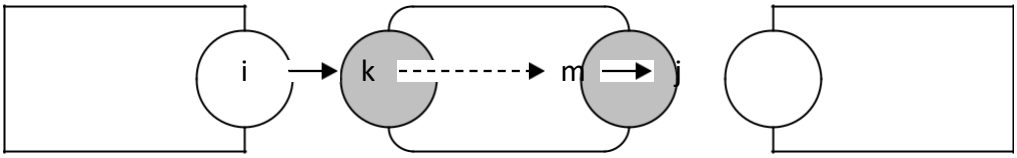
C_N_Path = { (i, j)} | i ∈ C_Interf, j ∈ N_Interf ve C_N_Relation(i, j) = 1}



Şekil 4.12 N_C_Path ve C_N_Path

4.10.5 Doğrudan Bileşenden Bileşen Yoluna

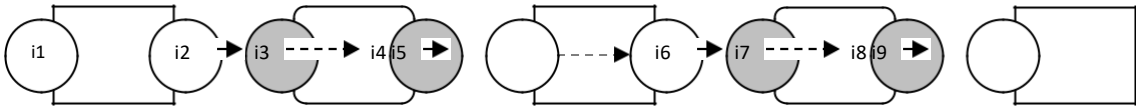
Bu refleksif ve simetrik ilişki, tüm C_N_Relation, N_C_Relation ve C_ İç İlişki yolda varsa var olur.



Şekil 4.12 DirectComponentPath

4.10.6 Dolaylı Bileşenden Bileşen Yoluna

Eğer N1 ve N2 bileşenleri doğrudan bileşen ilişkisine sahipse ve N2 ve N3 doğrudan bileşen ilişkisine sahipse ve N2,başlangıç ve çağrılan noktalar olarak doğrudan bileşen ilişkilerinin bir parçası olan arayüzleri arasındaki iç ilişkiye sahipse, o zaman N1 ve N3 arasındaki bileşen ilişkisine dolaylı bir bileşen vardır.



Şekil 4.13 IndirectComponentPath i2'den i9'a

Testin ne zaman durdurulacağına karar verebilmek için farklı kapsama stratejilerinin üzerindeki yolları kullanmak kullanılabilir. Bireysel bileşen arabirimi kapsamı, tanımlanan tüm bileşen iç aktarımını ve bileşen iç sıralama yollarını kapsayan bir test çalışması oluşturularak elde edilebilir. Bu, konektörlere de uygulanabilir. Doğrudan bileşenden bileşene kapsama alanı, tüm bileşenlerin bağlayıcıya, konektörün bileşene ve doğrudan bileşenin bileşen yollarına aktarılmasıyla elde edilebilir. Ve dolaylı bileşenden bileşene kapsama kümesi, tanımlanan bileşen yollarına tüm dolaylı bileşenleri içerecektir.

4.11 BG için Yol Tanımları

Davranış Grafiği, Arabirim Bağlantı Grafiği'nden daha fazla ayrıntıyı temsil eder. Yapısı statik bilgi verirken, revize edilmiş Petri Net'leri ateşleyerek dinamik özellikleri göstermek mümkündür. Bir Behavior Grafiğinde test amaçlı yollar aşağıda verilmiştir; bunlar, j geçiş/yer düğümleri kümesi olan BG yolunda ve i'th arkının i'th düğümü (i+1)'inci düğüme bağladığı j-1 yayları veya tam tersi şekilde bağlanır. Bu yol tanımları boyunca kullanılacaktır.

4.11.1 Bileşen Davranış Yolu

Bileşen Davranış Yolu, bileşen arabirimi alt ağı içinde bir başlangıç noktasından başlayıp bir bitiş noktasında biten bir yoldur. Döngüler varsa, yalnızca bir döngü dahil edilmelidir, böylece bir düğüm en fazla iki kez ziyaret edilir.

İstemci için Bileşen Davranış Yolları;

p 0 – açık – p1 –istek – p2 – sonuç?y – p1 – kapat – p3 – § - p4

p0 – açık – p1 – kapat – p3 – § – p4

p0 – açık – p1 – istek – p2 – sonuç?y – p0 – kapat – kapat – p3 – § - p4

p0 – kapat - kapat – p3 - § - p4

4.11.2 Bileşen Bağlantı Yolu

Bileşen bağlantı yolları, çıkış geçişi diğer bileşen alt ağına giden ve bu bileşen alt ağında ilk sırada biten bir bileşen alt ağının içindeki bir yerden başlayan yollardır.

Komponent bağlantı yolları;

client.p0 – client.open – p101 – server.open – server.p1

client.p1 – client.close – p102 – server.close – server.p3

client.p1 – client.request – p103 – server.request – server.p2

server.p2 – server.result?y – p104 – client.result?y – client.p1

4.12 Regresyon Testi

"Regresyon testi, daha önce test edilen koda yeni hataların eklenip eklenmediğini tespit etmek ve değişikliklerin doğru olduğuna dair güven sağlamak için değiştirilmiş yazılımı doğrulama işlemidir." (Graves, Harrold, Kim, Porter & Rothermel, 2001) Yazılım testinde ve ana yazılım bakım faaliyetinde en önemli konulardan biridir. Yazılım bakımı, aşağıdaki koşullara kadar maliyete tabidir:

Toplam yazılım üretiminin toplam maliyetinin üçte ikisi, dolayısıyla yeniden kullanılabilir bir test setine sahip olmak, yazılım geliştirmedeki en önemli konulardan biridir.

Birçok yazılım geliştirme grubu, çok küçük değişikliklerden sonra bile regresyon testi yapar, çünkü değişiklik sisteme hatalar getirebilir. Yazılımın kanal açma boyutlarıyla, otomatik bir test sürecine sahip olmak bile uzun zaman alır ve genellikle bir gecede çalışır, eğer insan dikkati gerekiyorsa daha da uzun sürer vedaha da fazla sonuç verir. Mutant yazılım üzerine uygulanacak test seti seçilerek bu maliyetin düşürülmesi için çalışmalar yapılmıştır. "Seçici yeniden test teknikleri, mevcut testleri yeniden kullanarak ve değiştirilmiş programın bölümlerini veya test edilmesi gereken spesifikasyonunu tanımlayarak regresyon testinin maliyetini azaltır." (Rothermel & Harrold, 1997) Yazılımın düşük seviyelerinde bir değişiklik olduğunu varsayan ve orijinal test girişlerini ve çıkışlarını kaydetmeye dayanan teknikler tanıtıldı. Diyelim ki P whi ch fonksiyonu var, örneğin daha hızlı olmak için P' olarak değiştirildi.

Yazılım sistemini ilk kez test ederken P'ye yapılan her çağrıyı günlüğe kaydedin, P'nin argümanlarının değerlerini ve P'nin atıfta bulunduğu genel verileri, her çağrıda ve karşılık gelen her dönüşte kaydedin. P'yi P için basit bir test sürücüsüne çağırdığında, P' tarafından üretilen sonuçları, aynı çağrı serisi için P tarafından üretilen kayıtlı sonuçlarla karşılaştırır. (Weide, 2001)

Yukarıdaki temele bağlı olarak bir grup test seçim tekniği,örneğin; minimizasyon tekniği, veri akışı tekniği, güvenli teknik, hepsini yeniden test etme ve rastgele teknikler gibi özümsenmiştir. Regresyon testlerinin maliyeti, seçim sürecinin maliyetine ve seçilen testleri çalıştırmanın maliyetine bağlıdır. Farklı teknikleri karşılaştırmak için bazı deneyler yapılmıştır . Bu deneylerin çoğu, çalışan yazılım sistemine sadece bazı hatalar sokmak ve bu tezin 2.3.3 bölümünde tanıtılan yöntemleri kullanarak mutantlar yaratmaktı.

Tanıttığımız mimarinin grafiksel ve matematiksel gösterimleri (ICG ve BG), regresyon testi için test seçimlerini taşımak için iyi bir alan görevi görebilir. Mutasyon teknikleriyle ortaya çıkan hatalar büyük olasılıkla geliştiricinin sahip olduğu u nit test seti ile yakalanır ve yazılım biriminde bir iç yapı değişikliği varsa, geliştiricinin birim testlerini yeniden uygulaması beklenir. Modül yeniden yazılırsa, testlerin de yeniden yazılması beklenir. Bu kadar kolay olan problem, muhtemelen modifiye edilmiş bir modülün diğer modüllerle olan ilişkilerine bağlı olarak tüm sisteme olası giriş hatalarını yakalamaktır.

Daha önce de belirtildiği gibi, architectural açıklama, otomatik olarak test edilecek gerekli yolları belirleyecek algoritmaları oluşturabilmek için matematiksel arka plan verir. İnsidans matrisi göz önüne alındığında, belirli bir mutant yazılım durumu için, değiştirilmiş bileşenler açıkça görüleceği gibi, çalıştırılacaktestlere karar vermek basittir. Değiştirilmiş bileşenlerin arayüzlerini temsil eden insidans matrisinin ilgili satır ve sütunları işaretlenebilir ve ilgili bileşen de dahil olmak üzere tüm yollar tekrarlanmalıdır. Bileşenin entegrasyonu tamamlandıktan sonra, bileşen arabirimleri de dahil olmak üzere tüm yollar tekrarlanmalıdır.

SYNC bileşeninin değiştirildiğini varsayarsak, insidans matrisinin son satırını ve sütununu işaretleyebiliriz. Sadece simetrik olan bağlantılar görüntülenir ve SYNC.i1'den SS.i2'ye bağlantıyı temsil eder ve sistemin genel yapısı SS konektörünün de dahil edilmesi gerektiğini verir, bu nedenle tekrarlanacak testler şunlardır;

bileşenden bağlayıcı yoluna : a8 (SYNC.i1 - SS.i2)

doğrudan bileşenden bileşen yoluna : a8-b5-a6 (SYNC.i1 - SERVER.i3)

dahili aktarım yolu : b5 (SD.i2 - SD.i1)

Mimari testlerin odak noktası, bileşenler arasındaki ilişkilerdir ve mimari tabanlı regresyon testlerinin odağı, yapılan değişikliklere bağlı olarak etkilenen ilişkileri test etmektir. Bu testler, doğrudan değiştirilen yazılıma bağlı olarak hataları yakalayamayabilir birim testleri düzgün bir şekilde yapılmalı ve bileşenin yeni sürümünün yeterince sağlam olduğundan emin olunmalıdır. Bileşen hazır olduktan sonra entegre edilmeli ve gerekli testler yapılmalıdır. SYNC bileşeninin değiştirilmesi, kapsanacak küçük bir yol kümesi olacaktır, ancak değişiklik verilen örnek için SERVER bileşeninde, testlerin çoğunun mutant sistemin regresyon testi için dahil edileceği anlamına gelecektir.

Bazı sistemler için daha fazla ayrıntı gösteren Davranış Grafiği kullanılabilir. Bölüm 4.11'de verilen yol tanımları ve Şekil 4.14'te verilen sistem için, eğer bileşen kendi iç mimari yapısını değiştirmeden değiştirilirse, tekrarlamamız gereken st'ler, değiştirilmiş bileşenin bileşen davranış patlarını ve bileşen bağlantı yollarını kapsayanlardır.

Bazen değişiklikler, en azından BG vakası için dahili olarak mimari değişikliklere neden olabilir. Bu gibi durumlarda yolların yeniden tanımlanması gerekir, ancak diğer bileşen değiştirilmediği için bağlayıcı alt ağındaki tüm düğümlerin aynı kalacağını unutmayın.

5.SONUÇ

Bu tez, yazılım geliştirme sürecinin farklı seviyelerinde kullanılacak genel test tekniklerini sunmuştur. Daha sonraki yıllarda projelerde yazılımın önemi artmış, böylece test edilmesi daha da önem kazanmıştır ve maalesef bu alanda bilgi eksikliği yaşanmaktadır. Geleneksel test tekniklerinin tam bir özetini vererek, yazılım mimarisine bağlı olarak yeni bir teknik sunarak ve bu tekniği regresyon testi için kullanmanın bir yolunu sunarak, bu tez, onu software geliştirmenin yaşam döngüsüne bağlayan test hakkında eksiksiz bir anket sunar.

İlk olarak, kod testi ve V modeli için iyi bilinen teknikler tanıtıldı. Daha sonra daha az bilinen bir teknik, tam yazılım üzerinden yol kapsamı uygulamaya çalışarak, yazılımın mimari açıklamasına bağlı olarak sunulmuştur. Bu yöntem, geliştiricilerin mimari tasarım aşamasındayken olduğu kadar erken test etmeye başlamalarını sağlayacaktır. Testin geliştirme sürecinin erken aşamalarına kadar gerçekleştirilmesiyle, riskler azaltılabilir. Hatalar ne kadar geç keşfedilirse, maliyeti o kadar artar.

Bu teknik belirli bir mimari Wright için sunulsa da, diğer mimari tanımlara uygulanabilir. Bu amaçla Arabirim Bağlantı Grafiği ve Davranış Grafiğine Eşleme uygulanmalıdır. Resmi bir ADL'ye sahip olmak uygulanan yazılımdır, petri ağlarında kilitlenme analizi gibi mimari doğrulama tekniklerinin kullanılmasına izin verecektir. Mimarinin doğrulanması bu tezin konusu değildir.

Tanımlanan mimari test metodolojisi kullanılarak bir regresyon testi seçim tekniği tanıtılmıştır, çünkü regresyon testi yazılım bakımının en maliyetli kısmıdır, test vakası azaltma önem kazanmaktadır.

Gelecekteki çalışmaların alanlarından biri, mimari tekniği farklı mimari tanımlama dillerine uygulamaktır. Bu teknik, sistemin yalnızca statik özelliklerini test eder, bu nedenle akış üzerinde yığın veya önbellek ıskalamaları gibi hatalar büyük olasılıkla bu metodoloji tarafından yakalanmayacaktır. Gelecekteki bir başka çalışma, bu yöntemi sistemin dinamik özelliklerini kapsayacak şekilde genişletmek ve metodolojinin bu genişletilmiş sürümü için regresyon testi seçimlerini tanımlamak olabilir. Bu teknik, Petri Ağları için geliştirilen teoriler uygulanarak daha da genişletilebilir.

6.KAYNAKÇA

İbrahim, S. (2018, Ocak 09). 12'den fazla kötü amaçlı yazılım türü örneklerle açıklanmıştır (tam liste), MalwareFox: https:// www.malwarefox.com/malware-types/

Albright, D., Brannan, P. ve Walrond, C. (2010). Stuxnet, Natanz Zenginleştirme Tesisi'nde 1.000 santrifüj çıkardı mı? Washington, D.C.: Bilim ve Uluslararası Güvenlik Enstitüsü.

Albright, D., Brannan, P. ve Walrond, C. (2011). Stuxnet Malware ve Natanz: ISIS 22 Aralık 2010 Raporu Güncellemesi. Washington, D.C.: Bilim ve Uluslararası Güvenlik Enstitüsü.

Bahmani, M. (2018, 07 Kasım). AI vs Machine Learning vs Derin Öğrenme Medium: https://medium.com/datadriveninvestor/ai-vs- machine-learning-vs-deep-learning-ba3b3c58c32

Baskin, B., Bradley, T., Faircloth, J., Schiller, C. A., Caruso, K., Piccard, P., . .. Piltzecker,

T. (2006). Bölüm 1 - Casus Yazılımlara Genel Bir Bakış. B. Baskin, T. Bradley, Combating Spyware in the Enterprise (Kuruluşta Casus Yazılımla Mücadele) (s. 1-25). Syngress.

Bencsáth, B., Pék, G., Buttyán, L. ve Félegyházi, M. (2011). Duqu: Vahşi doğada bulunan Stuxnet benzeri bir kötü amaçlı yazılım. Budapeşte: Kriptografi ve Sistem Güvenliği Laboratuvarı (CrySyS).

Bencsáth, B., Pék, G., Buttyán, L. ve Félegyházi, M. (2012). Stuxnet'in Kuzenleri: Duqu, Flame ve Gauss. Gelecekteki İnternet, 971-1003. doi:10.3390/fi4040971

Bostrom, N. (2016). Süper Zeka Yolları, Tehlikeleri, Stratejileri. Oxford: Oxford Üniversitesi Yayınları.

Cameron, J. (Yönetmen). (1984). Terminatör [Sinema Filmi].

Chen, P., Desmet, L. ve Huygens, C. (2014). Gelişmiş Kalıcı Tehditler Üzerine Bir Çalışma

Chien, E. (2010, 12 Kasım). Stuxnet: Bir Atılım. Symantec

Christou, G. (2016). Avrupa Birliği'nde Siber Güvenlik Yönetişim Politikasında Dayanıklılık ve Uyarlanabilirlik . Hampshire: Palgrave Macmillan.

Clarke, R. A. ve Knake, R. (2010). Siber Savaş: Ulusal Güvenliğe Bir Sonraki Tehdit ve Bu Konuda Ne Yapmalı? New York, NY: HarperCollins.

Dando, M. (2015). Sinirbilim İlerlemeleri ve Gelecekteki Savaş. J. dilinde Clausen ve N. Levy, Nöroetik El Kitabı (s. 1785-1800). New York: Springer Science + Business Media Dordrecht.

Davis, S. E. ve Smith, G. A. (2019). Savaşta Transkraniyal Doğru Akım Stimülasyon Kullanımı: Faydalar, Riskler ve Gelecekteki Beklentiler. İnsan Sinirbiliminde Sınırlar, 13, 1-18. doi:10.3389/fnhum.2019.00114

Eggenschwiler, J. ve Silomon, J. (2018). Siber silah norm yapımındaki zorluklar ve fırsatlar. Bilgisayar Dolandırıcılığı ve Güvenliği, 2018(12), 11-18. doi:10.1016/S1361-3723(18)30120-9

Falliere, N., O Murchu, L. ve Chien, E. (2011, Şubat 11). W32. Stuxnet Dosyası.

Erişim tarihi: 15 Haziran 2019, Symantec: https:// www.symantec.com/content/en/us/enterprise/media/security_response/white papers/w32_stuxnet_dossier.pdf

Fontugne, R., Bautista, E., Petrie, C., Nomura, Y., Abry, P., Goncalves, P.,. . . . Aben, E. (2019). BGP Zombies: A Nalysis of Beacons Stuck Routes. D. Choffnes ve

M. Barcellos (Ed.), Pasif ve Aktif Ölçüm. PAM 2019. Bilgisayar Bilimlerinde Ders Notları. 11419, s. 197-209. Cham: Springer.

Ford, R. (1999). Kötü Amaçlı Yazılım: Troya Yeniden Ziyaret Edildi. Bilgisayar ve Güvenlik, 18(2), 105-108. doi:10.1016/S0167-4048(99)80027-3

Gediya, J., Singh, J., Kushwaha, P., Srivastava, R. ve Wang, Z. (2019). 7 - Açık Kaynak Kodlu Yazılım. R. Oshana ve M. Kraeling (Eds.), Gömülü Sistemler için Yazılım Mühendisliği (s. 207-244). Cambridge, MA: Elsevier.

Gibney, A. (Yönetmen). (2014). Zero Days [Sinema Filmi]. Gibson, W. (1984). Neuromancer. New York: As.

Goodin, D. (2016, Eylül 29). Rekor kıran DDoS'un >145k saldırıya uğramış kameralar tarafından teslim edildiği bildirildi. Ars Technica: https://arstechnica.com/information-technology/2016/09/botnet-of-145k-cameras-reportedly-deliver-internets-biggest-ddos-ever/

Hallett, M. (2007). Transkraniyal Manyetik Stimülasyon: Bir Astar. Nöron, 187-199. doi:10.1016/j.neuron.2007.06.026

Herzog, S. (2011). Estonya Siber Saldırılarını Yeniden Ele Almak: Dijital Tehditler ve Çok Uluslu Yanıtlar. Stratejik Güvenlik Dergisi, 4(2), 49-60. doi:10.5038/1944-0472.4.2.3

Uluslararası Telekomünikasyon Birliği. (1994, Temmuz 01). ITU-T Öneriler Veritabanı., Uluslararası Telekomünikasyon Birliği

Kaplan, J. (2016). Yapay Zeka: Herkesin Bilmesi Gerekenler. Yeni York: Oxford Üniversitesi Yayınları.

Keane, S. (2019, 15 Temmuz). Huawei yasağı: Telefonlarının nasıl ve neden ateş altında olduğuna dair tam zaman çizelgesi. Cnet: https:// www.cnet.com/news/huawei-ban- full-timeline-on-how-and-why-its-phones-are-under-fire/

Kuehl, D. T. (2009). Siber Uzaydan Siber Güce: Sorunu Tanımlamak. F. D. içinde. Kramer, S. H. Starr ve L. K. Wentz (Eds.), Siber Güç ve Ulusal Güvenlik (s. 24-42). Washington, D.C.: Potomac Kitapları.

Langner, R. (2013). Bir Santrifüjü öldürmek için. Arlington: Langner Grubu.

Libicki, M. C. (2009). Siber caydırıcılık ve Siber Savaş. Santa Monica, Kaliforniya: RAND. Lindsay, J. R. (2013). Stuxnet ve Siber Savaşın Sınırları. Güvenlik Çalışmaları, 22(3),

365-404. doi:10.1080/09636412.2013.816122

Liska, A. ve Gallo, T. (2017). Dijital Gasp Karşı Savunma Fidye Yazılımı.

Sebastopol, CA: O'Reilly.

Lysne, O. (2018). Huawei ve Snowden Soruları, Güvenilmeyen Satıcılardan Elektronik Ekipman Doğrulanabilir mi? Güvenilmeyen Bir Satıcı Elektronik Ekipmana Güven Oluşturabilir mi? Cham: Springer.

Jin, Z. (2000), Yazılım Mimarisi Tabanlı Test Tekniği, *Yüksek Lisans Fakültesi. arasında. George MasonÜniversitesi*

Cook, T. (1999), Mimarlık Tanımlama Dilleri: Genel Bir Bakış, Microelectronics and Computer Technology Corporation

Graves, T.L., Harrold, M.J., Kim, J.M., Porter, A. & Rothermel, G. (2001), Regresyon Testi Seçim Tekniklerinin Ampirik Bir Çalışması, *Yazılım Mühendisliği ve Metodolojisinde ACM İşlemleri (TOSEM) 10 (* 2), 184-208

Weide, B.W. (2001), "Modüler Regresyon Testi": Bileşen Tabanlı Yazılıma Bağlantılar, *4· ICSE Bileşen Tabanlı Yazılım Mühendisliği Çalıştayı*

Rothermel, G. & Harrold, M.J. (1997), Güvenli, Verimli Bir Regresyon Testi Seçim Tekniği, *Yazılım Mühendisliği ve Metodoloji Lojisinde ACM İşlemleri* (*TOSEM) 6 (*2), 173-210

Özgeçmiş

Adı Soyadı:	Egemen İsa TARİH
E-mail (1):	y210234064@ogr.ikc.edu.tr
E-mail (2):	egementarih@gmail.com

Eğitim:

2012-2016	Bilecik Üniversitesi Elektrik-Elektronik Mühendisliği Bölümü
2019-2020	MEBS Subaylığı Temel Eğitimi

İş Deneyimi:

2017-2018	Milli Eğitim Müdürlüğü İnşaat Emlak Şubesinde Elektrik Mühendisi
2020 - Halen	Hava Kuvvetleri Komutanlığında Bilgi Sistemleri Subayı (2'nci Ana Jet Üs Komutanlığı/Çiğli)