

بسم الله الرحمن الرحيم

شناسایی متدولوژی های مهندسی نرم افزار

با هدف تدوین استاندارد فرآیند توسعه نرم افزار

در پژوهشگاه اطلاعات و مدارک علمی ایران

مجری: صدرا احمدی

همکاران: ناصر فروغی فر

زهره خلیلی

بهار 1386

7	بخش 1- مقدمه
17	بخش 2- بررسی وضعیت موجود معماری نرم افزار در پژوهشگاه اطلاعات و مدارک علمی ایران
23	بخش 3- رویکرد و روشهای شیء گرا
108	بخش 4- رویکرد، قابلیت ها و فن آوری های معماری سرویس گرا
142	بخش 5- بخش پنجم: رویکرد و روش های جنبه گرا
190	بخش 6- روش های چابک در توسعه نرم افزار
267	بخش 7- نتیجه گیری و ارایه چارچوب توسعه نرم افزار در پژوهشگاه اطلاعات و مدارک علمی ایران
292	بخش 9- منابع و ماخذ

7	بخش 1- مقدمه
8	1-1- رویکردهای غیر ساخت یافته
8	2-1- رویکرد ساخت یافته
9	3-1- رویکرد شی گرا
10	4-1- معماری سرویس گرا
12	5-1- رویکرد جنبه گرا (ASPECT ORIENTED)
14	6-1- روش های توسعه چابک نرم افزار
17	بخش 2- بررسی وضعیت موجود معماری نرم افزار در پژوهشگاه اطلاعات و مدارک علمی ایران
20	1-2- سیاست های مورد نظر در زمینه خرید نرم افزارهای کاربردی از بیرون از سازمان
21	2-2- سیاست های مورد نظر در زمینه برون سپاری توسعه نرم افزار در سیستم های خاص منظوره مورد نیاز سازمان
23	بخش 3- رویکرد و روشهای شی گرا
23	1-3- مقدمه
24	2-3- مقدمه ای بر شی گرایی
25	3-2-1- معرفی اصول شی گرایی برای مقابله با پیچیدگی نرم افزار
27	3-3- طراحی شی گرایی
27	3-3-1- تجرید (چکیده سازی)
30	3-3-2- دربرگیری (محصور سازی)
32	3-3-3- واحد بندی
33	3-3-4- سلسله مراتب
36	3-4- مزایای مدل شی و کاربردهای آن (شرکت ایزایران)
38	3-5- آشنائی با مفاهیم اولیه شی گرایی
38	3-5-1- مفاهیم اساسی
43	3-5-2- رابطه بین کلاسها
46	3-6- برنامه نویسی مبتنی بر اجزاء
46	3-6-1- استفاده از معماری مبتنی بر مؤلفه ها
48	3-7- تاریخچه
52	3-8- توسعه تکرار پذیر
52	3-8-1- توسعه تکرار پذیر یعنی چیست؟
53	3-8-2- چرا توسعه تکرار پذیر؟
55	3-8-3- مزایای روش تکرار پذیر
58	3-9- شرح کلی بر متدولوژی RUP
61	3-10- کاربرد RUP
63	3-11- فازهای RUP
۶۳	3-11-1- فاز بررسی
66	3-11-2- فاز تفصیل
69	3-11-3- فاز ساخت

74	12-3- الگوهای کاری
74	12-3-1- الگوی کاری مدلسازی حرفه
76	12-3-2- الگوی کاری نیازمندی ها
78	12-3-3- الگوی کاری تحلیل و طراحی
80	12-3-4- الگوی کاری پیاده سازی
82	12-3-5- الگوی کاری تست
84	12-3-6- الگوی کاری استقرار
86	12-3-7- الگوی کاری مدیریت پیکربندی و تغییرات
87	12-3-8- الگوی کاری مدیریت پروژه
88	12-3-9- الگوی کاری شکل دهی محیط پروژه
91	13-3- معرفی زبان استاندارد مدلسازی UML
91	13-3-1- مقدمه
92	13-3-2- دیگرام های UML
104	14-3- دلایل حرکت به سمت استفاده از COMPONENT در توسعه نرم افزار
105	15-3- مولفه نرم افزاری چیست؟
106	16-3- تفاوت توسعه نرم افزار مبتنی بر مولفه های نرم افزاری با روش های توسعه و برنامه نویسی شیئی گرا

بخش 4- رویکرد، قابلیت ها و فن آوری های معماری سرویس گرا

108	4-1- مقدمه
110	4-2- تاریخچه استفاده مجدد در تولید نرم افزار - افزایش انتزاع
110	4-3- SOA در مقابل معماری های نرم افزاری متعارف
111	4-4- SOA تغییر جهت از برنامه های کاربردی به سمت فرآیندها و سرویس ها
112	4-5- از توابع دانه ریز تا سرویس های دانه درشت
114	4-6- عناصر SOA سازمانی
117	4-7- ارتباط فرایند های کسب و کار با سرویس های کسب و کار
121	4-8- گام های اولیه برای مکان یابی و فراخوانی یک سرویس وب
123	4-9- آیا SOA همان سرویس وب است ؟
124	4-10- توانمندی های SOA
125	4-11- یکپارچه سازی در SOA
126	4-12- یکپارچه سازی نقطه به نقطه در SOA
127	4-13- استفاده از میان افزار یکپارچه سازی در SOA
130	4-14- ESB
131	4-15- پیاده سازی SOA
135	4-16- معماری ترکیبی پیشنهادی
138	4-17- یک الگوی معماری برای یک برنامه کاربردی سازمانی
141	4-18- نتیجه گیری

بخش 5- بخش پنجم: رویکرد و روش های جنبه گرا

142	5-1- مقدمه
-----	------------

143.....	2-5- رهیافت‌های جنبه‌گرا.....
143.....	1-2-5- تعریف.....
145.....	2-2-5- تحلیل جنبه‌گرایی دیدگاه-مبنا.....
145.....	3-5- متد آرکید.....
148.....	1-3-5- مصنوعات آرکید.....
149.....	2-3-5- فرایند آرکید.....
152.....	4-5- تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها.....
152.....	1-4-5- متد تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها.....
153.....	2-4-5- مصنوعات تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها.....
156.....	3-4-5- فرایند تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها.....
157.....	4-4-5- شناسایی و بررسی مباحث تقاطعی با تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها.....
158.....	5-5- مدل نمودن سناریوها بوسیله‌ی جنبه.....
158.....	1-5-5- متد مدل کردن سناریو با مدل جنبه‌ها.....
159.....	2-5-5- مصنوعات مدل کردن سناریوها با جنبه.....
162.....	3-5-5- فرایند مدل نمودن سناریوها توسط جنبه.....
163.....	4-5-5- شناسایی و بررسی مباحث تقاطعی با مدل کردن سناریوها بوسیله‌ی جنبه‌ها.....
164.....	6-5- رهیافت مورد کاربرد مبنای جنبه‌ای.....
164.....	1-6-5- متد مورد کاربرد مبنای جنبه‌ای.....
165.....	2-6-5- مصنوعات رهیافت مورد کاربرد مبنای جنبه‌ای.....
169.....	3-6-5- فرایند رهیافت مورد کاربرد مبنای جنبه‌ای.....
170.....	4-6-5- شناسایی و بررسی مباحث تقاطعی با رهیافت مورد کاربرد مبنای جنبه‌ای.....
171.....	7-5- مقایسه‌ی بین رهیافت‌های مختلف.....
173.....	8-5- جزئیات روش SMA.....
173.....	1-8-5- مقدمه.....
174.....	2-8-5- اثرات متقابل جنبه‌ای.....

190.....	6-1- مقدمه‌ای بر متدلوژی چابک توسعه نرم افزار: 190.....
191.....	1-1-6- تاریخچه متدلوژی چابک توسعه نرم افزار (سایت www.wikipedia.org).....
192.....	2-1-6- بررسی ویژگیها و خصوصیات متدلوژی های چابک توسعه نرم افزار 192.....
200.....	2-6- مقایسه روش چابک با دیگر روشها (سایت WWW.WIKIPEDIA.ORG).....
201.....	1-2-6- روشهای چابک در مقایسه با توسعه تکرار پذیر.....
202.....	2-2-6- روشهای چابک در مقایسه با مدل‌های آبشاری.....
203.....	3-2-6- روشهای چابک در مقایسه با کد نویسی Cowboy.....
205.....	3-6- مروری بر روشهای چابک موجود:.....
206.....	1-3-6- Extreme Programming.....
216.....	2-3-6- Scrum.....
224.....	3-3-6- Crystal.....
234.....	4-3-6- Feature Driven Development.....

241.....	Dynamic Systems Development Method-5-3-6
250.....	Adaptive Software Development-6-3-6
258.....	Open source software development-7-3-6
263.....	نتیجه گیری -4-6
267	بخش 7- نتیجه گیری و ارایه چارچوب توسعه نرم افزار در پژوهشگاه اطلاعات و مدارک علمی ایران
269.....	مقایسه رویکردهای توسعه -1-7
278.....	مقایسه روش های اجرایی توسعه نرم افزار -2-7
291.....	چارچوبی پیشنهادی توسعه نرم افزار در پژوهشگاه اطلاعات و مدارک علمی ایران -3-7
292	بخش 8- منابع و مآخذ

بخش 1 - مقدمه

در طول تاریخ توسعه و تحلیل سیستم‌های نرم‌افزاری سه رویکرد اصلی وجود داشته است:

1. رویکردهای غیر ساخت یافته

2. رویکردهای ساخت یافته

3. رویکردهای شیء‌گرا

و به تازگی رویکردهای

4. معماری سرویس‌گرا

5. رویکرد جنبه‌گرا

هم به آن اضافه شده‌اند. در این میان مجموعه روش‌های توسعه کاربردی نرم‌افزار نیز برای توسعه سریعتر و

پاسخگویی هر چه بیشتر به نیازمندی مشتریان تکامل یافتند و از این منظر نیز

6. روش‌های چابک توسعه نرم‌افزار

به عنوان مجموعه روش‌های عملی توسعه نرم‌افزار مطرح شدند.

در این پژوهش برآنیم که چهار حوزه اخیر (رویکردهای شیء‌گرا، معماری سرویس‌گرا، رویکرد جنبه‌گرا و

روش‌های چابک توسعه نرم‌افزار)، که امروزه در توسعه نرم‌افزارها مورد استفاده قرار می‌گیرند، را شناسایی و

نقاط قوت و ضعف هر یک را بررسی نماییم تا در نهایت به چارچوبه‌ای برای انتخاب روش توسعه نرم‌افزار

متناسب با شرایط آن برسیم.

در بخش‌های مختلف این گزارش مطالب زیر را ارایه می‌نماییم:

1- مقدمه: در این بخش تعریف اولیه هر یک از چهار حوزه مطروحه را بیان و منشاء پیدایشش را عنوان

می‌کنیم.

2- بررسی وضعیت موجود معماری نرم‌افزار در پژوهشگاه اطلاعات و مدارک علمی ایران: در

این بخش دسته‌بندی جامعی از نرم‌افزارهای موجود و در حال توسعه در پژوهشگاه اطلاعات و مدارک

علمی ایران ارایه می‌گردد.

3- بخش‌های سوم، چهارم، پنجم و ششم: در هر یک از این بخش‌ها به معرفی دقیق یکی از روش‌های مذکور می‌پردازیم.

1-1- رویکردهای غیر ساخت‌یافته

این رویکردها تا قبل از دهه‌ی 70 میلادی وجود داشتند. در اصل آنها رویکرد خاصی نبودند و بیشتر نظریات شخصی یک برنامه‌نویس برای ساخت یکی از برنامه‌های خود به حساب می‌آمدند. این دوره در اصل به علت نبود دیدگاهی کامل از تولید و توسعه‌ی سیستم‌ها بوجود آمده بود و می‌توان آن دوران را دوران سرگردانی نیز نام نهاد. از مشخصات این دوره این است که در آن هر سیستمی که پیاده‌سازی می‌شد قابلیت استفاده‌ی مجدد را نداشته و به جز برنامه‌نویس آن هیچ‌کس از منطق آن سردر نمی‌آورد.

1-2- رویکرد ساخت‌یافته

در اواخر دهه‌ی 70 تا اواسط دهه‌ی 80 میلادی این دیدگاه در تولید و توسعه‌ی سیستم‌ها دیدگاه قالبی بود. در این دیدگاه هر سیستم را از لحاظ عملیاتی به صورت درختی، آنقدر تقسیم می‌کردند تا به اجزاء اتمیک اجرایی برسند. نام این اجزاء اتمیک را برحسب موقعیت متد، روتین، روال و ... می‌نامیدند. که اگر تمامی این متدها را کنار هم می‌گذاشتند، می‌توانستند تمامی عملیات‌های سیستم را ببینند. در همین حال یک تحلیلگر و طراح پایگاه‌داده‌ها نیز ساختار پایگاه داده را مشخص می‌نمود. مزایای این رویکرد به شرح زیر است:

- برای سیستم‌های کوچک بسیار مناسب است.
- تمامی عملیات سیستم را به صورت تفکیک شده می‌توان مشاهده نمود.
- ایجاد اطلاعات عملیات در سطوح مختلف، امکان بررسی‌های گوناگون را از دید اشخاص مختلف امکان پذیر می‌سازد.

از معایب این رویکرد می‌توان به مسایل زیر اشاره نمود:

- جریان داده ها در برنامه (نه پایگاه داده) به صورت غیر تفکیکی و درهم تنیده است و اکثر داده ها یا دارای بازه‌ی دید درون متدی‌اند و یا دارای دید عمومی هستند. در حالی که در عمل ما داده‌هایی را داریم که تنها باهم ارتباط دارند و می‌توانند از دید عمومی خارج شوند. ولی رویکرد ساخت یافته تنها به تفکیک عملیاتی می‌پردازد. به همین دلیل، سیستم‌هایی که بزرگ باشند و در آنها جریان داده‌های زیادی وجود داشته باشد با این رویکرد به مشکل برخورد می‌کنند.
 - به همان علت اولی، چون داده‌های در جریان، از هم جدا نشده است؛ نمی‌توان بخش‌های مختلف تولید را به گروه‌های کاملاً مجزا از هم سپرد، زیرا هر گروه یا فرد می‌تواند در حال پیاده‌سازی، بر روی داده‌های دیگران اثرگذار باشد.
- به خاطر این مشکلات بود که برای رفع آنها به سمت رویکردهای جدید رفتند.

1-3- رویکرد شی‌گرا

متدولوژی شی‌گرایی از روش‌های شی‌گرایی که مدعی نوآوری در زمینه کنترل پیچیدگی نرم‌افزار هستند، استفاده می‌نماید. روشهای ارائه شده قبل از روش شی‌گرایی (که آنها را روشهای سنتی می‌نامیم) توانمندی لازم برای طرح یک نگرش عمیق و در عین حال ساده به سیستم‌ها، به میزانی که در روش‌های شی‌گرا مطرح بوده است، را نداشته‌اند. بعلاوه، تکنیک‌های سنتی موجود توان پاسخگویی به نیاز در طراحی و پیاده‌سازی سیستم‌های با مقیاس بسیار بزرگ را نداشته‌اند. برای مثال در مورد طراحی ساخت یافته¹ که در عمل بیشتر از بقیه روش‌های طراحی مورد استفاده قرار گرفته است، کاربردهای عملی نشان داده هنگامیکه تعداد خطوط برنامه از مرز صد هزار خط فراتر رود، کنترل برنامه که به شیوه ساخت یافته طراحی و پیاده‌سازی شده مشکل می‌گردد. سیستم‌های بسیار بزرگ و کاربردهای جدیدی که روز به روز همراه با پیشرفت سخت‌افزارها مطرح شده‌اند نیازمند ساختاری برای مدیریت مدل‌های پیچیده‌تری بوده‌اند که، به نظر می‌رسد، تنها مکانیزم‌های موجود در تکنولوژی شی‌گرایی پتانسیل برخورد با گستردگی و پیچیدگی این سیستم‌ها را دارند. ایده اصلی

¹Structured Design

برای فراهم‌آوری این پتانسیل تکیه بر مفهوم قابلیت استفاده مجدد² و تعریف وساخت مؤلفه‌های استاندارد با قابلیت کاربرد در محیط‌های مختلف است.

متد شی‌گرایی یک راه متفاوت مشاهده برنامه‌هاست. با متد شی‌گرایی می‌توان یک برنامه را به قطعات بسیار کوچک یا آبجکت‌هایی³ تقسیم کرد که تا اندازه‌ای مستقل از یکدیگر باشند. به محض اینکه تعدادی آبجکت‌های اساسی را در دنیای کامپیوتر ساخته شد، می‌توان به سادگی آنها را کنار هم گذاشت تا برنامه‌های جدید را ایجاد نمود.

یکی از امتیازات اساسی متدهای شی‌گرا این است که می‌توان یکبار اجزاء را ساخته و بارها و بارها استفاده کنید. همچنین می‌توان از یک قطعه طرح یا کد شی‌گرایی در نرم‌افزارهای مختلف همچون نرم‌افزار حسابداری یا یک نرم‌افزار بازرگانی یا یک نرم‌افزار پردازش سفارش استفاده نمود.

1-4- معماری سرویس‌گرا

در چند دهه اخیر پیچیدگی نرم‌افزارها روز بروز بیشتر شده و تقاضا برای نرم‌افزارهای قدرتمندتر افزایش یافته است. در این میان، به نظر می‌رسد که روشهای قدیمی (ساخت یافته و شی‌گرا) جوابگوی نیازهای در حال رشد کنونی نیستند و نیاز به ایجاد و بکارگیری روش‌هایی است که بوسیله آنها بتوان بر این پیچیدگی‌ها در زمان‌هایی کوتاه‌تر غلبه کرد. از طرفی امکان کنار گذاشتن سیستم‌های نرم‌افزاری موجود که تا به حال مشغول سرویس‌دهی به مشتریان بوده‌اند، وجود ندارد و می‌بایست سیستم‌های جدید را بصورت یکپارچه و در کنار سیستم‌های قبلی بوجود آورد. معماری سرویس‌گرا، با تکیه بر محاسبات توزیع شده و بر پایه شبکه‌ها و لایه‌های میانی و همچنین زبان‌هایی که تولید نرم‌افزارهای توزیع شده را فراهم می‌کنند، به‌عنوان راه حلی مناسب جهت از میان برداشتن مشکلات و مسائل مذکور مطرح گردیده است.

² Reusability

³ Objects

تعاریف گوناگونی از معماری سرویس گرا ارائه شده است که از جمله آن‌ها می‌توان به تعاریف زیر اشاره کرد: مجموعه قوانین، سیاست‌ها و چهارچوب‌هایی که نرم‌افزارها را قادر می‌سازد تا عملکرد خود را از طریق مجموعه سرویس‌های مجزا و در عین حال مربوط به هم در اختیار سایر درخواست کنندگان قرار دهند تا بتوانند بدون اطلاع از نحوه پیاده‌سازی و تنها از طریق رابط‌های استاندارد و تعریف شده، این سرویس‌ها را پیدا کرده و فراخوانی نمایند. و یا معماری سرویس گرا روشی برای ساخت سیستم‌های توزیع شده‌ای است که در آن‌ها عملکرد سیستم بصورت سرویس در اختیار کاربران و یا سایر سرویس‌ها قرار می‌گیرد.

از دیگر تعاریف ارائه شده می‌توان به "واحدهای نرم افزاری آماده در شبکه (Network-available Software Unit)" یا "سرویس‌های سطح حرفه (services Business-level)" اشاره کرد. در حال حاضر، تکنولوژی سرویس‌های وب (Web Services) و پیاده‌سازی نمونه‌های موفق از آن، نشان داده است که SOA می‌تواند به عنوان راه حلی عملی و دست یافتنی در طراحی سیستم‌های جدید و یکپارچه - سازی سیستم‌های بزرگ موجود، مطرح گردد. البته ذکر تفاوت سرویس‌های وب و SOA در اینجا لازم به نظر می‌رسد.

سرویس‌های وب مجموعه‌ای از تکنولوژی‌هایی همچون WSDL, SOAP, XML و UDDI می‌باشد که امکان ارائه راه حل و برنامه‌نویسی جهت رفع مشکلی خاص را فراهم می‌نماید. در حالی که SOA یک معماریست و از مجموعه‌ی مشخصی از تکنولوژی‌ها فراتر می‌باشد. اگرچه SOA بر اساس این تکنولوژی‌ها راه حل ارائه می‌نماید، اما در عین حال مستقل از هر یک از آنها است. آنچه اهمیت دارد تعریف سرویس به عنوان مهمترین عنصر این معماری می‌باشد.

سرویس؛ رفتار قراردادی تعریف شده‌ایست که هر قطعه‌ای می‌تواند آن را جهت استفاده سایر قطعات در سیستم تهیه و پیاده‌سازی نماید.

در این معماری، همه توابع به عنوان سرویس تعریف می‌گردند. این توابع شامل توابع حرفه (Business functions) و تراکنش‌های حرفه (Business transactions) می‌باشند که تراکنش‌های حرفه خود شامل

توابع سطح پایین (Low-level functions) و توابع سیستمی سرویس‌ها (System service functions) هستند.

سرویس‌ها به صورت مستقل طراحی و پیاده‌سازی شده و به عنوان جعبه سیاه عمل می‌نمایند. قطعات دیگر در خارج از این قطعه نیازی به دانستن نحوه انجام کار در این سرویس را ندارند و تنها به نتیجه آن نیازمندند. قطعات، سرویس‌های خود را از طریق رابط‌ها (interface) در اختیار قطعات دیگر قرار می‌دهند که این رابط‌ها قابل دستیابی و فراخوانی هستند، بدون اینکه محل قرار گیری آنها اهمیت داشته باشد (رابط‌های محلی یا دور). در ضمن این رابط‌ها می‌توانند به همان نرم افزار کاربردی یا به آدرسی در محل دیگری از شبکه مرتبط باشند.

رابط‌ها به عنوان کلیدی در برقراری این ارتباط‌ها هستند و از طریق آن‌ها نوع پارامترهای ورودی و نتایج (خروجی) مشخص می‌گردد. بعلاوه، با ایجاد قابلیت توزیع شدگی به شکلی مناسب و بدون وابستگی زیاد، می‌توان سرویس‌ها را در قسمت‌های مختلف شبکه و به صورت بعضاً تکراری (مخصوصاً برای سرویس‌های مهم) قرار داد، تا این سرویس‌ها همیشه در دسترس بوده و در صورت زیاد شدن درخواست‌ها بتوان با استفاده از تکنیک‌های Load Balancing به تمامی آنها به خوبی پاسخ داد.

5-1- رویکرد جنبه گرا (Aspect Oriented)

رویکرد جنبه گرا نیز همانند معماری سرویس‌گرا از رویکردهای جدیدی است که برای پاسخ‌گویی رفع مشکلات روش‌های ساخت یافته و شیء گرا توسعه یافته است. به علت تازگی مطالب این حوزه هنوز یک اجماع جامع و مانع از این مفاهیم در بین صاحب نظران بوجود نیامده است ولی همگی به وجود معضلاتی در شیوه‌های قدیمی اذعان دارند و زمان را برای تغییرات جدید مناسب می‌دانند. همچنین همگی به وجود مفهومی به نام جنبه‌ها در سیستم‌های کنونی باور داشته و به دنبال رهیافت‌هایی برای جداسازی این مفاهیم در بخش‌های مختلف تولید و توسعه‌ی سیستم‌ها هستند. ولی این رهیافت‌ها در بین نظریه پردازان هنوز به صورتی واحد

درنیامده است و هنوز بحث‌های متعددی در این زمینه وجود دارد. ولی به هر حال پیش بینی می‌شود که رهیافت‌های جنبه گرا مانند رهیافت شیء‌گرایی در سالهای 89 میلادی جای خود را در صنعت IT به زودی باز نمایند. هم‌اکنون زبان‌های برنامه نویسی بسیاری در جهت حمایت از این رهیافت‌ها بوجود آمده است که اکثرا به صورت متن باز (Open Source) هستند. از میان این برنامه‌ها دو زبان اصلی بزرگترین نقش را در حال حاضر بازی می‌کنند که هریک دیدگاه خاص خود را برای اجرای سیستم‌ها دارند. البته هردوی این زبانها در اصل ادامه دهنده‌ی راه جاوا هستند. (مانند C++ که ادامه دهنده‌ی راه C بود). این دو زبان عبارتند از: AspectJ و AspectWerkz.

تکنیک‌های مهندسی نیازمندی‌ای که به صورت واضح سعی در شناسایی اهمیت توضیح دادن نیازمندی‌های تقاطعی عملیاتی، غیرعملیاتی و آنهایی که غیر تقاطعی‌اند، دارند، به عنوان رهیافت‌های مهندسی نیازمندی جنبه‌گرا شناخته می‌شود. دلیل بوجود آمدن این رهیافت‌ها سه فاکتور اصلی بوده است:

- نیاز به ترکیب
- نیاز به ردیابی
- توسعه‌ی تکنولوژی‌های جدید

اولین فاکتور تاثیرگذار بر رهیافت‌های جنبه‌گرا، این واقعیت است که در اکثر متدولوژی‌های نیازمندی، ترکیب مصنوعات (مانند دیدگاه‌ها، اهداف، زیراهداف و مورد کاربردها، در رهیافت‌های مربوطه‌ی خود)، یکی از گلوگاه‌های اصلی است. برای مثال، بسیار مشکل است که دیدگاه‌های ذی‌نفعان شخصی را برای ساخت سیستم کلی، یکپارچه نمود. به‌عنوان مثال دیگر، بسیار سخت است که اهداف معمول امنیتی و زمان‌پاسخ، با دریافت توزیع شده‌ی داده، یکپارچه شود. سختی این کار به علت پراکندگی مباحث، در مصنوعات متفاوت، (برای مثال مباحث امنیتی در بسیاری از دیدگاه‌ها وجود دارد) و تداخل درونی میان مباحث، (برای مثال امنیت و زمان پاسخ برروی هم اثرگذارند و روی دریافت داده محدودیت ایجاد می‌کنند) است. این دو مورد در یکپارچه‌سازی، از یک دید ماژولار به سیستم، و تصمیم در رابطه با مباحثی که در چندین مستند مختلف آمده است، جلوگیری می‌کند. (مبحث امنیت در چندین دیدگاه مختلف آمده است، ولی هیچ نمود ماژولاری، ندارد).

نداشتن قابلیت تصمیمی صحیح، درباره‌ی مباحث تقاطعی در نیازمندی‌ها، باعث می‌شود، درک ما از نیازمندی‌های ذینفعان و انتظارات پایان نیابد.

فاکتور دوم، همان قابلیت ردیابی مباحث تقاطعی، در طول چرخه‌ی عمر سیستم نرم‌افزاری است. اینکه ما فقط نیازهای عملیاتی و غیرعملیاتی تقاطعی را شناسایی نموده و درباره‌ی آن‌ها تصمیم بگیریم، کافی نیست. ما نیاز داریم، زمانی که این نیازها را شناسایی کردیم، و درباره‌ی نقاط تعادل آنها تصمیم‌گیری نمودیم، بتوانیم آنها را در فازهای معماری، طراحی، پیاده‌سازی، حتی نگهداری و ارتقاء هم ردیابی نماییم. ماژولار نمودن خصوصیات تقاطعی در سطح نیازمندی‌ها، اولین گام به سمت قابلیت ردیابی آنهاست.

فاکتور سوم، ورود زبان‌های جنبه‌گراست. این زبان‌ها، برای ماژولار نمودن مفاهیم توزیع شده و به هم تنیده (تقاطع)، مکانیزم‌های جدید ترکیب (نقطه‌ی تقاطع و نقطه‌ی اتصال) را ارائه داده‌اند. شناسایی و رسیدگی به این خصوصیات تقاطعی، در مراحل اولیه‌ی تولید نرم‌افزار، باعث ایجاد یکرختی در تمام مراحل می‌گردد. همچنین، قابلیت ردیابی که در بالا اشاره شد افزوده می‌شود.

بنابراین، رهیافت‌های مهندسی نیازمندی‌هایی که جنبه‌گرا هستند، کانون توجه خود را به رسیدگی، بررسی، ترکیب‌کردن و درنهایت ردیابی سیستماتیک نیازمندی‌های عملیاتی و غیرعملیاتی تقاطعی با مکانیزم‌های تجزیه، نمایش و ترکیب مناسبی که مخصوص زمینه‌ی مهندسی نیازمندی‌ها است، معطوف نموده‌اند.

1-6- روش‌های توسعه چابک نرم‌افزار

مطالعه نشان می‌دهد که در روند توسعه نرم‌افزار متدلوژی‌های سنتی که در حوزه علم مهندسی نرم‌افزار وجود دارند در عمل به منظور توسعه سریع یک نرم‌افزار دارای کارایی کمی هستند؛ چرا که استدلال می‌شود که متدلوژی‌های سنتی در جزئیات ساختارمند عمل می‌نمایند که چنین سطحی از توسعه زمانی که توسعه‌دهندگان نرم‌افزار قصد دارند به صورت سریع نرم‌افزار مورد نظر را تولید کنند، آنها را با مشکل مواجه می‌کند.

پس از تجربیات گوناگون توسعه‌دهندگان به این نتیجه رسیدند که لازم است روش‌های جدیدی برای توسعه نرم‌افزارها ارائه گردد و به همین منظور روش‌های توسعه چابک، انتشار یافتند که درواقع معرفی این نوع از متدولوژی‌های توسعه نرم‌افزار حرکتی نوین در عرصه علوم مهندسی نرم‌افزار می‌باشد.

رویکرد توسعه نرم‌افزار به صورت چابک تاکید بر افراد، تبادل، کارکردن نرم‌افزار، مشارکت مشتریان و انعطاف‌پذیری نسبت به تغییرات فرایند، ابزار، قرارداد و طرح‌ها دارد.

نتایج بررسی‌ها نشان می‌دهد که توسعه نرم‌افزار با استفاده از روش‌های چابک از ویژگی‌های زیر برخوردار هستند:

- توسعه نرم‌افزار به صورت افزایشی: به این معنی که در اینگونه از روش‌ها تولیدات کوچک نرم‌افزار با چرخه‌های سریع انجام می‌پذیرد.
- ایجاد سطح مطلوبی از همکاری: به این معنی که در این روش‌ها مشتریان و توسعه‌دهندگان پیوسته با یکدیگر با ارتباطات نزدیک دارند.
- ایجاد سازگاری: به این معنی که نرم‌افزارهایی که با استفاده از این متد توسعه می‌یابند قادر به ایجاد تطبیق با تغییرات آتی می‌باشند.

هدف از استفاده از این نوع متدولوژی در واقع قادر ساختن افراد حرفه‌ای در نرم‌افزار، پروژه‌ها و سازمان‌ها برای انتخاب و به کار بردن روش درست و مناسب در زمان مناسب می‌باشد. در نتیجه تفکر چابک، یک نگرش فرد محور برای توسعه نرم‌افزار می‌باشد.

بررسی‌ها نشان دهنده این نکته است که محققین و متخصصین حوزه مهندسی نرم‌افزار بر این باورند که روش‌های توسعه چابک نرم‌افزار یک راه بدیع از مسائل مهندسی نرم‌افزار می‌باشد در حالی که این روشها قادر به حل همه مسائل نمی‌باشند.

در زیر به چند نمونه از مطالب مهم و ضروری در ارتباط با روشهای چابک اشاره داریم:

- برای پیشرفت در پروژه‌های چابک تجربه دارای اهمیت بسیار زیادی می‌باشد، اما تجربه واقعی از تجربه ای که از روش‌های چابک بدست می‌آید مهم‌تر می‌باشد. طی تخمینی 25% تا 33% از پروژه

های شخصی باید از لحاظ شایستگی و استفاده از تجارب دیگران بررسی شود ولی این مقدار تا 10% می تواند کاهش یابد.

- روش‌های چابک به آموزش رسمی کمتر نسبت به روش‌های سنتی دارند. استفاده از یک جفت برنامه نویس برای توسعه هر بخش نیاز به آموزش را کاهش می‌دهد چراکه هر کدام از اعضای این تیم دونفره می‌توانند در موارد خاص نقش مربی نفر دوم را ایفا کنند.

- اعتبار و ایمنی با استفاده از روش‌های چابک در پروژه‌های بحرانی فراهم می‌شود. در این نوع از متدها نیازمندی‌های عملکرد به صورت واضح تعیین می‌شوند و برنامه‌ریزی تست به صورت دقیق تشریح می‌شود.

- 3 فاکتور مهم در موفقیت پروژه‌هایی که از این نوع از متدولوژی‌ها به منظور توسعه استفاده می‌کنند عبارتند از: فرهنگ، افراد و ارتباطات. تاثیر این 3 فاکتور به این ترتیب است که روش‌های چابک برای دستیابی به موفقیت در استفاده نیاز برای به پشتیبانی فرهنگ دارند. همچنین تیم‌هایی که در این روش‌ها مورد استفاده قرار می‌گیرند لازم است دارای صلاحیت و انسجام لازم برای توسعه باشند. برای ایجاد تیم‌های کاری در این روش‌های چابک از تعداد افراد کمتر و با شایستگی بیشتر استفاده می‌شود. تیم‌ها مورد استفاده باید به صورتی آرایش یابند که در میان ارتباطات به سادگی ایجاد شود و مجموعه تیم آنرا را پشتیبانی کنند.

- بازسازی و بازنگری بر روی نرم‌افزار باید مکرراً و به صورت تکرارپذیر انجام شود بازسازی با مقیاس بالا مشکل نمی‌باشد و با استفاده از روش‌های چابک امکان پذیر می‌باشد.

در این روش‌ها لازم است پیش از توسعه نرم‌افزار تیم توسعه دهنده هزینه مد نظر خود را برای مستندسازی تعیین کند. بسیاری از سازمان‌ها بیشتر از سطحی که مستندات نیاز دارند از تیم مجری تولید مستندات را مطالبه می‌نمایند در صورتی در توسعه نرم‌افزار با استفاده از این متدولوژی لازم است که تیم هدف خود را بر ایجاد ارتباطات موثر قرار دهد و مستند سازی به عنوان آخرین گزینه مطرح باشد.

بخش 2 - بررسی وضعیت موجود معماری نرم افزار در پژوهشگاه اطلاعات و

مدارک علمی ایران

موضوع فناوری اطلاعات به عنوان یکی از موضوعات استراتژیک پژوهشگاه اطلاعات و مدارک علمی ایران از جایگاه ویژه‌ای برخوردار است. در راستای اجرای طرح‌ها و برنامه‌های مرتبط با موضوع استراتژیک گسترش فناوری اطلاعات، در پژوهشگاه فعالیت‌های متعددی صورت گرفته است. از مهم‌ترین فعالیت‌هایی که می‌توانن جهت اجرایی سازی این استراتژی نام برد، توسعه سیستم‌های اطلاعاتی متناسب با نیازمندی‌های سازمان است. با توجه به ماهیت کاری سازمان می‌توان نرم افزارهای مورد نیاز سازمان را به دو دسته کلی

1. نرم افزارهای عمومی

2. نرم افزارهای خاص منظوره

دسته بندی نمود. طی چند سال گذشته شیوه توسعه سیستم اطلاعاتی در سازمان به سه صورت توسعه سیستم به وسیله منابع انسانی درون سازمان، برون سپاری و در مورد سیستم‌های عمومی همچون سیستم مالی که مستقل از نوع سازمان، رویه‌های مشخص و استاندارد دارد، از بسته‌های نرم افزاری آماده استفاده نموده است. در سازمان‌ها روش‌هایی که تا کنون برای استفاده از نرم افزار و اجرایی سازی استراتژی توسعه فناوری اطلاعات استفاده شده‌اند عبارتند از:

1. توسعه سیستم اطلاعاتی به وسیله منابع انسانی درون سازمان.

2. توسعه سیستم اطلاعاتی با استفاده از برون سپاری.

3. خرید بسته‌های نرم افزاری و بومی سازی آنها با توجه به سطح نیاز سازمان.

نمودار زیر وضعیت موجود تهیه نرم افزار در سازمان را از 5 بعد

1. دسته بندی از منظر کاربرد سیستم

2. دسته بندی از منظر کاربرد سیستم

3. دسته بندی از منظر نحوه تهیه سیستم

4. دسته بندی از منظر سائز سیستم

5. دسته بندی از منظر تکنولوژی توسعه سیستم

نشان می دهد.

با توجه به دسته بندی اول یعنی دسته بندی از منظر کاربرد سیستم، در حال حاضر سیاست پژوهشگاه برای مکانیزه نمودن رویه های کاری و تحقق استراتژی فناوری اطلاعات را به صورت کلی می توان شامل دو حوزه زیر دانست:

1. خرید نرم افزار در سیستم های عمومی
2. برون سپاری توسعه سیستم در سیستم های خاص منظوره

با توجه به این نکته لازم است در هر کدام از این دو حوزه اصلی موارد زیر جهت انجام بهینه کار با توجه به نوع نرم افزار مورد نیاز سازمان، در زمان عقد قرار داد خرید یا توسعه سیستم در نظر گرفته شود.

2-1- سیاست های مورد نظر در زمینه خرید نرم افزارهای کاربردی از بیرون از سازمان

1. در صورتی که نرم افزار مورد نظر برای یک بخش یا واحد خاص از سازمان خریداری می گردد، لازم است پیش از خریداری نرم افزار نیازمندی های دقیق نرم افزار مورد نیاز توسط کارشناسان و خبرگان آن حوزه در سازمان تهیه گردد. با انج امان کار دو نکته زیر در تهیه نرم افزار شفاف می گردد:

ü حداکثر میزان تغییراتی که سازمان می تواند در انطباق با نرم افزار در خود به وجود آورد.

ü نکات کلیدی و غیر قابل تغییر در فرآیند کاری و فعالیت هایی که لازم است نرم افزارهای مدنظر خود را مطابق با آن تغییر و تطبیق دهند.

2. در صورتی که سطح استفاده از نرم افزار مورد نظر حالتی سازمان شمول داشته باشد، واحد تحقیق و توسعه موظف به تدوین شرح نیازمندی های نرم افزار می باشد.

3. میزان زمان مورد نیاز و روش پشتیبانی از نرم‌افزار توسط روند واحد خدمات ماشینی معین شده و در قرارداد مربوطه ذکر گردد

2-2- سیاست‌های مورد نظر در زمینه برون سپاری توسعه نرم‌افزار در سیستم‌های خاص منظوره مورد نیاز سازمان

1. در صورتی که نرم‌افزار مورد نظر برای یک بخش یا واحد خاص از سازمان خریداری می‌گردد، لازم است پیش از خریداری نرم‌افزار نیازمندی‌های دقیق نرم‌افزار در دو سطح فرآیندی و سیستمی به صورت کامل توسط کارشناسان و خبرگان آن حوزه در سازمان در قالب RFP نرم‌افزار تشریح گردیده و به عنوان پیوست فنی قرارداد مورد استفاده قرار می‌گیرد. با انجام چنین فعالیتی محدوده پروژه کاملاً مشخص شده و دو نکته زیر در توسعه نرم‌افزار شفاف می‌گردد:

ü سازمان دقیقاً می‌داند در زمان تحویل نرم‌افزار قرار است چه چیزی را از توسعه دهنده به عنوان محصول نهایی تحویل گیرد.

ü مجری محدوده کاری خود را دقیقاً با این سند شناسایی کرده و کاملاً منطبق با آن عمل می‌کند.

تبصره 1: در صورتی که حدود قیمت نرم‌افزاری که قرار است برای تولید برونسپاری گردد در محدوده بالای 200.000.000 ریال بوده و یا سازمان ملزم به برگزاری مناقصه برای واگذاری آن به پیمانکار باشد، در این صورت سازمان ملزم به تهیه RFP کامل برای پروژه می‌باشد در غیر این صورت تدوین شرح نیازمندی‌های سیستم با توجه به نوع مسئله با همکاری واحد مربوطه، خدمات ماشینی و تحقیق و توسعه به عنوان پیوست فنی قرارداد، کافی بوده و نیاز به تدوین RFP نمی‌باشد.

2. پیش از آغاز چنین پروژه‌ای و در زمان تدوین RFP ناظر پروژه با هماهنگی واحد درخواست دهنده، خدمات ماشینی و دفتر تحقیق و توسعه مشخص می‌گردد. ناظر اینگونه

از پروژه‌های نرم‌افزاری لازم است از دو سطح صلاحیت زیر برخوردار باشد:

ü نسبت به محدوده علمی که قرار است مکانیزه شود کاملاً مسلط باشد.

ü دارای سطح دانش فنی کافی در حوزه توسعه نرم‌افزار باشد.

تبصره 2: به دلیل اینکه در سیستم‌های خاص منظوره یافتن فردی که هردوی این مشخصات را با هم داشته باشد مشکل است، تیم تعیین ناظر می‌تواند دو نفر را یکی با صلاحیت علمی در حوزه فعالیت مورد نظر و دیگر دارای صلاحیت علمی در حوزه توسعه نرم‌افزار را مشخص نموده و روند تایید پروژه توسط آنها را به مقتضی هر نوع از پروژه‌ها مشخص نماید.

تبصره 3: لازم است پیش از تدوین RFP در برای یک پروژه نرم‌افزاری، در یک فرآیند مطالعه محدوده پروژه به صورت کلی مشخص گردد و سازمان پس از تایید محدوده به تدوین RFP بپردازد تا از هزینه‌کردهای اضافی که به علت عدم شناخت صحیح از محدوده پروژه حادث می‌شود دچار نگردد.

3. در صورتی که سطح استفاده از نرم‌افزار مورد نظر حالتی سازمان شمول داشته باشد، واحد تحقیق و توسعه موظف به تدوین RFP نرم‌افزار می‌باشد. در این خصوص این واحد با نظر خدمات ماشینی با توجه به شرایط ذکر شده در بند قبل به تعیین ناظر پروژه می‌پردازد.

4. میزان زمان مورد نیاز و روش پشتیبانی از نرم‌افزار توسط روند واحد خدمات ماشینی معین شده و در قرارداد مربوطه ذکر گردد

بخش 3 - رویکرد و روشهای شیء گرا

3-1- مقدمه

در دهه 1960 متخصصین حوزه علوم نرم افزار به حجم زیاد پیچیدگی در حوزه تولید نرم افزار پی بردند و سعی کردند تا راه حلی جامع برای رفع این پیچیدگی ارائه نمایند، همانطور که عنوان شد نتیجه بررسی های آنها دستیابی به دانش مهندسی نرم افزار بود. در این بخش از گزارش به دنبال آن هستیم تا متدولوژی های شیء گرا را که از یکی دیگر از متدولوژی هایی که در این حوزه مطرح است را مورد بررسی قرار دهیم. (شرکت ایزایران)

متدولوژی شیء گرائی از روشهای شیء گرائی که مدعی نوآوری در زمینه کنترل پیچیدگی نرم افزار هستند، استفاده می نماید. روشهای ارائه شده قبل از روش شیء گرائی (که آنها را روشهای سنتی می نامیم) توانمندی لازم برای طرح یک نگرش عمیق و در عین حال ساده به سیستم ها، به میزانی که در روش های شیء گرا مطرح بوده است، را نداشته اند. بعلاوه، تکنیک های سنتی موجود توان پاسخگویی به نیاز در طراحی و پیاده سازی سیستم های با مقیاس بسیار بزرگ را نداشته اند. برای مثال در مورد طراحی ساخت یافته⁴ که در عمل بیشتر از بقیه روش های طراحی مورد استفاده قرار گرفته است، کاربردهای عملی نشان داده هنگامیکه تعداد خطوط برنامه از مرز صد هزار خط فراتر رود، کنترل برنامه که به شیوه ساخت یافته طراحی و پیاده سازی شده مشکل می گردد. سیستم های بسیار بزرگ و کاربردهای جدیدی که روز به روز همراه با پیشرفت سخت افزارها مطرح شده اند نیازمند ساختاری برای مدیریت مدل های پیچیده تری بوده اند که، به نظر می رسد، تنها مکانیزم های موجود در تکنولوژی شیء گرائی پتانسیل برخورد با گستردگی و پیچیدگی این سیستم ها را دارند. ایده اصلی برای فراهم آوری این پتانسیل تکیه بر مفهوم قابلیت استفاده مجدد⁵ و تعریف و ساخت مؤلفه های استاندارد با قابلیت کاربرد در محیط های مختلف است. (شرکت ایزایران)

⁴Structured Design

⁵Reusability

در ادامه سعی بر آن است تا ابتدا به معرفی اصول شیء‌گرایی به بررسی مطرح‌ترین متدولوژی این دسته یعنی متدولوژی RUP پرداخته شود و بخش‌های مخالف آن به صورت اجمالی معرفی گردد. لازم به ذکر است که در این بخش به دلیل اینکه در سطح دنیا این متد به عنوان پراستفاده‌ترین و پرکاربردترین متد در حوزه شیء‌گرایی مطرح است، نیازی به بررسی دیگر متدولوژی‌های شیء‌گرا احساس نمی‌شود و از دیدگاه تهیه کنندگان این گزارش بررسی این متد از مجموع متدهای شیء‌گرا کافی به نظر می‌رسد.

3-2- مقدمه ای بر شیء‌گرایی

شیء‌گرایی⁶ لغتی است که امروزه در صنعت نرم‌افزار باب شده است. شرکتها به سرعت حرکت می‌کنند تا خود را با این تکنولوژی سازگار کنند و آن را در برنامه‌های موجود خود وارد نمایند. در حقیقت بیشتر برنامه‌ها امروزه با شیء‌گرایی توسعه می‌یابند.

متد شیء‌گرایی یک راه متفاوت مشاهده برنامه‌هاست. با متد شیء‌گرایی می‌توان یک برنامه را به قطعات بسیار کوچک یا آبجکت‌هایی⁷ تقسیم کرد که تا اندازه‌ای مستقل از یکدیگر باشند. به محض اینکه تعدادی آبجکت‌های اساسی را در دنیای کامپیوتر ساخته شد، می‌توان به سادگی آنها را کنار هم گذاشت تا برنامه‌های جدید را ایجاد نمود.

یکی از امتیازات اساسی متدهای شیء‌گرا این است که می‌توان یکبار اجزاء را ساخته و بارها و بارها استفاده کنید. همچنین می‌توان از یک قطعه طرح یا کد شیء‌گرایی در نرم‌افزارهای مختلف همچون نرم‌افزار حسابداری یا یک نرم‌افزار بازرگانی یا یک نرم‌افزار پردازش سفارش استفاده نمود.

اما این سوال مطرح می‌شود که بدون استفاده از روش‌های شیء‌گرا نیز می‌توان از کدهای توسعه داده شده قبلی نیز در برنامه‌های مختلف استفاده نمود و تفاوت متدهای شیء‌گرایی با روش سنتی توسعه

⁶ Object-Oriented

⁷ Objects

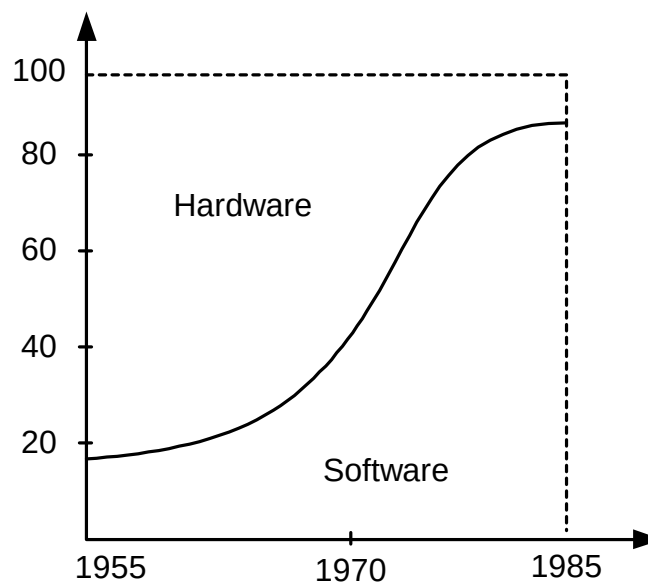
چیست؟ در پاسخ به این سوال می‌توان عنوان داشت که در روش سنتی، روش توسعه به میزان زیادی به اطلاعاتی که قرار است در سیستم نگهداری شود و به توسعه دهندگان وابسته است. در این روش نیازمندی‌های اطلاعاتی با مراجعه و پرس‌وجو از کاربران نهایی شناسایی می‌شوند و برای برطرف کردن نیازهای اطلاعاتی پایگاه داده‌ای را طراحی می‌شود که اطلاعات را نگهداری نماید. برای کار با اطلاعات و تولید اطلاعات صفحات تهیه می‌شوند. به عبارت دیگر در این روش تمرکز تیم توسعه‌دهنده بر روی اطلاعات است و کمتر به این موضوع توجه می‌شود که چه کاری قرار است با اطلاعات انجام شود و یا رفتار سیستم چگونه است. این روش Data Senteric یا مبتنی بر داده نامیده شده است و برای ایجاد سیستم‌های زیادی تا کنون استفاده شده است. در ادامه به بررسی اصول و قواعد حاکم بر مفهوم شی‌گرایی برای مقابله با پیچیده‌گی‌های روزافزون در عرصه صنعت نرم‌افزار پرداخته خواهد شد. (<http://en.wikipedia.org>)

3-2-1- معرفی اصول شی‌گرایی برای مقابله با پیچیدگی نرم‌افزار

با توجه به پیچیدگی روز افزون سیستم‌های نرم‌افزاری، روش‌های مقابله با آن نیز به نوبه خود تکامل یافته‌اند. در روزهای اولیه عصر کامپیوتر نقش نرم‌افزار در یک سیستم کامپیوتری نقش ثانویه تلقی شده و هزینه اساسی طراحی یک سیستم کامپیوتری برای سخت‌افزار پرداخت می‌شد. چون در آن روزها قابلیت سخت‌افزار بسیار محدود بود، برنامه‌ها نیز ساده و کوچک و زبان رایج همان زبان ماشین بوده است. سپس برای تسهیل در نوشتن برنامه‌های بزرگتر زبان اسمبلی ابداع شد. با اینکه در آن زمان سخت‌افزار همه منظوره وجود داشت اما نرم‌افزارها تک منظوره بودند، بدین معنی که برای کاربرد بخصوصی طراحی می‌شدند.⁸ بیشتر نرم‌افزارها بوسیله یک نفر طراحی، پیاده‌سازی، تست، نگهداری و حتی اجرا می‌شد و لذا ظاهراً نیاز چندانی به مستندسازی وجود نداشت. با توجه به طبیعت شخصی این محیط، فرآیند طراحی به صورت ضمنی در ذهن برنامه‌نویس صورت می‌گرفت بدون اینکه طرح خود را روی وسیله‌ای در خارج از ذهن خود مانند کاغذ نمایش دهد.

⁸Custom-Built Software

ولی در اوایل دهه 70 میلادی این وضعیت شروع به دگرگونی نمود. سخت‌افزار سریعتر، قابل اطمینان‌تر و ارزانتر شده و این پیشرفت شگفت انگیز سخت‌افزار باعث اقتصادی شدن فرآیند خودکار سازی بسیاری از کاربردهای صنعتی و تجاری و بدنبال آن افزایش تقاضا برای سیستم‌های نرم‌افزاری پیچیده‌تر گردید. در مقابل این فشار زبان‌های سطح بالا⁹ بعنوان ابزارهای مهم تولید سیستم‌های نرم‌افزاری وارد صحنه شدند. در واقع این سیستم‌ها بازدهی برنامه‌نویسان منفرد و تیم‌های نرم‌افزاری را افزایش داده بودند. در دهه 60 و 70 میلادی توجه پیشگامان رشته نرم‌افزار به "طراحی" بعنوان ابزاری مهم برای مقابله با پیچیدگی سیستم‌های نرم‌افزاری معطوف گشته است. لذا در این دو دهه و بعد از آن روش‌های زیادی ابداع شده‌اند که مهمترین آنها روش طراحی ساخت یافته بوده است. (طرح ملی استاندارد مستند سازی)



شکل 2- درصد هزینه های نرم‌افزار در مقابل هزینه های سخت‌افزار از سالهای 1955 تا

1985

⁹High Level Languages

* همانطوریکه در شکل 2 مشاهده می‌شود، در دهه 50 و 60 میلادی هزینه‌های سخت‌افزار درصد اصلی هزینه تولید یک سیستم کامپیوتری را تشکیل می‌داد، ولی در دهه 70 این وضعیت، کم کم، شروع به دگرگونی نمود تا سال 85 میلادی که در آن هزینه های نرم‌افزار حدود 90% کل هزینه تولید سیستم‌های کامپیوتری را تشکیل می‌داد.

3-3- طراحی شیء‌گرایی¹⁰

نگرش شیء‌گرایی یک روش نسبتاً جدید برای تفکر در مورد یک مسأله با استفاده از مفاهیم موجود در دنیای واقعی به جای مفاهیم موجود در دنیای کامپیوتر است. زیر ساخت اساسی برای این نوع تفکر همان مفهوم شیء¹¹ است که ضمن مدل‌سازی اشیاء واقعی (با توجه به میزان تجرید مورد نظر) قابلیت تلفیق ساختمان داده‌ها و رفتار و اعمال روی آن را در یک واحد به نام شیء داراست. در این نگرش، دنیای نرم‌افزار را به صورت سیستم‌هایی می‌توان در نظر گرفت که شامل مجموعه‌ای از اشیاء مستقل از همدیگر بوده ولی بین آنها روابطی حاکم است که بر اساس مدل‌سازی این روابط و اثرات متقابل آنها، ارائه راه حلی برای یک مسأله از طریق نرم‌افزار در دنیای واقعی میسر می‌شود. در مدل شیء تعریف یک سیستم و عناصر اصلی آن بر چهار اصل زیر استوار است: (شرکت ایزایران)

1. تجرید یا چکیده سازی¹²

2. دربرگیری یا محصور سازی¹³

3. واحدبندی¹⁴

4. سلسله مراتب¹⁵

اینک به ارائه توضیحی درباره هر کدام از این مفاهیم می‌پردازیم.

3-3-1- تجرید (چکیده‌سازی)

تجرید عبارتست از فرآیند متمرکز شدن و تاکید کردن روی ویژگی‌ها و رفتارهای اصلی یک پدیده و نادیده گرفتن ویژگی‌ها موقت و غیر مهم آن پدیده، از یک زاویه دید مشخص. (شرکت ایزایران)

¹⁰Object-Oriented Design

¹¹Object

¹²Abstraction

¹³Encapsulation

¹⁴Modularity

¹⁵Hierarchy

این فرآیند به ساختن یک مدل یا یک دید از آن شیء منتهی می‌گردد که به این مدل نیز تجرید (یا به عبارت دقیقتر مجرد) گفته می‌شود. مثال‌هایی از تجرید عبارتند از:

- دیدهای گوناگون از بدن انسان (مانند: سیستم گردش خون، سیستم عصبی، و...)
- بیان روابط میان اجزاء یک سیستم مکانیکی توسط یک معادله ریاضی.
- استفاده از یک نماد برای نمایش حضور یک موجودیت در یک صحنه خاص (مثلاً
این شکل  برای نمایش انسان)
- نمایش گرافیکی از رفتار یک سیستم.

Ø نقش تجرید در کنترل پیچیدگی

اصل تجرید به مدل‌ساز (اعم از تحلیلگر یا طراح) سیستم کمک می‌کند که به عوض در نظر گرفتن همه جزئیات مربوط به یک سیستم تنها ابعاد اساسی آن را، از زاویه دید خود، مد نظر قرار دهد. در واقع جزئیات بی‌شماری در مورد یک پدیده و یا یک سیستم می‌تواند مطرح باشد که توجه به همه آنها در آن واحد کاری غیر ضروری و نامطلوب است. بدین ترتیب می‌توان اشیاء موجود در یک سیستم را به طور مستقل و فارغ از سایر جزئیات غیر مهم (از دید مدل‌ساز) بررسی نمود. از این رو تجرید ابزاری مهم برای کنترل و غلبه بر پیچیدگی به شمار می‌آید.

Ø انواع تجرید

تجرید انواع مختلفی داشته که به ترتیب اهمیت، ذیلاً فهرست شده اند:

1. تجرید موجودیت¹⁶: شیئی که یک مدل مفید از یک موجودیت واقعی، که در قلمرو مسأله مطرح است، را نمایش می‌دهد. مثال (در OOP): کلاسی که نمایش دهنده موجودیت دانشجو باشد.

¹⁶Entity Abstraction

2. تجرید رفتار¹⁷: شی‌ای که یک مجموعه عمومی از اعمال¹⁸ که عملکرد یکسانی دارند را

نمایش می‌دهد. مثال: عمل اضافه کردن به لیست (صرفنظر از نحوه پیاده‌سازی آن)

3. تجرید مجازی¹⁹: در یک معماری لایه‌ای، شی‌ای که مجموعه‌ای از اعمال متکی بر

یک سطح پایین‌تر از خود را گروه‌بندی کرده و یا این که خود به عنوان یک سطح پایین‌تر

برای استفاده یک سطح بالاتر مورد استفاده قرار می‌گیرد. مثال: معماری لایه‌ای پروتکل

TCP/IP.

Ø ویژگیهای تجرید

§ از آنجا که همیشه تجرید با توجه به یک دید معین و برای یک منظور مشخص

ساخته می‌شود، بنابراین برای یک شی‌انواع گوناگونی از تجرید وجود دارد. در واقع هر

متخصص از زاویه تخصص خود به موضوع می‌نگرد لذا هر کدام، سایر جزئیات سیستم را

بعنوان یک مکانیزم همه‌جانبه را نادیده می‌گیرد، فقط روی جنبه مورد نظر خود

متمرکز می‌شود.

§ تجرید همواره با نمود خارجی²⁰ از یک شی سروکار دارد. به عبارت دیگر تجرید

روی اینکه یک شی چیست و چه کار می‌کند تاکید می‌نماید و جزئیات مربوط به نحوه

پیاده‌سازی آن شی را نادیده می‌گیرد.

§ تجرید سطوحی دارد، هرچه به سمت بالای آن سطوح حرکت کنیم به معنای تاکید

کردن و متمرکز شدن روی اطلاعات مهمتر است. همچنین حجم اطلاعاتی که باید بدانیم و

¹⁷ Action Abstraction or Procedural Abstraction

¹⁸ Operations

¹⁹ Virtual Abstraction

²⁰ Outside View

تعداد قطعه‌های اطلاعاتی کمتر می‌شود. برعکس هرچه به سمت پایین حرکت نماییم با جزئیات بیشتر، تعداد قطعات بیشتر و حجم اطلاعات گسترده تری روبرو می‌شویم. بنابراین با انتخاب سطح تجرید مناسب، می‌توان به میزانی از جزئیات که دلخواه و مورد نظر طراح باشد پرداخت و از تراکم سایر جزئیات جلوگیری نمود.

§ همه تجریدها دارای ویژگی‌های ساکن²¹ و پویا²² هستند. بعنوان مثال شی فایل را در نظر بگیرید، یک فایل دارای اسم، اندازه و محتویات است. خود این ویژگیهای ثابتند ولی مقادیر آنها متغیرند زیرا اسم، اندازه و محتویات در طول حیات فایل تغییر می‌یابد.

3-3-2- دربرگیری²³ (محصور سازی)

دربرگیری عبارت از عدم پذیرش تأثیرات ناخواسته و یا کنترل نشده و محدود کردن طرق دسترسی به و یا استفاده از یک شی است. (شرکت ایزایران)

1. این فرآیند با جداسازی ویژگی‌های اصلی یک شی، که برای اشیاء دیگر قابل دسترسی است، از جزئیات پیاده‌سازی این شی، که می‌بایستی از بقیه اشیاء مخفی شود، انجام می‌پذیرد. عبارت ساده‌تر محصورسازی همان فرآیند مخفی‌سازی جزئیات پیاده‌سازی یک شی است.

Ø نقش محصور سازی در کنترل پیچیدگی

§ در مورد دسترسی به اشیاء و استفاده از آنها در مواردی بسیار برای جلوگیری از خرابکاری‌های احتمالی و خطرات متقابل آنها نسبت به یکدیگر لازم است که اشیاء

²¹Static

²²Dynamic

²³Encapsulation

بشکل کاملاً حمایت شده و تعریف شده‌ای اعمال مرتبط با یکدیگر را انجام دهند و بدین صورت گستره خطاها در خود شیء محدود شده و نگهداری برنامه آسانتر می‌شود.

§ همچنین با استفاده از محصورسازی، تغییرات در پیاده سازی یک شیء، تا وقتی که واسط آن تغییر نکرده باشد، می‌تواند به آسانی و با قابلیت اعتماد بالا انجام شود و نیز بدون اثرگذاری روی دیگر اشیاء استفاده کننده از سرویس‌های این شیء صورت می‌گیرد.

§ مفهوم یک واسط چند پیاده‌سازی شده: با جدا ساختن واسط از پیاده سازی می‌توان برای یک واسط عمومی پیاده‌سازی‌های متفاوتی تعریف نمود. بدین صورت کاربر می‌تواند بر حسب نیاز یکی از آنها را انتخاب نموده که این ویژگی باعث افزایش قابلیت استفاده مجدد است.

Ø ویژگیهای دربرگیری

§ قاعده محصور سازی ایجاب می‌کند که ارتباط بین اشیاء تنها از راه واسطها باشد. به عبارت دیگر هیچ قسمتی از یک سیستم نرم‌افزاری نباید به جزئیات داخلی یک قسمت دیگر وابسته باشد.

§ گاهی بین دو مفهوم تجرید و محصور سازی خلط می‌شود، برای بیان رابطه این دو با یکدیگر باید گفت که می‌توان از تجرید به عنوان مکانیزم تعیین جزئیاتی که باید پنهان شود، استفاده نمود اما خود عمل پنهان‌سازی جزئیات (که شامل طراحی واسط و پنهان‌سازی پیاده‌سازی) را محصور سازی گویند.

§ محصور سازی یک مفهوم نسبی است: آن چیزی که در یک سطح از تجرید مخفی است می‌تواند نمود خارجی یک سطح دیگری باشد.

سیستمی را واحد بندی شده گویند که به مجموعه‌ای از واحدهای منسجم و با معنی که وابستگی بین آنها حداقل است، تجزیه شده باشد. (شرکت ایزایران)

Ø نقش واحد بندی در کنترل پیچیدگی

یکی از روش‌های مقابله با پیچیدگی سیستم‌ها شکستن یک مسأله به اجزایی کوچکتر است که میزان هزینه و تلاشی را که باید صرف حل چنین اجزای کوچکتری بنماییم در مجموع کمتر از زمانی است که بخواهیم کل مسأله را یکباره حل کنیم.

Ø ویژگیهای واحد بندی

§ اگر شرایط بیان شده در تعریف واحد بندی رعایت گردد آنگاه واحدهای بدست آمده قابلیت استفاده مجدد خواهد داشت.

§ تعداد اجزا: اگر مسئله به n قسمت شکسته گردد آنگاه $n(n-1)/2$ رابطه بین قسمت ها خواهیم داشت. حال اگر n بالا رود آنگاه با واحدهای بسیار کوچک روبرو خواهیم بود که خود یک مشکل پدید می‌آورد. یکی از وظایف طراح اینست که متوسط n را بدست آورد (n بر اساس سرویس‌های لازم در سیستم تعیین می گردد).

§ معیار تجزیه: تعیین معیاری مناسب برای شکستن یک سیستم به واحدهای کوچکتر مهمترین عامل موفقیت در استفاده از خاصیت واحد بندی بوده و در عین حال به هیچ وجه کار آسانی نیست (در واقع سختی این کار متناظر با سختی انتخاب مجموعه کلاس های مناسب یک مسأله است). لذا در انتخاب واحدها باید دقت کافی به خرج داد چه بسا انتخاب ناآگاهانه گاهی بدتر از عدم استفاده از خاصیت واحد بندی است. اگر این کار بدرستی انجام شود و همچنین

²⁴Modularity

واحدهای بدست آمده به خوبی مستند شوند آنگاه **مزیت سوم** واحدبندی روشن می‌شود زیرا تقسیم برنامه به واحدهای خوش تعریف²⁵ که به خوبی مستند شده باشند در آسان کردن درک برنامه و نگهداری آن نیز نقش مهمی ایفا می‌نماید.

§ واحدها باید ویژگیهای بلوکهای زیر بنایی را داشته باشند:

- استقلال
- واسطه‌های خوش تعریف

3-3-4- سلسله مراتب²⁶

تا به حال سه مفهوم از مفاهیم اساسی مدل شیء را مطرح نموده و نقش هر یک را در کنترل پیچیدگی نرم‌افزار بیان نمودیم. برای رعایت پیوستگی مطالب و روشن‌شدن نقش سلسله مراتب، تسلسل منطقی مکانیزم‌های مدل شیء برای مقابله با پیچیدگی سیستم‌های نرم‌افزاری را بیان می‌کنیم: **گام اول** در این راه متمرکز شدن روی ویژگی‌های اساسی سیستم و نادیده گرفتن جزئیات غیر مهم آن نسبت به یک دید معین که همان مفهوم تجرید است. در بیشتر سیستم‌های واقعی با وجود استفاده از این خاصیت معمولاً با کلاسهای (تجریدهای) زیادی که درک همزمان آنها برای ما بسیار مشکل است، روبرو خواهیم گشت. **گام دوم** پنهان نمودن نمای داخلی سیستم و محدود کردن روش دسترسی به اشیاء بوسیله استفاده از واسطها است. همچنین **واحدبندی (گام سوم)** از طریق ارائه راهی بمنظور دسته بندی تجریدهای منطقاً مرتبط کمک شایانی می‌کند. اما به تنهایی این کافی نیست و اغلب مجموعه ای از تجریدها خود تشکیل دهنده یک سلسله مراتب هستند. با تشخیص این سلسله مراتبها و در نظرگرفتن آنها در طراحی، درک ما نسبت به

²⁵Well-Defined Modules

²⁶Hierarchy

مسئله به صورت قابل توجهی افزایش می‌یابد. مفهوم سلسله مراتب به صورت زیر تعریف می‌شود:
(شرکت ایزایران)

سلسله مراتب عبارت از مرتب ساختن تجربدها در سطوح مختلف است.

سلسله مراتب دارای چند نوع بوده که مهمترین آنها سلسله مراتب ساختار کلاس (IS-A)²⁷ و سلسله مراتب ساختار شیء (PART-OF)²⁸ می‌باشد. وراثت مهمترین شکل سلسله مراتب IS-A بوده و یک عنصر اساسی در سیستمهای شیء‌گرا است. وراثت عبارت است از رابطه بین چند کلاس که در آن یک کلاس در ساختار، رفتار یا هر دو با یک کلاس (وراثت یگانه²⁹) یا چند کلاس (وراثت چندگانه³⁰) دیگر شرکت دارد. به عبارت دیگر وراثت عبارتست از سلسله مراتبی از تجربدها که در آن کلاس فرزند³¹ خصوصیات کلاس پدر³² را به ارث می‌برد. در واقع کلاس فرزند یک تخصیصی از کلاس عمومی تر (کلاس پدر) را نمایش می‌دهد. بدین صورت که کلاس فرزند علاوه بر فیلدها (ساختار) و عملیات (رفتار) اختصاصی خود شامل ساختار و رفتار کلاس پدر است. با توجه به این مطالب به خوبی روشن است که با استفاده از وراثت می‌توان سیستمهایی را ساخت که دارای ویژگی اقتصاد در بیان باشند. در شکل 3 روابط وراثتی کلاس پدر، کلاس، و کلاس فرزند نمایش شده‌اند.

²⁷Also Kind-Of or Generalization/Specialization Relationship

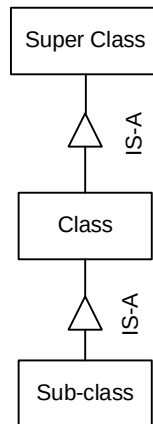
²⁸Also Aggregation or Whole/Part Relationship

²⁹Single Inheritance

³⁰Multiple Inheritance

³¹Sub-class

³²Super class



شکل 3- روابط وراثتی کلاس پدر، کلاس، و کلاس فرزند

سلسله مراتب دوم همان سلسله مراتب PART-OF است که در آن یک کلاس از یک یا چند کلاس دیگر تشکیل می یابد. مثلاً یک ماشین از اجزائی مانند بدنه، موتور، و چرخها تشکیل می شود.

Ø نقش سلسله مراتب در کنترل پیچیدگی

با سازمان دهی تجربیها در سلسله مراتب های IS-A و PART-OF درک ما نسبت به سیستم بهبود می یابد. اساساً ذهن انسان به گونه ای شکل گرفته که ترجیح می دهد برای مدیریت بهتر یک مسأله آنرا در لایه های متفاوتی از تجربیها مورد بررسی قرار دهد.

§ اهمیت سلسله مراتب PART-OF در این است که روابطی که بین اشیاء مختلف

در یک سیستم موجود بوده و نحوه همکاری این اشیاء را که به صورت الگوهایی³³ از

فعل و انفعالاتی³⁴ که بین آنها رخ میدهد بیان شده اند، را نمایش می دهد.

³³Patterns

³⁴Interactions

§ اهمیت سلسله مراتب IS-A در این است که افزونگی³⁵ موجود در سیستم را

مدیریت نموده و بوسیله آن سیستمهایی با خاصیت اقتصاد در بیان را می توان پیاده سازی کرد.

Ø ویژگیهای سلسله مراتب

§ استفاده از وراثت با پنهان سازی تام، تعارض دارد زیرا مستلزم دسترسی مستقیم کلاس فرزند به بعضی از عملیات و داده های اختصاصی³⁶ کلاس پدر است.

3-4- مزایای مدل شی و کاربردهای آن (شرکت ایزایران)

§ هدف نهایی تکنولوژی Object-Oriented انجام فرآیند تولید نرم افزار شبیه به روشی که در تولید سخت افزار استفاده می گردد، یعنی از طریق گروه بندی Object ها در لایه های مختلفی از تجرید است.

§ تکنیکهای سنتی موجود توان پاسخگویی به پیاده سازی گونه های مختلف از سیستم های پیچیده امروزی را ندارند. این سیستمها نیازمند ساختاری برای مدیریت مدل های پیچیده اند. مکانیزمهای موجود در تکنولوژی شی گرایی پتانسیل برخورد با گستردگی و پیچیدگی سیستمهای تجاری امروز را دارند.

§ از طریق کاربرد اشیاء به عنوان و احد مجتمع پذیر تفکیک ناشدنی، تاثیرات یک تغییر را محدود می نماییم که این خود، تشخیص تاثیرات تغییر پیشنهادی را آسانتر نموده و هزینه و زمان لازم

³⁵Redundancy

³⁶Private Members

برای تغییر را کاهش می دهد. از سوی دیگر این اشیاء وابستگی به اطلاعات دربرگرفته شده را محدودتر کرده و این خود نیز اثر تغییر را کاهش داده و نهایتاً زمان تولید نرم افزار را نیز کاهش خواهد یافت.

§ محصور سازی و جداسازی لایه های معماری نرم افزار باعث ایجاد مقیاس پذیری و قابلیت توسعه تدریجی یک سیستم می شود.

§ قابلیت انعطاف برای اجرای اشیاء بصورت توزیع شده در زمانیکه یک سیستم بزرگ از مجموعه ای از اشیاء تعریف شده است بوجود خواهد آمد. اشیاء نیازمند به قدرت پردازش بالا در سایت های قویتر قرار می گیرند و بر اساس نیازمندی های جغرافیایی این توزیع انجام می گیرد.

§ قابلیت استفاده مجدد: امروزه تمایل زیادی به سمت ساخت عناصر نرم افزاری استاندارد (مؤلفه ها³⁷) که می توان از آنها، بدون تغییر، در تولید نرم افزارهای مختلف استفاده نمود (معادل نرم افزاری IC وجود دارد. ساخت مؤلفه ها که بر دیدگاه شیء گرایی مبتنی است می تواند زمینه ساز چنین تحولی باشد. تکنولوژی های ایجاد مؤلفه ها مانند Microsoft COM/DCOM/COM+، Microsoft .NET Framework، Sun Java Beans، و OMG CORBA همگی بر دیدگاه شیء گرایی مبتنی هستند.

^{۴۶} Components

3-5- آشنائی با مفاهیم اولیه شیء گرائی

3-5-1- مفاهیم اساسی

در این بخش به تشریح مفاهیم اساسی شیء گرائی از جمله مفهوم شیء، کلاس، نوع داده مجرد، و مکانیزم ارتباط بین اشیاء خواهیم پرداخت.

Ø شیء

کلمه Object در فارسی تحت عنوان مفهوم، ایده، عینیت، هدف، و شیء ترجمه شده است که متأسفانه هیچیک از این عناوین بیانگر معنی دقیقی از کلمه Object نیستند. اما شاید نزدیکترین کلمه معادل فارسی همان "شیء" یا "چیز" باشد و البته مقصود ما از شیء یک مفهوم کلی است بگونه ای که دارای هویت بوده و قادر به بروز رفتار و ثبت حالات (وضعیت) خود باشد. مثلاً یک ماشین نمونه کاملی از یک شیء است. انسان، درخت، موجودات زنده و حتی عناصر بی جان نیز گونه ای از اشیاء هستند که در دنیای واقعی دارای هویتی مستقل بوده و از خود رفتار نشان داده (چه رفتار فعال [عمل] و چه رفتار غیر فعالانه [عکس العمل]) و نیز حالتی در آنها وجود دارد که گویای یکی از وضعیتهای کلی است که آن شیء می تواند در آن بسر ببرد. بسته به فضا و محیطی که به آن می اندیشیم و سطح تجرید مورد علاقه می توان اشیاء را از یکدیگر تمیز داده و بگونه ای کاملاً روشن تبیین نمود. (<http://en.wikipedia.org>)

از نظر تکنیک شیء گرائی، یک شیء دارای سه مشخصه ذاتی زیر است:

- 1- هویت³⁸: آن ویژگی از یک شیء است که آن را از سایر اشیاء متمایز می سازد. هویت نهفته در ذات شیء است لذا دو شیء که از همه جهات مشابه یکدیگر باشند، همچنان دو شیء به شمار می آیند (نه یک شیء).

³⁸ Identity

2- **حالت**³⁹: حالت یا وضعیت یک شیء در بردارنده تمام خواص (یا صفات) آن شیء (معمولاً ایستا) بعلاوه مقادیر جاری (معمولاً پویا) برای هر یک از این خواص است.

3- **رفتار**⁴⁰: چگونگی عمل و عکس العمل یک شیء در قالب تغییر حالت در مقابل دریافت و یا ارسال پیام را نشان می دهد.

بر اساس تعریف فوق مفاهیمی مانند عمل، صفت، و زمان نمونه هایی از موجودیتهایی هستند که یک شیء محسوب نمی شوند.

می توان اشیاء را به دو قسمت تقسیم نمود: فعال⁴¹ و غیر فعال⁴². شیء فعال روی اشیاء دیگر تاثیر می گذارد (با فرستادن پیام) و شیء غیر فعال تاثیر می پذیرد. اگر شیء بتواند توأمأً فعال و غیر فعال باشد آنگاه عامل⁴³ نامیده می گردد.

از دیدگاه زبانهای شیء-گرا (OOP) یک شیء متشکل از ساختمان داده ها و الگوریتمهایی است که روی این داده ها اعمال شده و بوسیله آنها سرویسهای شیء به سرویس گیرندگان آن ارائه می گردد. بنابراین می توان گفت که:

الگوریتم ها + ساختمان داده ها = شیء

اگر به جهان اطراف خود بنگریم اشیاء زیادی را پیدا می کنیم که نمونه هایی از یک مفهوم کلی تر بنام کلاس را تشکیل می دهند. برای مثال ماشین پیکان به شماره x و متعلق به آقای y یک شیء است. این شیء نمونه ای از یک مفهوم عمومی تر که همان ماشین است، می باشد.

³⁹ State
⁴⁰ Behavior
⁴¹ Active
⁴² Passive
⁴³ Agent

Ø تعریف کلاس

مجموعه ای از اشیاء که دارای ساختار و رفتار مشترکی باشند را کلاس می نامند. مزیت گروه بندی اشیاء در مفهوم کلاس مدیریت بهتر و قابلیت استفاده مجدد آنهاست. در واقع کلاس به عنوان ابزار اصلی تجرید در شیءگرائی عمل می کند. در زبان مدلسازی یکنواخت⁴⁴، که بمنظور یکسان سازی نحوه نمایش مستندات نرم افزارها در محیط شیءگرائی تعریف شده است، کلاس به صورت زیر نمایش داده می شود. (<http://en.wikipedia.org>)

Class Name
Attribute 1 ... Attribute n
Operation 1 ... Operation n

شکل 4- نماد کلاس در UML

همانطوریکه در شکل 4 می بینیم، هر کلاس از سه قسمت زیر تشکیل می گردد:

- 1- نام کلاس: یک شناسه منحصر به فرد.
- 2- صفات کلاس: ویژگیهایی از کلاس که نگهداری آن برای ما مهم است.
- 3- اعمال کلاس: رفتار کلاس در مقابل کلاسهای دیگر.

نوع دستیابی⁴⁵ به صفات و رفتار یک کلاس بوسیله مکانیزمهای ذیل کنترل می گردد:

- 1- دستیابی عمومی⁴⁶: برای همه استفاده کنندگان قابل دسترسی است.

⁴⁴ Unified Modeling Language (UML)

⁴⁵ Access

⁴⁶ Public Access

2- دستیابی اختصاصی⁴⁷: تنها برای همان کلاس و دوستان آن کلاس قابل دسترسی است.

3- دستیابی حفاظت شده⁴⁸: تنها برای خود کلاس، دوستان کلاس و زیر کلاسها⁴⁹ قابل دسترسی است.

Ø مفهوم نوع داده مجرد⁵⁰

ADT امکانی را فراهم می آورد که باعث شناسائی (معرفی) اشیاء از طریق بیان ساختار و رفتار آنها بدون نیاز به اطلاع از نحوه پیاده سازی آن ساختار یا رفتار باشد. (<http://en.wikipedia.org>)

مفهوم ADT بر روی جداسازی اعلان کلاس از پیاده سازی آن تاکید می ورزد لذا در اکثر زبانهای OOP کلاس از دو قسمت واسط و پیاده سازی تشکیل می گردد.

1- واسط: بیان کننده سرویسهایی است که یک کلاس به سرویس گیرندگان خود ارائه می دهد. واسط شامل معرفی عملیات (متدها)⁵¹، تعداد و نوع پارامترهای آنها، و تشریح آنچه کلاس انجام می دهد، بدون بیان چگونگی آن است. زبانهای OOP از مورد اخیر به صورت مستقیم پشتیبانی نمی نمایند. لذا، معمولاً، آنچه کلاس انجام می دهد به صورت Comment بیان می شود. البته، راه بهتر استفاده از یک زبان تعریف واسط⁵² است که در تکنولوژیهای CORBA و COM مورد استفاده قرار می گیرد.

⁴⁷ Private Access

⁴⁸ Protected Access

⁴⁹ Sub-Class

⁵⁰ Abstract Data Type (ADT)

⁵¹ Operations (Methods) Declaration

⁵² Interface Definition Language (IDL)

2- پیاده سازی: در این قسمت جزئیات داخلی ساختار و رفتار کلاس پیاده سازی می گردد. به عبارت دیگر، در اینجا چگونگی فراهم آوردن سرویسهای کلاس بیان می شود. تقسیم کلاس به واسط و پیاده سازی مزایای زیر را دربردارد:

- اعمال اصل محصور سازی که باعث جدائی ساختار داخلی کلاس از نحوه استفاده از آن شده، و این به معنای
- استقلال از نحوه پیاده سازی است، که باعث
- حذف پدیده انتشار تغییرات⁵³، و به عبارت دیگر
- نگهداری آسانتر خواهد بود.

Ø مکانیزم ارتباط بین اشیاء

ارتباط بین اشیاء و بهرمندی از سرویس های(عملیات) آنها از طریق مکانیزم تبادل پیام⁵⁴ صورت می گیرد.

در شئ گرائی تعامل اشیاء یک سیستم تنها از طریق مکانیزم تبادل پیام اعمال می گردد. برای تصور بهتر تبادل پیام به توضیح مفهوم سرویس گیرنده/سرویس دهنده⁵⁵ نیاز داریم. شئ ارسال کننده پیام سرویس گیرنده (تقاضا دهنده) حساب شده و شئ دریافت کننده پیام، سرویس دهنده محسوب می گردد. هر پیام عبارت از یک درخواست از طرف سرویس گیرنده به سرویس دهنده برای اجرای یک عمل⁵⁶ است. این درخواست باید مطابق با امضای عمل⁵⁷ باشد. امضای

⁵³ Change Propagation: اگر سرویس گیرندگان کلاس به ساختار داخلی آن وابسته باشند آنگاه هرگونه تغییر در این ساختار مستلزم تغییری متناظر در سرویس گیرندگان خواهد بود

⁵⁴ Message Passing

⁵⁵ Client/Server

⁵⁶ Method Call

⁵⁷ Method Signature

عمل دربردارنده نوع، تعداد، و ترتیب پارامترها و نوع مقدار برگشتی عمل است).

(<http://en.wikipedia.org>)

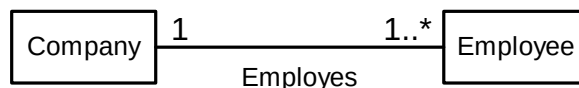
3-5-2- رابطه بین کلاسها

در ذیل به بیان انواع ارتباطات اساسی که بین کلاسها وجود دارد، پرداخت شده است. توجه داشته باشید که انواع دیگری از ارتباطات وجود دارند. بعنوان نمونه می توان از رابطه استفاده از⁵⁸ و رابطه ترکیب⁵⁹ (که یک صورت قویتری از رابطه تجمعی) نام برد. به صورت کلی سه نوع ارتباط اصلی بین کلاسها وجود دارد که به شرح ذیل است.

Ø رابطه انجمنی⁶⁰

نوعی وابستگی معنایی⁶¹ بین نمونهایی از کلاسهای متفاوت است که با حذف این وابستگی عملاً هیچ ارتباط دیگری بین دو کلاس وجود نخواهد داشت. برای مثال رابطه شرکت با کارمند.

(<http://en.wikipedia.org>)



شکل 5- رابطه انجمنی

در هر رابطه، نوع رابطه (مفهوم رابطه) و درجه وابستگی باید مشخص گردد. درجه وابستگی⁶² نشان دهنده تعداد نمونه هایی است که از دو طرف رابطه، در آن رابطه شرکت دارند. در

⁵⁸ Using

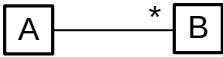
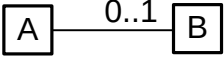
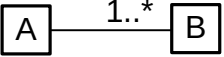
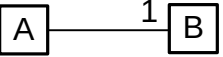
⁵⁹ Composition

⁶⁰ Association Relationship

⁶¹ Semantic Relationship

⁶² Multiplicity or Cardinality

مثال قبل هر شرکت همیشه یک کارمند یا بیشتر استخدام می نماید. در UML درجات مختلف وابستگی به صورت زیر نمایش داده می شود. (<http://en.wikipedia.org>)

یک نمونه از A همیشه با هیچ، یک نمونه، یا بیشتر از B ارتباط دارد.	یک نمونه از A همیشه با هیچ یا فقط یک نمونه از B ارتباط دارد.	یک نمونه از A همیشه با یک نمونه یا بیشتر از B ارتباط دارد.	یک نمونه از A همیشه با یک نمونه از B ارتباط دارد.
			

شکل 6- نمادهای درجات وابستگی گوناگون در UML

Ø رابطه تجمعی⁶³

زمانی که یک کلاس از تلفیق تعدادی کلاس دیگر تشکیل گردد، این رابطه را تجمعی گویند. نام دیگر این نوع رابطه را، رابطه کل به جزء⁶⁴ است. در شیء‌گرایی از عبارات ذیل برای نشان دادن این رابطه استفاده می‌شود: (<http://en.wikipedia.org>)

PART-OF (پایین به بالا)

HAS-A (بالا به پایین)

برای مثال:

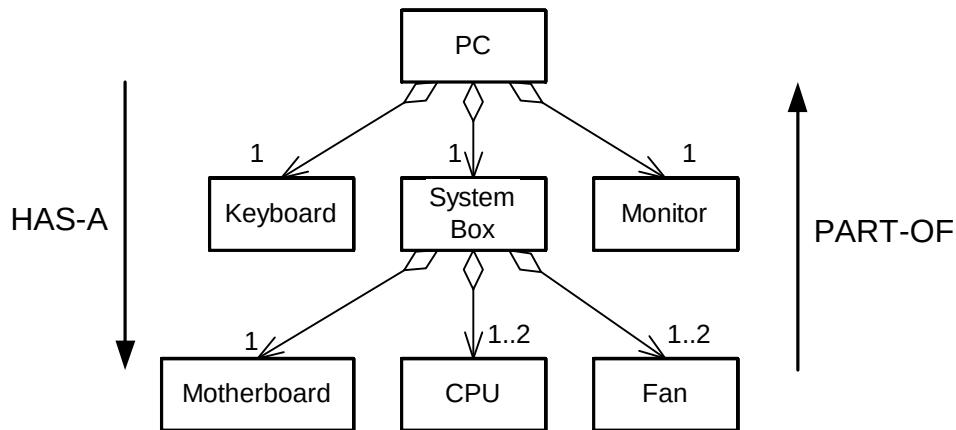
A PC HAS-A Keyboard

A Monitor is PART-OF a PC

⁶³ Aggregation Relationship

⁶⁴ Whole-Part

بخشی از شکل 7 نمایش دهنده مثال فوق است (در UML):



شکل 7 - نمونه‌ای از رابطه تجمعی و نحوه نمایش آن توسط UML

Ø رابطه وراثت⁶⁵

یک رابطه طبقه بندی⁶⁶ بین یک عنصر عمومی تر و یک عنصر اختصاصی تر به طوریکه عنصر اختصاصی تر ساختار و/یا رفتار عنصر عمومی تر را به ارث برده و علاوه بر آن از ویژگیها و/یا رفتارهای اضافی نیز برخوردار باشد. عبارت دیگر وراثت عبارتست از سلسله مراتبی از تجربدها⁶⁷ که در آن کلاس فرزند⁶⁸ خصوصیات کلاس پدر⁶⁹ را به ارث می برد. در واقع کلاس فرزند یک تخصیص⁷⁰ از کلاس پدر را نمایش داده و همزمان کلاس پدر یک تعمیم⁷¹ از کلاس فرزند به حساب می آید. در وراثت یگانه⁷² برای هر زیر کلاس فقط یک کلاس پدر وجود دارد. در حالیکه در وراثت چندگانه⁷³ یک زیر کلاس از دو کلاس یا بیشتر ارث می برد. نام دیگر

⁶⁵ Inheritance, Generalization/Specialization , IS-A Relationship

⁶⁶ Taxonomic Relationship

⁶⁷ Abstract Hierarchy

⁶⁸ Child Class or Sub-Class

⁶⁹ Parent Class or Super Class

⁷⁰ Specialization

⁷¹ Generalization

⁷² Single Inheritance

⁷³ Multiple Inheritance

این رابطه عام و خاص است. در شیء‌گرایی از عبارات ذیل برای نشان دادن این رابطه استفاده می

شود: (<http://en.wikipedia.org>)

IS-A / Generalized-to (پایین به بالا)

Specialized-to (بالا به پایین)

3-6- برنامه نویسی مبتنی بر اجزاء

3-6-1- استفاده از معماری مبتنی بر مؤلفه‌ها

همانطوریکه قبلاً بیان کردیم، برای تولید یک سیستم نرم‌افزاری باید به سیستم از زوایای گوناگونی بنگریم. زیرا معمولاً هر یک از سهامداران از دیدگاه‌های مختلفی و در زمانهای متفاوت به سیستم توجه می‌کنند. به نظر می‌رسد که معماری سیستم مهمترین روش مدیریت و کنترل دیدهای گوناگون و در نتیجه کنترل و مدیریت توسعه تدریجی و تکراری سیستم است. معماری سیستم عبارتست از تعیین ساختار کلی سیستم و روشهایی که این ساختار را قادر به تامین کلیه ویژگیهای کلیدی سیستم می‌سازد. در واقع، معماری سیستم شامل تصمیم‌گیریهایی در سطح کلان در موارد ذیل است. (<http://www.enterpriseunifiedprocess.com>)

§ نحوه سازمان دهی سیستم نرم‌افزاری

§ انتخاب عناصر ساختاری و واسطه‌های آن‌ها (مؤلفه‌ها)، که سیستم را تشکیل می‌دهند، همراه با مشخص نمودن رفتار این عناصر که عملکرد کلی سیستم از همکاری این عناصر با یکدیگر بوجود می‌آید.

§ سازمان دهی این عناصر در گروه‌های بزرگتر (زیر سیستمها).

§ سبک معماری⁷⁴ مورد استفاده که تعداد گزینه‌های مطرح به عنوان الگوی معماری را کاهش می‌دهد. همچنین درجه‌ای از یکنواختی را بر معماری تحمیل می‌نماید. سبک معماری،

⁷⁴Architectural Style

معمولاً، بوسیله انتخاب یک میان افزار⁷⁵، مجموعه ای از الگوهای مشخص یا چارچوب معینی برای معماری مشخص می گردد. مثالهایی از سبکهای معماری عبارتند از: سبک های خادم/مخدوم⁷⁶، Pipe and Filter⁷⁷ و رویداد گرا⁷⁷.

در واقع نقش معماری تنها در تعیین ساختار و رفتار مؤلفه های سیستم خلاصه نمی شود، بلکه با مواردی از قبیل کارایی، انعطاف پذیری، استفاده مجدد و محدودیت های تکنولوژی و اقتصادی نیز سروکار دارد. یکی از شیوه های مهم معماری نرم افزار، توسعه مبتنی بر مؤلفه ها⁷⁸ است زیرا این روش امکان استفاده مجدد از مؤلفه های موجود را به ما می دهد.

این روش دارای ویژگی های ذیل است:

§ مؤلفه ها به ما کمک می کنند که یک معماری کشسان⁷⁹ را داشته باشیم: معماری کشسان امکان استفاده مجدد و بهینه سازی قابلیت نگهداشت را فراهم نموده و نیز در آسان نمودن تقسیم کار بین تیمهای مختلف نقش موثری ایفا می نماید.

§ قابلیت استفاده مجدد از مؤلفه ها و تکنولوژی های موجود را افزایش می دهد.

§ مؤلفه یک پایه خوبی برای مدیریت پیکربندی است.

واژه جزء یا اجزاء یکی از پرکاربردترین واژه ها در حوزه مهندسی نرم افزار است. مفهوم تولید نرم افزار مبتنی بر اجزاء از دهه 60 زمانی آغاز شد که در کنفرانس ناتو بحث بحران نرم افزار پدیدار شد و صاحب نظران در این حوزه معتقد بر این نکته بودند که تولید نرم افزار فرآیند پیچیده ای است و لازم است راهکاری استاندارد برای رهایی پیدا کردن از این بحران اندیشیده شود. در مقام بررسی، این

⁷⁵Middleware

⁷⁶Client/Server Architecture

⁷⁷Event Driven Architecture

⁷⁸Component-Based Development (CBD)

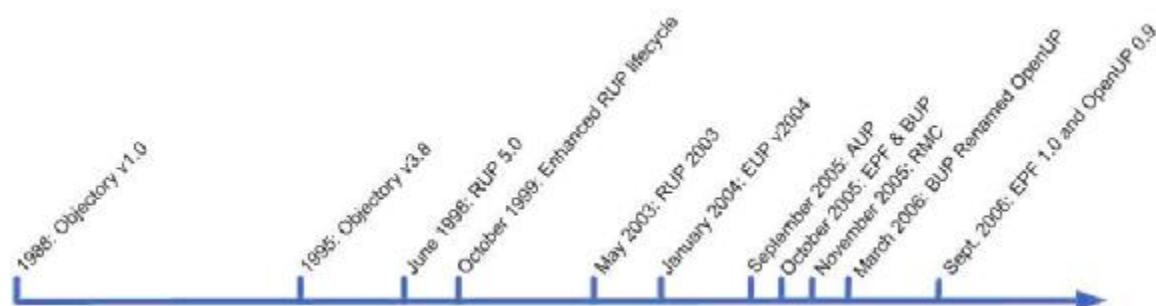
⁷⁹Resilient Architecture

مفهوم با تولید محصولات در قرن 19 مقایسه می‌شود. محصولات تولید شده در آن دوره محصولاتی با تولید کاملاً دستی، گران و با خطای بسیار زیاد بودند.

3-7- تاریخچه

برای اینکه بتوان تاریخچه متدولوژی RUP را به درستی درک کرد لازم است در گام اول به تاریخچه مفهوم شی گزایی توجه کنیم. به طور کلی RUP برای اولین بار توسط شرکت IBM به وسیله بخش IBM's Rational division ارائه شد و برای توسعه این متد نرم‌افزار Rational به بازار عرضه شد. این نرم‌افزار به وسیله زبان مستند سازی UML پشتیبانی می‌شود. شکل زیر روند توسعه متدولوژی RUP را نمایش می‌دهد.

(<http://www.enterpriseunifiedprocess.com>)



شکل 8- نمودار تشریح روند توسعه متدولوژی RUP

مدل Unified Process را می‌توان مدلی دانست که به وسیله Ivar Jacobson برای اولین بار در شرکت اریکسون در اواخر دهه 1960 ارائه گردید. جاکوبسون به وسیله همکارانش یک سیستم ارتباطی بسیار بزرگ را مدل نمودند. در این مدل سازی آنها از یک روش مدلسازی چند لایه

استفاده نمودند که دارای ساختاری بلوک بندی شده بود بلوک های سطح بالا به وسیله های لایه های پایینتر بلوک بندی پشتیبانی می شدند.

همانطور که ذکر شد در این مدل دو سطح از بلوک بندی وجود داشته است یکی بلوک بندی سطح بالا که نشانگر اجزا⁸⁰ اصلی سیستم و بلوک های سطح پایینتر که پشتیبان بلوک های سطح بالا بودند و به traffic cases معروف بودند. در زبان مدل سازی امروزی یعنی زبان مدل سازی UML به بلوک های سطح بالا اجزا سیستم و به بلوک های پشتیبان و سطح پایینتر use cases گفته می شود به کمک این ابزار می توان کلیه بخش های تحلیل، طراحی، پیاده سازی و تست سیستم را پشتیبانی نمود. لازم به ذکر است بسیاری دیگر از نمودارهایی که در زبان UML مطرح هستند نیز حاصل بررسی ها و نتایج کاری است که در شرکت اریکسون انجام شده است. جدول زیر نشانگر مثالی مقایسه بین نمودارهای UML را با نمودارهایی که در شرکت اریکسون برای تحلیل و توسعه سیستم مورد استفاده می شد، است.

Diagrams Ericsson	UML Diagrams
object interaction diagram at the time	sequence diagram
state transition graph	activity diagrams

شکل 9- مقایسه نمودارهای sequence و activity

جاکوبسون در سال 1987 از شرکت اریکسون جدا شد و با کمک همکارانش شرکت Objectory AB را تاسیس نمود و به انجام تحقیق و بررسی بر روی مفاهیم شی گرای پرداخت. وی با مطرح

⁸⁰ Component

کردن مفهوم شی گزایی به دنبال یافتن مفهومی بود که هم فرآیند و هم محصول را با یک راهکار جامع در خود داشته باشد. وی در سال 1995 کتابی را تحت عنوان مهندسی نرم افزار شی گرا منتشر کرد. در این کتاب تمرکز وی بر توصیف فرآیندهای شی گرا بود.

(<http://www.awprofessional.com>)

چندی بعد از انتشار این کتاب، شرکت Rational شرکت Objectory AB را خریداری نمود. در ادامه مسیر حرکت جاکوبسون تصمیم گرفت تا با استفاده از مفهوم فرآیند شی گرا ابزاری برای توسعه سیستم و مدیریت پروژه نرم افزار ارائه نماید. در همان ایام آقایان بوچ⁸¹ رامبوچ⁸² نیز در شرکت Rational مشغول به کار بودند. جاکوبسون تصمیم گرفت با کمک این دو نفر متد ارائه شده خود را توسعه دهد. به این ترتیب بود که این سه نفر به عنوان شناخته شده ترین افراد و صاحبان متدولوژی RUP شناخته شدند. ایشان در ابتدا متدولوژی ROP⁸³ را ارائه نمودند و سپس با تکامل آن متدولوژی RUP را معرفی نمودند.

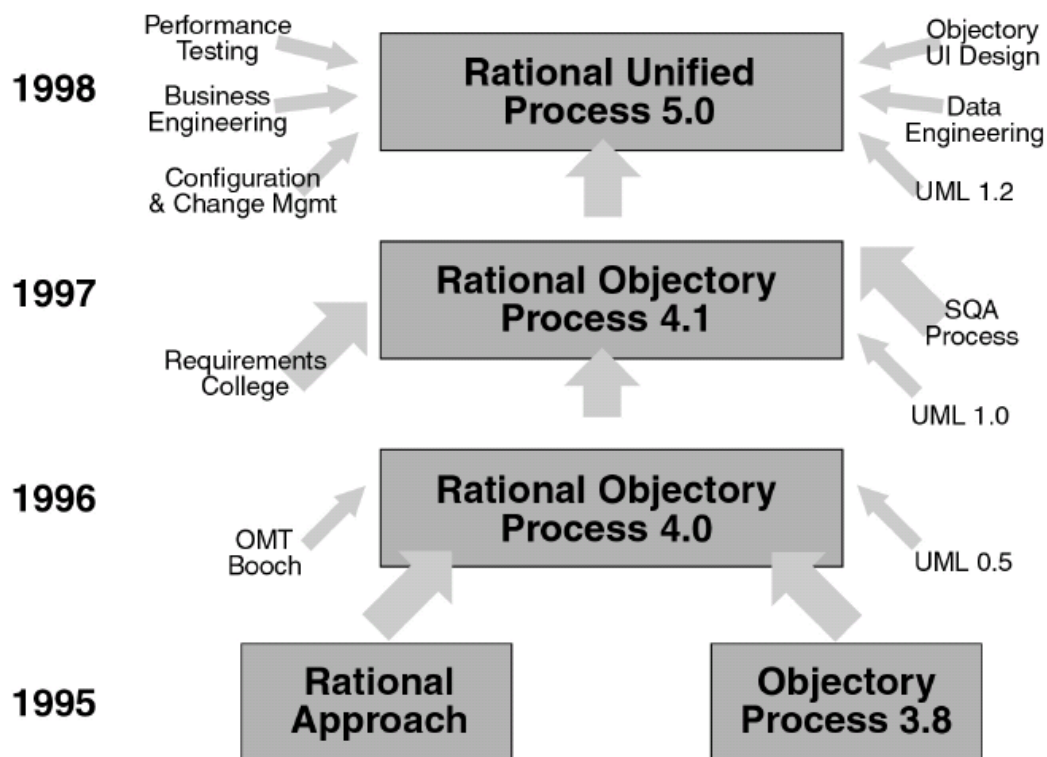
توسعه متدولوژی RUP حاصل تجربه بسیاری از افرادی است که در حوزه مهندسی نرم افزار فعالیت نموده اند. روند توسعه اصلی این متد بین سالهای 1995 تا 1999 بوده است که شکل زیر نشانگر این روند توسعه است:

(Best Practices for Software Development Teams.Rational Unified Process)

⁸¹ Grady Booch

⁸² Jim Rumbaugh

⁸³ Rational Objectory Process



شکل 10- روند توسعه RUP

در ادامه رویدادهای مهمی که در روند توسعه متدلوژی RUP وجود داشته است ارائه خواهد شد.

(<http://www.enterpriseunifiedprocess.com>)

1988	Objectory v1.0 by Jacobson
1995	Objectory v3.8
1996	Rational Objectory Process 4.0
Jun-98	Rational Unified Process 5.0
Feb-99	Unified Software Development Process is published
Jun-99	RUP 5.5
Oct-99	extend RUP with Scott Ambler
Feb-00	RUP 2000
2000-2002	The Unified Process series of books
Mar-02	Agile Modeling

Dec-02	IBM purchases Rational Corporation.
May-03	RUP 2003
Spring 2003	EUP
Jan-04	EUP v2004
Feb-05	Extending the Rational Unified Process by Prentice Hall PTR.
Sep-05	The Agile Unified Process
Oct-05	The Eclipse Project Framework (EPF) and Basic Unified Process (BUP) announced by IBM.
Nov-05	The Rational Method Composer (RMC)
Mar-06	BUP is renamed OpenUP
Sep-06	EPF 1.0 and OpenUP 0.9

شکل 11- روند توسعه RUP

3-8- توسعه تکرار پذیر⁸⁴

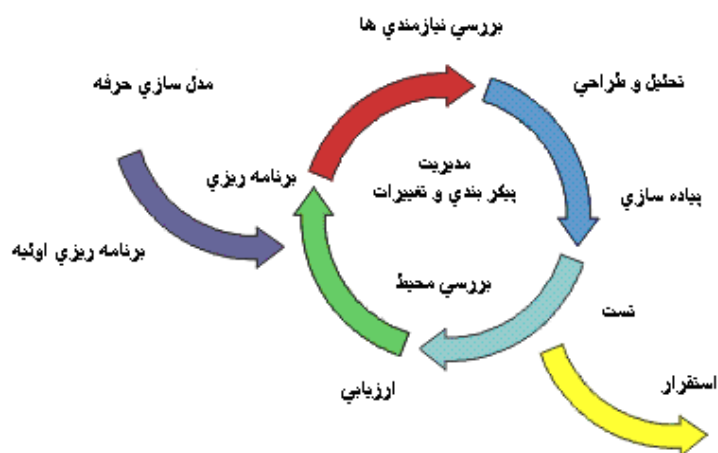
3-8-1- توسعه تکرار پذیر یعنی چیست؟

پروژه‌ای که از توسعه تکرارپذیر استفاده می‌کند چرخه حیاتی متشکل از چند تکرار دارد. هر تکرار از ترکیب الگوهای کاری مرتبط از قبیل: مدل سازی حرفه، تحلیل و طراحی، پیاده سازی، تست و استقرار تشکیل شده که وابسته به محل قرارگرفتن تکرار در چرخه حیات توسعه می باشد. تکرارها در فاز بررسی و تفصیل متمرکز بر مدیریت، نیازمندیها و فعالیتهای طراحی بوده و تکرارها در فاز ساخت متمرکز بر طراحی، پیاده سازی و تست بوده و نهایتاً تکرارها می بایست براساس روش جدول زمانی⁸⁵ مدیریت شوند بدین معنا که زمانبندی هر تکرار ثابت در نظر گرفته می شود و محدوده فعالیتهای هر تکرار به شکلی مدیریت شود که به زمانبندی برسیم.

(<http://www.enterpriseunifiedprocess.com>)

⁸⁴ Develop Iteratively

⁸⁵ Time boxed



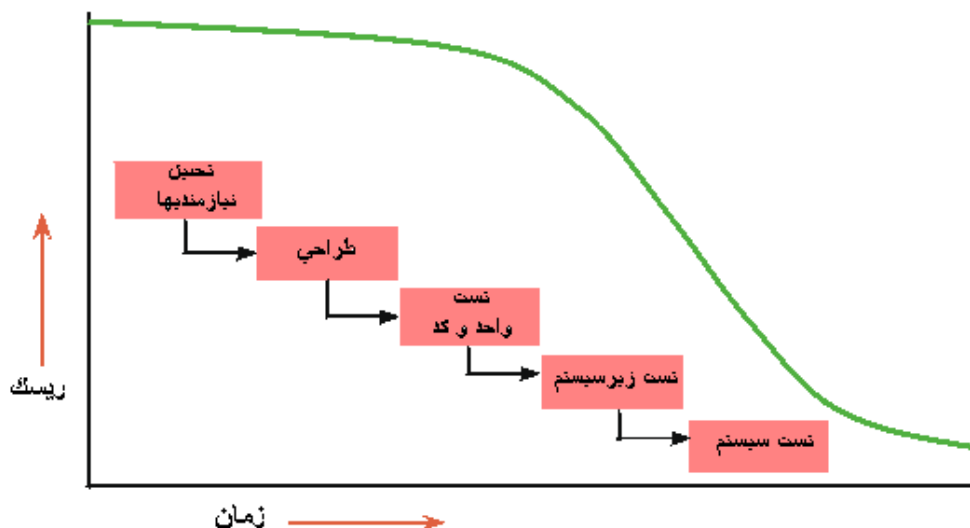
شکل 12- توسعه تکرار پذیر

3-8-2- چرا توسعه تکرار پذیر؟

طراحی یکباره ای که بواسطه نیازهای کلیدی آن زیر سؤال برود و فهم دیر هنگام نقایص طراحی منجر به اجرای سیستم با هزینه زیاد و حتی بعضی مواقع لغو پروژه می گردد.

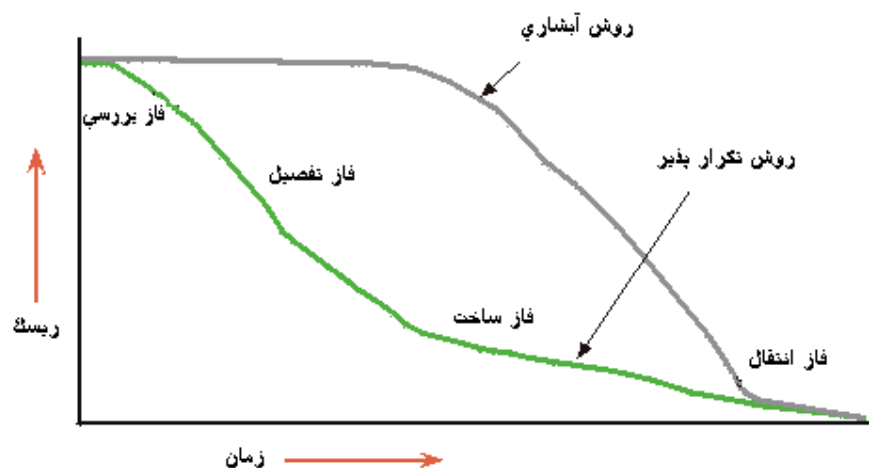
تمامی پروژه ها درگیر مجموعه ای از ریسک ها می باشند. در اوایل چرخه حیات به منظور جلوگیری از بروز ریسک ها می تواند برنامه ریزی دقیق تری انجام داد. بسیاری از ریسک ها تا زمانی که یکپارچه سازی سیستم شروع نشود بروز نمی کنند. شما نمی توانید تمام ریسک را پیش بینی کنید مگر مواردی که تیم توسعه تجربه قبلی درباره آنها داشته باشند.

(<http://www.enterpriseunifiedprocess.com>)



شکل 13- روند توسعه آبشاری

همانطور که در نمودار فوق مشهود است در چرخه حیات آبشاری⁸⁶، ریسک های زیادی تا اواخر چرخه حیات همراه پروژه می باشند.



شکل 14- مقایسه روند توسعه آبشاری با روند توسعه تکرار پذیر

در چرخه حیات تکرارپذیر⁸⁷، شما میزان توسعه و گسترش در هر تکرار را براساس ریسک های کلیدی تعیین می نمایید. بعد از تهیه یک آزمون عملی در هر تکرار شما می توانید تعیین کنید که چه ریسک هایی را می توان کاهش داد.

⁸⁶ Waterfall Lifecycle

⁸⁷ Iterative Lifecycle

3-8-3- مزایای روش تکرارپذیر

یک روش تکرارپذیر بطور کلی از روشهای خطی و یا آبشاری به دلایل متعددی ارجح تر می باشد

که عبارتند از: (<http://www.enterpriseunifiedprocess.com>)

§ ریسکها سریعتر کاهش می یابند زیرا که عناصر بصورت تدریجی یکپارچه می شوند.

§ درخواستهای تغییر و راهکارهای مورد نیاز مورد توافق قرار می گیرند.

§ توسعه و خالص سازی محصول راحت تر می باشد در نتیجه محصول کاملتری ایجاد می شود.

§ سازمان ها می توانند در طی این روش اقدام به یادگیری نموده و فرآیند های خود را بهبود دهند.

§ قابلیت استفاده مجدد افزایش می یابد.

یک مشتری درباره توسعه تکرارپذیر چنین گفته است: "در روش آبشاری، تمام موارد تا اواخر پروژه خوب به نظر می رسد در اواسط یکپارچه سازی بعضی موارد بروز می کند و سپس همه چیز به هم می ریزد ولی در روش تدریجی مخفی نگه داشتن حقایق برای زمان طولانی بسیار مشکل می باشد".

Ø کاهش ریسک

روش تکرارپذیر این اجازه را به شما می دهد که بسیار سریع ریسک ها را کاهش دهید زیرا که بسیاری از ریسک ها در طول یکپارچه سازی معلوم شده و تشریح گردیده اند.

زمانی که شما وارد اولین تکرار می شوید، درگیر تمام الگوهای کاری شده و جنبه های مختلفی از پروژه را از قبیل ابزارها، نرم افزارها، مهارت های انسانی و موارد مشابه را در عمل تجربه می نمایید. همچنین معلوم خواهد شد که بعضی ریسک های تعیین شده واقعاً ریسک نبوده و ریسک های جدیدی نمایان خواهند شد.

یکپارچه سازی یکباره و در پایان پروژه انجام نخواهد گرفت بلکه عناصر بصورت تدریجی ترکیب خواهند شد در واقع، روش تدریجی یک یکپارچه سازی مستمر می باشد و در این روش کار طولانی، نامطمئن و سختی - انتقال 40% از کار به انتهای پروژه - که برنامه ریزی دقیق آن نیز بسیار مشکل می باشد به شش تا نه یکپارچه سازی جزئی که بر روی عناصر بسیار کوچکتر انجام می گیرد تقسیم می شود.

Ø انعطاف پذیری در مواجهه با تغییرات

روش تکرارپذیر این اجازه را به شما می دهد که تغییر نیازمندیها منجر به تغییرات قابل قبولی در ادامه میسر گردد. تغییر نیازمندیها دلیل اصلی مشکلات یک پروژه بوده که منجر به تأخیر در تحویل، از دست رفتن برنامه زمانبندی، نارضایتی مشتری و بیهوده ماندن کار توسعه دهندگان می گردد.

نیازمندیهای کاربران در طول زمان تغییر می کند و این خصلت بشر می باشد. اجبار کردن کاربران برای پذیرش سیستمی که نشأت گرفته از تصورات اولیه بوده کار درستی نمی باشد. کاربران عقایدشان را تغییر می دهند زیرا که مفاهیم تغییر کرده و آنها از محیط و تکنولوژی چیزهای بیشتری یاد می گیرند و آنها ابعاد میانی محصول را در حالی که توسعه می یابد، درک می کنند.

چرخه حیات تکرار پذیر راهی را برای مدیریت باز می کند تا تغییرات را بطرز ماهرانه ای در محصول اعمال نماید. برای مثال، برای رقابت با محصولات موجود، تصمیم می گیرد تا یک نسخه اجرایی که از حجم کاری کاربران می کاهد ارائه نماید و یا اینکه فروشنده تکنولوژی خاصی را به مجموعه خود اضافه نماید.

تکرارها همچنین این اجازه را به شما می دهد که تغییرات تکنولوژیکی در میانه راه داشته باشید. اگر تکنولوژی تغییر کرده و یا تکنولوژی جدیدی بعنوان استاندارد ظاهر گردیده، پروژه شما می

تواند از مزایای آن استفاده کند. این حالت بخصوص در مورد تغییر پلت فرم و ساختارهای پایینی آن صدق می کند.

Ø دستیابی به کیفیت بالاتر

یک روش تکرار پذیر منتج به یک معماری استوارتری می گردد زیرا که خطاها در طی تکرارهای متعددی اصلاح می شوند. نخستین معایب در زمانی که محصول در طول تکرارهای اولیه در حال تکامل و توسعه است، آشکار می شود. گلوگاههای عملکردی به جای اینکه در آستانه تحویل محصول شناخته می شوند، در طی این تکرارها شناخته شده و کاهش می یابد. توسعه تکرارپذیر برخلاف تست های زمان. اجراء که یکبار در آخر پروژه انجام می گیرند، منجر به تست های بیشتری در سرتاسر پروژه می شود. فعالیتهای بحرانی سیستم مجال بیشتری برای تست شدن در طول تکرارهای متعدد می یابند و خود تست ها و نرم افزارهای تست فرصت تکامل پیدا می کنند.

Ø یادگیری و بهبود

توسعه دهندگان می توانند در طول حرکت یادگیری داشته و شایستگی ها و تخصص ها بصورت کامل در طول چرخه حیات بکار گرفته شوند و بجای معطلی طولانی برای برنامه ریزی، انجام دهندگان تست می توانند در ابتدا تست را شروع کرده و نویسندگان فنی نیز کار خود را آغاز کنند همچنین نیاز به آموزش هایی اضافی و کمک های جانبی بسیار سریع و در بازبینی نتایج ارزیابی اولین تکرار قابل تشخیص است.

فرآیندها در طول توسعه بهبود یافته و خالص می شوند. ارزیابی انتهای، تکرارها فقط به لحاظ مقایسه وضعیت پروژه از لحاظ زمانبندی نبوده بلکه تحلیلی بر تغییرات مورد نیاز در سازمان و فرآیندها به منظور اجرای هرچه بهتر تکرار بعدی می باشد.

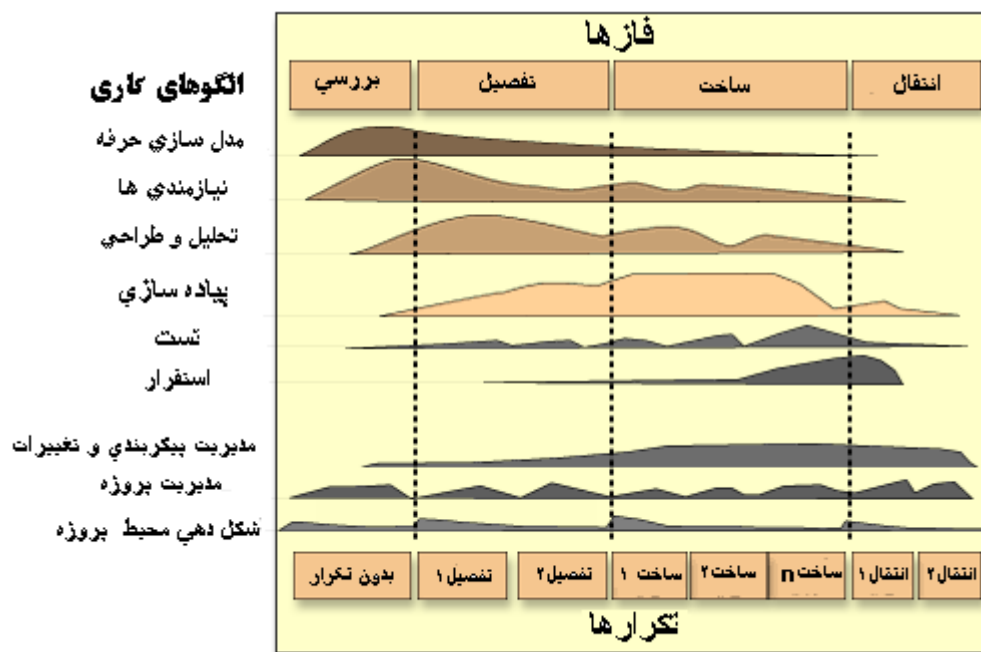
Ø افزایش قابلیت استفاده مجدد

یک چرخه حیات تکرار پذیر استفاده مجدد را آسان می سازد. تعیین مشترکات اجزاء طراحی شده و پیاده سازی شده در مقایسه با حالتی که از قبل تمام مشترکات تعیین گردیده باشد، بسیار آسان تر می گردد.

بازنگری طراحی در اولین تکرارها به معمار نرم افزار اجازه می دهد تا موارد واضح، پتانسیل استفاده مجدد و تکرارهای متوالی را که اجازه توسعه بیشتر و تکامل متن برنامه های مشترک را می دهد، تعیین نمایند. بکارگیری روش تکرارپذیر دستیابی به مزایای یک محصول را آسان تر ساخته و تکرارهای متعددی برای انتخاب، یکپارچه سازی و اعتبار سنجی آنها از لحاظ تناسب با معماری فراهم می سازد.

3-9- شرح کلی بر متدولوژی RUP

متدولوژی RUP یک فرآیند مهندسی نرم افزار می باشد که رویه های ساخت یافته ای را برای تخصیص فعالیت ها و مسئولیت ها به سازمان توسعه دهنده نرم افزار، تعریف می کند. هدف RUP تضمین دستیابی به محصول نرم افزاری با کیفیت بالا و مطابق با نیازمندیهای کاربران با توجه به زمانبندی و بودجه پیش بینی شده می باشد. شکل زیر ساختار کلی RUP را نشان می دهد. (فخرایی، سید وحید)



شکل 15- متدولوژی RUP

همانطور که در نمودار فوق مشهود است RUP دارای دو بعد می باشد:

- محور افقی بیانگر زمان و چرخه حیات (دوره عمر) فرآیندها می باشد.
- محور عمودی بیانگر مجموعه فعالیتهای قابل انجام در طول پروژه می باشد.

بعد اول: بیانگر جنبه های پویای فرآیندها می باشد که با لغاتی از قبیل فازها، تکرارها و نقاط عطف بیان می شوند.

بعد دوم: بیانگر جنبه های ایستا (ثابت) فرآیندها می باشد که با لغاتی از قبیل، الگوهای کاری، فعالیت ها، گردش کار، دستاوردها و نقش ها بیان می شوند.

هر مرحله از چرخه حیات با تعیین نیاز کاری آغاز و با پایاده سازی، آزمون و نهایتاً "نصب نرم افزار خاتمه می یابد. هر مرحله از تعدادی تکرار یا قدم تشکیل شده است. تعداد قدمهای یک تکرار وابسته به پیچیدگی پروژه متغیر است. نسبت تلاش و زمان لازم برای انجام هر مرحله در جدول زیر

مشخص شده است. همانگونه که در زیر مشخص شده، عمده وقت و تلاش، صرف ایجاد برنامه‌ها می‌شود.

شناخت اولیه	تشریح	ایجاد	تحويل
% تلاش	5	20	65
% زمانبندی	10	30	50
			10

شکل 16- درصد تلاش و زمان لازم برای انجام مراحل آ.یو.پی

بر طبق جدول فوق تاکید اصلی در چرخه حیات آ.یو.پی بر مرحله پیاده سازی است. هدف ایجاد برنامه است. لذا، از ابتدا باید این هدف را مد نظر داشت. پیاده سازی در قالب نمونه های سریع در دو مرحله شناخت اولیه و تشریح به ترتیب جهت درک بیشتر نیازها و ارائه نمونه ای ظاهری از نرم افزار مطرح است.

نمودار نشان می دهد که در سرتاسر زمان پروژه اهمیت فعالیتها چه تغییراتی می کند. بعنوان مثال در اولین تکرار تمرکز بر روی نیازمندیهای سیستم بوده و در آخرین تکرار عمده تمرکز بر روی پیاده سازی سیستم می باشد. توسعه تدریجی سیستم در تکرارهای متعدد یکی از مزایای اساسی RUP می باشد که در ادامه به آن می پردازیم. با توجه به مطالب گفته شده، نکته بسیار مهم آن است که در این متد به دلیل توسعه تکرارپذیری که در آن مطرح است نمی توان انجام فازهای مختلف RUP را دارای تواتر زمانی دانست و به مانند متدهای آبشاری عنوان نمود که یک فعالیت در یک فاز شروع می شود و در انتهای فاز پایان می یابد و هیچ زمان دیگری برای اصلاح آن در متد در نظر گرفته نشده است. در این متد کلیه محصولات که در الگوهای کاری 9 گانه تولید می شوند در هر 4 فاز قابل بررسی و اصلاح هستند. به این معنی که در هر 4 فاز لازم است به تمامی محصولات کاری تهیه شده مراجعه شود اما سطح مراجعه و اهمیت آن بسته به زمان پروژه متغیر است.

همانطور که ذکر شد در متدلوژی RUP در بخش‌های مختلف انجام کار به محصولات تولید شده در RUP مراجعه می‌شود. اما لازم است بدانیم تعریف این متد از محصول چیست؟ این تعریف عبارت است از: «محصول یکی از انواع مختلف دستاوردهای قابل سنجشی است که در زمان توسعه نرم‌افزار تولید می‌شود. یک محصول، قطعه اطلاعاتی است که تولید، اصلاح و یا استفاده می‌شود. محصولات عناصر حیاتی و قابل لمس پروژه هستند و توسط نقش‌ها شروع به ایجاد شدن می‌شود. محصولات ممکن است حالات گوناگونی داشته باشند.» (<http://en.wikipedia.org>)

3-10- کاربرد RUP

متدلوژی RUP را می‌توان درست از ابتدای شروع پروژه جدید بکار گرفته و استفاده از آن در چرخه بعدی توسعه تا مدت زیادی پس از انتهای پروژه اولیه ادامه یابد ولی به هر حال روش استفاده از RUP متناسب با نیازهای شما تغییر خواهد نمود. ملاحظات اندکی وجود دارد که زمان و نحوه استفاده از اجزاء RUP را دگرگون می‌سازد که عبارتند از: (فخرایی، سید وحید)

§ چرخه حیات پروژه (تعداد تکرارها، طول زمان هر فاز، طول زمان پروژه)

§ اهداف حرفه مرتبط با پروژه، چشم‌انداز پروژه، محدوده آن و ریسک‌های موجود.

§ میزان فعالیتهایی که برای توسعه نرم‌افزار باید انجام گیرد.

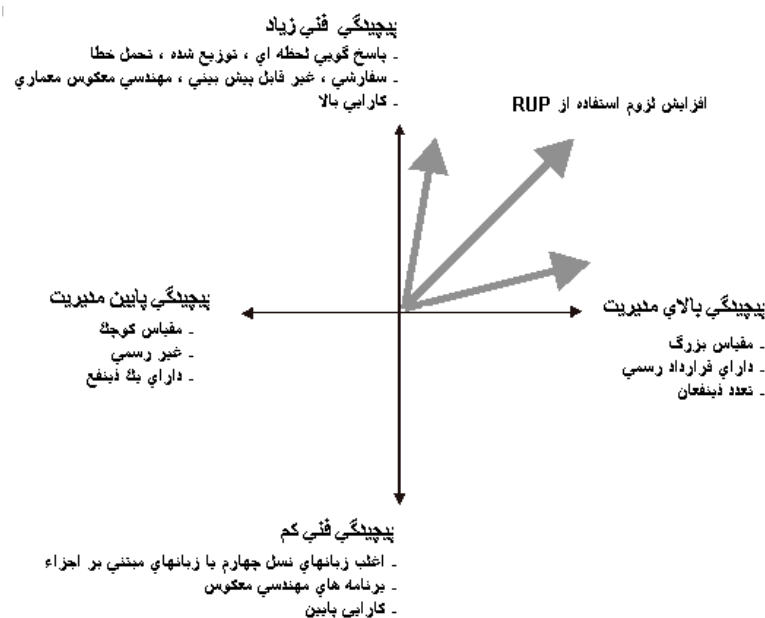
به صورت کلی می‌توان استفاده از هر نوع متدلوژی در توسعه هر امری را حاصل از بررسی دو سطح پیچیدگی فنی و پیچیدگی مدیریتی دانست. در توسعه نرم‌افزار نیز این تعریف معنی دارد، بدین معنی که تیم توسعه‌دهنده نرم‌افزار لازم است با در نظر گرفتن این دو سطح از پیچیدگی متدلوژی مورد نظر خود را برای توسعه نرم‌افزار انتخاب نمایند و بر اساس آن نرم‌افزار را توسعه داده و پیاده‌سازی نمایند.

طبق تعاریف زمانی که پیچیدگی فنی کم باشد اغلب از زبان‌های نسل چهارم یا زبان‌های مبتنی بر اجزا استفاده می‌شود و در توسعه نرم‌افزار از روش‌های مهندسی معکوس استفاده می‌شود. در این سطح معمولاً کارایی نرم‌افزارهای مورد استفاده پایین است. اما زمانی که پیچیدگی فنی زیاد است،

پاسخگویی لحظه‌ای زیاد بوده و سیستم می‌بایست توزیع شده و دارای قابلیت تحمل خطا باشد. معمولاً سیستم‌هایی که در این سطح تولید می‌شوند سفارشی بوده و غیر قابل پیش‌بینی می‌باشند و برای توسعه آنها از روش‌های مهندسی معکوس استفاده می‌شود. این سیستم‌های از نظر سطح استفاده معمولاً دارای کارایی بالا هستند.

زمانی که پیچیدگی مدیریتی پایین است معمولاً مقیاس سیستم کوچک است و می‌توان ذی‌نفع استفاده کننده از سیستم را یک نفر دانست. اما زمانی که پیچیدگی بالای مدیریتی وجود دارد معمولاً مقیاس سیستم‌های مورد استفاده بزرگ است نرم‌افزارها دارای قراردادهای رسمی هستند و تعداد ذی‌نفعان استفاده نرم‌افزار زیاد است.

بنابراین با توجه به مطالب ذکر شده زمانی که در توسعه نرم‌افزار پیچیدگی مدیریتی و پیچیدگی فنی هر دو در سطح بالایی قرار دارد از متدولوژی RUP استفاده می‌شود. نمودار زیر میزان اهمیت استفاده از RUP را در حالت‌های مختلف بررسی می‌نماید.



شکل 17- مواقع استفاده از RUP

3-11- فازهای RUP

3-11-1- فاز بررسی⁸⁸

Ø اهداف

برجسته ترین هدف فاز Inception دستیابی به یک توافق و تفاهم در میان کلیه ذینفعان⁸⁹ بر سر اهداف دوره عمر پروژه می باشد. اصولاً "فاز Inception برای سیستمهایی که از ابتدا ایجاد می شوند اهمیت بیشتری دارد، سیستمهایی که در آن موضوعات مبهم و مخاطرات نیازمندیها بایستی قبل از شروع پروژه مورد توجه قرار گیرد. برای پروژه هایی متمرکز شده بر روی ارتقاء یک سیستم موجود، فاز Inception مختصرتر می باشد اما با اینحال بر روی اطمینان از عملی بودن و توجیه اقتصادی داشتن پروژه تمرکز دارد. (فخرایی، سید وحید)

اهداف اصلی فاز **Inception** شامل موارد زیر می باشد :

- بنا نهادن محدوده نرم افزاری پروژه و شرایط حدود، شامل یک چشم انداز قابل استفاده، پارامترهای پذیرش و آنچه قصد داریم که در محصول تعبیه کنیم و آنچه نمی خواهیم داشته باشیم.
- مشخص کردن usecase های (قالب های عملیاتی) بحرانی سیستم، سناریوهای اصلی از عملیاتی که از نکات اصلی طراحی هستند.
- بیان و احتمالاً "توجیه حداقل یک معماری منتخب در مقابل برخی سناریوهای اصلی.
- تخمین تمامی هزینه و زمان لازم برای انجام پروژه (همچنین جزئیات بیشتری برای فاز تفصیل⁹⁰) که بلافاصله بعد از آن خواهد آمد.
- تخمین مخاطرات بالقوه (از منابع غیر قابل پیش بینی)

⁸⁸ Inception

⁸⁹ Stakeholders

⁹⁰ Elaboration

- آماده سازی محیط پشتیبانی برای پروژه

Ø فعالیتهای اصلی

- فرموله کردن محدوده پروژه :
- این فعالیت درک مفهوم و مهمترین نیازمندیها و محدودیتها تا حدی که بتوان پارامترهای پذیرش برای محصول نهایی را از آنها استخراج کرد، را در بر می گیرد.
- آماده سازی و طراحی یک **Business Case** :
- ارزیابی گزینه ها از جهت امتیازات مدیریت مخاطرات، منابع انسانی، طرح پروژه و هزینه/زمان بندی / سود دهی
- ترکیب بندی یک معماری پیشنهادی :
- ارزیابی امتیازات طراحی و ساخت / خرید / دوباره بکارگیری نظیر هزینه، زمان بندی و منابع قابل تخمین.

در اینجا فرض بر این است که اثبات موجه بودن طرح از طریق بعضی دلایل مفهومی انجام می گیرد. که ممکن است در قالب یک مدل که خواسته های مورد نیاز را شبیه سازی می کند و یا یک Prototype اولیه که آنچه در ناحیه پر خطر مطرح شده است را نشان می دهد، باشد.

عملیات ساخت Prototype در خلال فاز Inception بایستی به اطمینان از عملی بودن راه حل محدود شود راه حلی که در خلال فاز تفصیل و ساخت⁹¹ محقق می شود.

- آماده سازی محیط برای پروژه :

⁹¹ Elaboration & Construction

ارزیابی پروژه و سازمان، انتخاب ابزارها، تصحیح در مورد اینکه کدام بخش فرآیند بایستی اصلاح شود.

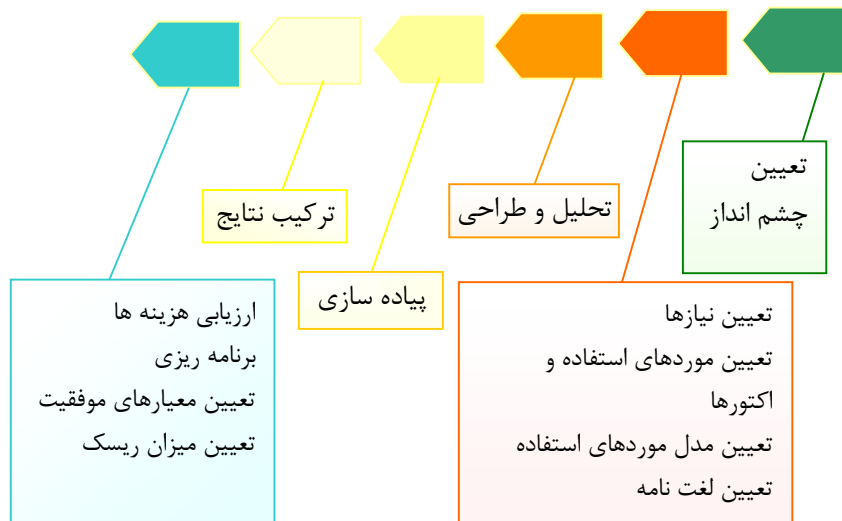
Ø نقطه عطف: اهداف دوره عمر (چرخه حیات)

در انتهای فاز Inception اولین Milestone اصلی پروژه یا Milestone اهداف دوره عمر می باشد. در این نقطه، شما اهداف دوره عمر پروژه را می آزمایید و تصمیم می گیرید که پروژه را ادامه دهید یا آن را Cancel کنید.

Ø ارزیابی پارامترها

- توافق ذینفعان بر سر تعریف محدوده و تخمین های زده شده روی هزینه / زمان بندی.
- توافق بر سر اینکه مجموعه درستی از نیازمندیها گرد آمده است و درک مشتری از مفهوم این نیازمندیها وجود دارد.
- توافق بر سر تخمینهای هزینه / زمان بندی، اولویت ها، مخاطرات و فرآیند توسعه مناسب.
- کلیه مخاطرات بایستی شناسایی شده و یک استراتژی تخفیف برای هر کدام وجود داشته باشد.

پروژه در این Milestone ممکن است نیمه کاره رها شود و یا بطور جدی در مورد شکست آن اندیشیده شود.



شکل 18- عملیات مراحل شناخت اولیه

3-11-2- فاز تفصیل⁹²

اهداف

هدف از این فاز پایه گذاری معماری سیستم و رسیدن به بنیانی ثابت برای متمرکز کردن فعالیتهای طراحی و پیاده سازی در فاز بعدی (فاز ساخت) می باشد. از طریق مطالعه نیازمندیهای مهم مرتبط با معماری سیستم و ارزیابی ریسک ها به معماری مناسب خواهیم رسید و پایداری و ثبات معماری حاصله از طریق یک یا چند نمونه برنامه اجرایی ارزیابی می شود. (فخرایی، سید وحید)

اهداف اصلی فاز تفصیل عبارتند از:

- برای حصول اطمینان کافی از قابل اعتماد بودن معماری، نیازمندیها و طرحها و همچنین کاهش ریسک ها در حد قابل پیش بینی، می بایست هزینه ها و زمانبندی فرآیند توسعه را تعیین نمود.
- ذکر تمامی ریسک های مهم پروژه که مرتبط با معماری هستند.

⁹² Elaboration

- تعیین مبنایی برای معماری با در نظر گرفتن سناریوهای مهم معماری که شامل ریسکهای فنی و مهم پروژه می باشد.
- تهیه نمونه برنامه تکاملی از اجزاء کیفی محصول به شکلی که بتواند منجر به کاهش ریسک های تعیین شده زیر باشد:
- متعادل کردن نیازمندیها و طراحی
- استفاده مجدد از اجزاء
- اثبات امکان پذیری دسترسی به محصول به سرمایه گذاران، مشتریان و کاربران سیستم تشریح می شوند.
- اثبات اینکه معماری پایه ریزی شده نیازمندیهای سیستم را با هزینه منطقی و در زمان منطقی برآورده می سازد.
- بوجود آوردن محیط پشتیبان
- به منظور دستیابی به اهداف مذکور، آماده سازی محیط پشتیبان پروژه شامل ایجاد چارچوب توسعه، نمونه کار، راهنماها و ابزار کار، حائز اهمیت می باشد.

Ø فعالیتهای اساسی

- تعریف، صحنه گذاری و پایه ریزی معماری با حداکثر سرعت ممکنه.
- بازنگری چشم انداز براساس اطلاعات جدید بدست آمده در طی فاز و پایه ریزی درک کاملی از قالب های عملیاتی مهم که تصمیمات اخذ شده در رابطه با معماری و طرح ریزی سیستم را متاثر می سازد.
- تعریف تکرارهای فاز ساخت و جزئیات مربوطه
- بازنگری چارچوب توسعه و ارائه محیط مناسب توسعه شامل روش کار، ابزارها و پشتیبان های مکانیزه مورد نیاز به منظور حمایت از تیم ساخت سیستم.

• بازنگری در معماری و انتخاب اجزاء بدین شکل که اجزاء موجود ارزیابی شده و در مورد اجزائی که می بایست ساخته شده، خریداری شده و یا مجدداً مورد استفاده قرار گیرد، تصمیم گیری می گردد، در نتیجه هزینه فاز ساخت تعیین شده و برنامه زمانبندی قابل اطمینانی حاصل گردد. اجزاء انتخاب شده یکپارچه شده و در مقابل سناریوهای اصلی معماری مورد ارزیابی قرار می گیرند، نتایج حاصل از این کار ممکن است منجر به طراحی مجدد معماری، انتخاب گزینه های دیگر طراحی و یا تجدید نظر در نیازمندیها گردد.

Ø نقطه عطف: اهداف دوره عمر (چرخه حیات)

در پایان فاز تفصیل به دومین نقطه مهم⁹³ پروژه می رسیم. در این زمان اهداف سیستم و محدوده آن، معماری انتخاب شده و تحلیل ریسک های مهم ارزیابی می گردند.

Ø محدوده ارزیابی

- چشم انداز حاصله و نیازمندیهای مطرح شده پابرجا می باشند.
- معماری پابرجا می باشد.
- روشهای کلیدی استفاده شده در تست و ارزیابی تعیین گردیده اند.
- تست و ارزیابی نمونه برنامه اجرایی نشان داده است که فاکتورهای مهم ریسک که مشخص گردیده اند، قابل بررسی هستند.
- برنامه تکرار فاز ساخت به اندازه کافی ریز شده است و منجر به پیشرفت صحیح کار خواهد گردید.
- تمامی ذینفعان هم عقیده اند که اجرای برنامه های جاری در چارچوب معماری فعلی، ما را به نگرش تدوین شده خواهد رساند.

⁹³ Milestone

- هزینه های منابع واقعی مصرف شده در مقابل هزینه های برنامه ریزی شده قابل قبول می باشند.

در صورت نرسیدن به این نقطه ممکن است که پروژه بی نتیجه مانده و یا از اهداف مورد نظر بسیار دور شود.

3-11-3- فاز ساخت⁹⁴

Ø اهداف

هدف از این فاز تعیین نیازمندیهای باقیمانده و تکمیل توسعه سیستم براساس معماری پایه ریزی شده می باشد. فاز ساخت از بعضی جهات یک فرآیند تولیدی بوده که بر روی مدیریت منابع و کنترل عملیات به منظور بهینه سازی هزینه ها، زمانبندی و کیفیت متمرکز می باشد. (فخرایی، سید وحید)

در این مورد تفکرات مدیریتی درگیر تبدیل ویژگی های منطقی بدست آمده در طول فازهای بررسی و تفصیل به محصول قابل تحویلی در طول فاز ساخت و انتقال می باشد.

اهداف اصلی فاز ساخت عبارتند از:

§ به حداقل رساندن هزینه توسعه توسط بهینه سازی منابع و جلوگیری از اتلاف وقت و

دوباره کاری

§ دستیابی به کیفیت مناسب با حداکثر سرعت ممکنه

§ دستیابی به ویرایش های مفیدی از محصول (آلفا، بتا و سایر نسخ تستی) با حداکثر سرعت

ممکنه

§ به اتمام رساندن تحلیل، طراحی، توسعه و تست تمامی عملیات مورد نیاز

⁹⁴ Construction

§ توسعه تکرارپذیر و افزایشی یک محصول کامل و آماده انتقال به تمامی کاربران و ضمناً

تشریح قالب های عملیاتی و سایر نیازمندیهای باقی مانده و همچنین تثبیت طراحی و

کامل کردن پیاده سازی و انجام تست های مورد نیاز بر روی نرم افزار

§ تصمیم گیری درباره اینکه آیا نرم افزار، محل کار و کاربران برای استقرار برنامه های کاربردی

آمادگی دارند.

§ همزمان شدن کار تیم توسعه، حتی در پروژه های کوچک نیز که امکان توسعه اجزاء

بصورت مستقل از یکدیگر نیز فراهم می باشد، اجازه موازی کارکردن تیم پروژه و دسترسی

مشترک به منابع را بدهید. با وجود اینکه همزمانی و موازات به کار تیم توسعه سرعت

زیادی می بخشد ولی در عین حال مدیریت منابع و همزمان سازی گردش کاری را پیچیده

تر می سازد و در صورت استفاده از موازات در کار باید یک معماری قوی داشته

باشیم، فعالیتهای اساسی

§ مدیریت منابع و بهینه سازی فرآیند و کنترل ها

§ به اتمام رساندن توسعه اجزاء و تست آنها در محدوده ارزیابی تعریف شده

§ ارزیابی محصول آماده تحویل در مقابل محدوده پذیرش تعیین شده در چشم انداز

Ø نقطه عطف: اولین قابلیت استفاده

در این Mileston محصول برای تحویل به تیم انتقال آماده می باشد. تمام عملیات مورد نیاز

نرم افزار ایجاد گردیده و تمام تست های آلفا بطور کامل انجام شده اند. همچنین به همراه نرم افزار

ایجاد شده یک راهنمای کاربران و همچنین شرحی بر محصول آماده تحویل ارائه می گردد.

Ø محدوده ارزیابی

محدوده ارزیابی فاز ساخت شامل پاسخ گویی به سوالات زیر می باشد :

§ آیا محصول آماده تحویل به اندازه کافی به بلوغ و ثبات رسیده است تا تحویل مجموعه کاربران گردد ؟

§ آیا منابع واقعی صرف شده در قیاس با برنامه ریزی صورت گرفته هنوز قابل قبول است ؟

§ در صورتی که پروژه به این نقطه عطف Milestone نرسد، ممکن است که انتقال نسخه اجرائی نرم افزار به تعویق بیافتد.

3-11-4- فاز انتقال⁹⁵

Ø اهداف

فاز انتقال حول محور حصول اطمینان از اینکه نرم افزار در دسترس پایین ترین سطح کاربران خود قرار گرفته است، متمرکز گردیده است. فاز انتقال ممکن است دارای تکرارهای متعددی بوده و شامل تست محصول به منظور آماده سازی آن برای تحویل و انجام اصلاحات اندکی که از نتایج بازخورد کاربران نشأت گرفته است، باشد. در این نقطه از چرخه حیات، بازخوردهای کاربران می بایست بر تنظیم و نصب مناسب نرم افزار و قابل استفاده بودن مستندات متمرکز گردیده و تمام مستندات ساختاری و اساسی می بایست قبلاً تعبیه گردیده باشند.

در انتهای فاز انتقال می بایست به اهداف چرخه حیات رسیده باشیم و پروژه آماده خاتمه پذیرفتن قرار بگیرد. در بعضی مواقع پایان چرخه حیات جاری می تواند منطبق بر شروع چرخه حیات محصول مشابهی باشد که منجر به تحویل نسخه جدیدتری از محصول می گردد و در سایر پروژه ها پایان فاز انتقال ممکن است همزمان با تحویل کامل محصول به سه دسته از افراد که مسئول بهره برداری، پشتیبانی و توسعه سیستم تحویل شده می باشند، گردد.

⁹⁵ Transition

فاز انتقال بسته به نوع محصول ممکن است، در محدوده سهل و آسان تا بی نهایت پیچیده تغییر یابد. جایگزین کردن یک نسخه جدید از محصول عمومی موجود ممکن است بسیار ساده باشد در حالیکه جایگزینی یک سیستم ترافیک هوایی بین الملل بی نهایت پیچیده می باشد. فعالیت‌های انجام شده در طول هر تکرار در فاز ساخت وابسته به اهداف می باشد. برای مثال، هنگامی که رفع خطاها، پیاده سازی و تست در حد معمول انجام گردیده و کافی به نظر می رسد، حتی اگر قابلیت های جدیدی نیز می بایست اضافه گردد، ولی تکرارها در فاز ساخت همگی مشابه به هم بوده و مثلاً نیاز به تحلیل و طراحی دارند.

زمانی که امکان استقرار محصول در قلمرو کاربران فراهم گردیده باشد فاز انتقال آغاز خواهد شد و این امر مستلزم کامل شدن مجموعه قابل استفاده ای از سیستم با کیفیت قابل قبولی به همراه مستندات کاربری بوده تا انتقال محصول به مشتری از هر لحاظ نتایج مثبتی را در پی داشته باشد.

(<http://en.wikipedia.org>)

اهداف اصلی فاز عبارتند از:

- § انجام تست بتا به منظور اعتبار سنجی سیستم جدید در قبال توقعات مشتری
- § انجام تست بتا و کار کردن موازی بر روی سیستمی که قرار است سیستم جدید جایگزین آن گردد.
- § انتقال بانک اطلاعاتی مورد نیاز
- § آموزش کاربران و پشتیبان ها
- § سازمان دهی بازاریابی، توزیع و کارمندان فروش
- § توسعه دانش های مهندسی از قبیل: ارتباطات شبکه، بسته بندی تجاری و تولید، سازمان دهی فروش و آموزشهای تخصصی اشخاص
- § تنظیم فعالیتهایی از قبیل رفع خطا، افزایش کارایی و قابلیت استفاده

- § ارزیابی خطوط مبنای استقرار در مقابل چشم انداز نهایی و محدوده پذیرش محصول
- § بدست آوردن حمایت شخصی کاربران
- § حصول توافق با ذینفعان در مورد اتمام یافتن استقرار
- § حصول توافق با ذینفعان درمورد اینکه محصول استقرار یافته سازگار با محدوده های ارزیابی تعیین شده در چشم انداز می باشد.

Ø فعالیتهای اساسی

- § اجرای طرح استقرار
- § نهایی ساختن پشتیبان های کاربری
- § تست کردن محصول تحویلی در محلی که باید استقرار یابد.
- § تحویل نسخه محصول
- § جمع آوری بازخوردهای کاربران
- § تنظیم محصول بر اساس بازخوردها
- § قرار دادن محصول در دسترس آخرین لایه کاربران

Ø نقطه عطف: تحویل محصول

در پایان فاز انتقال به چهارمین نقطه مهم پروژه می رسیم. در این نقطه تعیین می شود که آیا به اهداف تعیین شده رسیده ایم و یا اینکه باید چرخه توسعه دیگری را آغاز نمائیم. در بعضی موارد، این نقطه ممکن است که منطبق بر پایان فاز بررسی برای چرخه بعدی باشد.

پس از اینکه بازنگری پذیرش پروژه توسط مشتری با موفقیت همراه بود و مشتری محصول را قابل تحویل گیری تشخیص دهد ما به نقطه مذکور رسیده ایم.

Ø محدوده ارزیابی

محدوده اصلی ارزیابی برای فاز انتقال شامل پاسخ گویی به سوالهای زیر می باشد:

§ آیا کابر راضی شده است؟

§ آیا منابع واقعی مصرف شده در مقابل برنامه مصرف منابع قابل قبول می باشد؟

در این **Milestone** از محصول بهره برداری شده و چرخه پشتیبانی از آن آغاز می گردد. این امر ممکن است منجر به شروع یک چرخه جدید شده و یا اینکه صرفاً پشتیبانی هایی بر روی محصول انجام گیرد.

3-12- الگوهای کاری⁹⁶

الگوهای کاری فقط لیست کردن همه نقش ها، فعالیت ها و محصولات یک فرآیند نیست، بلکه راهی برای توضیح دادن توالی معنادار از فعالیت هایی که بعضی نتایج ارزشمند را تولید می کنند و همچنین برای نشان دادن کنش بین نقش ها، به کار می رود. الگوهای کاری به اشکال مختلفی وجود دارد. دو جریان کاری رایج، یکی دیسیپلین ها هستند که جریان کار سطح بالا هستند و دیگری جزئیات جریان کاری که جریان کاری موجود در یک دیسیپلین هستند. در ادامه به بررسی و ارائه هر یک از جریان کاری های 9 گانه که به دیسیپلین های RUP معروف هستند پرداخته خواهد شد. (<http://en.wikipedia.org>)

3-12-1- الگوی کاری مدلسازی حرفه

Ø اهداف

اهداف مدلسازی حرفه عبارتند از :

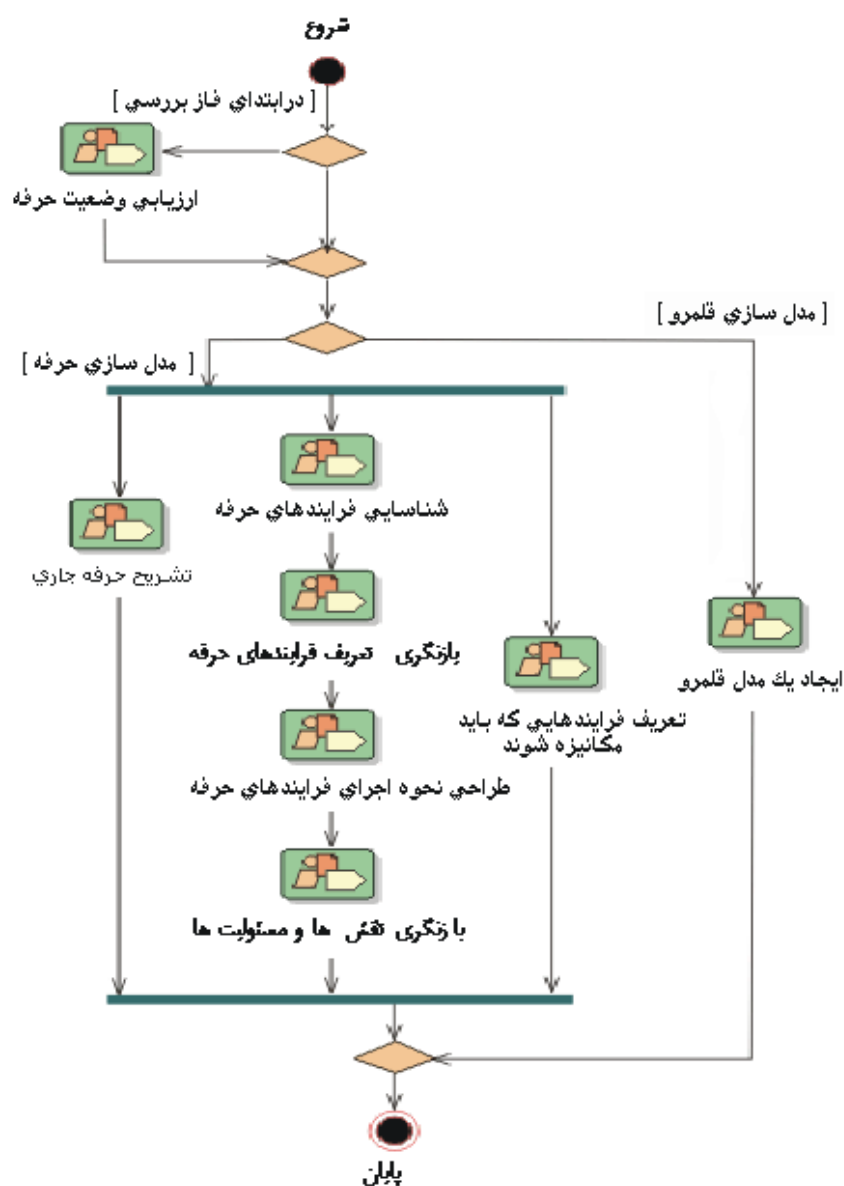
§ شناخت ساختار و رفتارهای سازمانی که قرار است سیستم در آنجا توسعه یابد.

⁹⁶ Workflow

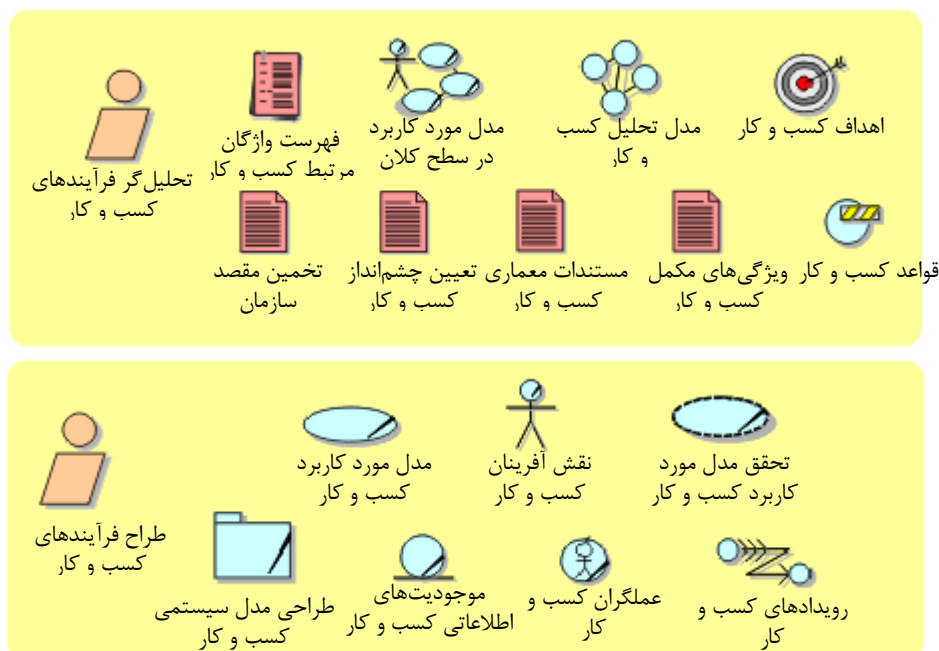
§ درک شکل فعلی سازمان و تعیین پتانسیل

§ اطمینان از اینکه مشتریان، کاربران و توسعه دهندگان سیستم به درک مشترکی از سازمان رسیده اند.

§ استنتاج نیازهای سیستمی که می بایست در سازمان برآورده شود.



شکل 19- الگوی کاری مدلسازی حرفه



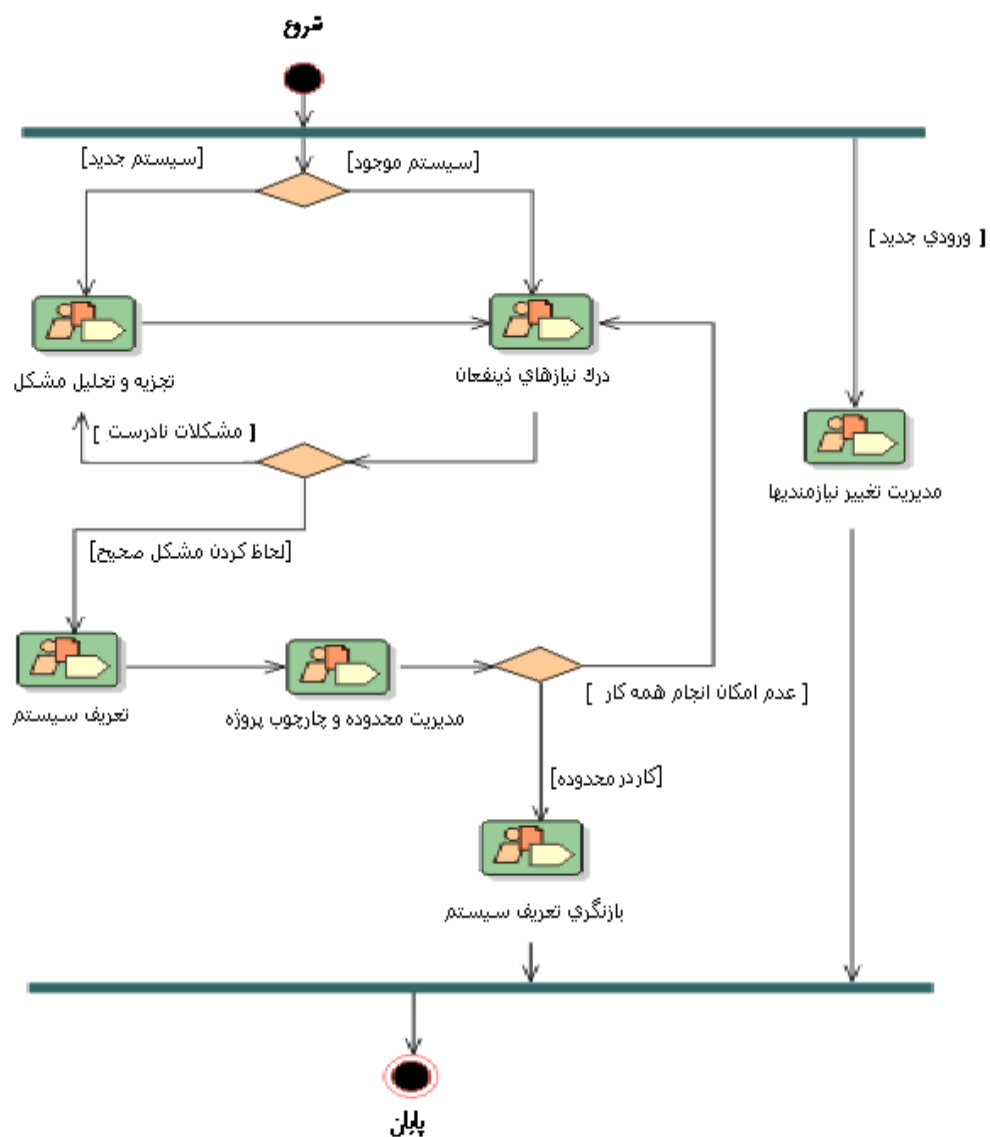
شکل 20- نقش ها و محصولات الگوی کاری مدل سازی حرفه

3-12-2- الگوی کاری نیازمندی ها

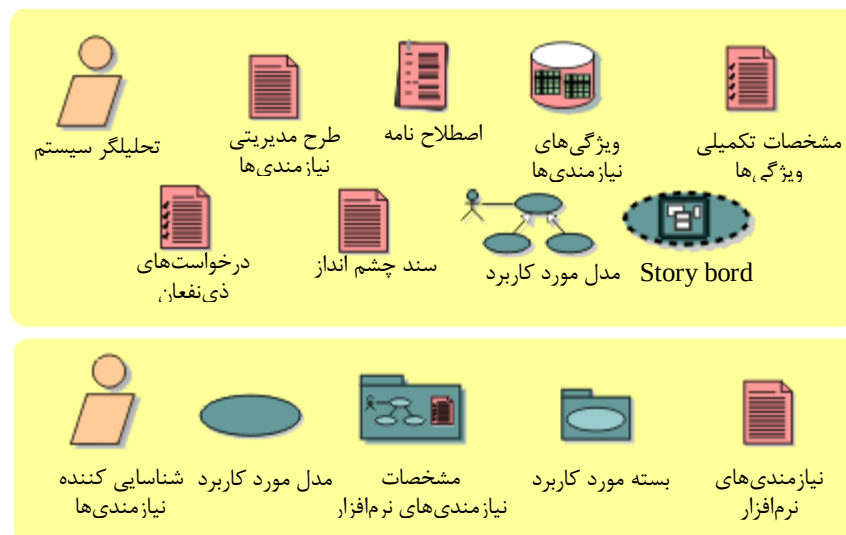
اهداف

اهداف الگوی کاری نیازمندیها عبارتند از :

- § توافق با مشتری و ذینفعان در مورد اینکه سیستم چه کارهایی می بایست انجام بدهد.
- § بوجود آمدن درک بهتری از نیازمندیهای سیستم برای توسعه دهندگان
- § تعیین حدود و ثغور سیستم
- § بوجود آمدن اساسی برای برنامه ریزی اقدامات فنی در هر فاز
- § بوجود آمدن اساسی برای تخمین زمان و هزینه توسعه سیستم
- § تعریف فرمهای محاوره کاربران به منظور تمرکز بر روی نیازها و اهداف کاربران



شکل 21- الگوی کاری نیازمندی ها



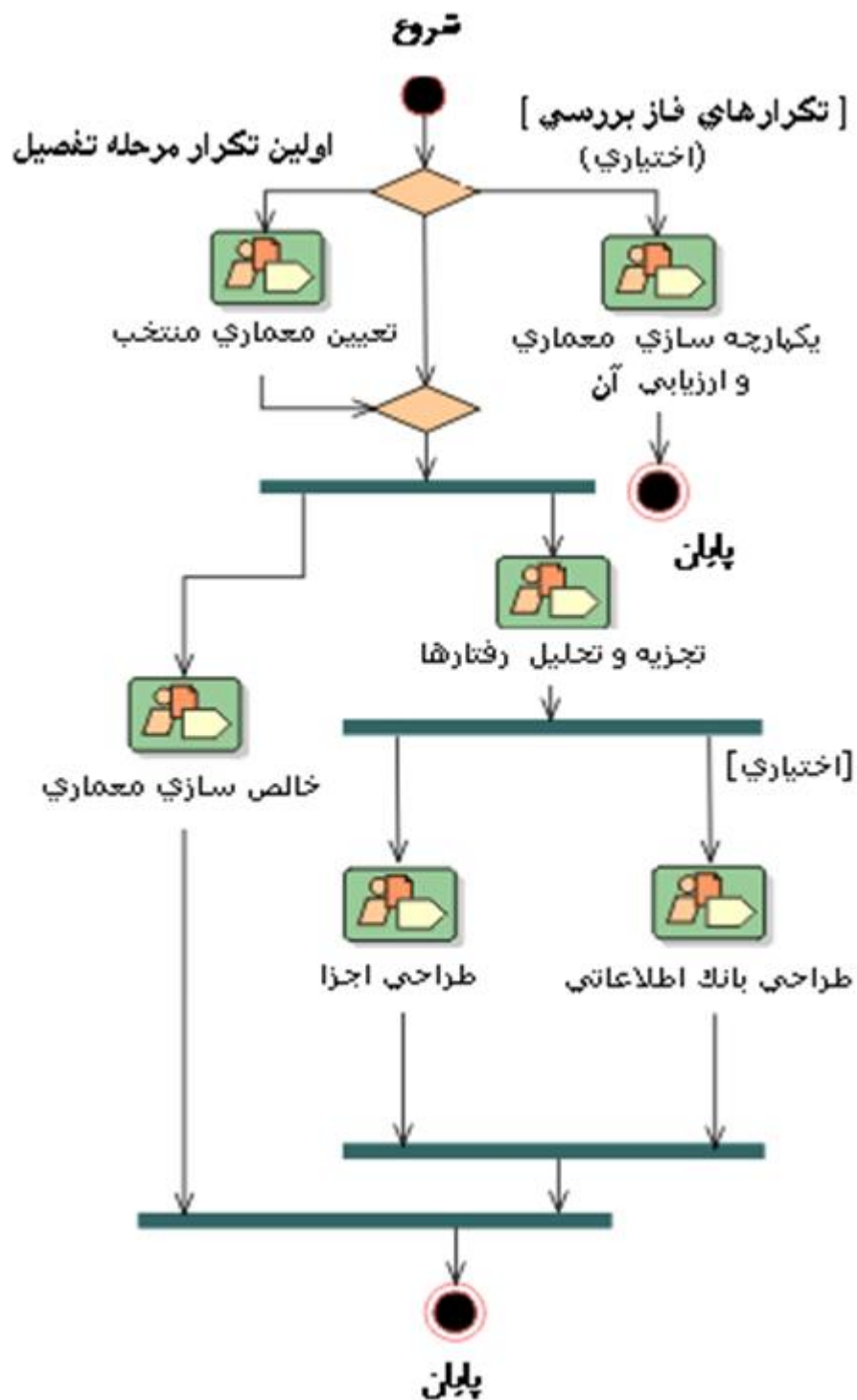
شکل 22- نقش‌ها و محصولات الگوی کاری مدل سازی حرفه‌ای

3-12-3- الگوی کاری تحلیل و طراحی

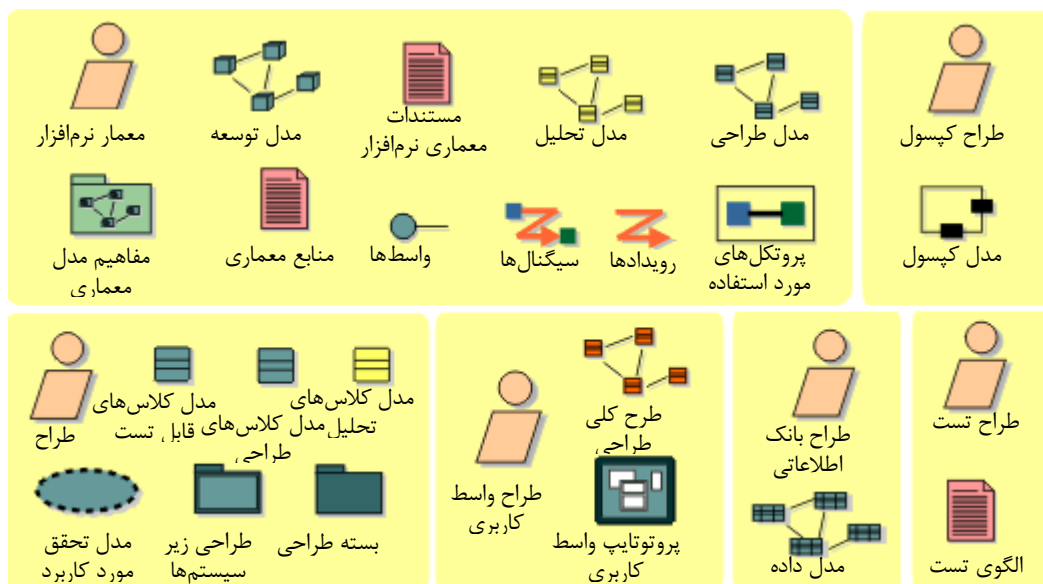
اهداف

اهداف مجموعه فعالیت‌های مربوط به الگوی تحلیل و طراحی عبارتند از :

- § تبدیل نیازمندی‌ها به طراحی سیستم
- § استخراج یک معماری قوی برای سیستم
- § سازگار کردن طراحی با محیط پیاده سازی



شکل 23- الگوی کاری تحلیل و طراحی



شکل 24- نقش ها و محصولات الگوی کاری تحلیل و طراحی

3-12-4- الگوی کاری پیاده سازی

اهداف

هدف از پیاده سازی عبارت است از :

§ تعریف سازماندهی متن برنامه (کد) از لحاظ پیاده سازی زیر سیستم های تفکیک شده در

لایه ها

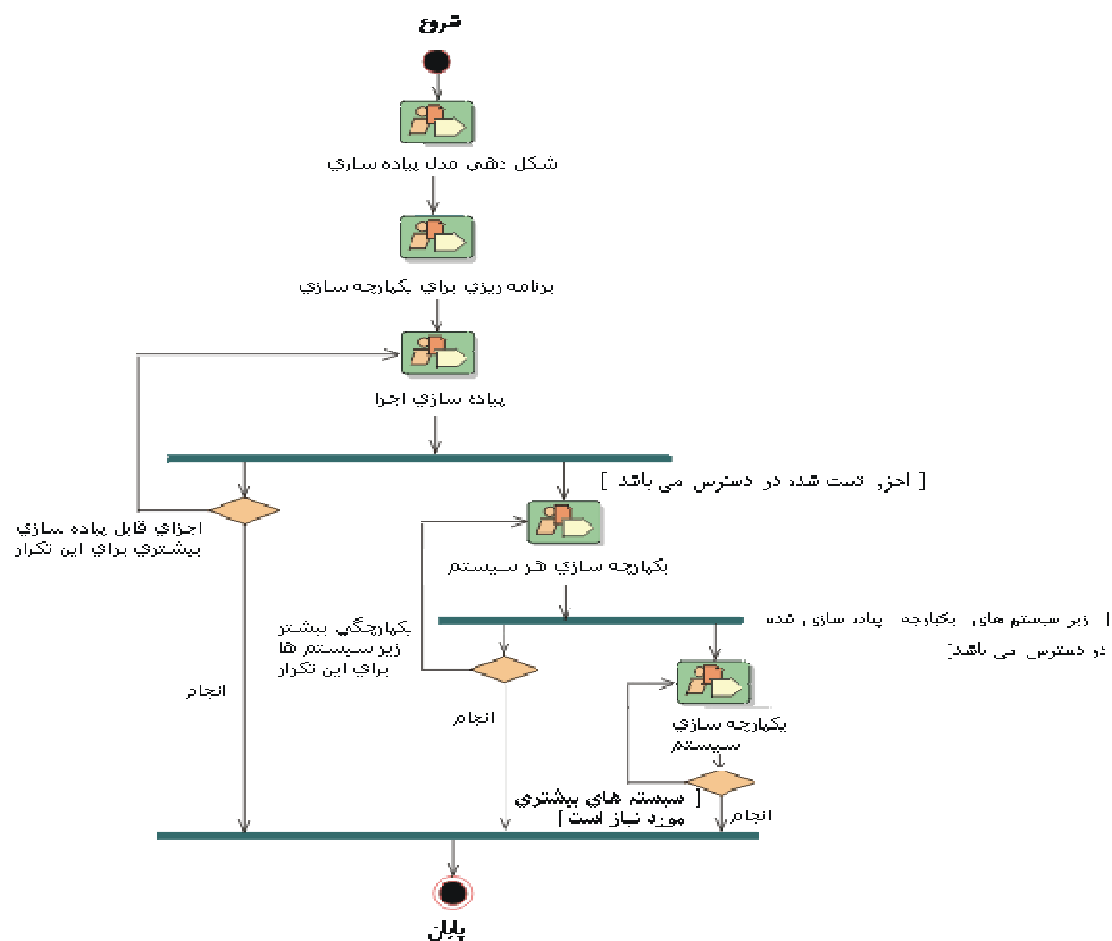
§ پیاده سازی اجزاء شامل کلاس ها و اشیاء ها

§ تست کردن اجزاء ایجاد شده بعنوان یک unit

§ یکپارچگی سازی نتایج کار برنامه نویسان یا تیم های برنامه نویسی در قالب یک برنامه

اجرائی

§ در الگوی پیاده سازی فقط به تست کلاس پرداخته می شود و تست سیستم و تست یکپارچگی در الگوی تست توضیح داده شده است.



شکل 25- الگوی کاری پیاده سازی



شکل 26- نقش ها و محصولات الگوی کاری پیاده سازی

3-12-5-الگوی کاری تست

Ø اهداف

الگوی کاری تست در بسیاری از موارد به عنوان یک سرویس دهنده به سایر الگوهای کاری عمل می نماید. مجموعه فعالیتهای مرتبط با تست عمدتاً بر ارزیابی و اظهار نظر در مورد کیفیت محصول از طریق اقدامات اساسی زیر متمرکز می باشد :

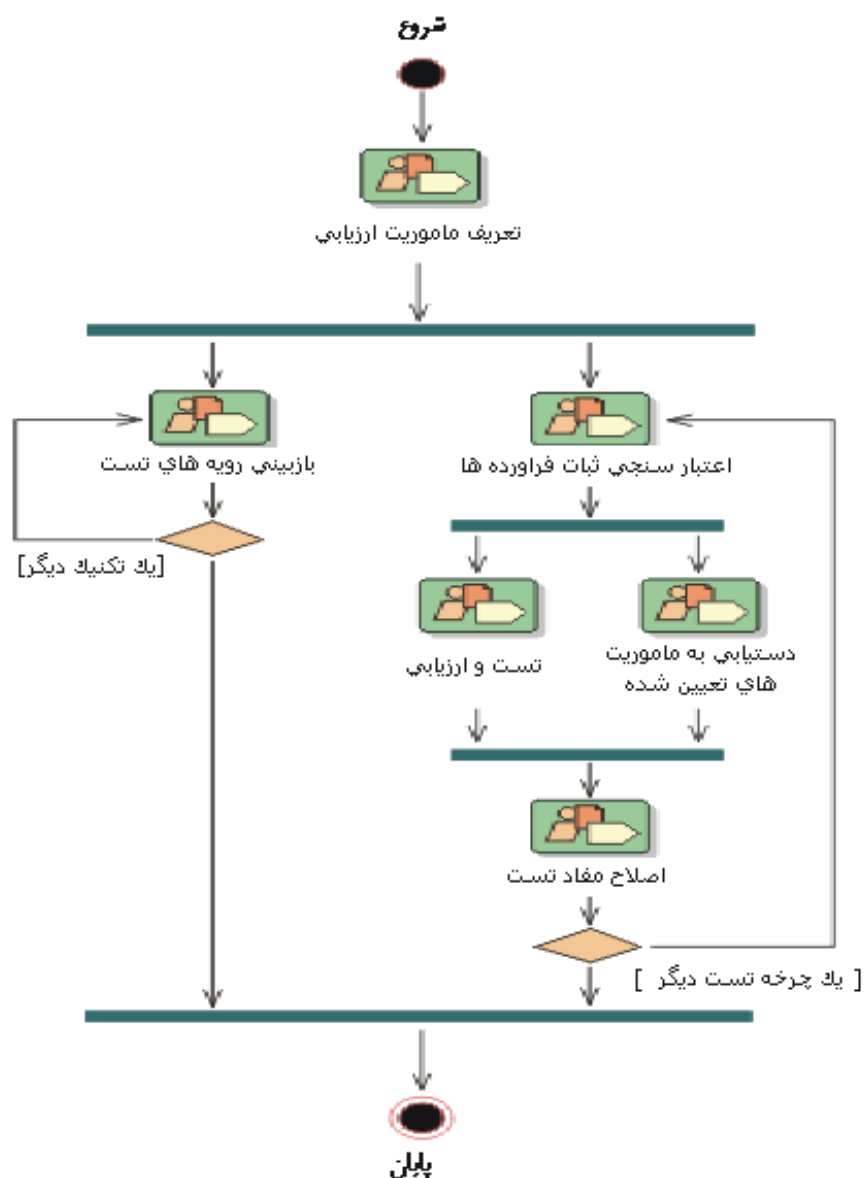
§ یافتن و مستند کردن نقایص کیفی نرم افزار

§ قضاوت عمومی درباره کیفیت نرم افزار دریافت شده

§ اثبات اعتبار فرضیات مطرح شده در طراحی فرمها مشخصات نیازمندیها از طریق نمایش های ارائه شده

§ اعتبار سنجی توابع محصول نرم افزای بر اساس طراحی

§ اعتبار سنجی اینکه نیازمندیها بطور مناسب پیاده سازی شده اند



شکل 27- الگوی کاری تست



شکل 28- نقش ها و محصولات الگوی کاری تست

3-12-6- الگوی کاری استقرار

اهداف

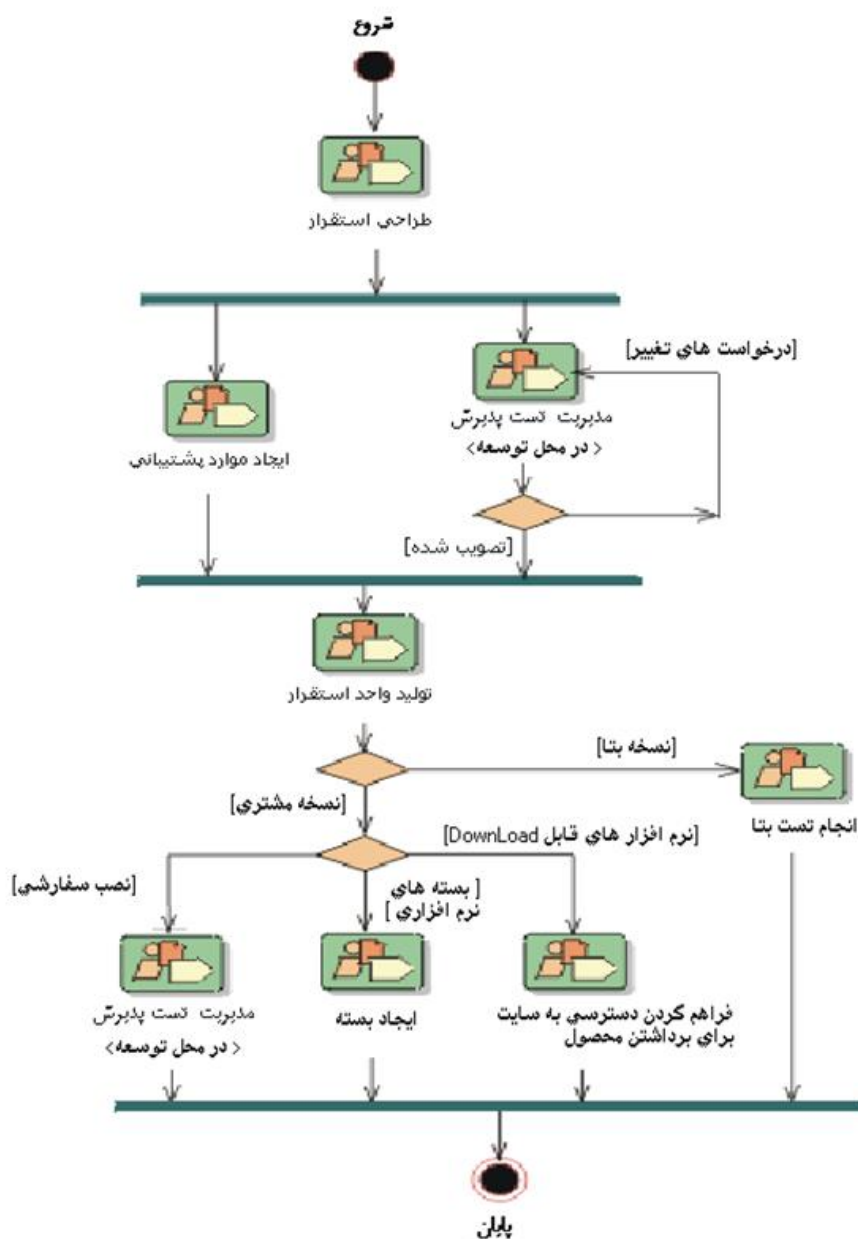
الگوی کاری استقرار مرتبط به فعالیتهای به منظور کسب اطمینان از دسترسی آخرین سطح کاربران به محصول نرم افزاری می باشد.

الگوی کاری استقرار سه روش را برای استقرار محصول تشریح می نماید:

§ نصب سفارشی

§ ارائه محصول بصورت بسته نرم افزاری

§ دسترسی به نرم افزار از طریق اینترنت



شکل 29- الگوی کاری استقرار

در هر حالت، تأکید بر روی تست محصول در محل توسعه و ایجاد نرم افزار بوده و می بایست تست Beta بر روی محصول انجام شده و سپس محصول تحویل مشتری گردد.

اگرچه پیک کاری استقرار در فاز انتقال می باشد اما به منظور طراحی و آماده سازی زمینه های استقرار بعضی فعالیت ها در فازهای قبلی انجام می شوند.



شکل 30 - نقش ها و محصولات الگوی کاری استقرار

3-12-7- الگوی کاری مدیریت پیکربندی و تغییرات

براساس مدل بلوغ قابلیتها (CMM) مربوط به انجمن مهندسی نرم افزار، الگوی کاری پیکربندی و مدیریت تغییرات اعمال شده در دستاوردهای پروژه و یکپارچگی آنها با سایر مستندات را کنترل می نماید.

پیکربندی و مدیریت درخواستهای تغییر شامل موارد زیر می باشد :

- § ویرایش های متعدد
- § تعیین موارد پیکربندی شده
- § محدود کردن تغییرات به موارد فوق
- § ممیزی تغییرات انجام شده در موارد مذکور
- § تعریف و مدیریت پیکربندی موارد مذکور

متدها، فرآیندها و ابزار استفاده شده در تغییرات و مدیریت پیکربندی یک سازمان را می توان تحت عنوان سیستم مدیریت تغییرات یک سازمان تعریف نمود.

سیستم مدیریت پیکربندی و درخواستهای تغییر اطلاعات کلیدی را درباره توسعه محصول، رشد محصول، پیاده سازی و فرآیندهای نگهداری و نگهداری از دستاوردهای قابل استفاده که از اجرای این فرآیند حاصل شده اند، نگهداری می کند.

سیستم مدیریت تغییرات یک قسمت اساسی و کامل از سرتاسر فرآیندهای توسعه می باشد.



شکل 31- نقش ها و محصولات الگوی کاری مدیریت پیکره بندی و تغییرات

3-12-8- الگوی کاری مدیریت پروژه

اهداف

هدف ما در این بخش ساده تر کردن فعالیتها بوسیله فراهم ساختن برخی مفاهیم برای مدیریت پروژه می باشد. این توضیحات دستور العملی برای موفقیت نیست اما رویه ای را برای مدیریت پروژه ارائه می دهد که بطور مشخص احتمال موفقیت آمیز بودن نرم افزار را افزایش می دهد.

هدف از مدیریت پروژه عبارت است از:

§ فراهم نمودن چارچوبی برای مدیریت پروژه های نرم‌افزاری

§ فراهم نمودن راهنماهای عملی برای برنامه ریزی، تامین منابع انسانی، اجرا و ردیابی پروژه

ها

§ فراهم نودن چارچوبی برای مدیریت ریسک

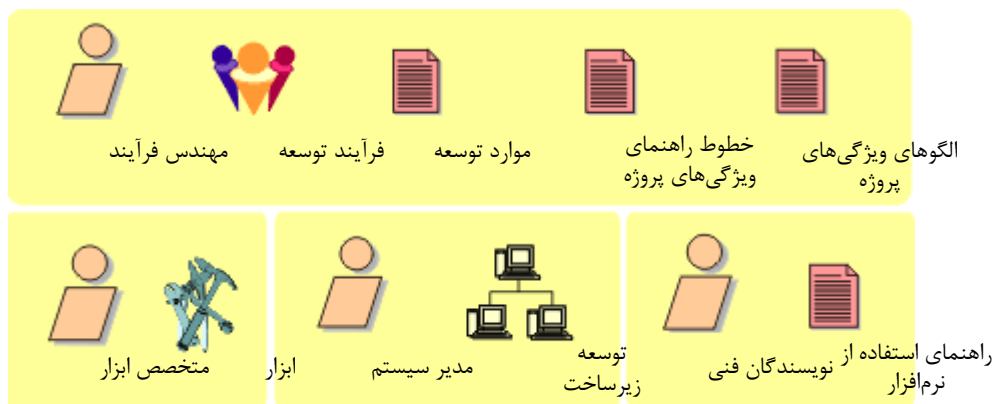


شکل 32- نقش‌ها و محصولات الگوی کاری مدیریت پروژه

3-12-9- الگوی کاری شکل دهی محیط پروژه

اهداف

الگوی محیط پروژه بر روی فعالیتهای لازم برای شکل دهی فرآیندهای پروژه متمرکز شده و فعالیتهای مورد نیاز برای توسعه راهنماهای پروژه را تشریح می نماید و هدف از این کار ایجاد سازمان توسعه نرم فزار به همراه محیط توسعه نرم‌افزار (یعنی هر فرآیندها و هم ابزار) به منظور حمایت از تیم توسعه می باشد.



شکل 34 - نقش‌ها و محصولات الگوی کاری شکل دهی محیط پروژه

3-13- معرفی زبان استاندارد مدلسازی UML

3-13-1- مقدمه

زبان UML یک زبان استاندارد برای نمایش، ایجاد و مستندسازی سیستم های نرم افزاری مبتنی بر روشهای شی گرا می باشد. قبل از UML نیز روشهای شی گرایی متعددی توسط افراد مختلف برای مدل سازی سیستم های شی گرا ارائه شده بود. اتفاقی که باعث ایجاد UML شد بدین صورت بود که Rumbough، طراح متدلوژی OMT به شرکت Rational که متعلق به Booch بود پیوست و آنها تلاش خود را برای ایجاد یک زبان مدلسازی شی گرای متحدالشکل بکار گرفتند. ترکیب دومتدلوژی و ایجاد زبان UML اعتبار ویژه ای به آن بخشید. در سال 1995، شرکت Rational آماده بود تا اولین مستندات UML (نسخه 0.8) را ارائه نماید، اما در یک اقدام ناگهانی امتیاز شرکت Jacobson را که مالک Objectory بود، خریداری نمود. پس از این اقدام، شرکت Rational با ترکیب سه متدلوژی سطح بالا قادر به ارائه یک استاندارد در روشهای شی گرا بود. (OMG Unified Modeling Language)

در سال 1997، UML بعنوان یک زبان استاندارد مدلسازی شی گرا از طرف گروه ⁹⁷OMG پذیرفته شد. مهمترین قابلیت این زبان ارائه مدلهایی بصورت دیاگرام برای کل چرخه حیات نرم افزار است و می تواند بصورت یک زبان ارتباطی بین تمام گروههای یک تیم پروژه استفاده شود. از قابلیت های دیگر آن اینست که سازگاری خود را با اغلب روشهای متداول مانند OMT، Booch و OOSE حفظ نموده است.

از دید مستندسازی، UML قادر است کل چرخه حیات سیستم را در قالب نمودارهایی بصورت کلی و قابل فهم ارائه نماید که می تواند مستقل از متدلوژی ساخت ارائه شود هرچند که برخی از

⁹⁷ Object Management Group

متدلوژیها دیاگرامهای خاص خود را دارند. اما با توجه به نزدیکی متدلوژیهای شیء‌گرا و شباهت دیاگرامهای آنها می‌توان UML را در بسیاری از متدلوژی‌های شیء‌گرا استفاده نمود.

زبان مدل‌سازی ⁹⁸UML مستقیماً روشهای Booch, Rumbaugh و Jacobson را با هم ادغام کرده است، اما محدوده آن بیشتر از سه متدلوژی فوق است. این زبان همانطور که از نامش پیدا است یک زبان مدل‌سازی است تا یک متدلوژی. بطور معمول، هر متدلوژی شامل حداقل یک زبان مدل‌سازی و یک پروسه ساخت است. زبان مدل‌سازی شامل نمودارهایی است که هر متدلوژی برای نمایش تحلیل و طراحی سیستم‌ها از آن استفاده می‌کند. اما یک پروسه ساخت شامل دستورات، راهنمایی‌ها و قدم‌های لازم برای انجام تحلیل و طراحی سیستم‌ها می‌باشد. افرادی که یک متدلوژی را به کار می‌برند معمولاً بیشتر با زبان مدل‌سازی آن سروکار دارند. هدف طراحان UML بیشتر تدوین یک زبان مدل‌سازی شیء‌گرا بوده تا ارائه یک پروسه ساخت استاندارد.

3-13-2- دیاگرام‌های UML

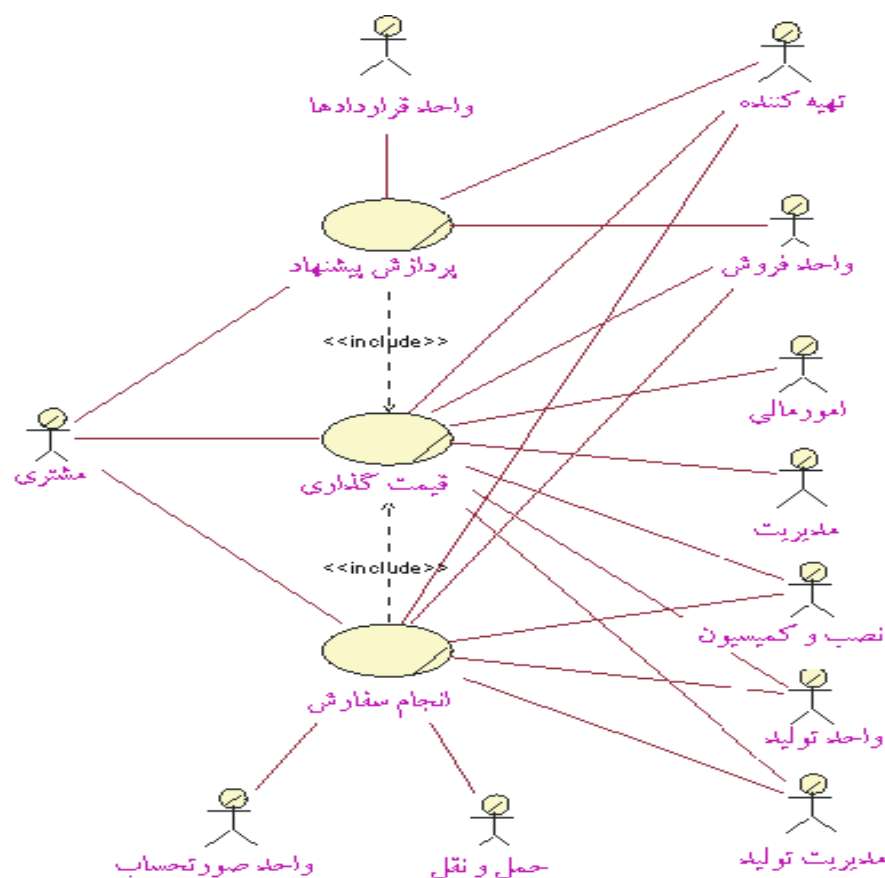
یکی از بزرگترین اهداف طراحی برنامه‌های سیستم نرم‌افزاری ایجاد برنامه‌های صحیح است به نحوی که نیازهای کاربران را بدرستی و با هزینه قابل قبولی برآورده نماید. فهم نیازهای کاربران که مستلزم ایجاد ارتباط با آنهاست، یکی از نکات کلیدی در ایجاد نرم‌افزارهای مفید می‌باشد. روشی که در UML برای نشان دادن این خواسته‌ها بکارگرفته می‌شود Use Case نام دارد. مجموعه تمامی Use Case ها، تصویر خارجی سیستم را تشکیل می‌دهد. یک مجموعه خوب از Use Case ها وقتی حاصل خواهد شد که طراح بداند کاربران چه چیزی از سیستم می‌خواهند. Use Case ها همچنین ابزار خوبی برای پیشبرد پروژه می‌باشند زیرا ساخت سیستم به طریق تکراری را کنترل می‌کند. در واقع از قابلیت‌های ویژه متدلوژی‌های شیء‌گرا این است که می‌توان یک مجموعه‌ای از نیازها را انتخاب نموده و طراحی و پیاده‌سازی آنرا به پیش برد و بدین صورت ساخت

⁹⁸ Unified Modeling Language

سیستم به صورت افزایشی را مقدور می‌سازد همچنین با توجه به این که تیم طراح و برنامه‌نویس در هر مرحله با Use Case ها سروکار دارد، در هر مرحله امکان بازگشت سریع به مراحل قبل وجود دارد. این روش خود تکنیک با ارزشی است، زیرا نتایج را مرحله به مرحله به کاربر باز خور می‌دهد. در ادامه این بخش، دیاگرام‌هایی که در UML برای مدل‌سازی سیستم‌های نرم‌افزاری استفاده می‌شود به صورت اجمالی معرفی می‌گردد.

Ø دیاگرام Use Case

در ساده‌ترین حالت Use Case ها بوسیله مصاحبه با کاربران و انتظاراتی که آنها از سیستم دارند تعریف می‌شود. برای نمایش اطلاعات مربوط به نیازهای کاربران که در قالب Use Case ها جمع‌آوری شده است از دیاگرام Use Case استفاده می‌شود. شکل زیر نمونه‌ای از این دیاگرام را نشان می‌دهد



شکل 35 - نمونه‌ای از دیاگرام Use Case

نمادهای آدمک که در شکل دیده می‌شود، بازیگر⁹⁹ نامیده می‌شوند. هر بازیگر می‌تواند یک کاربرسیستم باشد که نقش یا نقش‌هایی را در سیستم بازی می‌کند. یک بازیگر لزومی ندارد حتماً یک انسان باشد، یک سیستم خارجی نیز میتواند یک بازیگر باشد.

چهار نوع رابطه در دیاگرام Case Use وجود دارد که عبارتند از :

الف - رابطه Communication: نشان می‌دهد یک بازیگر، یک Use Case را استفاده می‌کند.

ب - رابطه Extend: ارتباط بین دو Use Case که حالات خاص یکی در دیگری قرار داده می‌شود.

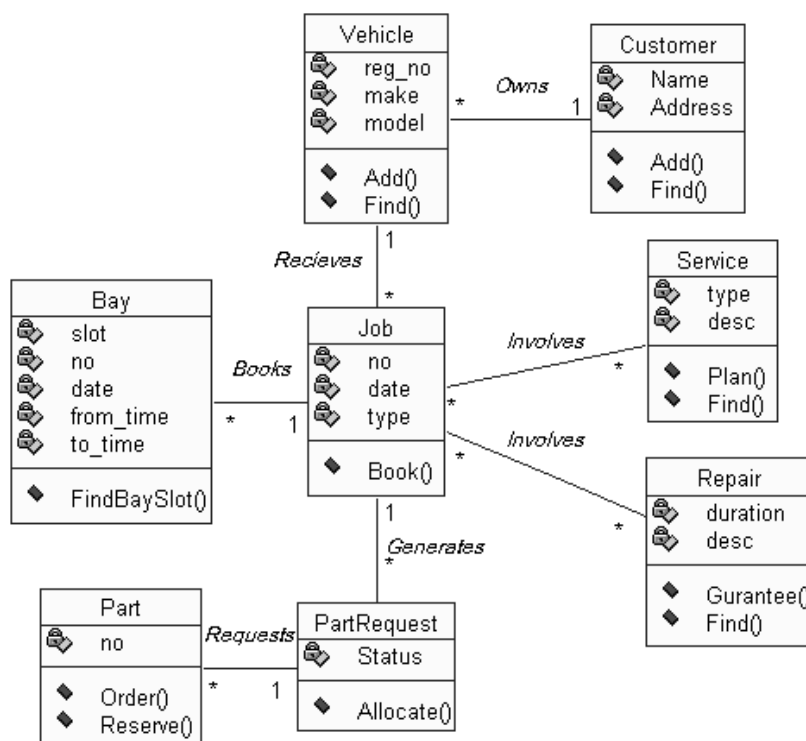
ج - رابطه Uses: شبیه Extend می‌باشد ولی در اینجا حالات خاص را در یک Use Case قرار نمی‌دهد، بلکه اگر رفتاری در چند Use Case مشترک باشد، آنرا جدا کرده و در یک Use Case مجزا قرار می‌دهند.

Ø دیاگرام کلاس

دیاگرام کلاس یکی از دیاگرام‌های مهم و اساسی در متدلوژیهای شی‌گرا می‌باشد که هر متدلوژی حالات مختلفی از آنرا استفاده می‌کند. دیاگرام کلاس شامل اشیاء و روابط مابین آنها می‌باشد. همچنین دیاگرام کلاس شامل صفات و رفتار کلاس‌ها می‌باشد.

در دیاگرام کلاس اگر فقط اشیاء (یعنی نمونه‌های کلاس‌ها) و روابط آنها نشان داده شود، آن را دیاگرام شیء¹⁰⁰ گویند. شکل شماره زیر نمونه‌ای از دیاگرام کلاس را نمایش می‌دهد. در ادامه مفاهیمی که در دیاگرام کلاس استفاده می‌شود بطور مختصر شرح داده می‌شود.

⁹⁹ Actor



شکل 36- نمونه‌ای از دیاگرام کلاس

Ø کلاس

کلاس، مشخصه‌ای برای مجموعه‌ای از اشیاء با صفات، رفتار و روابط مشابه می‌باشد. زبان UML برای نمایش کلاس از نماد مستطیل سه قسمتی استفاده می‌کند. قسمت فوقانی این نماد برای نمایش مشخصات نام کلاس، قسمت میانی برای نمایش مشخصات صفات و قسمت تحتانی برای نمایش مشخصات رفتار و اعمال کلاس بکار می‌رود. برخی عناصر که از نظر مفهوم شبیه کلاس هستند نیز از همین نماد و خواص استفاده می‌کنند.

کلاس‌های مشتق شده از یک کلاس معمولاً با مفهوم کلیشه¹⁰¹ بیان می‌شود. این مفهوم در UML استفاده زیادی دارد و حالات خاص و مختلف از یک چیز را نشان می‌دهد. انواع مختلفی که از نظر

¹⁰⁰ Object Diagram

¹⁰¹ Stereotype

مفهومی شبیه کلاس هستند با ذکر یک کلیشه، در قسمت نام کلاس مشخص می‌شوند. سه نمونه از این مفاهیم، تایپ¹⁰²، کلاس پیاده‌سازی¹⁰³، و واسط¹⁰⁴ نام دارند.

تایپ، نشان‌دهنده یک نقش قابل تغییر است که یک شیء می‌تواند انتخاب کند و سپس آنرا کنار بگذارد. یک کلاس پیاده‌سازی، نشان‌دهنده ساختار فیزیکی و توابع یک شیء است که در یک زبان، پیاده‌سازی می‌شود و قابل تغییر نیست. یک شیء می‌تواند دارای چندین تایپ باشد اما فقط یک کلاس پیاده‌سازی دارد. بین یک تایپ و یک کلاس پیاده‌سازی می‌توان رابطه Realize برقرار نمود که نشان می‌دهد یک تایپ توسط کدام کلاس پیاده‌سازی شده است. یک واسط نشان‌دهنده اعمال یک کلاس است که قابل رویت توسط دیگران است. واسطها فاقد صفت، حالت و روابط انجمنی هستند و فقط شامل اعمال می‌باشند. واسطها می‌توانند روابط کلی/ اختصاصی داشته باشند.

کلاس پارامتری¹⁰⁵ معرف کلاسی است که تعدادی پارامتر داشته ولی می‌تواند خانواده‌ای از کلاسها را تعریف کند. هر کلاس از این خانواده با تعریف مقادیر واقعی پارامترها، مشخص می‌شود. یک کلاس پارامتری با همان نماد کلاس نمایش داده می‌شود. کلاس‌هایی که به یک کلاس پارامتری محدود یا مقید می‌شوند، عناصر مقید¹⁰⁶ نامیده می‌شوند.

شیء نمونه‌ای از یک کلاس است که صفاتش دارای مقادیر حقیقی هستند. نماد شیء مانند نماد کلاس است که در زیر نام آن خط کشیده می‌شود. یک شیء مرکب¹⁰⁷ شیء سطح بالایی است که در درون خود دارای شیء‌هایی است که این شیء‌ها می‌توانند دارای روابط فی مابین باشند.

¹⁰² Type

¹⁰³ Implementation Class

¹⁰⁴ Interface Class

¹⁰⁵ Parameterized Class

¹⁰⁶ Bound Element

¹⁰⁷ Composite Object

Ø روابط مابین کلاس‌ها

روابط مابین کلاسها و اشیاء در زیر به انواع روابط بین کلاسها و اشیاء اشاره می‌شود.

الف - رابطه انجمنی

یک رابطه انجمنی می‌تواند بین دو کلاس و یا چند کلاس برقرار شود. مشخصات کلی این رابطه توسط چند خصیصه تعیین می‌شود که عبارت است از مقدار کثرت، ترتیب، علامت جهت و

مشخصه اجماع¹⁰⁸

ب - رابطه در برگیری¹⁰⁹

این رابطه یک رابطه قویتر از رابطه اجماع می‌باشد که در آن وجود کلاس جزء وابسته به وجود کلاس کل می‌باشد. رابطه دربرگیری با یک خط راست که در طرف کلاس کل یک لوزی توپر قرار دارد نمایش داده می‌شود.

ج - رابطه کلی - اختصاصی¹¹⁰

رابطه‌ای است که بین یک کلاس عمومی و کلاس‌های اختصاصی آن کلاس برقرار می‌شود. در این نوع رابطه، کلاس‌های اختصاصی تمامی خصوصیات کلاس عمومی را به ارث برده و علاوه بر آن می‌توانند یکسری خصوصیات دیگر را نیز تعریف نمایند. این رابطه را رابطه ارث‌بری نیز می‌نامند.

د - رابطه وابستگی

این رابطه یک رابطه معنایی را بین دو عنصر تعریف می‌کند که تغییر در یک عنصر ممکن است باعث تغییراتی در عنصر دیگر شود. از انواع این روابط می‌توان به `uses.bind` اشاره نمود.

هـ - رابطه Realize

رابطه‌ای است که بین کلاس پیاده‌سازی و تایپ برقرار می‌شود.

¹⁰⁸ Aggregation

¹⁰⁹ Composition Whole-Part

¹¹⁰ Generalization / Specialization

Ø دیاگرام تعامل

یک سیستم علاوه بر ساختار ایستا دارای رفتار پویا، مبتنی بر ساختار ایستای خود است که بیان کننده نحوه و ترتیب ارتباط اجزای مختلف سیستم با یکدیگر است. این جنبه بوسیله نمودار تعامل مدلسازی می گردد. نحوه تعامل و ترتیب فرستادن پیامها میان کلاسها در یک سیستم شیء‌گرا باید صریحاً مدلسازی و تعیین گردد. مدلسازی تعامل به این جنبه از سیستم توجه دارد. در واقع پرداختن به بیان رفتار ایستای سیستم به هیچ وجه نمی‌تواند جایگزین مشخص کردن رفتار پویای آن باشد.

در مدلسازی تعامل، همکاری گروهی از اشیاء در انجام یک عمل معین از طریق بیان ترتیب تبادل پیامها میان آنها، نشان داده می‌شود. این عمل می‌تواند یک مورد کاربری باشد که در این صورت نمودار تعامل رفتار یک مورد کاربری را با توجه به اشیاء شرکت کننده در آن و ارتباطات ما بین این اشیاء نشان خواهد داد.

دیاگرام تعامل¹¹¹ بیانگر رفتار اشیاء در روابط با یکدیگر است. معمولاً هر دیاگرام تعامل، رفتار یک Use Case را نشان می‌دهد. دیاگرام تعامل خود به دو نوع دیاگرام ترتیب¹¹² و دیاگرام همکاری¹¹³ تقسیم می‌شود.

¹¹¹ Interaction Diagram

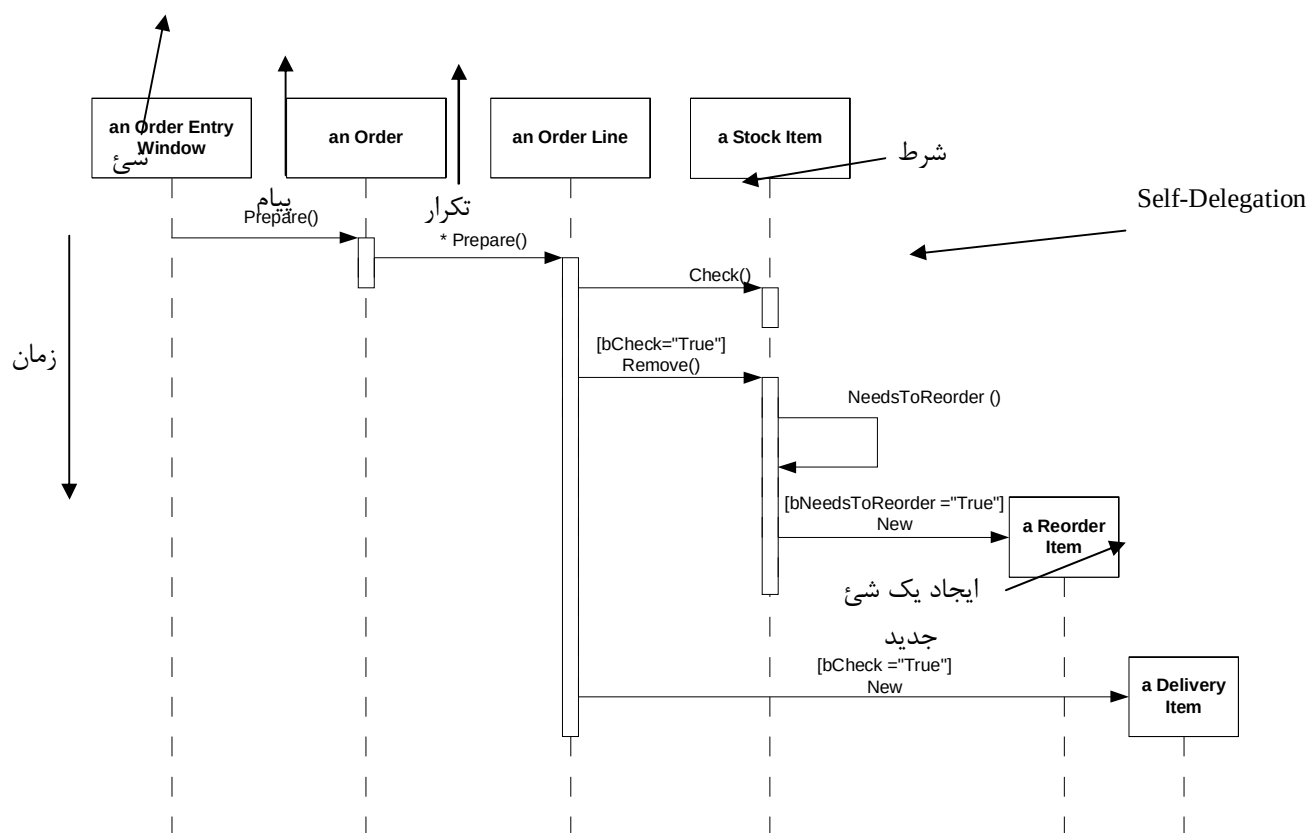
¹¹² Sequence Diagram

¹¹³ Collaboration Diagram

الف - دیاگرام ترتیب

یک نمونه از این دیاگرام در شکل زیر نشان داده شده است. عملیات بصورت ارسال پیام ما بین اشیا انجام می‌گیرد. ترتیب ارسال پیام از بالا به پایین می‌باشد. هر پیام دارای یک نام بوده و می‌تواند پارامتر و اطلاعات کنترلی نیز داشته باشد. دو نوع اطلاعات کنترلی می‌تواند در کنار پیام نوشته شود، یکی شرط ارسال است که با بر آورده شدن آن پیام ارسال می‌شود و دیگری علامت تکرار است که

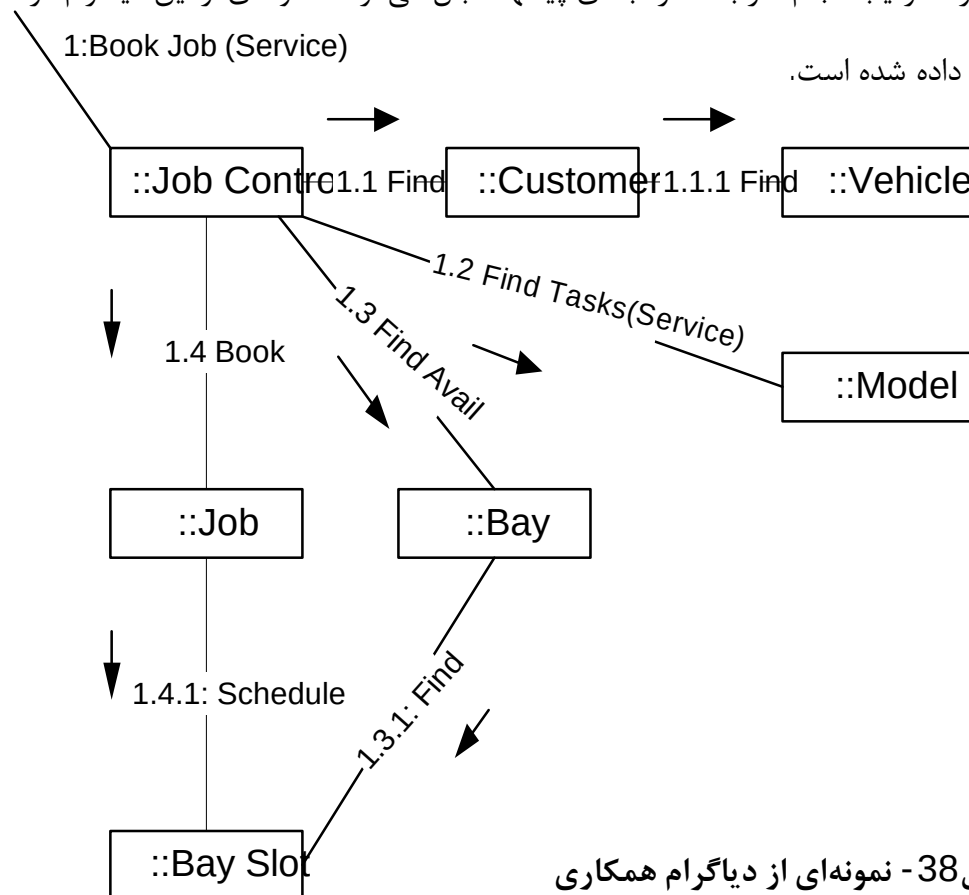
مشخص می‌کند پیام چند مرتبه به اشیا فرستاده می‌شود. اشیاء



شکل 37 - نمونه‌ای از دیاگرام ترتیب

ب - دیاگرام همکاری

نمودار همکاری بر روی سازماندهی اشیاء شرکت کننده در یک تعامل تاکید می‌کند. این نمودار معمولاً برای بیان نحوه اجرای سناریوها استفاده می‌شود. در این نمودار اشیاء در قالب مستطیلها نمایش داده می‌شوند و فلشها نیز نشان دهنده پیامهای رد و بدل شده بین اشیاء هستند. اما در این نمودارها ترتیب انجام کار با شماره بندی پیامها دنبال می‌گردد. نمونه‌ای از این دیاگرام در شکل زیر نشان داده شده است.



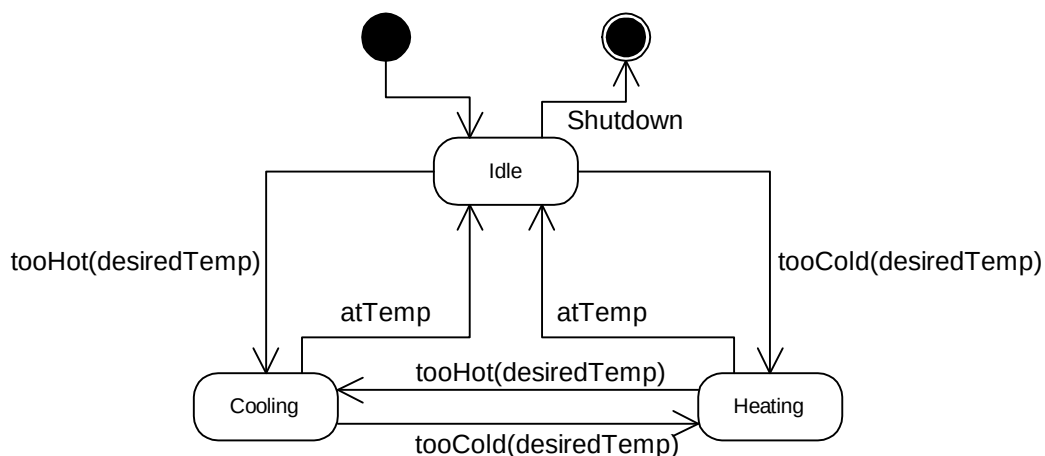
شکل 38 - نمونه‌ای از دیاگرام همکاری

Ø دیاگرام حالت¹¹⁴

دیاگرام حالت یکی از دیاگرامهای مرسوم در توصیف رفتار سیستم می‌باشد. از دیاگرام حالت می‌توان برای نمایش حالات مختلفی از نمونه‌های کلاس استفاده نمود. هر حالت با نماد مستطیل

¹¹⁴ State Diagram

خواهید با گوشه‌های مدور نمایش داده می‌شود. در داخل مستطیل نام حالت نوشته می‌شود. همچنین اگر در یک حالت، عملی انجام شود در اینصورت به شکل Activity / Do نوشته می‌شود. نمودار حالت بر روی بیان رفتار یک شیء در دوران حیات خود تمرکز می‌یابد. توجه داشته باشید که اینجا مفهوم «شیء» می‌تواند یک نمونه از یک کلاس، یک مورد کاربری، یک زیر سیستم، و یا حتی تمام سیستم باشد. در واقع، مفهوم کلاس جنبه ایستای یک موجودیت را دربر می‌گیرد در حالیکه مفهوم حالت روی رفتار آن موجودیت تمرکز می‌یابد. شکل زیر یک دیاگرام حالت را نمایش می‌دهد.



شکل 39 - نمونه‌ای از دیاگرام حالت

Ø دیاگرام فعالیت

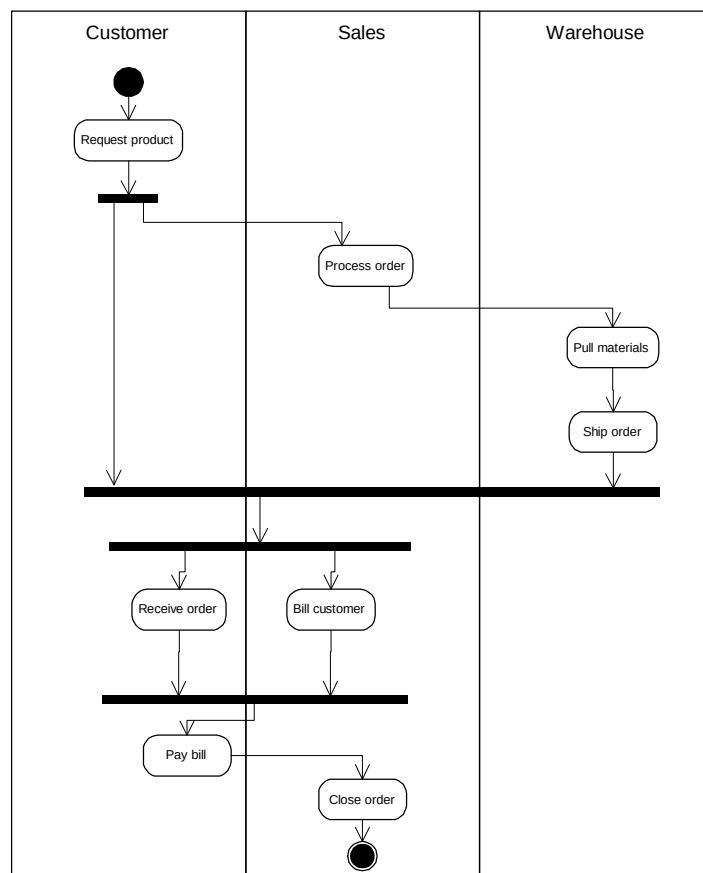
نمودار فعالیت نوعی فلوچارت است که جریان کنترل از یک فعالیت به فعالیت دیگری را نمایش می‌دهد. تفاوت این مدل با یک فلوچارت معمولی این است که از نمایش جریانهای همزمان نیز پشتیبانی می‌کند. همانطوریکه بیان کردیم نمودار تعامل بر روی مدلسازی جریانهای کنترلی میان

اشیاء تاکید نموده در حالیکه نمودار فعالیت بر روی مدلسازی جریان کنترلی میان فعالیتها که هر کدام منتسب به یک شیء هستند تاکید می‌کند.

در واقع، نمودار فعالیت نوع ویژه‌ای از نمودار ماشین حالت محسوب می‌شود، بدین صورت که در یک ماشین حالت اگر به بیشتر حالتها یک فعالیت (یا کنش) منسوب کنیم و آن فعالیت (یا کنش) را بعنوان ویژگی اصلی این حالت بدانیم و نیز بیشتر انتقالها به محض اتمام فعالیتهای مبدا اجرا شده باشند، آنگاه یک **نمودار فعالیت** خواهیم داشت. نمودار فعالیت در مدلسازی توالی و همزمانی اجرای فعالیتهای مربوط به یک موجودیت استفاده می‌شود. این موجودیت می‌تواند تقریباً شامل هر عنصر مدلسازی مانند یک سیستم کامل، یک زیر سیستم، یک عمل، و یا یک کلاس باشد. اما، بیشتر اوقات از نمودار فعالیت به دو شکل ذیل استفاده می‌گردد:

1- **مدلسازی یک گردش کار:** در اینجا بر فعالیتهایی تاکید می‌شود که از دیدگاه عواملی که با سیستم ارتباط دارند، قابل مشاهده است. گردش کارها، معمولاً در مدلسازی فرآیندها کاری یک سازمان استفاده می‌شوند.

2- **مدلسازی یک عمل:** در اینجا از نمودار فعالیت بعنوان یک فلوچارت برای مدلسازی جزئیات انجام یک عمل استفاده می‌شود.



شکل 40- نمونه‌ای از دیاگرام فعالیت

3-14- دلایل حرکت به سمت استفاده از Component در توسعه نرم افزار

در اواخر دهه 60 که بحران نرم افزار، به عنوان یکی از مشکلات عمده سازمان های توسعه دهنده نرم افزار مطرح شد. تا به آن زمان نرم افزارهای تولید شده کاملاً مبتنی بر سفارش بودند و برای تولید نرم افزارهای مختلف و هرچند مشابه فرآیند خاص توسعه نرم افزار طی می شد. انجام چنین کاری باعث می شد تا سازمان ها زمان و هزینه زیادی را برای توسعه هر نرم افزار مصرف نمایند و به همین دلیل نرم افزارها بسیار هزینه بر بوده و سازمان های می بایست برای دستیابی به آنها هزینه های زیادی را می پرداختند.

با کمی دقت می توان متوجه شد که چرخه و فرآیند تولید نرم افزار مانند فرآیند تولید صنعتی است. پس از انقلاب صنعتی و به وجود آمدن مفهوم تولید انبوه سازمان های صنعتی به این فکر افتاد که تولید خود را با کیفیت تر و کم هزینه تر نمایند. به طور مثال می توان به شرکت فورد اشاره نمود که برای تولید هر نوع ماشین در این شرکت سنگ آهن از معدن وارد کارخانه می شد و اتومبیل از کارخانه خارج می شد. انجام کل چنین فرآیند بزرگ و پیچیده ای در یک سازمان هزینه بسیار زیادی را برای سازمان ها ایجاد می کرد. بنابراین سازمان ها به این سمت حرکت کردند که فرآیند تولید خود را به بخش های مختلف تقسیم کنند و هر بخش را به سازمان های خارجی برونسپاری کنند و سازمان اصلی صرفاً مسئولیت مونتاژ بخش های مختلفی را که در دیگر سازمان ها تولید شده است بر عهده می گیرد. انجام چنین کاری بسیاری از هزینه های تولید و کیفیت را کاهش می دهد و سازمان های خارجی برای اینکه در بازار رقابتی باقی بمانند موظف به رعایت استانداردهای کنترلی خاص و دقیق می شوند. هدف سازمان های صنعتی بزرگ از انجام چنین کاری دو نکته بود:

(<http://en.wikipedia.org>)

1. کاهش هزینه های تولید و کیفیت
2. مستقل کردن بخش های مختلف تولیدی به صورتی که بتوان از هر کدام به صورت تکرارپذیر در بخش ها و محصولات مختلف سازمان استفاده برد. (به این معنی که برای

تولید هر محصولی مانند یک اتومبیل جدید که شباهت‌های زیادی با مدل‌های قبلی خود دارد، دیگر لازم نیست سازمان کل قطعات مربوط به آن را در یک فراگرد جدید و کاملاً مستقل از روندهای تولیدی قبلی، تولید نماید. بدین ترتیب سازمان می‌تواند از قطعاتی که در مدل‌های قبلی استفاده می‌شده است و در مدل جدید نیز قابل استفاده است، استفاده نماید. (

لازم به ذکر است همین روند یعنی تقسیم کردن نرم‌افزار به بخش‌های مختلف حوزه شرکت‌های نرم‌افزاری نیز می‌تواند صورت پذیرد. بدین معنی که سازمان‌های نرم‌افزاری نیز با الگو گرفتن از این رویکرد به دنبال یافتن راهی برای کاهش زمان و هزینه‌های تولید نرم‌افزار بودند. به این معنی که آنها به دنبال یافتن راهکاری بودند که بتواند از بخش‌های مختلف نرم‌افزار تولیدی خود استفاده کنند، تا بتوانند سرعت و هزینه تولید نرم‌افزار خود را تا جایی که ممکن است کاهش دهند. به همین دلیل سازمان‌ها به سمت شکستن نرم‌افزار به اجزاء¹¹⁵ مختلف حرکت کردند. هدف اصلی از استفاده از اجزاء در علوم نرم‌افزاری، استفاده از قابلیت استفاده مکرر هر جزء در تولید نرم‌افزار می‌باشد.

3-15- مولفه نرم‌افزاری¹¹⁶ چیست؟

مولفه نرم‌افزار عبارت است از اجزاء یک سیستم که به صورت سرویس‌های از پیش تعریف شده پیشنهاد می‌شوند و قابلیت ارتباط با دیگر مولفه‌ها را دارند. زیپرسکی¹¹⁷ و مسرچمیت¹¹⁸ در بررسی‌های خود 5 ویژگی را برای مولفه‌های نرم‌افزاری تعریف می‌کنند که عبارتند از:

(<http://en.wikipedia.org>)

¹¹⁵ Component

¹¹⁶ Software Component

¹¹⁷ Clemens Szyperski

¹¹⁸ David Messerschmitt

1. قابلیت تکرارپذیری در استفاده¹¹⁹
 2. دارای ویژگی غیر متنی¹²⁰
 3. قابلیت تطابق با دیگر مولفه‌ها
 4. به صورت بسته‌ای بودن و بدون نیاز به بررسی مجدد برای استفاده
 5. واحدی است که به صورت مستقل تولید شده و به صورت مستقل نیز قابل استفاده است.
- به عبارت مشخص‌تر مولفه نرم‌افزاری را می‌توان شیئی¹²¹ دانست که به منظور و با هدف دستیابی به یک ویژگی خاص نوشته می‌شود.
- با توجه به مطالب ذکر شده می‌توان عنوان داشت که مولفه‌های نرم‌افزاری همان اشیائی هستند که در متدهای شیء‌گرا مورد استفاده قرار می‌گیرند با این تفاوت که هدف از تولید این مولفه‌ها استفاده از قابلیت استفاده مجدد آنهاست که اشیاء از این قابلیت برخوردار نیستند.

3-16- تفاوت توسعه نرم‌افزار مبتنی بر مولفه‌های نرم‌افزاری با روش‌های توسعه و برنامه‌نویسی شیء‌گرا¹²²

اساس برنامه‌نویسی شیء‌گرا مبتنی بر این اصل است که در روش برنامه‌نویسی نرم‌افزار باید بر مبنای یک مدل ذهنی که به وسیله اشیاء تبدیل به یک مدل واقعی شده است، ارائه شود. روش‌های OOP و دیسپلین‌های مرتبط با آن همچون تحلیل و طراحی شیء‌گرا تمرکز بر مدل‌سازی تعاملاتی دارند که در دنیای واقعی اتفاق می‌افتند و به دنبال آن هستند زبان و خواسته‌های افرادی که به عنوان کاربران مطرح هستند را با برنامه‌نویسان که وظیفه تولید کد نرم‌افزار را دارند به یکدیگر شبیه سازند.

¹¹⁹ Multiple-use

¹²⁰ Non-context-specific

¹²¹ Object

¹²² Object Oriented Programming

درمقابل نرم افزارهایی که تولیدشان مبتنی بر مولفه های نرم افزاری است فرض دیگری را پیش روی خود قرار می دهند و آن عبارت است از اینکه نرم افزار باید به وسیله چسباندن مولفه های مختلف نرم افزاری که همانند اجزاء یک قطعه مکانیکی به هم متصل می شوند، مونتاژ و تولید شود. نکته مهم آن است که مولفه های نرم افزاری از لحاظ ظاهر شباهت زیادی به اشیاء دارند اما در واقعیت از نظر تعریف تفاوت های زیادی با هم دارند. همین تفاوت ها منجر به این شده است که امروز رویکرد جدیدی تحت عنوان برنامه نویسی مولفه گرا¹²³ مطرح شده است. (<http://en.wikipedia.org>)

با توجه به مطالب ذکر شده برای اینکه بتوان از مولفه های نرم افزاری به خوبی بهره برد باید مولفه ها دارای ویژگی های زیر باشند:

1. ویژگی های هر مولفه باید کاملاً مستند شده باشد.
2. باید صحت و اعتبار آنها کاملاً آزمایش شده باشد و بدون خطا باشند.
3. قابلیت برطرف نمودن پیام های خطاهای به وجود آمده در مسیر استفاده از آنها وجود داشته باشد.
4. باید با این آگاهی تولید شوند که ممکن در آینده در هر بخشی از نرم افزار تولید مورد استفاده قرار گیرند و محل استفاده آنها مهم نباشد.

¹²³Component Oriented Programming

بخش 4 - رویکرد، قابلیت ها و فن آوری های معماری سرویس گرا

رویکرد SOA¹²⁴ رویکردی در حال ظهور است که از محاسبات توزیع شده با پیوند ضعیف (حداقل وابستگی به اجزایی که محاسبه در آنها توزیع شده است) مبتنی بر استاندارد و مستقل از پروتکل پشتیبانی می کند. برای ساخت یک معماری سرویس گرا به یک زیر ساخت کاملاً توزیع شده برای ارتباطات و یکپارچه سازی آنها نیاز است؛ که این وظایف توسط ESB ارائه می شود. چارچوب ESB یک چارچوب برای یکپارچه سازی است که از استانداردهای سرویس های وب برای پشتیبانی از الگوهای ارتباطی گوناگون بر روی چندین پروتکل انتقال استفاده می کند و توانمندی های با ارزش دیگری نیز برای معماری های سرویس گرا فراهم می کند.

این بخش مروری بر تاریخچه معماری های نرم افزاری متعارف می کند و آنها را با SOA مقایسه می کند سپس عناصر تشکیل دهنده تشکیل دهنده و توانمندی های SOA را بررسی می کند. در اینجا به جای تمرکز روی فن آوری سرویس های وب روی مباحث معماری سرویس ها و رویکردهای یکپارچه سازی ارائه دهنده سرویس با مصرف کننده سرویس و همچنین نقش دلال سرویس پرداخته می شود. ادامه سناریوهای مختلفی سرویس گرایی برای ایجاد یک معماری یکپارچه ارائه می شود. در نهایت لایه بندی دو نمونه از معماری های سرویس گرای متعارف با جزئیات بیشتری مورد بررسی قرار می گیرد.

4-1- مقدمه

سازمانهای مدرن امروز نیاز دارند به طور موثر و سریعی به فرصت های رقابتی و بازارهای جهانی پاسخ دهند. سازمانها برای رسیدن به چابکی تجاری باید فرآیند های کسب و کار موجود خود را روان کرده و عملکرد برنامه های کاربردی بسته بندی شده و سفارشی متعدد خود را که سرتاسر سازمان را پوشش می دهند به یک شکل کاملاً استاندارد در معرض نمایش بگذارند و یکی از رویکردهای معاصر برای حل این مسائل استفاده از سرویس های وب است که به آسانی می توانند

¹²⁴ Service Oriented Architecture

روی هم قرارگیرند تا تشکیل یک مجموعه از فرآیندهای کسب و کار مستقل و با پیوند ضعیف بدهند. ظهور استانداردهای سرویس وب و توسعه این سرویس ها برای پشتیبانی از یکپارچه سازی فرآیندهای کسب و کار سبب توسعه فنی عمده ای در فضای نرم افزارهای یکپارچه سازی خصوصاً معماری سرویس گرا SOA شده است. هدف این معماری برآورد نیازهای محاسبات توزیع شده با پیوند ضعیف مبتنی بر استاندارد و مستقل از پروتکل است به شکلی که تناظر یک به یک بین سیستم های اطلاعاتی سازمانی و فرآیندهای کسب و کار کلی سازمان برقرار شود.

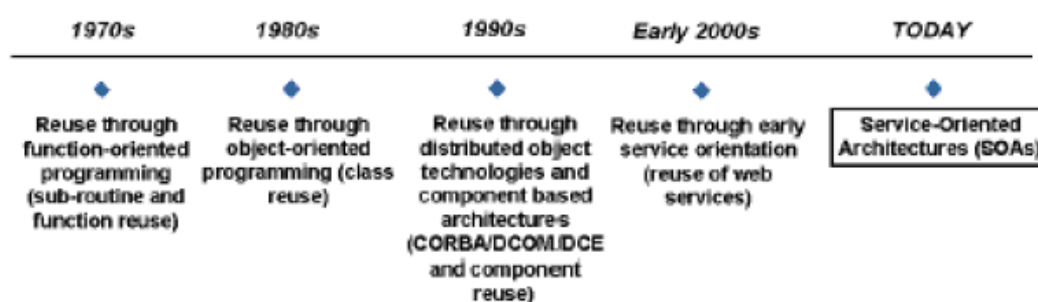
رویکرد SOA به توسعه دهندگان این امکان را می دهد تا بررسیاری از چالش های سازمانی توزیع شده مانند یکپارچه سازی برنامه کاربردی مدیریت تراکنش ها و سیاست های امنیتی غلبه کرده در عین حال امکان پشتیبانی از چندین پلتفرم و پروتکل و بهره برداری از قدرت چندین دستگاه دسترسی و سیستم های موروثی را نیز داشته باشند. هدف پیش برنده SOA حذف این موانع به صورتی است که برنامه های کاربردی یکپارچه شده و به صورت بدون درزی اجرا شوند. بعد از حذف این موانع SOA می تواند انعطاف و چابکی مورد نیاز کاربران کسب و کار را ارائه داده و سرویس های دانه درشت را که قابل ترکیب شدن و استفاده مجدد هستند به عنوان بلوک های ساخت معماری نرم افزاری سازمان تعریف کنند تا نیازهای مداوم و در حال تغییر کسب و کار را برآورده کند. با ظهور SOA فرآیندهای کسب و کار که خود حاصل ترکیب سرویس های کسب و کار هستند جایگزین برنامه های کاربردی می شوند. سرویس های وب نخستین فناوری هستند که به عنوان یک مکانیزم توسعه SOA به کار رفته اند اما باید توجه کرد که برای بهره بردن کامل از مزایایی که سرویس های وب و SOA ارائه می دهند علاوه بر دانستن فناوری ها و پروتکل های سرویس های وب فهم مبانی معماری سرویس گرایی نیز بسیار مهم و ضروری است و توصیف این مبانی کاری است که ما به دنبال آن هستیم.

این بخش در ابتدا مروری بر معماری های نرم افزاری متعارف داشته و آنها را با SOA مقایسه می کند سپس عناصر تشکیل دهنده SOA و توانمندی های آن را بررسی میکند. در ادامه به بررسی

رویکردهای یکپارچه‌سازی ارائه دهنده و مصرف کننده سرویس می‌پردازد. بعد از بررسی رویکردهای مختلف ESB را به عنوان بهترین مدل برای کاربردهای سازمانی و بزرگ معرفی و توصیف میکند. سپس سناریوهای مختلفی که سرویس‌گرایی در آنها کاربرد دارد و ارائه شده و در نهایت لایه‌بندی دو نمونه از معماری‌های سرویس‌گرا با جزئیات بیشتری مورد بررسی قرار می‌گیرند.

4-2- تاریخچه استفاده مجدد در تولید نرم افزار - افزایش انتزاع

شکل 40 روند تکامل استفاده مجدد در توسعه نرم‌افزار را نشان می‌دهد از دهه 1970 تا دهه 1980 توسعه نرم افزاری از زبان ماشین تا زبان اسمبلی تا سطح بالاتر زبان های کامپایل شده (موسوم به زبان های نسل سوم) تا زبان‌های نسل چهارم و ابزارهای CASE سطح انتزاع به طرز چشم‌گیری افزایش یافته است. در دهه 1980 برای استفاده مجدد از معماری های شی گرا استفاده شده سپس در دهه 1990 برای پیاده‌سازی سیستم های توزیع شده پیچیده‌تر و بزرگتر حرکت به سمت معماری های اشیاء توزیع شده و توسعه بر مولفه COM/DCCOM/CORBA آغاز شد و در اوایل دهه 2000 استفاده مجدد از نرم افزار در قالب سرویس‌های وب پدیدار گشت و امروزه معماری‌های سرویس گرا ظاهر شده‌اند و در ادامه به مقایسه این معماری‌های متعارف نرم‌افزاری می‌پردازیم.



شکل 41- تکامل استفاده مجدد در توسعه نرم‌افزار

4-3- SOA در مقابل معماری های نرم افزاری متعارف

یک قاعده پذیرفته شده در مهندسی نرم‌افزار، برای برنامه‌ها این است که هر برنامه شامل دو جزء ساختارهای داده‌ای و الگوریتم‌ها است. برنامه نویسی ساخت یافته (رویه‌ای) روی الگوریتم‌ها تمرکز

دارند. یعنی اینکه چه فرآیندی داریم (باید چه کاری انجام دهیم) و ساختارهای داده‌ای در مرحله دوم اهمیت قراردارند. برنامه نویسی شی‌گرا روی ساختارهای داده‌ای تمرکز دارد ابتدا اشیا در نظر گرفته می‌شوند و الگوریتم‌ها (نحوه تعامل بین اشیا) در رده بعدی اهمیت قرار می‌گیرند. برنامه نویسی سرویس‌گرا مشابه برنامه نویسی رویه‌ای دوباره روی الگوریتم‌ها (نحوه پردازش) تمرکز دارد؛ اما در یک سطح بالاتر ارتقا کلیدی سرویس‌ها نسبت به متدها این است که آنها از اشیا پیچیده‌تری برای انجام وظایف مورد نظر خود استفاده می‌کنند و به طور خلاصه می‌توان گفت: معماری‌های متعارف (مانند شی‌گرا با مبتنی بر مولفه) به دنبال این هستند که یک سیستم از چه اشیا تشکیل شده اما SOA به دنبال این است که چه نیازهایی وجود دارد که سیستم نرم افزاری نهایت باید آنها را برآورده کند. برخلاف معماری های نرم‌افزاری متعارف که در ابتدا ساختار یک سیستم را در قالب زیر سیستم‌ها و ارتباطات آنها مشخص می‌کنند SOA از منطقی برای طراحی سیستم نرم‌افزاری استفاده می‌کند که به کاربر نهایی با سایر سرویس‌ها که در یک شبکه توزیع شده اند از طریق رابط‌های منتشر شده و قابل کشف سرویس ارائه می‌کند.

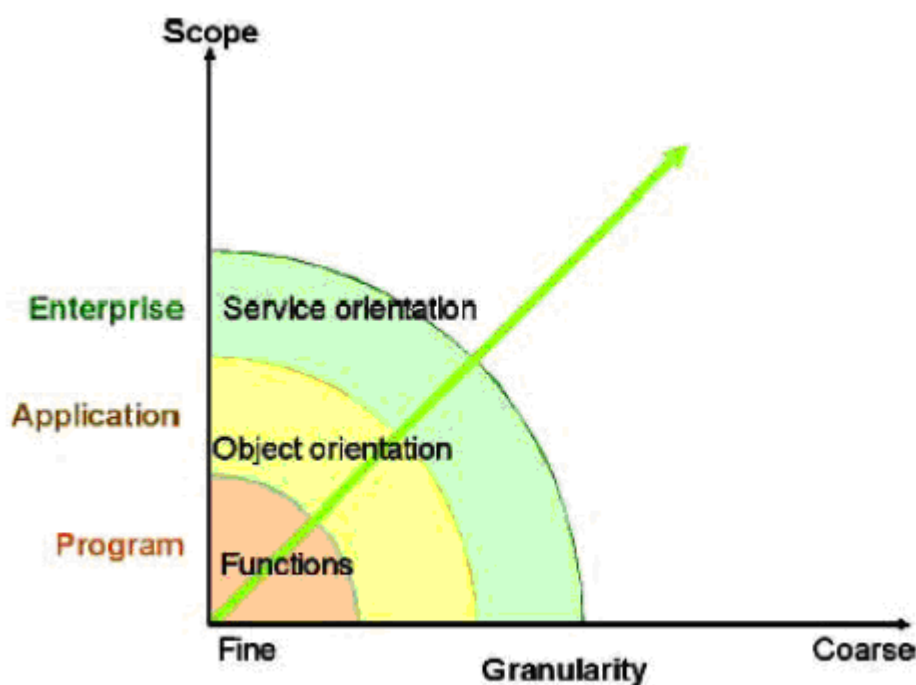
4-4- SOA تغییر جهت از برنامه های کاربردی به سمت فرآیندها و سرویس ها

معرفی SOA سبب تغییر جهت تدریجی از برنامه های کاربردی به سمت عناصری است که از لحاظ معماری به انتزاعات سطح کسب و کار نزدیکتر هستند و اصولاً با ظهور SOA برنامه های کاربردی حذف خواهند شد چون امروزه این برنامه های کاربردی بایستی با فرآیندهای کسب و کار انعطاف پذیر که در بالای سرویس های کسب و کار سازمانی قرار می گیرند جایگزین شوند. امتیاز SOA ایجاد سیستمهای نرم افزاری به صورت سرویسهای نرم افزاری به صورت سرویس هایی است که با همدیگر کار می کنند به شکلی که هر سرویس یک واقعیت کسب و کار یا وظیفه بخصوص را که در زمینه کل کسب و کار تعریف شده پیاده می کند.

4-5- از توابع دانه ریز تا سرویس های دانه درشت :

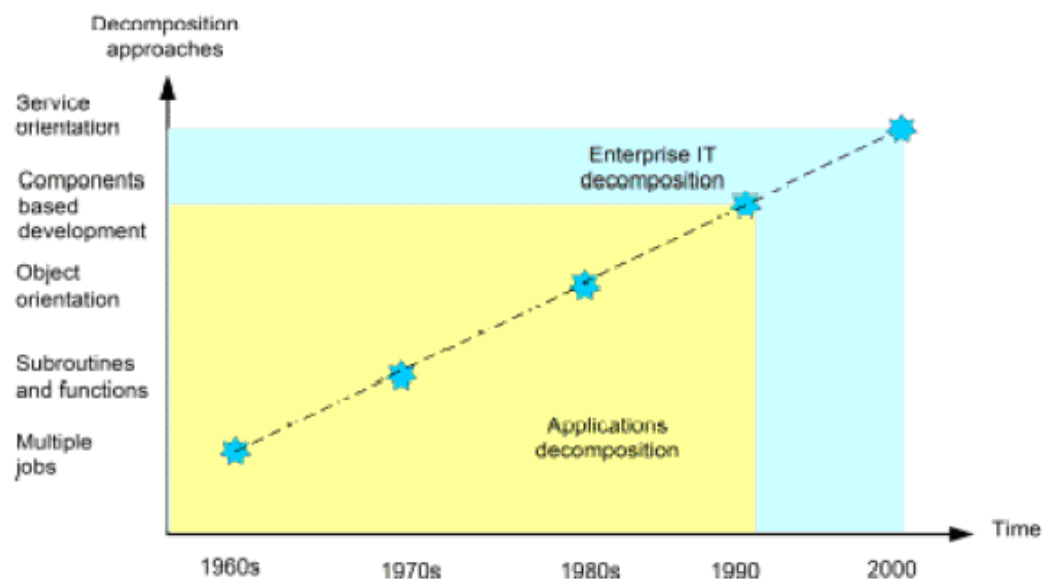
همانطور که در شکل 41 مشاهده می شود در رویکردهای برنامه نویسی ساخت یافته (تابعی) متمرکز صرفاً روی ساخت یک برنامه است و از توابع دانه ریز برای پیاده سازی توابع و زیر روال های این برنامه ها استفاده می شود. در روش شی گرا افق پیش رو گسترش یافته و روی برنامه های کاربردی متمرکز می شود. در نتیجه اشیایی با اندازه متوسط به وجود می آیند که در برنامه های کاربردی قابل استفاده مجدد هستند اما باز هم چون استفاده مجدد صرفاً در سطح برنامه کاربردی دیده شده اشتراک اشیا با سایر برنامه های کاربردی یا پلتفرم ها مشکل است.

در معماری سرویس گرایی متمرکز روی یک سازمان است و چندیتن شی در قالب عملکرد های (سرویس های) کسب و کار دانه درشت ترکیب می شوند و میتوانند مجدداً استفاده شوند و در هر جایی از این طرف تا آن طرف مرزهای سازمانی به اشتراک گذاشته شوند.



شکل 42- از توابع دانه ریز تا سرویس های دانه درشت

همانطور که در شکل 3 مشاهده می شود معماری های نرم افزاری متعارف (مانند ساخت یافته یا شی گرا) برنامه کاربردی محور هستند و بر توسعه یک برنامه کاربردی تمرکز دارند بنابراین به دنبال تجزیه یک برنامه کاربردی به توابع یا اشیاء کسب و کار هستند اما با ظهور معماری توسعه مبتنی بر مولفه حرکت از برنامه کاربردی محوری به سمت سازمان محوری آغاز شد و معماری سرویس گرا به آن تحقیق بیشتری بخشید. SOA به جای برنامه های کاربردی روی سازمان تمرکز دارد و در آن سرویس ها کل یک سیستم IT سازمانی را تجزیه می کنند (نقشی که قبلاً توسط برنامه های کاربردی ایفا می شد).



شکل 43- رویکرد تجزیه - از برنامه کاربردی محوری تا سازمان محوری

در شکل 44 تفاوت های معماری های برنامه کاربردی محور (مانند معماری ساخت یافته یا شی گرا) و معماری سرویس گرا به طور خلاصه ارائه شده است.

مشخصات	معماری برنامه کاربردی محور	معماری سرویس گرا
	تابع گرا	هماهنگی گرا
طراحی و پیاده سازی	ساخت برای نهایی شدن	ساخت برای تغییر
	چرخه توسعه طولانی	ساخت و بکارگیری به صورت تدریجی
سیستم نهایی	سیلوهای از برنامه کاربردی	راه حل های سازمانی
	پیوند قوی	پیوند ضعیف
	تعاملات شی گرا	تعاملات پیغام معنایی گرا

شکل 44- جدول تفاوت های بین معماری های برنامه کاربردی محور و معماری سرویس گرا

4-6- عناصر SOA سازمانی

رویکرد SOA سبکی از معماری است که تفکر فرایند های را ترویج می کند که این فرایند های کسب و کار سازمانی اجرای سرویس های کسب و کار سازمانی را هماهنگ و موزون می کنند. SOA سازمانی یک مجموعه از سرویس های IT منطبق با کسب و کار را که مورد نیاز افراد درون سازمان و سازمان های همکار است. تعریف می کند که این سرویس ها در کنار هم فرآیندها و اهداف سازمانی را برآورده می کنند. توالی و ترکیب این سرویس ها می تواند درون راه حل های کسب و کار سازمانی تعریف شود و این سرویس ها می توانند از طریق پروتکل های استاندارد فراخوانی شوند.

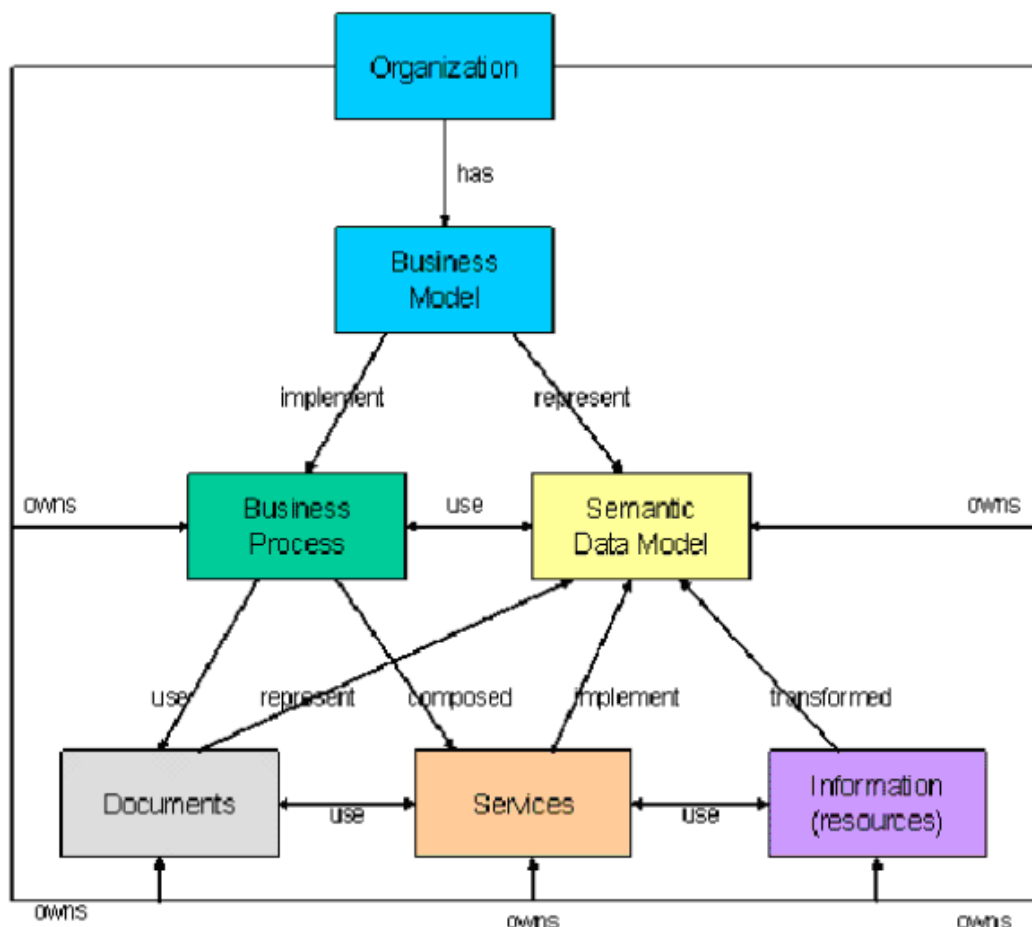
شکل 44 عناصر اصلی SOA سازمانی را نشان می دهد. این عناصر عبارتند از:

سازمان مالک تمام عناصر مورد استفاده در SOA (مدل ها سرویس ها فرایندها و منابع) است و بر ایجاد استفاده دسترسی و نگهداری از آنها نظارت می کند. سایر عناصر از سازمان و اهداف آن پشتیبانی می کنند.

مدل کسب و کار بازنمایی ابتدایی منابع و فرآیندهای کسب و کار است که برای رسیدن به اهداف عملیاتی تاکتیکی و راهبردی کسب و کار باید تکمیل انجام شوند. SOA چارچوبی برای ساختار بندی و انطباق IT برای رسیدن به این اهداف است. برای تطبیق موفقیت آمیز سرویس ها با اهداف و نیازمندی کسب و کار استفاده یک مدل کسب و کار الزامی است.

مدل داده ای معنایی اشیاء استاندارد کسب و کار (مانند مشتری و قرارداد) را برای یک سازمان معین تعریف می کند این اشیاء با تعریف موثر مفاهیم (و محتوای آنها) به داده های سازمانی موجودیت داده و وظایف سازمان را توصیف می کنند. استفاده از این مدل داده ای برای تعریف رابط های سرویس های کسب و کار منجر به ایجاد تعاریف رابط معنایی می شود که این اشیا می توانند با هم تعامل داشته باشند و نتیجه کار یک معماری سرویس گرای معنایی خواهد بود.

اسناد نمایند ه موجودیت های قانونی (مانند اسناد مالی قراردادهای و ادعاهای بیمه و قوانین دولتی) هستند که وظایف سازمان و همکاری آنها را تعریف می کنند. اسناد بخشی اساسی از سازمان های نوین هستند و باید در پیاده سازی معماری سرویس گرا به عنوان شهروند درجه یک در نظر گرفته شوند.



شکل 45- عناصر SOA سازمانی

سرویس ها و وظایف کسب و کار خصوصی را پیاده می کنند و به داده ها و منابع کسب و کار دسترسی دارند سرویس هایی که به خوبی تعریف شده اند و با کسب تطبیق یافته اند یک عنصر بحرانی و مهم از یک معماری انعطاف پذیر و قابل توسعه اند. ساختار سرویس ها اجازه می دهند تا آنها به صورت مستقل توسعه یافته و به کار گرفته شوند. تعریف و تطبیق درست سرویس ها با مدل کسب و کار و مدل داده ای معنایی منجر به استفاده PLUG AND PLAY از آنها میشود که اجازه می دهد به طور موثری از آنها در فرآیند های مختلف کسب و کار سازمان استفاده شود. اطلاعات نماینده منابع داده ای سازمان است که این داده ها در چندین مخزن برنامه کاربردی و همچنین قالب متفاوت قرار دارند. سطوح متفاوت از ساختار های SOA از سطوح متفاوت داده ای

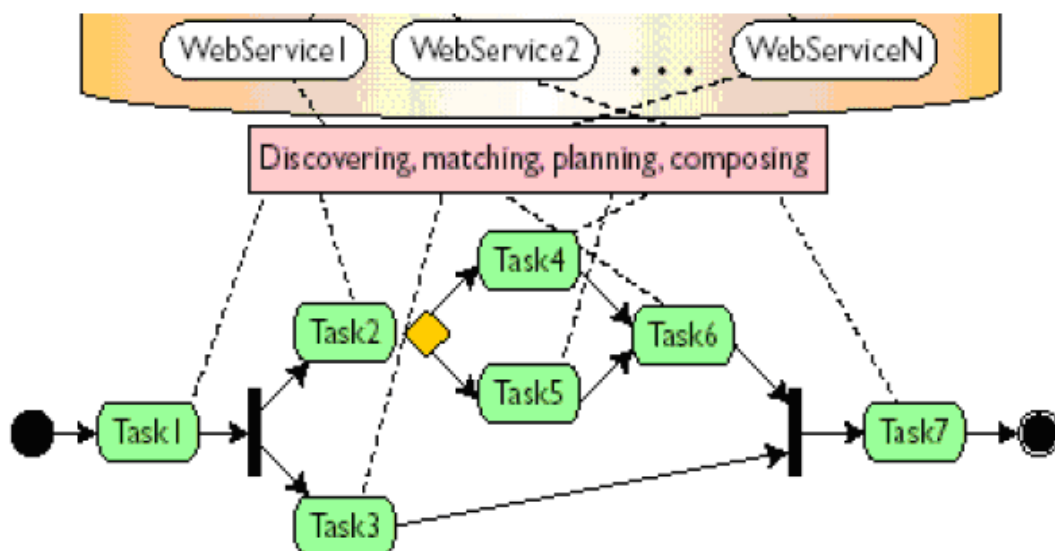
استفاده می کنند مدل داده ای معنایی داده های فرآیند ها و سرویس هاسی کسب و کار را تعریف می کند SOA مکانیزم تبدیل داده ها را از قالب عملیاتی به داده های معنایی که برای سرویس های کسب و کار ضروری است تعریف می کند.

فرایند های کسب و کاری چوب پردازش سفارش یا پردازش مطالبات اجرای سرویس های کسب و کار را هماهنگ و موزون می کنند تا وظایف کسب و کار همانطور که در مدل کسب و کار تعیین شده پیاده سازی شوند. فرآیند های کسب و کار معمولاً با اهداف عملیاتی و اهداف کسب و کار پیوند خورده اند (مانند پردازش مطالبات بیمه) هر فرآیند کسب و کار مشخص شرایط راه اندازی (شروع) تعریف شده ای دارد (مانند رسیدن یک مطالبه) و برای زمان اتمام آن نیز خروجی های تعریف شده ای وجود دارد.

4-7- ارتباط فرایند های کسب و کار با سرویس های کسب و کار

با توجه به اهمیت ارتباط سرویس و کار در معماری SOA در ادامه ارتباط این دو را با تفصیل بیشتری تشریح خواهیم کرد. شاید گفته شود که استفاده از فرآیند های کسب و کار به شکلی قابل انعطاف مبحث جدیدی نسبت ودرسیستم های مدیریت فرایندهای کسب و کار BPM نیز این مفاهیم مطرح بود. اما باید توجه شود که BPM تنها روی فرآیند های کسب و کار تمرکز دارد اما SOA روی دو گانگی کسب و کار فرایند کسب و کارسیر میکند (شکل 5) در واقع در SOA سه توانمندی زیر در ارتباط با سرویس ها و فرآیندها مطرح هستند .

- سرویس های کسب و کار می توانند با همدیگر ترکیب شوند تا توابع با فرایندهای کسب و کار ترکیبی تحویل دهند.
- چندین فرایند کسب و کار می توانند مشترکاً از یک سرویس کسب و کار منفرد استفاده کنند.
- یک سرویس کسب و کار خودش می تواند به صورت یک فرایند پیاده سازی شود.



شکل 46- هر فرآیند کسب و کار ترکیب چندین سرویس کسب و کار است

نقش ها و عناصر اصلی تشکیل دهنده معماری سرویس گرا

نقش های کلیدی SOA عبارتند از:

- درخواست کننده های سرویس (مشتریان یا مصرف کنندگان سرویس) که شامل انسان ها برنامه های کاربردی عامل ها /حساسه ها فرایندهای کسب و کار و سایر سرویس ها هستند.
- ارائه دهندگان سرویس را میتوان به دو دسته کلی ارائه دهندگان درون سازمانی و برون سازمانی تقسیم کرد. منظور از ارائه دهندگان سرویس درون سازمانی برنامه های کاربردی سازمان است که با در معرض نمایش گذاشتن سرویس های آنها میتوان به سایرین سرویس ارائه داد. ارائه دهندگان برون سازمانی درازای دریافت هزینه ارائه سرویس به شما سرویس ارائه می دهند.

- دلایل یا رجیستری سرویس که نقش میانجی را بین ارائه دهنده و مصرف کننده سرویس ایفا می کند البته این یک نقش اختیاری است و الزامی در استفاده از آن وجود ندارد. در ادامه در مبحث یکپارچه سازی در SOA این نقش با تفصیل بیشتری مورد بحث و بررسی قرار خواهد گرفت

عناصر تشکیل دهنده SOA عبارتند از:

- 1- سرویس: در SOA منابع نرم افزاری به صورت سرویس ها بسته بندی می شوند هدف اساس یک سرویس بازنمایی یک واحد کاری کامل کسب و کار است که بتوان از آن مجدداً استفاده کرد. از نظر منطقی هر سرویس شامل دو عنصر کنار هم است که عبارتند از:

- رابط (قرارداد) سرویس هویت سرویس و روشهای فراخوانی (درخواست) آن را تعریف میکند قرارداد مشخصات روش تعامل مصرف کننده سرویس با ارائه دهنده سرویس را تعیین می کند و حد و مرز گرامر و معنای سرویس را مشخص میکند به صورت کلی یک قرارداد به صورت یک توافق انتزاعی تعریف می شود که در هر دو زمان کامپایل و اجرا استفاده می شود. چون رابط های سرویس مستقل از پلتفرم اند هر کلاینت می تواند توسط هر وسیله ارتباطی به کمک هر پلتفرم می با هر سیستم عاملی و از هر زبان برنامه نویسی به سرویس دسترسی داشته باشد.

- پیاده سازی سرویس به عملیاتی که سرویس برای انجام آن طراحی شده عینیت بخشیده و انرا پیاده سازی کند پیاده سازی یک سرویس با هر معماری زبان برنامه نویسی پلتفرم و سیستم عاملی می تواند انجام شود.

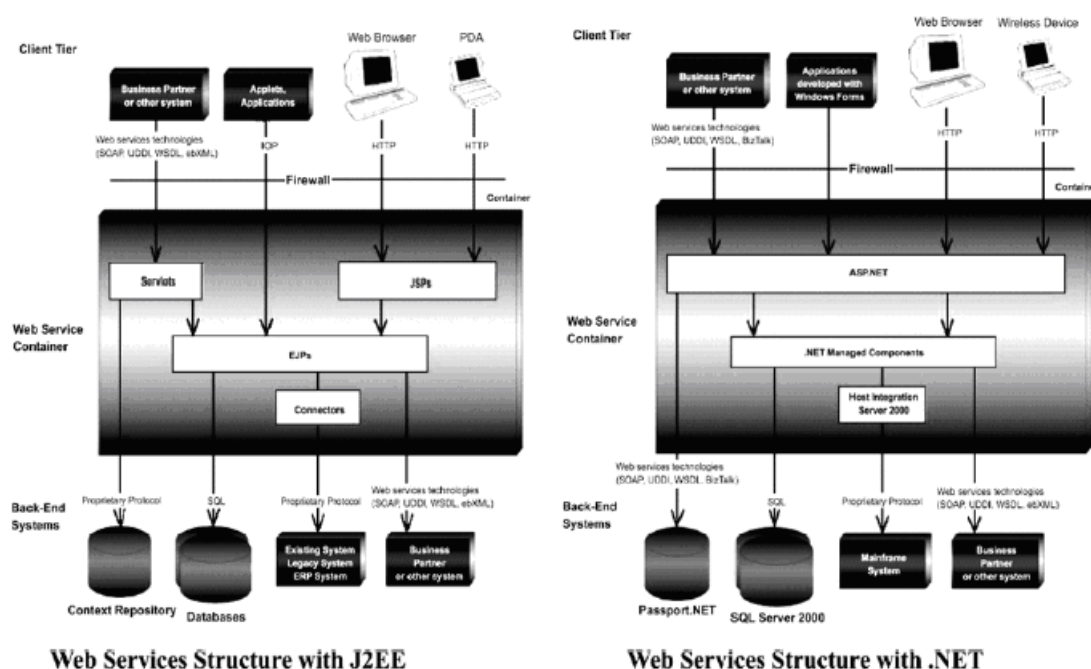
اگر چه هستی دو جنبه سرویس بسیاری به همدیگر وابسته است اما آنها متمایز هستند و به صورت اقلام متمایز طراحی و نگهداری می شوند .

یک سرویس در SOA باید مشخصه های اساسی زیر را داشته باشد:

- سرویس ها ماژول های خویش تعریفی هستند که عملکرد کسب و کار استانداری را ارائه می کنند.
 - یک سرویس کامل و تودار است یعنی تمام چیزهایی را که لازم دارد درون خود دارد و حالت خود را نگهداری می کند.
 - سرویس هامستقل از پلتفرم هستند منظور اینست که رابط قرارداد سرویس به اعلان های مستقل از پلتفرم محدود شده است. محلی یا راه دور بودند. طرح یا پروتکل اتصال روی فراخوانی اثر نمی گذارد در ضمن برای برقرای اتصال با آن از هر دستگاه ارتباطی به مولفه زیر ساخت خاصی نیاز نیست.
 - سرویس ها می توانند به صورت پویا مکان یابی فراخوانی و مجدداً ترکیب شوند.
 - تمام سرویس ها خود مختار بوده و مستقل از حالت یا زمینه سایر سرویس ها هستند. عملیات آنها برای مولفه های خارجی غیر شفاف است و فعال سازی و غیر فعال سازی سرویس ها مستقل از یکدیگر است.
- 2- پیغام های توصیفی برای ارتباط مستقل از پلتفرم بین ارائه دهنده و مصرف کننده سرویس به کار گرفته می شوند آنها رفتار سیستم را پیشنهاد میکنند و چرئیات مختص انتقال پیغام را براساس پروتکل استانداری خاصی کپسوله میکنند. ساختار پیغام ها براساس قراردادهای تعیین می شود این عنصر توسعه سیستمهای توزیع شده را بسیار آسانتر می کند. SOAP یکی از پروتکل های استاندارد سرویس وب است که پیغام ها را میتوان در آن قالب کپسوله کرد.
- 3- کانتینر سرویس: کانتینر محیطی برای اجرای سرویس ها است که دسترس پذیری و امنیت کد را مدیریت می کند. کانتینرهای سرویس مشابه کانتینرهای J2EE (مانند EJB و JSP) تسهیلاتی مانند مکان یابی، مسیر یابی فراخوانی و مدیریت سرویس را ارائه می کنند همچنین کانتینرها تسهیلات مدیریت دوره عمر را مانند راه اندازی پایین

آوردن و پاک کردن منابع ارائه می کنند. یک کانتینر می تواند میزان چندین سرویس باشد حتی اگر آنها بخشی از فرآیند توزیع شده یکسانی نباشند. سرویس دهنده های برنامه کاربردی ESB نمونه هایی از کانتینرهای سرویس هستند در شکل 6 کانتینرها

سرویس در J2EE و NET نمایش داده شده است



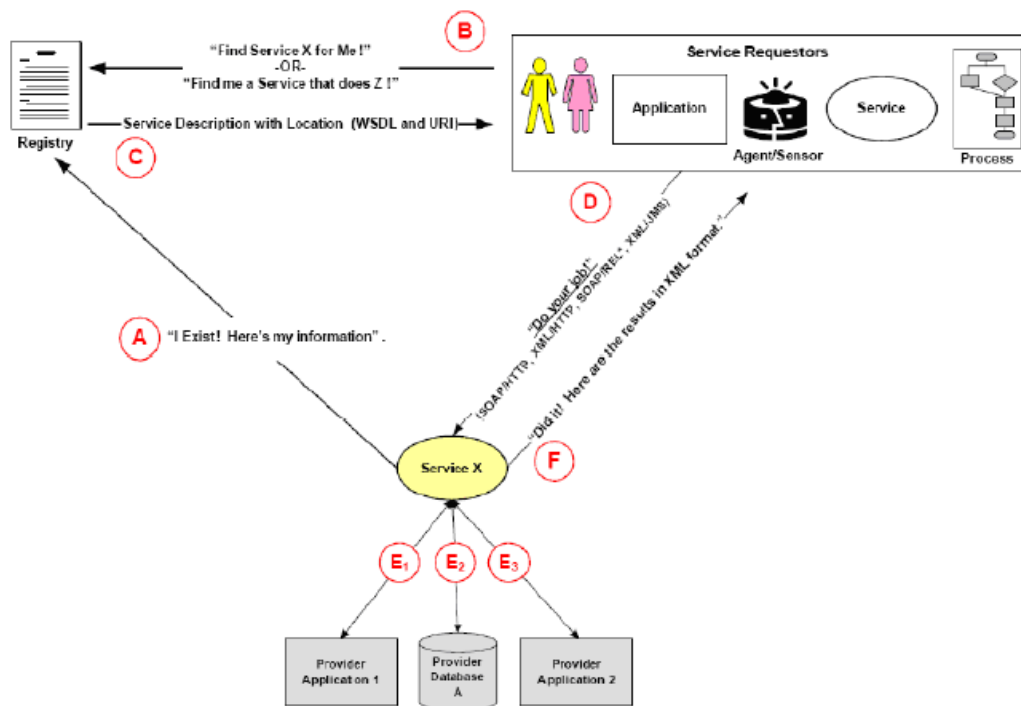
شکل 47- کانتینرهای سرویس در J2EE و .NET.

4-8- گام های اولیه برای مکان یابی و فراخوانی یک سرویس وب

در اینجا برای درک ساختار کلی یک سرویس وب نحوه اعلان جستجو و فراخوانی آن گام های اولیه فراخوانی یک سرویس وب ارائه شده است (شکل 7)

1. ارائه دهندگان سرویس، سرویسهای خود را درون یک رجیستری مشترک که بر مبنای استاندارد UDDI است منتشر می کنند. اطلاعات منتشر شده شامل مکان و توصیف جزئی سرویس در قالب یک سند XML با پروتکل WSDL است. فرض کنید سرویس منتشر شده سرویس X است.

2. درخواست کننده سرویس (مثلاً یک برنامه کاربردی) برای کشف سرویس مورد نظر خود یک درخواست به سمت رجیستری سرویس ارسال می کند یعنی از رجستری می خواهد سرویس X یا سرویسی که یک عمل خاص را انجام می دهد پیدا کند .
3. رجیستری توصیف سرویس (WSDL) و مکان سرویس (URL) را به درخواست دهنده سرویس می فرستد.
4. درخواست کننده سرویس خود را به آن سرویس خاص مقید می کند مثلاً اگر می خواهید از این سرویس برای انجام بخشی از فرآیند کسب و کار Y استفاده کند این سرویس را به عنوان بخشی از آن فرآیند در نظر می گیرد بعد از عمل انقیاد این سرویس مانند یک سرویس محلی برای درخواست کننده سرویس است با این تفاوت که پردازش در سرویس دهنده انجام خواهد شد. سپس درخواست دهنده سرویس برای دریافت سرویس مورد نظر با ارسال یک پیغام در قالب SOAP به URL ارائه دهنده سرویس آن را فراخوانی خواهد کرد.
5. ارائه دهنده سرویس درخواست رسیده را بر اساس وظایف تعریف شده اش که از دید کاربر خارجی پنهان است پردازش می کند. برای انجام عملیات مورد نظر ارائه دهنده سرویس ممکن است مجبور باشد با چند مولفه یا برنامه کاربردی درگیر شود یا حتی چند سرویس دیگر را از سایر ارائه دهندگان سرویس فراخوانی کند.
6. ارائه دهنده سرویس نتایج را در قالب یک سند xml به برنامه کاربردی درخواست دهنده برگشت می دهد.



شکل 48- گام‌های اولیه برای مکان‌یابی و فراخوانی یک سرویس وب

4-9- آیا SOA همان سرویس وب است ؟

به نظر می‌رسد که سرویس‌های وب فناوری پیاده‌سازی ارجح برای تحقق بخشیدن به SOA هستند اما باید گفت که فلسفه طراحی مبتنی بر SOA به هیچ فناوری بخصوص وابسته نیست. هر سرویس وب فقط رابط استاندارد است که مشخص می‌کند چگونه میتوان با هر سرویس ارتباط برقرار کرد. در حقیقت سرویس‌ها را به کمک فناوریهای متنوعی مانند سرویس‌های وب EJB/RMI، JMS و یا CORBA/COM/DCOM ساخت. برای این کار فقط باید چندین محدودیت روی پیاده‌سازی رابط سرویس اعمال کرد.

4-10- توانمندی های SOA

بهره برداری بالاتر: با توجه به دانه درشت بودن سرویس ها و همچنین امکان ترکیب آنها توسعه اجزای پردازشی بزرگتر امکان پذیر خواهد بود که این سبب بهره وری بالاتر در توسعه نرم افزاری می شود.

پیوند ضعیف: SOA با اعمال دو محدودیت زیر از پیوند ضعیف بین مولفه های نرم افزاری که با هم تعامل دارند پشتیبانی می کند .

- رابطها (قراردادهای) ساده استاندارد و همگانی برای تمام مولفه های نرم افزاری که قراراست به کارگرفته شوند. آنها فقط معانی همگامی را بیان می کنند. و بایستی برای تمام ارائه دهندگان و مصرف کنندگان سرویس به طور جامعی در دسترس باشند.
- پیغام های تشریحی که ساختار آنها براساس قراردادها تعیین می شود رفتار سیستم را پیشنهاد می کنند.

قابلیت توسعه و نگهداری: با توجه به پیوند ضعیف بین ارائه دهنده و مصرف کننده سرویس برنامه های کاربردی با معماری سرویس گرا امکان گسترش تغییر و نگهداری بسیار آسان تری نسبت به روشهای متعارف دارند.

پشتیبانی از پردازش توزیع شده: سرویس های SOA براساس کمیت داده ها و دستیابی پذیری داده ها با امکان گزینش مالکیت داده هایی که باید پردازش شوند یا مالکیت حقوق دسترسی به داده ها از پردازش توزیع شده پشتیبانی می کنند .

جدا سازی نگرانی ها: مصرف یک سرویس معمولاً ارزان تر و بهینه تر از درگیر شدن در طراحی آن است. این یک اصل خوب در مهندسی نرم افزاراست که جداسازی نگرانیها نامیده می شود. جدا سازی نگرانی ها منجر به خوانایی بیشتر میشود. چون مصرف کنندگان سرویس فقط به فهم قراردادهای ارائه دهندگان سرویس نیاز دارند نه پیاده سازی سرویس ها آن همچنین سبب می شود که هر برنامه کاربردی مسئول نگهداری منطق کسب و کار خودش باشد. با وجود اینکه سرویس ها

برای درخواست دهنده سرویس قابل رویت هستند اما مولفه های زیرین آنها ناپیدا هستند. مادامی که ارائه دهنده سرویس عملکرد مورد نیاز را انجام داده و کیفیت سرویس مطلوب را ارائه کند مصرف کننده سرویس نگران نحوه پیاده سازی آن نیست. تصمیمات طراحی و معماری اصلی که ارائه دهنده سرویس باید در نظر بگیرد طراحی و واقعیت بخشیده به مولفه های ارائه دهنده سرویس (پیاده سازی سرویس) در معرض نمایش گذاشتن آنها و نحوه مدیریت آنها است. انتخاب بین کارایی یا انعطاف: برخلاف معماری های متعارف (مانند شی گرای) که انعطاف کار را افزایش داده اما سبب کاهش کارایی شده اند در SOA مصرف کنندگان سرویس می توانند براساس نیازشان به کارایی یا انعطاف سرویس های متفاوتی را انتخاب کنند.

4-11- یکپارچه سازی در SOA

همانطور که در ابتدا بیان شد یک از اهداف اصلی SOA یکپارچه سازی برنامه های کاربردی است. در اینجا به بررسی جزئی تر یکپارچه سازی برنامه های کاربردی که جزء قابلیت هاس SOA است اصولاً در SOA ارائه دهندگان و مصرف کنندگان سرویس خود مولفه های نرم افزاری هستند که قرار است باهم تعامل داشته و به یکی از دو شکل زیر یکپارچه شوند (شکل 48)

1. روش نقطه به نقطه: آماده سازی مازول مصرف کننده برای انطباق دقیق با مشخصات

ماژول سرویس دهنده ای که فراخوانی خواهند شد. منثور اینست که دوه دو بین هر دو نقطه ارتباط مستقیم برقرارشود. این روش به توسعه یک رابط برای هراتصال نیاز دارد. مدیریت و نگهداری این توپولوژی سخت است چون به یک پیوند قوی برای یکنواخت کردن پروتکل های انتقال قالب های سند سبک های تعاملی و غیره نیاز دارد. علاوه براین روش نقطه به

نقطه مقیاس پذیر نیست و با بالا رفتن تعداد سیستم های درگیر در یکپارچه سازی، پیچیدگی بسیار بالا می رود و سیستم نهایی به سرعت غیر قابل مدیریت شود.

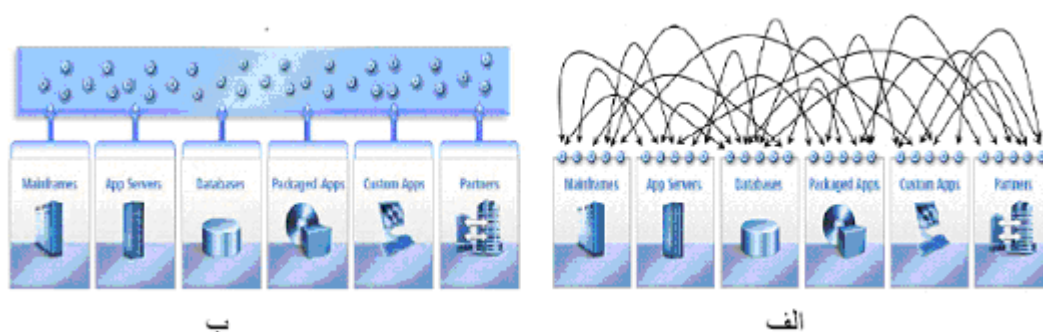
2. استفاده از میان افزار یکپارچه سازی: درج یک لایه منطق یکپارچه سازی بین

ماژول های مصرف کننده و ارائه دهنده سرویس. زمانی که تعداد سیستم های درگیر

در یکپارچه سازی بالا می رود، استفاده از یک زیر ساخت SOA ضرورت پیدا می

کند. رویکرد دوم، یک لایه یکپارچه سازی معرفی می کند که از تعامل بین زیر ساخت

بکار گرفته شده و برنامه های کاربردی پشتیبانی می کند.



شکل 49- استفاده از روش نقطه به نقطه – شکل الف و روش میان افزار یکپارچه سازی -

شکل ب

4-12- یکپارچه سازی نقطه به نقطه در SOA

در SOA برای پیاده سازی روش نقطه به نقطه، از دو الگو می توان استفاده کرد:

رجیستری سرویس ایستا: این الگو، مدل سنتی برای مکان یابی و انقیاد سرویس ها در SOA است.

این مدل از رجیستری برای انتشار قراردادهای سرویس توسط ارائه دهندگان سرویس و همچنین

جستجوی این سرویس ها توسط مصرف کنندگان سرویس استفاده می کند. این مدل، تنها به

استانداردهای هسته ای سرویس های وب نیاز دارد که شامل SOAP، WSDL و UDDI است.

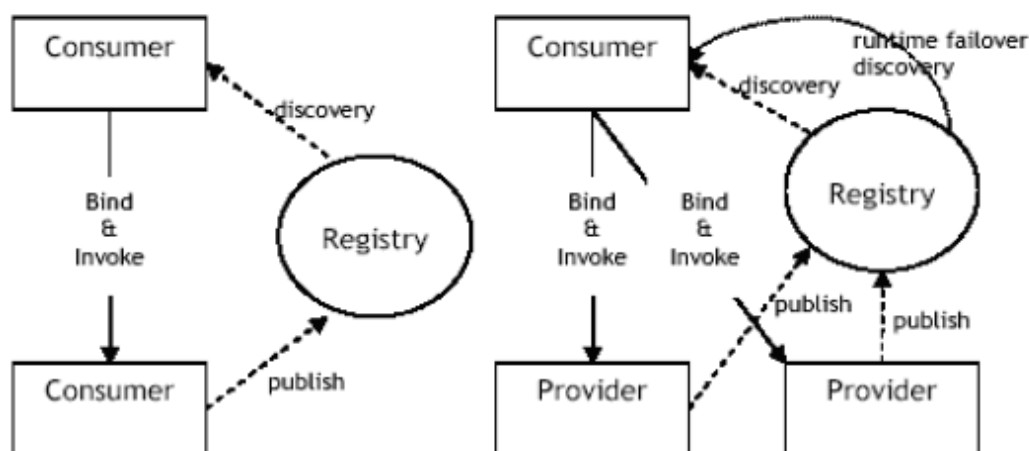
بزرگترین مشکل این الگو آنست که انقیاد سرویس، ایستا است و در زمان طراحی تعریف می شود

که منجر به پیوند نسبتاً قوی بین سیستم های به هم پیوسته شده می شود.

رجیستری سرویس پویا: این الگو، مشابه مدل قبلی است، با این تفاوت که پیوند آن ضعیف تر است

و جستجوی سرویس ها و انقیاد به آنها توسط مصرف کننده سرویس، در زمان اجرا انجام می شود.

با توجه به اینکه جستجو و انقیاد در زمان اجرا انجام می شود مصرف کننده سرویس بایستی بر اساس رابطه سرویس دریافتی (مثلاً WSDL)، پیغام های توصیفی خود را پیکربندی و ارسال کند. در این مدل، UDDI این امکان را در اختیار می گذارد که هنگامی که قرارداد سرویس یکسانی، توسط چندین ارائه دهنده سرویس پیشنهاد می شود، بتوان سرویس مورد نظر خود را انتخاب کرد. این قابلیت، امکان موازنه بار و افزایش قابلیت اطمینان را فراهم می کند، چون فهرست سرویس، قادر است یک سرویس خاص را از بین تمام ارائه دهندگان سرویس که در حال حاضر موجودند، یافته و ارائه دهد. اگر در حین اجرا خطایی در دریافت سرویس رخ دهد، رجیستری می تواند سرویس مشابه دیگری را در اختیار مصرف کننده سرویس قرار دهد تا وی خود را به سرویس جدید مقید کرده و آن را فراخوانی کند.



شکل 50- مدل رجیستری سرویس ایستا (سمت چپ) و رجیستری سرویس پویا (سمت راست)

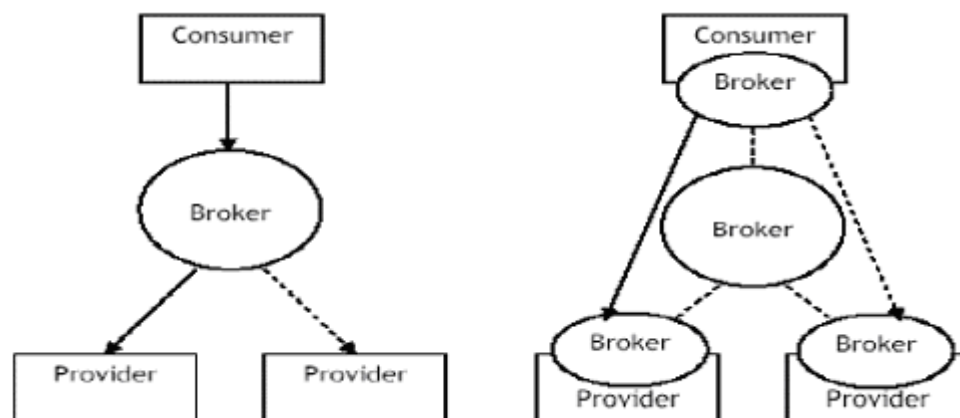
4-13- استفاده از میان افزار یکپارچه سازی در SOA

همانطور که مشاهده کردید، در دو مدل ذکر شده، ارائه دهنده و درخواست دهنده سرویس، عمل انقیاد و فراخوانی سرویس را به صورت مستقیم انجام می دهند و تنها برای جستجو یا انتشار قراردادها از یک واسطه استفاده می کنند، اما در صورت استفاده از یک لایه یکپارچه سازی، هیچ

ارتباط مستقیمی بین ارائه دهنده و مصرف کننده سرویس وجود ندارد و تمام تعاملات باید از درون این لایه عبور کند. برای پیاده سازی میان افزار یکپارچه سازی در SOA می توان یکی از الگوهای دلال سرویس، دلال سرویس توزیع شده، باس سرویس یا باس سرویس توزیع شده را به کار گرفت.

دلال سرویس: در این مدل، تمام پیغام ها از میان لایه دلال پیغام منتقل می شوند و این لایه قادر به انجام عملیاتی مانند مسیریابی براساس محتوا، بازیافت خرابی، ترجمه پیغام، موازنه بار و فیلترینگ است. دلال پیغام، قادر است با ارائه اطلاعات اضافی درباره سرویس ها به رجیستری ارائه دهندگان سرویس ارزش اضافه کند که ممکن است شامل تفاوت های قابلیت اطمینان، قابلیت اعتماد، کیفیت سرویس (QOS)، موافقتنامه های سطح سرویس (SLA) و مسیریابی براساس دریافت پاداش از ارائه دهنده سرویس باشد. البته باید توجه کرد که دلال سرویس، ذاتاً، یک تنگنای بالقوه سیستم و یک نقطه منفرد خرابی است. با توجه به اینکه سبک ارتباطات در الگوی دلال سرویس، همزمان است و از صف بندی پیغام پشتیبانی نمی شود، هم ارائه دهنده سرویس و هم مصرف کننده سرویس بایستی در زمان درخواست موجود باشند. برای موازنه بهتر بار، می توان از یک پیکربندی خوشه ای استفاده کرد، اما هنوز هم تمام پیغام ها از میان دلال سرویس عبور می کند که کارایی کلی را کاهش می دهد.

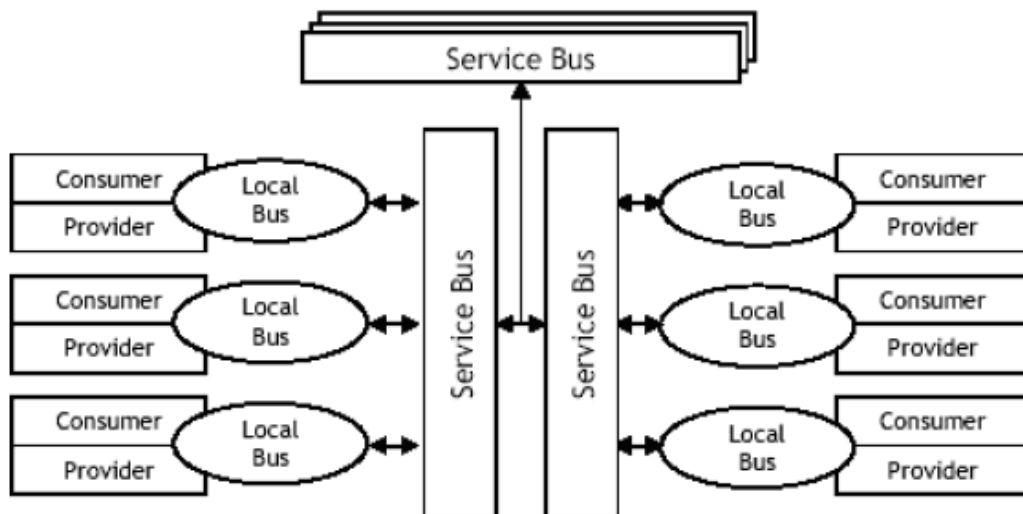
دلال سرویس توزیع شده: یک دلال سرویس توزیع شده، یک مدل توسعه یافته تر است که هر پلتفرم سرویس، یک دلال محلی دارد که قادر است با دلال سرویس مرکزی و همچنین با دلال های سرویسی که روی سایر پلتفرم های سرویس موجودند، ارتباط داشته باشد.



شکل 51- دلال سرویس (شکل سمت چپ) و مدل توزیع شده (شکل سمت راست)

باس سرویس: در حال حاضر، این مدل توسط محصولات SOA با اندازه بزرگ (مانند BEA و IBM) با نام ESB به کار گرفته شده است. باس سرویس در بین تمام این الگوها، ضعیف ترین پیوند را دارد، چون سرویس ها نه اتصال مستقیم دارند و نه از لحاظ زمانی به هم وابسته اند، چون زیر ساخت آن به طور ذاتی از پیغام رسانی غیر همزمان پشتیبانی می کند، اصول پایه ای این مدل، براساس الگوی انتشار / اشتراک است که پیغام رسانی غیر همزمان را امکان پذیر می سازد. در این الگو، مصرف کنندگان درخواستهای خود را روی باس منتشر می کنند، در حالی که ارائه دهندگان سرویس، به اشتراک درخواست های معنی در می آیند. نوآوری این مدل این است که مصرف کننده و ارائه دهنده سرویس، لازم نیست همزمان در حال اجرا باشند.

باس توزیع شده: گاهی اوقات، به یک باس سرویس توزیع شده نیاز است که در آن چندین لایه باس سرویس وجود دارد. طبیعت تعامل SOA سبب شده، پیوند باس های سرویس که توسط سازمان های متفاوت نگهداری می شوند، درون یک شبکه از باس های سرویس نسبتاً آسان باشد. یک باس سرویس محلی، ارتباطات داخلی یک نود منفرد را مدیریت می کند و در صورت نیاز، مسیریابی پیغام ها را به سمت باس سرویس مرکزی انجام می دهد.



شکل 52- باس سرویس توزیع شده

ESB-14-4

اگر قرار باشد یکی از رویکردهای یکپارچه سازی فوق الذکر، برای پیاده سازی معماری های سرویس گرای سازمانی پیچیده و بزرگ، انتخاب شود، با توجه به معماری توزیع شده، قابلیت انتشار / اشتراک، توسعه پذیری، قابلیت نگهداری و غیره، ESB بهترین انتخاب خواهد بود. در ادامه ESB با تفصیل بیشتری مورد بررسی قرار می گیرد.

ESB از نظر مفهومی، تکامل یافته مکانیزم ذخیره و ارسال محصولات میان افزار مانند افزار پیغام گرا (MOM) است و اکنون فناوری های متعارف EAI را با فناوری های هماهنگ و موزون سازی، سرویس های وب، XML و XSLT ترکیب کرده است. ESB امکان طراحی، توسعه، استقرار و مدیریت SOA را فراهم کرده و چالش های هماهنگ و موزون سازی سرویس ها، همگام سازی داده ای و مانیتورینگ فعالیت های کسب و کار (BAM) را برطرف می کند. از نظر فیزیکی، ESB ستوان فقرات مورد نیاز برای پیاده سازی SOA را ارائه می کند. ESB به برنامه های کاربردی و مولفه های یکپارچه سازی گسسته، این امکان را می دهد تا با همدیگر تعامل کرده و مجموعه ای از سرویس ها را برای تشکیل فرآیندهای کسب و کار ترکیبی ایجاد کنند تا در نهایت

فرآیندهای کسب و کار سازمان را خودداری کنند. در ادامه، فهرستی از قابلیت ها و وظایف پایه ای ESB به طور خلاصه ارائه شده است:

- پشتیبانی از سرویس های وب: UDDL،Ws-Reliable messeging،Wssecurity ، SOAP،WSDL و غیره
- یکپارچه سازی با برنامه های کاربردی موروثی
- پیغام رسانی، تحویل غیر همزمان پیغام با روش ذخیره و ارسال با چندین سطح کیفیت سرویس (Qos)
- تبدیل داده ای و تبدیل پروتکل ها
- استقلال از پلتفرم
- مسیریابی پیغام ها و تحویل درخواست ها به ارائه دهنده سرویس مناسب
- جایگزینی پیاده سازی یک سرویس با دیگری بدون تاثیر روی مصرف کنندگان سرویس
- پشتیبانی از، نیازهای امنیتی، سیاسی، قابلیت اطمینان و حسابرسی
- توانایی اجرای فرآیندهای بلند مدت کسب و کار و پشتیبانی از سازگاری تراکنشی و امکانات بازیابی خرابی
- توانمندی های مدیریت و مانیتورینگ
- مقیاس پذیری

4-15- پیاده سازی SOA

برای پیاده سازی SOA، هم برنامه های کاربردی و هم زیرساخت SOA باید از اصول سرویس گرایی پشتیبانی کنند. زیر ساخت SOA (ESB)، باید دو توانمندی اصلی داشته باشد. نخست آنکه بتواند درخواست های مصرف کنندگان سرویس را به سمت ارائه دهنده سرویس مناسب هدایت کند. توانمندی کلیدی دیگر، پشتیبانی از جانشینی بی خطر پیاده سازی یک سرویس با دیگری بدون تحت تاثیر قرار گرفتن مشتریان سرویس است. لازمه این امر آنست که علاوه بر اینکه

رابط های سرویس براساس اصول SOA توصیف شده باشند، زیر ساخت هم اجازه دهد، برنامه کاربردی مصرف کننده، سرویس ها را بدون مراعات مکان سرویس و پروتکل ارتباطی بکار رفته فراخوانی کند.

توانمند سازی یک برنامه کاربردی برای SOA، درگیر ایجاد رابط های سرویس روی توابع موجود یا جدید، به صورت مستقیم یا از طریق استفاده از آداپترها یا لفافه است. در واقع می توان برنامه های کاربردی ارائه دهنده سرویس را که قرار است رابط سرویس آنها در معرض نمایش گذاشته شود، به سه دسته کلی زیر تقسیم کرد:

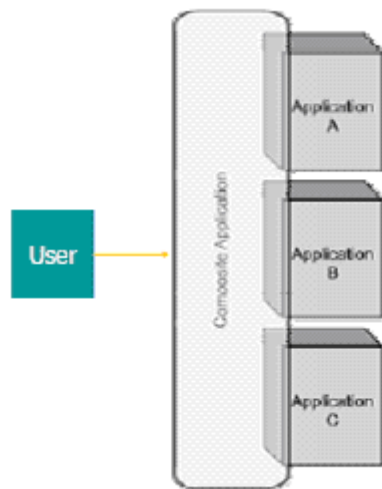
- پیاده سازی های سرویس کاملاً نو که مستقیماً رابط خود را در قالب یک قرارداد استاندارد (مانند WSDL) در معرض نمایش گذاشته اند.
- قطعاتی از برنامه های کاربردی قدیمی که چون قبلاً توسعه داده شده اند، از سرویس گرایی پشتیبانی نمی کنند. برای در معرض نمایش گذاشتن سرویس های آنها، باید از یک آداپتر مختص برنامه کاربردی استفاده کرد تا به عنوان یک واسط بین API برنامه کاربردی و مصرف کننده سرویس قرار گرفته و درخواست های سرویس دهند را به یک قالب قابل فهم توسط ارائه دهند سرویس و بالعکس تبدیل کند.
- قطعاتی از برنامه های کاربردی قدیمی یا آداپتر ها که سرویس ارائه می دهند، اما سرویس آنها از پروتکل مورد نظر سایر برنامه های کاربردی تبعیت نمی کند. به طور مثال اگر قرار باشد از سرویس های وب که قالب پیغام هایشان SOAP است، استفاده شود، باید از لفافه ها برای تبدیل قراردادها و پیغام های سرویس به قالب مناسب استفاده کرد.

سه نوع سناریوی سرویس گرایی

هنگامی بکارگیری فناوری وب سرویس برای حل مسائل یکپارچه سازی، سه الگو یا معماری وجود دارد که عبارتند از:

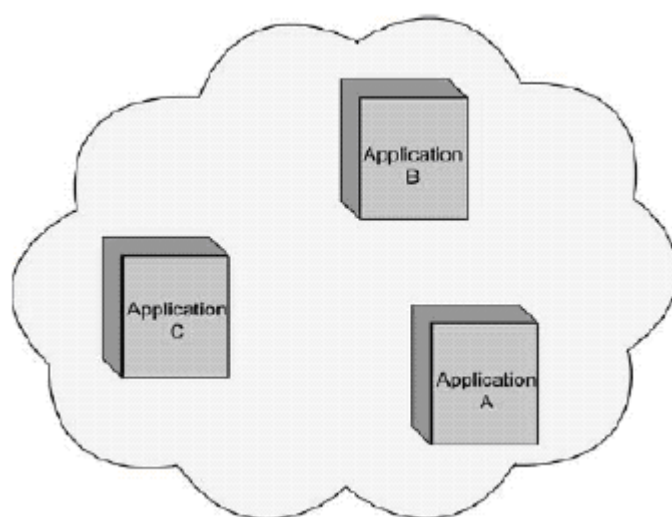
معماری رویداد گرا: در این معماری، برای پشتیبانی از یک تراکنش کسب و کار بخصوص، داده ها از یک سیستم به دیگری منتقل می شود. این معماری، به جای تجميع سرویس های برنامه های کاربردی، بیشتر با انتقال اطلاعات سر و کار دارد، ما در اینجا، همچنان نیاز به دسترسی به سرویس های برنامه های کاربردی وجود دارد. در این معماری، پیوند ضعیفی بین برنامه های کاربردی وجود دارد و خرابی یک برنامه های کاربردی، روی سایر برنامه ها تاثیر جدی نمی گذارد. این معماری، یک گزینه خوب برای همگام سازی و فدراسیون داده ای است.

شکل 53- معماری رویداد گرا



شکل 54- معماری برنامه کاربردی ترکیبی

معماری توزیع شده خود مختار: به معماری هایی اشاره دارد که سرویس های برنامه های کاربردی، طوری به صورت قوی با هم پیوند خورده اند که برنامه های کاربردی، به صورت یک برنامه کاربردی منفرد به نظر می آیند. این معماری، هدف نهایی سرویس های وب است، یعنی پیوند چندین برنامه کاربردی درون سازمانی و بین سازمان با همدیگر به شکلی که در نهایت به صورت یک کل منفرد و متحد الشكل دیده شوند. البته تا گسترش این معماری سال ها راه است.



شکل 55- معماری توزیع شده خود مختار

در ادامه نمونه هایی از معماری های مبتنی بر سرویس با تفصیل بیشتری آورده شده است.

معماری اول، ترکیبی بهینه از معماری سرویس گرا و مبتنی بر مولفه ارائه می دهد که برای کاربردهای سازمانی و همچنین برای ارائه دهندگان سرویس های اینترنتی قابل استفاده است.

توانمندی های سرویس ها نسبت به مولفه ها: سرویس ها عملکرد بیشتری را در بر می گیرند و نسبت به مولفه ها، دانه درست تر بوده و پیوند بین آنها ضعیف تر است. باتوجه به پیوند ضعیف سرویس ها و این حقیقت که مستقل از زبان و پلتفرم هستند، آنها برای به هم پیوستن سیستم های متمایز که به صورت مستقل از هم، پیاده سازی و راه اندازی شده اند، مناسب تر هستند.

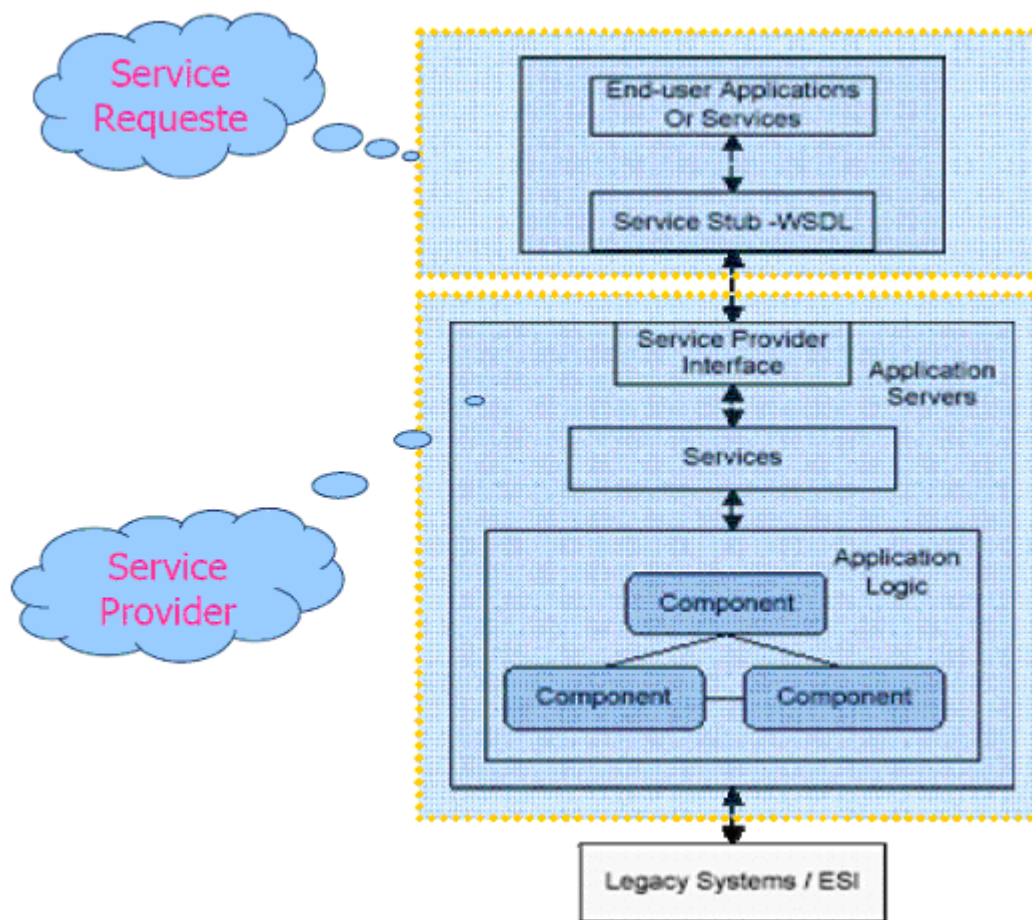
توانمندی های مولفه ها نسبت به سرویس ها: در موقعیت هایی که کارایی بیشتر مورد نیاز باشد تعاملات مبتنی بر سرویس، زیاد مناسب نیست، چون این تعاملات، منجر به ارتباطات شبکه ای و درگیر شدن با پروتکل ها می شود و به ناچار روی کارایی کلی سیستم تاثیر منفی می گذارد. مولفه ها باتوجه به پیوند قوی تر و مستقیم تری که دارند، کارایی بهتری نسبت به سرویس ها که پیوند ضعیف تری داشته و درگیر ارتباطات شبکه ای می شوند، ارائه می دهند. برای نمونه، تعاملات اشیای COM درون یک ماشین منفرد، کاراتر از فراخوانی سرویس خواهد بود، چون مدل پردازشی COM، می تواند از سربار مورد نیاز برای تعاملات سرویس های وب که شامل پردازش WSDL، درخواست سرویس و پاسخ به سرویس است، جلوگیری کند.

البته هیچ کدام از این فناوری ها به تنهایی تمام مسائل مقیاس پذیری، انعطاف پذیری، تعامل با همدیگر و کارایی برنامه های کاربردی مبتنی بر شبکه را که نباید محدودیتی روی زبان ها، پلتفرم ها و مرزهای سازمانی اعمال کنند، حل نمی کند. به همین دلیل در این مورد، برای معماری سیستم از ترکیب این دو فناوری استفاده شده است.

4-16- معماری ترکیبی پیشنهادی

معماری پیشنهادی، یک معماری چند لایه است که هم از قدرت طراحی مبتنی بر مولفه و هم سرویس گرایی بهره برداری می کند، به این شکل که از لایه سرویس به عنوان مکانیزم اتصال با

یکدیگر و از لایه مولفه برای پیاده سازی به هم پیوسته و مستحکم سرویس هایی که قرار است، توسط لایه سرویس ارائه شوند، استفاده می کند. در این معماری، سیستم های درگیر در تعامل، عملکرد خود را از طریق لایه سرویس در معرض نمایش می گذارند. این معماری چندلایه، بدون قربانی کردن کارایی سیستم، با بهره بردن از فناوری های مبتنی بر مولفه و SOA، به انعطاف پذیری و مقیاس پذیری نائل می شود. مولفه ها با پیوند قوی تر و مستقیم تر نسبت به سرویس ها که پیوند ضعیف تری داشته و اغلب درگیر ارتباطات شبکه ای می شوند، سبب کارایی بهتر می شوند. در ادامه، لایه های این معماری با تفصیل بیشتری مورد بررسی قرار می گیرند. (شکل 15)



شکل 56- معماری سیستم با استفاده از سرویس ها و مولفه ها

لایه اول، شامل برنامه های کاربردی کاربران نهایی، مرورگرهای وب و یا سایر سرویس هایی است که به برنامه های کاربردی و سرویس هاس مستقر دسترسی دارند، برنامه های کاربردی کاربردی

کاربران نهایی و سایر سرویس‌ها، از استاب‌های سرویس برای تعامل با لایه سرویس استفاده می‌کنند.

لایه دوم، لایه سرویس است که رابط‌های سرویس آنها به خوبی با استفاده از WSDL تعریف شده است. این رابط‌های انتشار یافته، تنها نقاط دسترسی هستند که برنامه‌های کاربردی کاربران نهایی یا سایر سرویس‌های می‌توانند از طریق آنها به سرویس‌ها دسترسی پیدا کنند.

لایه سوم، شامل منطق کسب و کار است که با استفاده از مولفه‌ها پیاده‌سازی می‌شود.

در این معماری، سرویس‌های وب، نقش لفافه برنامه‌های کاربردی موروثی و سایر سیستم‌های اطلاعاتی سازمانی (ELS) را ایفا می‌کنند تا برای یکپارچه‌سازی آنها با سایر برنامه‌های کاربردی، عملکرد آنها را در معرض نمایش بگذارند.

در مجموع می‌توان گفت معماری پیشنهادی، یک راه حل انعطاف پذیر و مقیاس پذیر برای طراحی و یکپارچه‌سازی سیستم‌های بزرگ و توزیع شده است.

گام‌های فراخوانی یک سرویس وب در این معماری به صورت زیر است. (شکل 16).

1. وقتی که برنامه کاربردی کلاینت، نیاز به فراخوانی سرویس وب دارد، استاب کلاینت را فرا

می‌خواند. استاب کلاینت، این فراخوانی محلی را به یک درخواست مناسب در قالب

SOAP تبدیل می‌کند که به این عمل marshaling یا serializing می‌گویند.

2. درخواست SOAP با استفاده از پروتکل HTTP روی شبکه ارسال می‌شود.

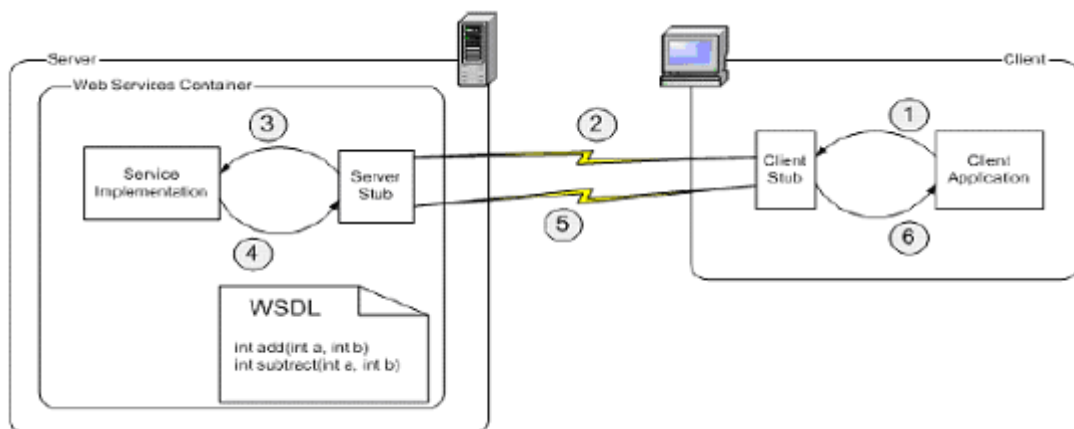
کانتینر سرویس، درخواست SOAP را دریافت کرده و آن را در اختیار استاب سرویس

دهنده، قرار می‌دهد. استاب سرویس دهنده، درخواست SOAP را به قالبی تبدیل می‌کند

که بخش پیاده‌سازی سرویس توان درک آن را داشته باشد (به این عمل

unmarshaling یا deserializing می‌گویند).

3. پیاده سازی سرویس، درخواست را از استاب سرویس دهنده دریافت می کند و کاری را که از وی خواسته اند، انجام می دهد. برای مثال، اگر متد add(inta, intb) فراخوانی شده باشد، پیاده سازی سرویس، عملیات جمع را انجام می دهد.
4. نتیجه عملیات درخواستی در اختیار استاب سرویس دهنده قرار می گیرد، در اینجا نتیجه عملیات تبدیل به یک پاسخ در قالب SOAP می شود.
5. پاسخ در قالب SOAP با استفاده از پروتکل HTTP روی شبکه ارسال می شود. استاب کلاینت، پاسخ SOAP را دریافت کرده و آن را به چیزی تبدیل می کند که برای برنامه کاربردی کلاینت قابل فهم باشد.
6. در نهایت، برنامه کاربردی، نتایج فراخوانی سرویس را دریافت کرده و از آن استفاده می کند.



شکل 57- گام های فراخوانی یک سرویس وب در معماری پیشنهادی

4-17- یک الگوی معماری برای یک برنامه کاربردی سازمانی

الگوی معماری دوم که مورد بررسی قرار می گیرد، توسط IBM ارائه شده است. این الگو، یک معماری نسبتاً جامع چند لایه برای پیاده سازی SOA سازمانی ارائه داده است که سیستم های موروثی و موجود، مولفه ها، سرویس های مورد نیاز، گردش فرایند کسب و کار و لایه ارائه را در بر می گیرد (شکل 17). این معماری با بهره گیری از ESB، مسائل یکپارچه سازی، مدیریت،

مانیتورینگ و کیفیت سرویس (Qos) را نیز در برمی گیرد. توصیف بیشتر لایه های این معماری در ادامه آمده است.

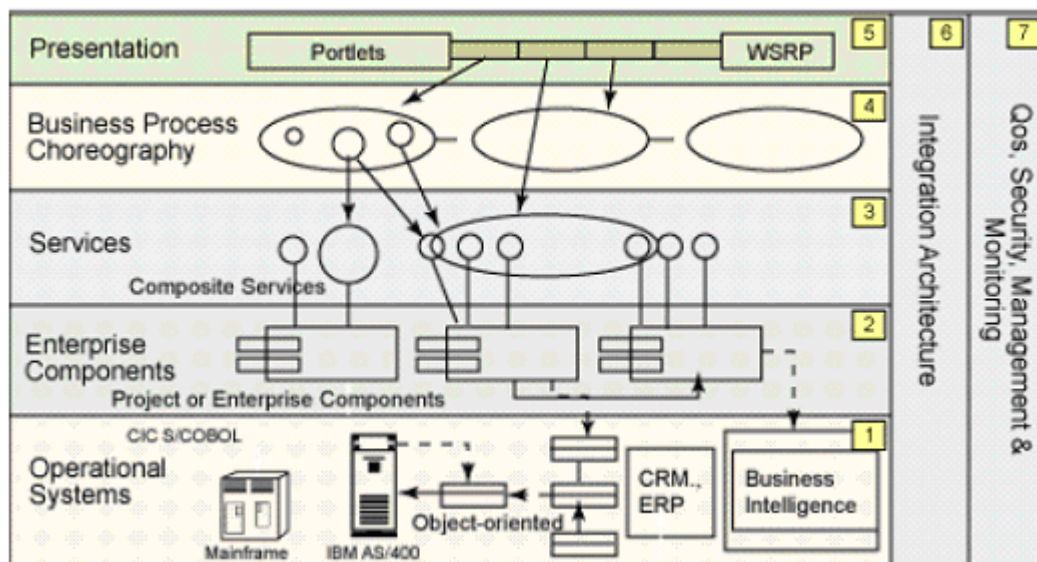
لایه سیستم های عملیاتی (لایه 1): این لایه، شامل برنامه های کاربردی سفارشی ساخته شده موجود، برنامه های کاربردی بسته بندی شده CRM و ERP موجود، سیستم های شی گرای قدیمی که قبلاً پیاده سازی شده اند و برنامه های کاربردی هوش تجاری (BI) است.

لایه مولفه های سازمانی (لایه 2): لایه مولفه های سازمانی، مسئول تحقق بخشیدن به عملکرد و نگهداری کیفیت سرویس (Qos) سرویس های در معرض نمایش گذاشته شده است. این لایه، معمولاً از فناوری های مبتنی بر کانتینر مانند سرویس دهنده های برنامه کاربردی برای پیاده سازی مولفه ها، مدیریت حجم کار، دسترس پذیری بالا و توازن بار استفاده می کند.

لایه سرویس ها (لایه 3): سرویس هایی را که کسب و کار انتخاب می کند تا آن را پایه گذاری کرده و در معرض نمایش بگذارد، در این لایه مستقر می شوند. سرویس ها می توانند کشف شده یا به طور ایستا مقید شده و سپس فراخوانی شوند یا ممکن است درون یک سرویس ترکیبی رقصانده شوند.

لایه ترکیب فرایند کسب و کار یا رقصاندن (لایه 4): ترکیب و رقصاندن سرویس هایی که در لایه 3 در معرض نمایش گذاشته شده اند، در این لایه تعریف می شود. سرویس ها از طریق هماهنگی و توازن یا رقصاندن در قالب یک روند گرد هم می آیند، بنابراین مانند یک برنامه کاربردی منفرد با یکدیگر کار می کنند. این برنامه های کاربردی از Case use اختصاصی و فرایندهای کسب و کار پشتیبانی می کنند.

لایه ارائه یا دسترسی (لایه 5): اگر چه این لایه معمولاً خارج از حوزه مباحثه درباره SOA است، اما به تدریج شایع تر می شود. با توجه به اینکه استاندارد هایی مانند WSRP نسخه 2 و سایر فناوری های مشابه سمت همگرایی بیشتر پیش می روند و به دنبال استفاده از قدرت سرویس های وب در سطح رابط برنامه های کاربردی یا ارائه هستند، این لایه در اینجا آورده شده است.



شکل 58- معماری SOA سازمانی

ذکر این نکته مهم است که SOA رابط کاربری را از مولفه ها مجزا می کند و این همان چیزی است که شما در نهایت برای ارائه یک راه حل انتها - تا - انتها از کانال دسترسی تا یک سرویس یا ترکیبی از سرویس ها به آن نیاز دارید.

معماری یکپارچه سازی (ESB - لایه 6): این لایه، یکپارچه سازی سرویس ها را از طریق معرفی مجموعه قابل اعتماد از توانمندی هایی مانند مسیریابی هوشمندانه، وساطت پروتکلی و سایر مکانیزم های تبدیلی امکان پذیر می سازد. تمام این قابلیت ها، تحت عنوان ESB قرار می گیرند و ESB یک مکانیزم یکپارچه سازی ارائه می دهد که به موقعیت ارائه دهنده یا مصرف کننده سرویس وابسته نیست.

کیفیت سرویس (لایه 7): این لایه، توانمندی هایی را ارائه می کند که برای مانیتورینگ، مدیریت و نگهداری کیفیت سرویس (QoS) مانند امنیت، کارایی و دسترسی پذیری مورد نیاز است. این یک فرایند پس زمینه ای است که از طریق مکانیزم های حس و پاسخ و ابزارهای مانیتورینگ سلامتی برنامه های کاربردی SOA مانند تمام استانداردهای مهم پیاده سازی WS-Management و سایر پروتکل ها و استانداردهای مرتبط که کیفیت سرویس SOA را تضمین می کنند، پیاده می شود. برای مدیریت سرویس ها، آنها باید ممیزی، نگهداری و پیکره بندی مجدد شوند. درپیکره

بندی مجدد، لازم است که بتوان بدون بازنویسی سرویس ها یا برنامه های کاربردی زیرین، تغییراتی روی فرایندها اعمال کرد.

4-18- نتیجه گیری:

معماری های نرم افزاری متعارف به دنبال این هستند که یک سیستم از چه اشیایی تشکیل شده، اما SOA به دنبال این است که چه نیازهایی وجود دارد که سیستم نرم افزاری نهایی، باید آنها را برآورده کند. با ظهور SOA فرایندهای کسب و کار که خود حاصل ترکیب سرویس های کسب و کار هستند، جایگزین برنامه های کاربردی می شوند. SOA با کمک رابط های سرویس و پیام های توصیفی، نیازمندی های محاسبات توزیع شده مانند پیوند ضعیف، مبتنی بر استاندارد و مستقل از پروتکل را برآورده می کند. هر مصرف کننده سرویس، می تواند توسط هر وسیله ارتباطی، به کمک هر پلتفرمی، با هر سیستم عاملی و از هر زبان برنامه نویسی به سرویس دسترسی داشته باشد. برای پیاده سازی SOA در سطح سازمانی باید از میان افزار یکپارچه سازی استفاده کرد که ESB بهترین نمونه از این میان افزارها برای پشتیبانی از SOA است. ESB با ارائه توانمندی هایی مانند مسیریابی هوشمندانه، جایگزینی سرویس ها، عملیات تبدیل داده ها و همچنین یکپارچه سازی با سایر برنامه های کاربردی، یک زیرساخت مناسب برای پشتیبانی از پیاده سازی، بکارگیری، نگهداری و مدیریت یک سیستم سازمانی مبتنی بر سرویس ارائه می دهد.

بخش 5- بخش پنجم: رویکرد و روش‌های جنبه گرا

5-1- مقدمه

در این بخش به مفهومی جدید به نام جنبه‌ها در تحلیل سیستم می‌پردازیم و اثرات این مفهوم را بر روی تحلیل سیستمهای کامپیوتری بررسی کنیم. به علت تازگی دیدگاههای جنبه گرا در سیستمهای کامپیوتری هنوز یک اجماع جامع و مانع از این مفاهیم در بین صاحب نظران بوجود نیامده است ولی همگی به وجود معضلاتی در شیوه‌های قدیمی اذعان دارند و زمان را برای تغییرات جدید مناسب می‌دانند. همچنین همگی به وجود مفهومی به نام جنبه‌ها در سیستمهای کنونی باور داشته و به دنبال رهیافتهایی برای جداسازی این مفاهیم در بخشهای مختلف تولید و توسعه‌ی سیستمها هستند. ولی این رهیافتها در بین نظریه پردازان هنوز به صورتی واحد در نیامده است و هنوز بحثهای متعددی در این زمینه وجود دارد. ولی به هر حال پیش بینی می‌شود که رهیافتهای جنبه گرا مانند رهیافت شیء‌گرایی در سالهای 89 میلادی جای خود را در صنعت IT به زودی باز نمایند. هم اکنون زبان های برنامه نویسی بسیاری در جهت حمایت از این رهیافتها بوجود آمده است که اکثرا به صورت متن باز هستند. از میان این برنامه‌ها دو زبان اصلی بزرگترین نقش را در حال حاضر بازی می‌کنند که هریک دیدگاه خاص خود را برای اجرای سیستمها دارند. البته هردوی این زبانها در اصل ادامه دهنده‌ی راه جاوا هستند. (مانند C++ که ادامه دهنده‌ی راه C بود). این دو زبان عبارتند از: AspectJ و AspectWerktz. در این بخش سعی در بیان دیدگاههای برنامه نویسی در پشت جنبه ها نداریم. به همین دلیل اطلاعات زیادی در ارتباط با این دو زبان نخواهیم داد و تنها در برخی بحثها اشاره‌ای مختصری به هر یک داشته خواهیم داشت.

5-2- رهیافت‌های جنبه‌گرا

5-2-1- تعریف

تکنیک‌های مهندسی نیازمندی‌ای که به صورت واضح سعی در شناسایی اهمیت توضیح دادن نیازمندی‌های تقاطعی عملیاتی، غیرعملیاتی و آنهایی که غیر تقاطعی اند، دارند، به عنوان رهیافت‌های مهندسی نیازمندی جنبه‌گرا شناخته می‌شود. دلیل بوجود آمدن این رهیافت‌ها سه فاکتور اصلی بوده است:

- نیاز به ترکیب
- نیاز به ردیابی
- توسعه‌ی تکنولوژی‌های جدید

اولین فاکتور تاثیرگذار بر رهیافت‌های جنبه‌گرا، این واقعیت است که در اکثر متدولوژی‌های نیازمندی، ترکیب مصنوعات (مانند دیدگاه‌ها، اهداف، زیراهداف و مورد کاربردها، در رهیافت‌های مربوطه‌ی خود)، یکی از گلوگاه‌های اصلی است. برای مثال، بسیار مشکل است که دیدگاه‌های ذی‌نفعان شخصی را برای ساخت سیستم کلی، یکپارچه نمود. به‌عنوان مثال دیگر، بسیار سخت است که اهداف معمول امنیتی و زمان‌پاسخ، با دریافت توزیع شده‌ی داده، یکپارچه شود. سختی این کار به علت پراکندگی مباحث، در مصنوعات متفاوت، (برای مثال مباحث امنیتی در بسیاری از دیدگاه‌ها وجود دارد) و تداخل درونی میان مباحث، (برای مثال امنیت و زمان پاسخ بر روی هم اثر گذارند و روی دریافت داده محدودیت ایجاد می‌کنند) است. این دو مورد در یکپارچه‌سازی، از یک دید مازولار به سیستم، و تصمیم در رابطه با مباحثی که در چندین مستند مختلف آمده است، جلوگیری می‌کند. (مبحث امنیت، در پری‌ویو، در چندین دیدگاه مختلف آمده است، ولی هیچ نمود مازولاری، ندارد.) نداشتن قابلیت تصمیمی صحیح، درباره‌ی مباحث تقاطعی در نیازمندی‌ها، باعث می‌شود، درک ما از نیازمندی‌های ذینفعان و انتظارات پایان نیابد.

فاکتور دوم، همان قابلیت ردیابی مباحث تقاطعی، در طول چرخه‌ی عمر سیستم نرم‌افزاری است. اینکه ما فقط نیازهای عملیاتی و غیرعملیاتی تقاطعی را شناسایی نموده و درباره‌ی آنها تصمیم بگیریم، کافی نیست. ما نیاز داریم، زمانی که این نیازها را شناسایی کردیم، و درباره‌ی نقاط تعادل آنها تصمیم‌گیری نمودیم، بتوانیم آنها را در فازهای معماری، طراحی، پیاده‌سازی، حتی نگهداری و ارتقاء هم ردیابی نماییم. ماژولار نمودن خصوصیات تقاطعی در سطح نیازمندی‌ها، اولین گام به سمت قابلیت ردیابی آنهاست.

فاکتور سوم، ورود زبان‌های جنبه‌گراست. این زبان‌ها، برای ماژولار نمودن مفاهیم پخش‌شده و به‌هم پیچیده (تقاطعی)، مکانیزم‌های جدید ترکیب (نقطه‌ی تقاطع و نقطه‌ی اتصال) را ارائه داده‌اند. شناسایی و رسیدگی به این خصوصیات تقاطعی، در مراحل اولیه‌ی تولید نرم افزار، باعث ایجاد یکرختی در تمام مراحل می‌گردد. همچنین، قابلیت ردیابی که در بالا اشاره شد افزوده می‌شود. بنابراین، رهیافت‌های مهندسی نیازمندی‌هایی که جنبه‌گرا هستند، کانون توجه خود را به رسیدگی، بررسی، ترکیب کردن و درنهایت ردیابی سیستماتیک نیازمندی‌های عملیاتی و غیرعملیاتی تقاطعی با مکانیزم‌های تجزیه، نمایش و ترکیب مناسبی که مخصوص زمینه‌ی مهندسی نیازمندیها است، معطوف نموده‌اند.

ما در زیر، انواع کلاس‌های رهیافت‌های جنبه‌گرا را آورده‌ایم. این کلاس‌ها همانند رهیافت‌های غیرجنبه‌گرا هستند، بجز اینکه یک کلاس (برای مهندسی نیاز چند بعدی) به آنها اضافه شده است. چنداناً از این رهیافت‌ها را به صورت کامل توضیح خواهیم داد.

- گروه تحلیل جنبه‌گرای دیدگاه-مبنا، که براساس کارهای آرکید، شکل گرفته است.
- گروه تحلیل جنبه‌گرای هدف‌گرا، که براساس مدل‌های هدف نیازمندی‌هاست.
- گروه تحلیل جنبه‌گرای موارد کاربرد و سناریو-مبنا که براساس مورد کاربردها، مدل نمودن سناریوها بوده و با رهیافت جنبه‌ها و رهیافت مورد کاربرد هدایت‌گر جنبه‌ای نشان داده می‌شود.

- گروه تجزیه‌ی چند بعدی مباحث که با کازموس و مهندسی نیازمندی‌های مبحث-گرا نمود پیدا می‌کند.
- گروه تحلیل جنبه‌گرای مولفه-مبنا، که با مهندسی نیازمندی‌های جنبه‌گرا، برای سیستم‌های نرم افزاری مولفه-مبنا، نمود پیدا می‌کند.
- دیگر رهیافت‌هایی که با عنوان تم/داک، نمود پیدا می‌کنند.

5-2-2- تحلیل جنبه‌گرای دیدگاه-مبنا

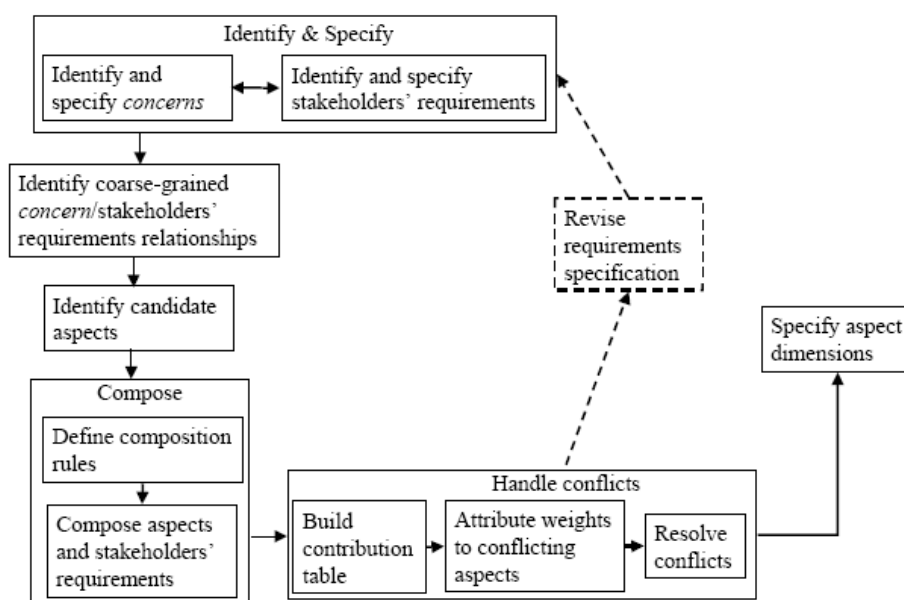
رهیافت‌های جنبه‌گرای دیدگاه-مبنا، از دیدگاه-مبنای کلاسیک مشتق شده است. تنها، نمادهای مباحث تقاطعی و ترکیب به آنها اضافه شده‌است. در حال حاضر، تنها رهیافت شناخته شده‌ی دیدگاه مبنا، آرکید است.

مهندسی نیازمندی‌های جنبه‌گرا با آرکید

این متد، اولین متدی بود که به شناسایی اولیه‌ی جنبه‌ها روی آورد. باقی متدها، در هر کلاسی دنباله‌روی مفهومی بودند که این متد معرفی نمود: شناسایی اولیه‌ی جنبه‌ها. این متد با نام مهندسی نیازمندی‌های جنبه‌گرا معرفی شد، ولی چون با نام باقی رهیافت‌ها، تداخل دارد، ما به آن نام آرکید را می‌دهیم.

5-3- متد آرکید

این متد، برای جداسازی نیازمندی‌های جنبه‌ای، غیرجنبه‌ای و قوانین ترکیب آنها بوجود آمده است. کار این رهیافت، یک فرایند عمومی را برای مهندسی نیازمندی‌ها توسعه داده است و می‌تواند توسط هر متدی مورد استفاده واقع شود. این فرایند را می‌توانید در شکل زیر مشاهده کنید، و درآینده توضیح داده خواهد شد.



شکل 59- مدل فرایند، برای مهندسی نیازمندی‌های جنبه‌گرا

به صورت حالتی خاص، مدل‌ها را توسط دیدگاه‌های شبیه به پری‌ویو نمایش داده و مکانیزم ترکیب را به صورت ایکس‌ام‌ال-مبنا انجام می‌دهیم. در این حالت خاص، نیازمندی‌های جنبه‌ای همانند مباحث پری‌ویوی -نیازمندی‌هایی با بازه‌ی وسیع‌تر، که نیازمندی‌هایی را از دیدگاه‌های بسیاری جمع کرده- قطع می‌کند. اما در پری‌ویو، هدف، شناسایی نیازمندی‌ها و مستند جزئیات نیازمندی‌هاست. آرکید، علاوه بر آن، سعی در مازولار کردن و ترکیب مباحث تقاطعی، در سطح نیازمندی، دارد. از طرف دیگر، می‌خواهد انسجام را نیز حفظ نماید. این کار بوسیله‌ی شناخت تضادها در ترکیب نیازمندی‌ها و رسیدگی به این تضادهای شناخته شده در بین دیدگاه‌ها و مباحث و نیازمندی‌ها، انجام می‌شود.

در این رهیافت، سعی می‌شود مصنوعات تولیدی، برای کسانی که قرار است به آنها نمایش داده شوند، تا حد امکان ساده باشد. در نهایت، این رهیافت برای نمایش مصنوعات از ایکس‌ام‌ال استفاده کرده، تا بتواند به آنها ساختار داده، و با وجود این هنوز قابل درک و ساده بماند. مزیت دیگر این

رهیافت در استفاده از ایکس ام ال برای قوانین ترکیب، و عملگرها، این است که می تواند تغییر کند و گسترش های مورد نیاز را، به سادگی، برای یک پروژه به همراه بیاورد.

ترکیب، در این روش از طریق یکسری عملگرها، رفتارها (Actions) و خروجی های ترکیب پشتیبانی می شود. یک نمونه از قوانین ترکیب را در زیر مشاهده می کنید.

<pre> <?xml version="1.0" ?> <Viewpoint name="ATM"> <Requirement id="1"> The ATM sends the customer's card number, account number and gizmo identifier to the system for activation and reactivation. <Requirement id="1.1"> The ATM is notified if the activation or reactivation was successful or not. <Requirement id="1.1.1">In case of unsuccessful activation or reactivation the ATM is notified of the reasons for failure. </Requirement> </Requirement> </Viewpoint> </pre>	(a)	<pre> <?xml version="1.0" ?> <Concern name="Compatibility"> <Requirement id="1"> The system must be compatible with systems used to: <Requirement id="1.1">activate and reactivate gizmos; </Requirement> <Requirement id="1.2">deal with infraction incidents; </Requirement> <Requirement id="1.3">charge for usage. </Requirement> </Requirement> </Concern> </pre>	(b)
--	-----	--	-----

شکل 60 - نمونه ای از تعریف دیدگاه و تعریف مبحث

همچنین آرکید، برای آماده سازی نیازمندی ها در مراحل دیگر تولید نیز کمک هایی را انجام میدهد. این کمک ها از طریق فراهم نمودن روش های نگاشت، امکان پذیر شده است. مستندات نیازمندی ها و روشن نمودن تصمیمات برای تضادها، که در متد آرکید، ایجاد شده است، به عنوان ورودی های چارچوب پراب، استفاده می شود. پراب، برای ایجاد رابطه میان نیازهای جنبه ای و مصنوعات پسین آنها، در مراحل بعدی تولید، توسعه یافته است.

مهمترین مسئله ای که این رویافت را پراهمیت نموده است، قابلیت جداسازی و سپس ترکیب نیازمندی های تقاطعی و غیرتقاطع، بوسیله ی پشتیبانی قابل انعطاف، ساده و در عین حال

قدرتمند از ترکیب، شناسایی و روشن نمودن تضاد، و قابلیت ردیابی و صحت‌یابی نیازمندی‌ها در طول چرخه‌ی عمر تولید نرم‌افزار است.

5-3-1- مصنوعات آرکید

مهمترین مفاهیم در نمونه‌های پری‌ویو با متد آرکید، دیدگاه‌ها و مباحث هستند. یک نمونه از دیدگاه و مبحث را در شکل قبلی مشاهده کردید.

در آرکید، هم دیدگاه‌ها، (شکل 61 قسمت a) و هم مباحث، (شکل 61 قسمت b) دارای نیازمندی‌ها و زیرنیازمندی‌هایی (که به عنوان بچه‌های بالاترین نیازمندی شناخته می‌شود.) است، و با یک عدد شناسه شناخته می‌شود. این اعداد شناسه، برای اشاره به نیازمندی خاص، در زمان ترکیب، همان‌طور که در شکل زیر (قسمت a) نمایش داده شده است، به کار می‌رود.

```
<?xml version="1.0" ?>
```

```
<Composition>
```

```
<Requirement aspect="Compatibility" id="1.1">
```

```
<Constraint action="ensure" operator="with">
```

```
<Requirement viewpoint="ATM" id="all" />
```

```
</Constraint>
```

```
<Outcome action="fulfilled" />
```

```
</Requirement>
```

```
</Composition>
```

Constraint Action:

Enforce: used to impose an additional condition over a set of viewpoint requirements.

Ensure: used to assert that a condition that should exist for a set of viewpoint

Constraint Operator:

During: describes the temporal interval during which a set of requirements is being satisfied.

With: Describes that a condition will hold for two sets of requirements with respect to each other.

Outcome Action:

Satisfied: used to assert that a set of viewpoint requirements will be satisfied after the constraints of an aspectual requirement have been applied.

Fulfilled: used to assert that the constraints of an aspectual requirement have been successfully

(a)

(b)

شکل 61- نمونه‌ای از تعریف ترکیب

در مستند ترکیب، یک تگ، به نام محدودیت‌ها (constraints) وجود دارد که نمایش‌دهنده‌ی چگونگی محدود شدن نیازمندی‌های یک دیدگاه، توسط یک‌سری جنبه است. تگ محدودیت‌ها، دارای خصوصیات: عمل، عملگر و خروجی است. یک‌سری از این خصوصیات را، می‌توان هرگاه نیاز

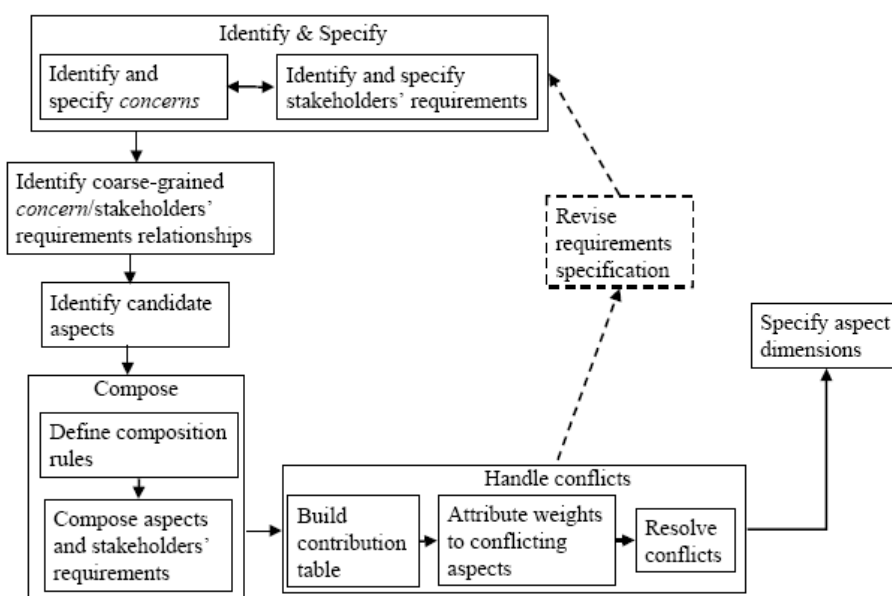
بود به دایره المعارف تحلیل اضافه نمود. یک مثال از این کار را در قسمت b ی شکل 7، مشاهده می کنید.

قانون ترکیب، وجود رابطه ای را بین نیازمندی های جنبه ای (نیازمندی هایی که برای مباحث، تعریف شده است) و نیازمندی های دیدگاهی، (نیازمندی های مربوطه به یک دیدگاه) مشخص می کند. اگر کسی بخواهد تمامی نیازمندی های مربوط به یک دیدگاه را وارد و یا خارج از این قانون، ترکیب نماید، می تواند به قسمت آی دی در تگ نیازمندی، مقدار «کل» بدهد. عملگرهای ترکیب در تگ محدودیت، نوع رابطه ی بین نیازمندی ها را مشخص می کند. قسمت عمل، نوع ابراز این رابطه را مشخص می کند. خروجی، مشخص می کند که به عنوان خروجی این ترکیب، انتظار چه چیزی را داشته باشیم.

علاوه بر این المانهای ایکس ام ال-مبنا، آرکید، از ماتریس هایی برای ارتباط دادن بین مباحث و دیدگاه ها به هم، به همراه تخصیص وزن به نیازهای متضاد در هنگام روشن نمودن تضادها، استفاده می کند.

5-3-2- فرایند آرکید

فرایند آرکید را در شکل زیر مشاهده می کنید:



شکل 62- فرایند آرکید

فرایند، با شناسایی نیازمندی‌های ذینفعان آغاز می‌شود. وقتی با پری‌ویو مقایسه می‌شود، این همانند شناسایی دیدگاه‌هاست. کلکسیونی از نیازمندی‌های جمع می‌شود. این نیازمندی‌ها، سپس توسط الگوهای ایکس‌ام‌ال نمایش داده می‌شوند.

قسمت بعدی فرایند، به شناسایی و جزئیات مباحث می‌پردازد. این متد، برعکس پری‌ویو، که مباحث را اول شناسایی کرده و سپس به شناسایی نیازمندی‌ها، با توجه به آن مباحث می‌پردازد، عمل می‌کند. در اصل آرکید، مباحث را با تحلیل نیازمندی‌های اولیه کشف می‌کند و همچنین امکان شناسایی نیازمندی‌ها و مباحث را به صورت بازگشتی می‌دهد. این کار توسط شناسایی رابطه‌های اصلی بین مباحث و دیدگاه‌ها، امکان‌پذیر شده است. برای این کار یک ماتریس تهیه می‌شود، که چه مبحثی روی چه دیدگاهی اثر می‌گذارد. اگر یک مبحث بر روی چند دیدگاه اثر گذار بود، به عنوان یک کاندید برای جنبه شدن، در نظر گرفته می‌شود.

سپس تولیدگران، قوانین ترکیب را برای نیازمندی‌های جنبه‌ای و نیازمندی‌های دیدگاهی، در سطح یک نیاز مطرح می‌سازند. در اینجا یک مهندس نیازمندی، می‌بایست مشخص سازد نیازمندی‌های دیدگاهی مشخص شده، چگونه می‌بایست با مباحث ترکیب شوند. سپس در صورت لزوم، عملگرهای

جدیدی را برای این تغییر مشخص سازد. سپس مبحث و دیدگاه با توجه به قوانین ذکرشده، ترکیب می‌شوند.

پس از ترکیب، رسیدگی به تضادها در دو سطح پشتیبانی می‌شود. اول، با توجه به توانایی ترکیب نیازمندی‌های جنبه‌ای و غیر جنبه‌ای به صورت دانه‌های ریز، نیاز به تحلیل تضادهای ممکن، برای وقتی که چندین مبحث، به چندین نیاز در یک دیدگاه اثر می‌گذارند. دوم، مکانیزمی برای روشن نمودن و بررسی تضادها در زمانی که مباحث بر روی یک نیازمندی اثر می‌گذارند. در این حالت یک ماتریس روابط می‌بایست تشکیل شود، در حالتی که روابط منفی هستند، یک مشکل ایجاد شده است. سپس فواصل وزنی‌ای براساس فواصل فازی را تولید، و استفاده می‌کنیم. سپس نیازمندی‌های متضاد را اولویت‌بندی می‌کنیم. تنها در زمانی که دو نیاز متضاد وزن‌های یکسان داشتند، می‌بایست بین دینفعان پروژه مذاکره انجام پذیرد.

درنهایت، فرایند با مشخص نمودن نقاط جنبه، پایان می‌یابد. (یعنی مشخص شود که هر جنبه می‌بایست تبدیل به یک عملیات، تصمیم یا یک جنبه در مراحل بعدی نرم‌افزار گردد)

تمامی مستندات و مصنوعات که در فرایند بالا شرح داده شد، یک ورودی را برای چارچوب پراب شکل می‌دهد.

5-4-4- تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها

تحلیل و تولید جنبه‌گرا با موردکاربردها، این دید را دارد که هر مورد کاربرد مباحث تقاطعی‌ای را نشان می‌دهد، در حالیکه نمود واقعی هر موردکاربرد چندین کلاس را تحت‌الشعاع قرار می‌دهد. این رهیافت، یک دید شبه-زبان-پیاده‌سازی، بر موردکاربردها به عنوان جنبه دارد. این دید، شدیداً توسط زبان‌های AspectJ و HyperJ تحت‌الشعاع قرار گرفته است. تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها از تکه‌های زبانی به شیوه‌ی AspectJ، (مانند نقاط تقاطع و نقاط اتصال) ماژول‌های تجزیه‌ای همانند HyperJ (مانند برش‌ها) را با خود وفق می‌دهد.

5-4-4-1- متد تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها

تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها، رهیافت کلاسیک مورد کاربرد را توسعه می‌دهد. به همراه این توسعه‌ی دوالمان اصلی معرفی می‌گردد: نقاط تقاطع، برای موردکاربردها و جمع مصنوعات تولید در برش‌های موردکاربرد و ماژول‌های مورد کاربرد.

- نقاط تقاطع، در اینجا گروهی از نقاط اتصال مورد کاربرد که با نقاط گسترش‌یافته نمایش داده می‌شوند و همانند المان‌هایی مانند کلاس‌ها و عملگرها و ... در درون یک مورد کاربرد هستند. در حالی که نقاط گسترش‌یافته، (که در جزئیات مورد کاربرد به عنوان نقاط شناخته می‌شود) در مدلسازی مورد کاربرد حتی قبل از تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها هم وجود داشتند، ولی تنها برای اشاره به یک مورد کاربرد گسترش‌یافته شدند که به جداسازی مورد کاربردهای پایه و توسعه‌یافته کمک کرده و اجازه‌ی شناسایی نقاط تقاطع را در فازهای معماری و طراحی می‌دهد.
- یک برش موردکاربرد، شامل جزئیات یک موردکاربرد را در یک فاز تولیدی داده شده است، مانند فاز تحلیل یا طراحی. یک ماژول موردکاربرد در مقابل، شامل تمام جزئیات مربوط به یک موردکاربرد در تمام فازهای تولیدی می‌گردد.

این رهیافت، بین دو نوع مورد کاربرد تفاوت قائل میشود: نظیر و گسترش. مورد کاربردهای نظیر، به صورت مشخص و مستقل از همدیگرند، هریک می‌تواند به صورت مجزا، بدون داشتن مرجع به همدیگر استفاده می‌شوند، این‌ها نیازمندی‌های پایه‌اند، وقتی که مورد کاربردهای نظیر ترکیب می‌شوند، تمامی فعالیت‌ها بدون تداخل در عملیاتشان، ترکیب می‌شوند. ولی ترکیب مورد کاربردهای نظیر، نیاز به توجهاتی نسبت به رفتارهای پوشش و تضاد بین کلاس‌هایی که برای واقعی‌سازی مورد کاربردهای مختلف استفاده دارد. گسترش‌ها در اصل قابلیت‌های اضافی‌ای بر روی مورد کاربردهای پایه هستند. بنابراین گسترش‌ها را می‌توان به صورت مجزا از موارد پایه تعریف نمود، ولی به طور نرمال به همراه پایه‌ها استفاده می‌شود. وقتی که گسترش‌ها با موارد پایه ترکیب می‌شوند، عملیات‌های آنها معمولاً با اجرای عملیات‌های موارد پایه تداخل دارد.

همچنین، متد تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها، نیازهای غیرعملیاتی را نیز به عنوان مورد کاربردها دریافت می‌کند. این کار از طریق سازه‌ای به نام زیربنای موردکاربرد، به فعالیت‌هایی که زیر بنای سیستم، باید برای برآورده ساختن نیاز کاربران انجام می‌دهد.

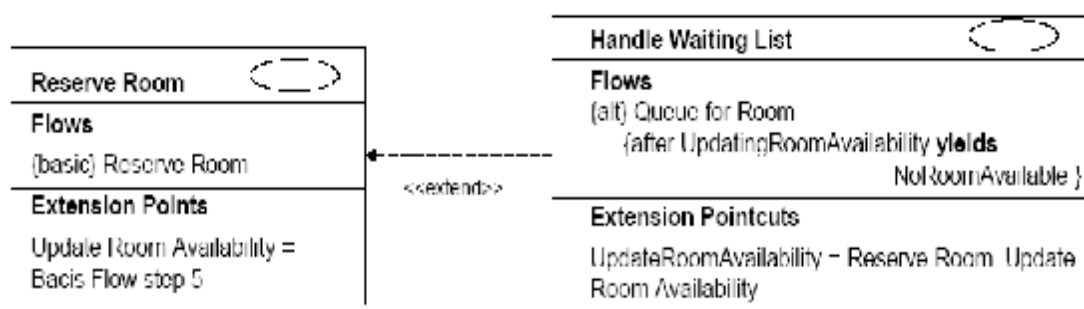
5-4-2- مصنوعات تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها

تمامی مصنوعاتی که در قسمت موردکاربردها معرفی گردید، قابل اعمال است. اگرچه بعضی از آنها تا حدودی تغییر می‌کند.

نمایش موردکاربردها در یوامال، به عنوان یک کلاسه‌کننده، می‌تواند از طریق شکل زیر(به صورت بیضی) باشد. این تفاوت نمایش مورد کاربرد به عنوان کلاسه‌کننده، اطلاعات بیشتری را در اختیار ما می‌گذارد.

یک کلاسه، بوسیله‌ی تگهای `Basic/alt/` و `sub/` برای جریان‌ات پایه، جایگزین و اجرایی به صورت مجزا، گسترش‌یافته است. تگ `basic`، مشخص کننده‌ی این است که مورد کاربرد می‌تواند توسط بازیگر شروع گردد، ولی تگ `sub`، مشخص کننده‌ی این است که این جریان می‌تواند توسط یک

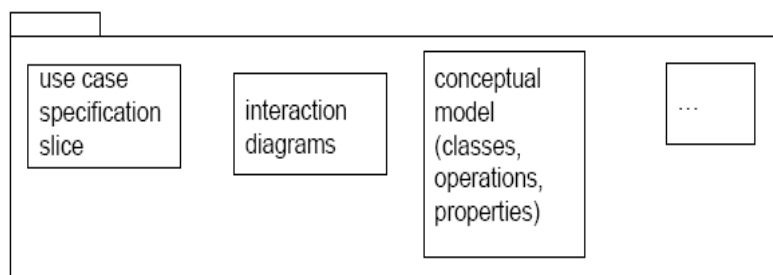
جریان دیگر ارجاع داده شود، یا در درون یک جریان مشمول گردد. جریان جایگزین به عنوان مورد کاربردهای گسترش یافته تعریف گردد.



شکل 63 - نمونه‌ای مورد کاربردهای اختصاصی، به همراه کلاسه کننده‌های آن

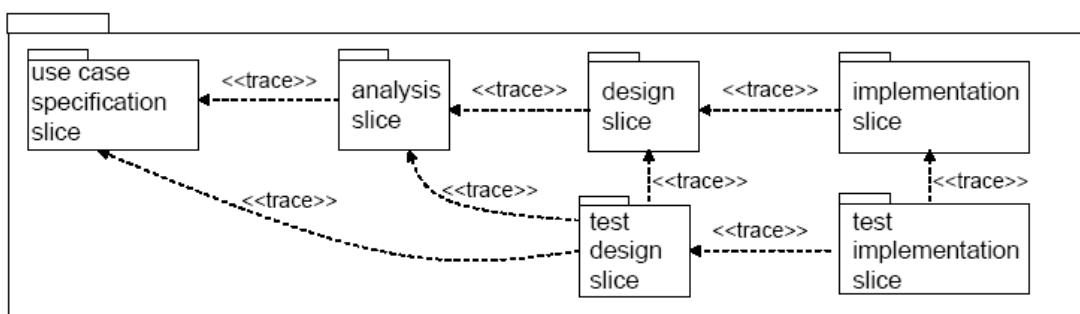
همانطور که در شکل 63 دیده می‌شود، مورد کاربردهای گسترش یافته می‌توانند توسط مورد کاربردهای پایه، بوسیله‌ی نام‌های زیرجریان‌ها (که نقاط گسترش نامیده می‌شوند و در قسمت نقاط گسترش مورد کاربردها آورده می‌شوند)، ارجاع داده می‌شود. این نقاط گسترش هم در عوض در مورد کاربردهای گسترش یافته، از طریق نقاط گسترش ارجاع داده می‌شوند. تعریف جریان در مورد کاربردهای گسترش یافته، می‌تواند توسط یک شرط گسترش و کلمات کلیدی: قبل (before)، بعد (after) و اطراف (around)، تکمیل گردد. این دو بخش نمایان‌گر شرایط و ترتیب اجرای نقاط گسترش در جریان مورد کاربرد پایه‌اند. نقاط گسترش و تقاطع، بعدها برای توسعه‌ی نقاط تقاطع سطح طراحی، استفاده می‌شوند.

دو نماد جدید برش مورد کاربردها و مازول مورد کاربرد، در تحلیل جنبه‌گرا بوسیله‌ی مورد کاربردها برای محلی‌سازی مورد کاربردهای مرتبط و مصنوعات تکمیلی، در مرحله‌ی خاص تولید و در کل مراحل استفاده می‌شوند. بعضی از المان‌هایی که در سطح نیازمندی‌ها، در برش مورد کاربردها استفاده می‌شود را، در شکل زیر مشاهده می‌کنید.:



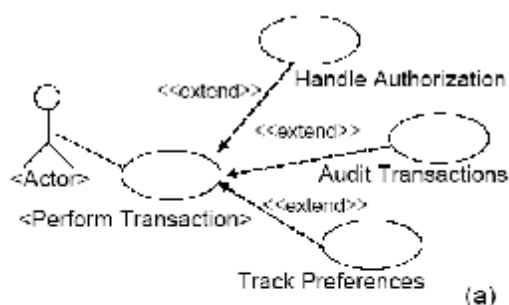
شکل 64- بعضی المان‌هایی که در برش مورد کاربردهای سطح نیازمندی‌ها

برش‌های مورد کاربرد را، می‌بایست برای مراحل دیگر تولید نیز بوجود آورد، که المان‌های مخصوص خود را داراست. برای مثال، برش طراحی، می‌بایست شامل دیاگرام‌های اثرمتقابل، کلاس‌ها و گسترش و ... برای هر مورد کاربرد باشد. تمامی این برش‌ها را در مازول یک مورد کاربرد جمع کرده و مانند شکل 65 نمایش می‌دهیم:



شکل 65- پیمانه‌ی موردکاربردها

زیرساختار موردکاربرد خاص «تراکنش اجرا»، برای دریافت نیازمندی‌های غیرعملیاتی استفاده می‌شود. الگوی نیازمندی‌های غیرعملیاتی می‌بایست به صورت موردکاربردهای گسترش‌یافته، برای «تراکنش اجرا»، همانند آنچه در شکل زیر آمده است، نمایش داده شود:



Handle Authorization	
Flows	
{basic} Define Permission	
{all} Check Authorization (around PerformTransactionRequest)	
Extension Pointcuts	
PerformTransactionRequest = Perform Transaction	
Perform Request	

(c)

Use-Case Specification: <Perform Transaction>

Basic Flows

The use case begins when an actor instance performs a transaction to view the values of the entity instance.

1. The system prompts the actor to identify the desired entity instance.
2. The actor instance enters the values and submits this request.
3. The system retrieves the entity instance from its data store and displays its values.
4. The use case terminates.

Alternative Flows

A1: Access Control

If in step 3 of the basic flow the request requires authorization, the system checks the actor instance's access rights.

1. If the actor instance does not have access rights, the request is rejected. The use case terminates.
2. Otherwise, the use case proceeds.

A2: User Preference

Special Requirements:

All retrieval of records should take no longer than 2 seconds.

(b)

شکل 66- یک مثال از زیربنای مورد کاربرد در فرایند جنبه‌گرا توسط مورد کاربردها

مزایای استفاده از الگوی مورد کاربرد، این است که کمک می‌کند زمینه‌ی مکانیزم زیر ساختار را تصویر کند، (مانند مجوزگیری و ...) و به عنوان یک پایه برای شناسایی سیستماتیک نقاط گسترش استفاده می‌شود.

همچنین این رهیافت، یک سری مصنوعات سطح طراحی، مانند نقاط تقاطع سطح طراحی، برش مورد کاربرد پارامتری شده و ... را فراهم می‌کند، اما این مصنوعات به این بخش مربوط نبوده و در اینجا نیز آنها را نمی‌آوریم.

5-4-3- فرایند تحلیل جنبه‌گرا بوسیله‌ی مورد کاربردها

تحلیل جنبه‌گرا بوسیله‌ی مورد کاربردها، یک رهیافت تولید است، که تمامی چرخه‌ی عمر تولید را پوشش می‌دهد؛ از جمع آوری نیازمندی‌ها تا پیاده سازی. ما در اینجا تنها به بخش مهندسی نیازمندی‌ها، توجه می‌کنیم.

برای بخش مهندسی نیازمندی‌ها، فرایند تحلیل جنبه‌گرا بوسیله‌ی مورد کاربردها، بسیار شبیه به فرایند مورد کاربردهای کلاسیک است، تنها تفاوت اساسی، وجود مورد کاربردهای مجزا برای

نیازمندی‌های غیرعملیاتی است. این‌ها را می‌بایست توسط زیرساخت‌های مورد کاربرد شناسایی، تعریف و ذخیره سازی نماییم. در همان حال می‌بایست مورد کاربردهای عملیاتی را نیز شناسایی و بررسی نماییم. تفاوت جزئی، جمع نمودن هر مورد کاربرد و المان‌های مربوط به آن در برش مورد کاربرد است.

5-4-4- شناسایی و بررسی مباحث تقاطعی با تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها

شناسایی و بررسی مباحث تقاطعی با تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها، در مرحله‌ی مهندسی نیازمندی‌ها، بسیار شبیه رهیافت مورد کاربردهاست. تنها تفاوت جزئی، استفاده از نمایش کلاسه‌ها در بخش نقاط تقاطعی گسترش برای نمایش مورد کاربردهاست.

به علاوه، تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها برای بررسی نیازمندی‌های غیرعملیاتی نیز قابلیت‌هایی را معرفی می‌کند: «تا وقتی که نیازمندی‌ها، نیازبه پاسخ مشخص برای برنامه‌نویسی شدن در سیستم را داشته باشند، شما می‌توانید از مورد کاربردها استفاده نمایید.» و «تا وقتی که شما بتوانید یک تست را تعریف کنید، می‌توانید یک مورد کاربرد را برای آن تعریف کنید.» همان طور که در شکل قبل مشخص بود. آنها را توسط گسترش‌هایی برای الگوی مورد کاربرد «تراکنش اجرا» مدل نمودیم.

البته آرگومان تعریف مورد کاربرد غیرعملیاتی، با تعریف معمول یک موردکاربرد که می‌گوید: «چیزی که کاربر انجام می‌دهد و چیزی که سیستم پاسخ می‌گوید»، مخالف است. یک کاربر برای نیازمندی‌های غیرعملیاتی هیچ کاری نمی‌کند، او فقط به آنها احتیاج دارد که آنها باشند. بنابراین هیچ حرکتی از طرف کاربر برای اکثر موارد انجام نمی‌گیرد. برای مثال، وقتی که یک کاربر از طریق اینترنت به سیستم متصل می‌گردد، هیچ کار خاصی برای اینکه پاسخ خود را در کمتر از 2 ثانیه دریافت کند، انجام نمی‌دهد، یا برای هماهنگی query خود با دیگر queryهایی که در حال بروزرسانی پایگاه داده است، کاری را نمی‌کند، اینها به عنوان جزئی از عملکرد سیستم، مد نظرند. بنابراین شناسایی مورد کاربرد غیرعملیاتی در این مورد، می‌بایست طبق ترتیب زیر باشد:

1. آنچه کاربر می‌خواهد.

2. چگونه تست خواهد شد.

3. طراحی تست

4. تبدیل آن به مورد کاربرد

5. گسترش قابلیت‌ها، با مورد کاربردهای جدید

یکی دیگر از مناطق نامطمئن در تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها، تعریف جنبه‌ها در آن است. همانطور که قبلاً اشاره شد، یک مورد کاربرد، به عنوان یک جنبه شناخته می‌شود، اگر و تنها اگر چندین کلاس را قطع نماید. البته این، خصوصیت را به مبحث سطح نیازمندی‌ها، براساس نمود طراحی آن تخصیص می‌دهد. یک چنین مبحثی حتماً در سطح طراحی، تقاطعی خواهد بود، (با کلاس‌های شیء‌گرایی)، ولی این برای اینکه آن را در سطح نیازمندی‌ها هم تقاطعی بدانیم، کافی نیست.

5-5- مدل نمودن سناریوها بوسیله‌ی جنبه

مدل کردن سناریو بوسیله‌ی جنبه‌ها، به تاثیرات آنها در رفتار مدل‌سازی تمرکز می‌نماید، با توجه به دیدگاه یوام‌ال-مبنا، در اینجا رفتارهایی تکراری را در جنبه‌ها عمومی‌سازی می‌کنیم. سپس در زمان مناسب از آنها استفاده می‌کنیم.

5-5-1- متد مدل کردن سناریو با مدل جنبه‌ها

انگیزه‌ی حرکت این رهیافت، کمک به توسعه‌ی مجموعه‌ی کامل‌تر و منسجم‌تری از سناریوها بوسیله‌ی حذف نیاز به توجه مکرر به سناریوهای یکسان، تقاطعی (مانند نارسایی‌ها، رخ داده‌ای اتفاقی و ...)، که در بسیاری از سناریوها اتفاق می‌افتد، است.

قسمت‌های غیرتقاطع و جنبه‌ای سناریوها به صورت مجزا از هم مدل می‌شوند و سپس در صورت نیاز، با هم ترکیب می‌گردند و یک سناریوی کامل را بوجود می‌آورند و سپس سناریوهای کامل را اگر کنار هم قرار دهیم، می‌توانیم برای اهداف صحت‌یابی اجرایشان کنیم.

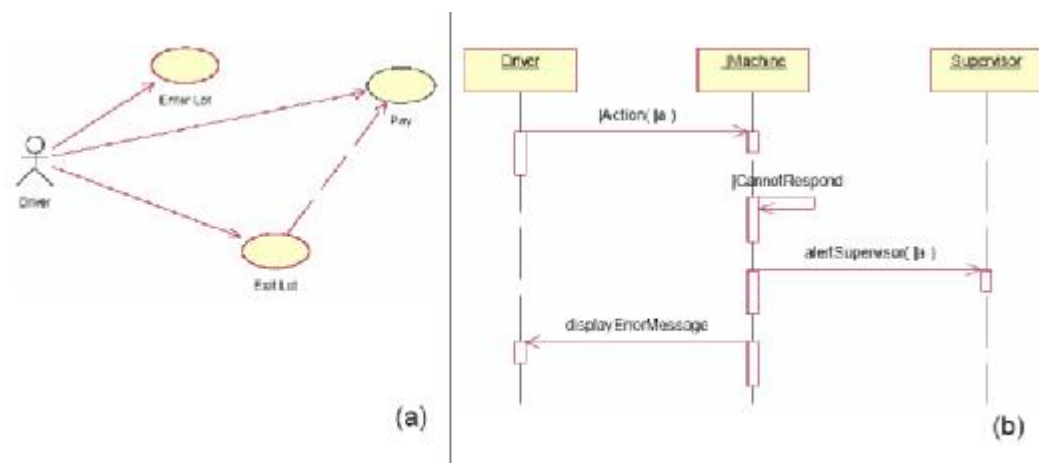
5-5-2- مصنوعات مدل کردن سناریوها با جنبه

این رهیافت، از موردکاربردها برای شناسایی نیازمندی‌های عملیاتی، استفاده می‌کند. بنابراین تمامی مصنوعاتی که در رهیافت موردکاربرد کلاسیک نام برده شدند، اینجا هم مورد استفاده دارند. یک دیگرام مورد کاربرد را به صورت نمونه در شکل زیر قسمت a، برای استفاده و پرداخت یک پارکینگ مشاهده می‌کنید (البته این مثال در انتهای بحث به صورت کامل تشریح می‌گردد). علاوه برآن، دیگرام ترتیب برای نمایش سناریوهای تقاطعی استفاده می‌شود.

سناریوهای جنبه‌ای به صورت جزییات الگوی اثرات متقابل (IPS)¹²⁵ نمایش داده می‌شوند، به صورتی که جزییات الگوی اثرات متقابل، یک الگوی ارتباطی را بین اجزای درگیرش با توجه به نقش‌هایی که اجزای درگیر می‌بایست پرکنند، تشریح می‌کند. نمونه‌ای از این الگو را در قسمت b شکل زیر مشاهده می‌کنید. با اینکه بسیار شبیه به دیگرام اثرات متقابل¹²⁶ در یوام‌ال است، ولی بعضی از پیام‌ها و پارامترهای آن دارای علامت | قبل از خود هستند. این علامت نمایان‌گر نقشی است که نیاز دارد با کسی که از الگو استفاده می‌کند، برگردد.

¹²⁵ Interaction Pattern Specification

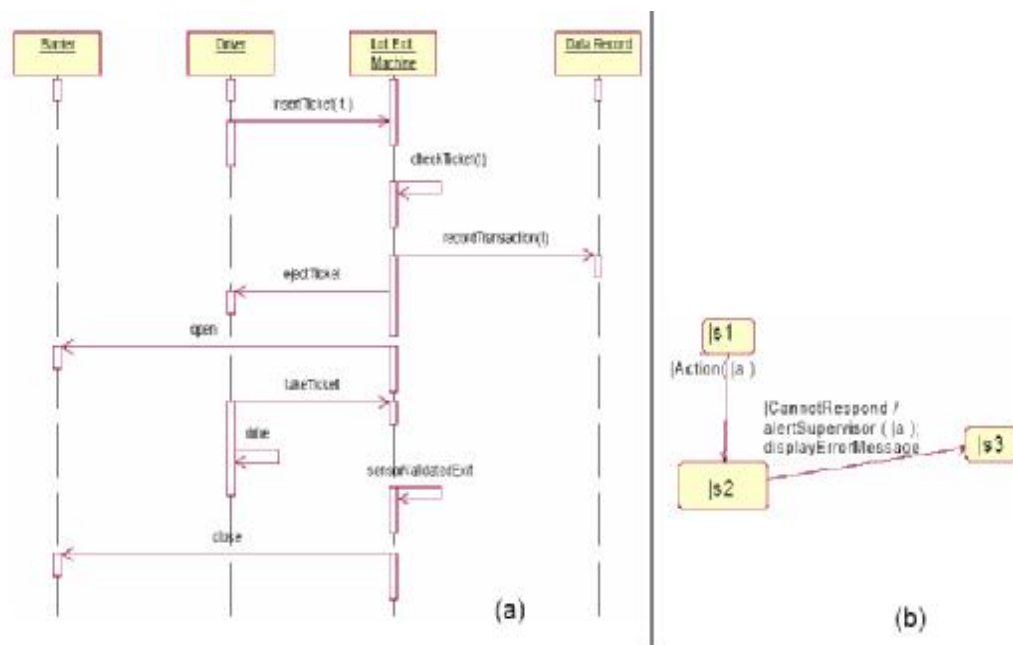
¹²⁶ Interaction Diagram



شکل 67- مصنوعات مدل کردن سناریوها بوسیله‌ی جنبه‌ها: (a) یک مثال از مورد کاربرد،

(b) یک مثال از جزئیات الگوی اثرات متقابل

سناریوهای جنبه‌ای از جزئیات الگوی اثرات متقابل، به جزئیات الگوی ماشین وضعیت (State Machine Pattern Specification، SMPS)، توسط الگوریتم سنتز ماشین وضعیت، تبدیل می‌شوند. یک مثال از جزئیات الگوی ماشین وضعیت، شبیه به دیاگرام ماشین وضعیت در یوامال است ولی در اینجا هم نقش‌هایی وجود دارد که نیاز دارند پر شوند.



شکل 68- (a) دیاگرام ترتیب یوامال، (b) جزئیات الگوی ماشین وضعیت

قبل از اینکه جزئیات الگوی ماشین وضعیت برای سناریوهای جنبه‌ای و دیاگرام ماشین وضعیت معمولی (FSM, Finite State Machine) برای سناریوهای غیرجنبه‌ای ترکیب شوند، نقش‌هایی که درون جزئیات الگوی ماشین وضعیت تعریف شده‌اند، نیاز دارند که به المان‌هایی در جزئیات ماشین وضعیت معمولی، نگاشت شوند. به همین دلیل جزئیات اتصال ماشین وضعیت (SMBS, State Machine Binding Specification) را می‌بایست تهیه نمود. برای مثال، یک نمونه از این مستند را برای اتصال قسمت bی شکل قبل و قسمت aی شکل زیر به صورت زیر آورده‌ایم:

s1 به T1 متصل شود.

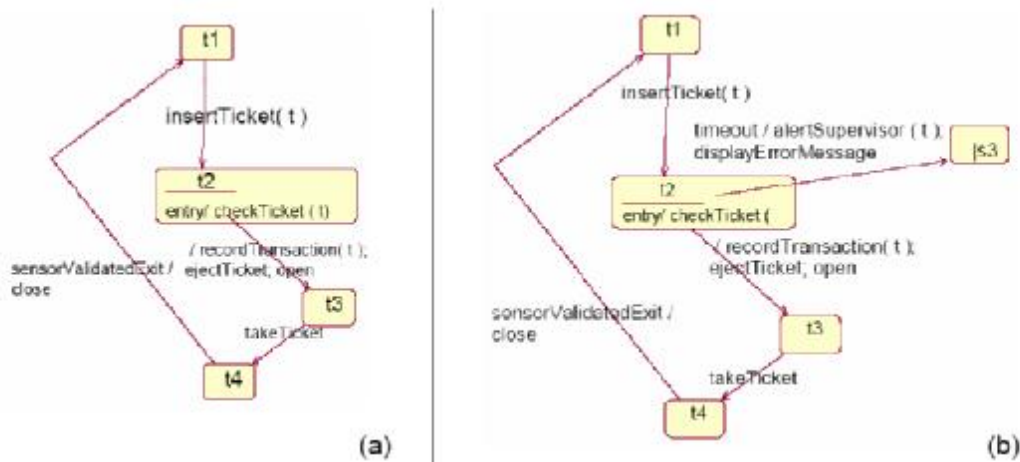
s2 به T2 متصل شود.

action به داخل کردن بلیط متصل شود.

a به t متصل شود.

نتوانستن پاسخ (cannot respond)، به پایان زمان (timeout) متصل شود.

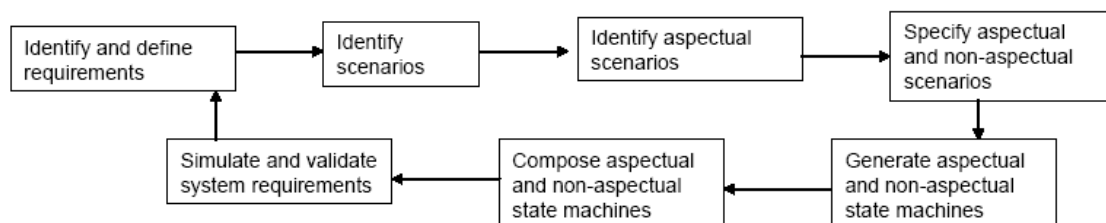
پس از اینکه این جزئیات تهیه گردید، دیاگرام‌های گفته شده را می‌توان متصل نمود و با هم ترکیب کرده، تا یک دیاگرام ماشین وضعیت کامل، همانند شکل زیر قسمت بوجود آید:



شکل 69- A) ماشین وضعیت نهایی ای که از دیاگرام اثرات متقابل از شکل 14- a بدست آمده است. B) ماشین وضعیت نهایی ترکیب شده از ماشین وضعیت شکل 14- b و 15- a

5-3-5- فرایند مدل نمودن سناریوها توسط جنبه

شکل زیر فرایند رهیافت مدل کردن سناریوها توسط جنبه ها را نشان می دهد:



شکل 70- فرایند مدل کردن سناریوها بوسیله رهیافت جنبه ای

فرایند، با شناسایی و تعریف نیازمندی ها آغاز می شود. این کار بوسیله ی موردکاربردها، برای شناسایی نیازمندی های عملیاتی و الگوها، برای نیازمندی های غیرعملیاتی انجام می دهیم. این قسمت، با بازبینی موردکاربردها و شناسایی سناریوها ادامه می یابد. در اینجا نیازمندی های غیرعملیاتی به سناریوهای تقاطعی تبدیل می شوند، همچنین سناریوهای تقاطعی دیگری نیز برای

بعضی از نیازمندی‌های عملیاتی شناسایی می‌گردد. پس از آن، سناریوهای غیر تقاطعی را به دیاگرام ترتیب¹²⁷ یوامال تبدیل می‌کنند. سپس ماشین وضعیت معمولی را به وسیله‌ی الگوریتم سنتز ماشین وضعیت تشکیل می‌دهیم.

سناریوهای جنبه‌ای را به جزئیات الگوی اثرات متقابل تبدیل می‌کنیم، که آنها هم به جزئیات الگوی ماشین وضعیت، بوسیله‌ی الگوریتم سنتز ماشین وضعیت تبدیل می‌شود.

سپس ماشین‌های وضعیت جنبه‌ای، نمونه‌سازی شده و در نهایت با ماشین وضعیت غیرجنبه‌ای با استفاده از جزئیات اتصال ماشین وضعیت ترکیب می‌شوند. در این جزئیات الگوی ماشین وضعیت، روابط متقابل بین وضعیت‌های هر یک از ماشین وضعیت‌ها، فراهم می‌کند.

در این حالت، ماشین وضعیت ترکیب‌شده را بوسیله‌ی ابزارهای شبیه‌سازی می‌توان شبیه‌سازی نمود، به صورت جایگزین می‌توان از ابزارهای تحلیل که برای تکنیک‌های چک کردن مدل معمولی ایجاد شده، برای تأیید صحت و اعتبار سناریوها، در برابر نیازمندی‌ها استفاده نمود.

در جای دیگر، در مقابل قسمت آخر فرایند، که در سطح ماشین وضعیت، عمل ترکیب را اجرا می‌کند، فرایندی است که پس از تشکیل جزئیات الگوی اثرات متقابل برای جنبه‌ها، یک دیاگرام اثرمتقابل¹²⁸ تولید می‌گردد و سپس دیاگرام‌های ارتباطی نیازمندی‌های جنبه‌ای و غیرجنبه‌ای را ترکیب کرده و از روی دیاگرام ارتباطی ترکیب شده یک دیاگرام ماشین وضعیت را تولید می‌کند. در این حالت در دیاگرام ارتباطی، عملگرهایی برای جزئی‌سازی پیام‌های ردوبدل شده در بین سناریوهای مختلف بکار می‌رود. اگرچه مسیر این دو جزء فرایند، جدا است، ولی چون در نهایت یک چیز را تولید می‌کنند، هر دو قابل قبولند.

5-5-4- شناسایی و بررسی مباحث تقاطعی با مدل کردن سناریوها بوسیله‌ی جنبه‌ها

این رهیافت به مسئله‌ی شناسایی سناریوهای جنبه‌ای توجه شایانی ندارد و تنها مشخص می‌سازد که سناریوهایی که نماینده‌ی مورد کاربردهایی‌اند که بر دیگر مورد کاربردها اثر می‌گذارند (مانند

¹²⁷ sequence diagram

¹²⁸ Interaction Diagram

دردست گرفتن خطاها)، جنبه‌ای هستند. این رهیافت بر اینکه یک نیاز غیرعملیاتی، تقاطعی است، باور دارد. مانند سناریوهایی چون، جلوگیری از اتصال کاربر در زمانی که کار دیگری در حال انجام است. با اینکه این می‌تواند بخشی از نیازمندی غیرعملیاتی باشد، ولی بحث بیشتری برای روشن نمودن اینکه همچون سناریوئی، چگونه می‌تواند نمایانگر نیاز غیرعملیاتی باشد، موجود است. ولی اگر نیازهای جنبه‌ای و غیرجنبه‌ای شناسایی شده باشند، در این رهیافت بررسی و عملیات روی نیازمندی‌ها به صورت کامل تشریح شده است.

5-6- رهیافت مورد کاربرد مبنای جنبه‌ای

این رهیافت از جهت جداسازی عملیات‌های تقاطعی به مورد کاربردهای شمول و گسترش و همچنین مراحل تبدیل صفات کیفی به مورد کاربردها، به تحلیل جنبه‌گرا بوسیله‌ی موردکاربردها، شبیه است. اما در اینجا، صفات کیفی به مورد کاربرد زیرساختار تبدیل نمی‌شوند، همچنین در این رهیافت الگوهایی برای شناسایی مباحث تقاطعی فراهم آورده شده است.

5-6-1- متد مورد کاربرد مبنای جنبه‌ای

این رهیافت سعی در گسترش مدل مورد کاربردها برای یکپارچه‌سازی نیازمندی‌های غیرعملیاتی و شناسایی موردکاربردهای عملیاتی تقاطعی دارد. موردکاربردهای مجزا نیز، برای تمامی مباحث نیز می‌بایست در یک مدل کامل با استفاده از مکانیزم‌های ترکیب، یکپارچه گردند. این متد کارهای انجام شده برای شناسایی و یکپارچه‌سازی صفات کیفی تقاطعی را گسترش می‌دهد. به این صورت که، صفات کیفی به عنوان ابتدائی‌ترین پایه‌های مدل کردن حساب می‌شوند، این متد همچنین از در دست گرفتن تغییرات نیازمندی‌ها هم پشتیبانی می‌کند.

5-6-2- مصنوعات رهیافت مورد کاربرد مبنای جنبه‌ای

چون این رهیافت بر مبنای رهیافت کلاسیک مورد کاربردها بنانهاده شده است، تمامی مصنوعات موجود در آن رهیافت (همانند دیاگرام مورد کاربردها) در اینجا هم‌کارایی دارند.

علاوه بر آن، در این رهیافت صفات کیفی را نیز در ابتدا توسط الگویی که در زیر آمده است، مشخص می‌کنیم. در اینجا برای آنکه بدانیم که یک صفت کیفی، تقاطعی است، لازم است که به ردیف مکان و نیازمندی‌ها در شکل زیر توجه کنیم، اگر در این دو ردیف چندین مورد کاربرد/مدل و نیازمندی را مشاهده کنیم، صفت تقاطعی است:

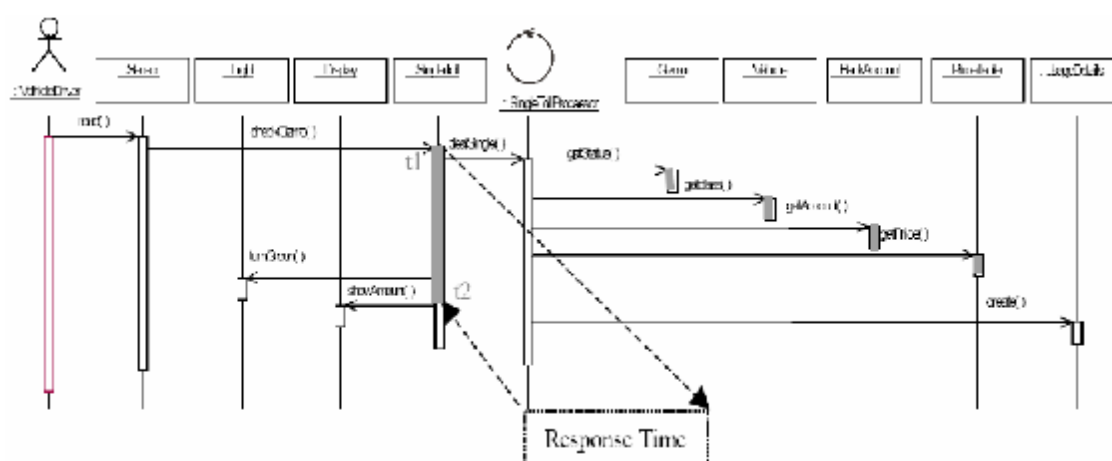
Name	The name of the quality attribute
Description	Executive description
Focus	A quality attribute can affect the system (i.e. the end product) or the development process
Source	Source of information (e.g. stakeholders, documents)
Decomposition	Quality attributes can be decomposed into simpler ones. When all (sub) quality attributes are needed to achieve the quality attribute, we have an AND relationship. If not all the sub quality attributes are necessary to achieve the quality attribute, we have an OR relationship
Priority	Expresses the importance of the quality attribute for the stakeholders. A priority can be MAX, HIGH, LOW and MIN
Obligation	Can be optional or mandatory
Influence	Activities of the software process affected by the quality attribute
Where	List of the actors influenced by the quality attribute and also a list of models (e.g. use cases and sequence diagrams) requiring the quality attribute
Requirements	Requirements describing the quality attribute
Contribution	Represents how the quality attribute affects other quality attributes. This contribution can be positive (+) or negative (-)

شکل 71- جدول الگویی برای صفات کیفی

علاوه بر آن، این رهیافت، نمادهای زیر

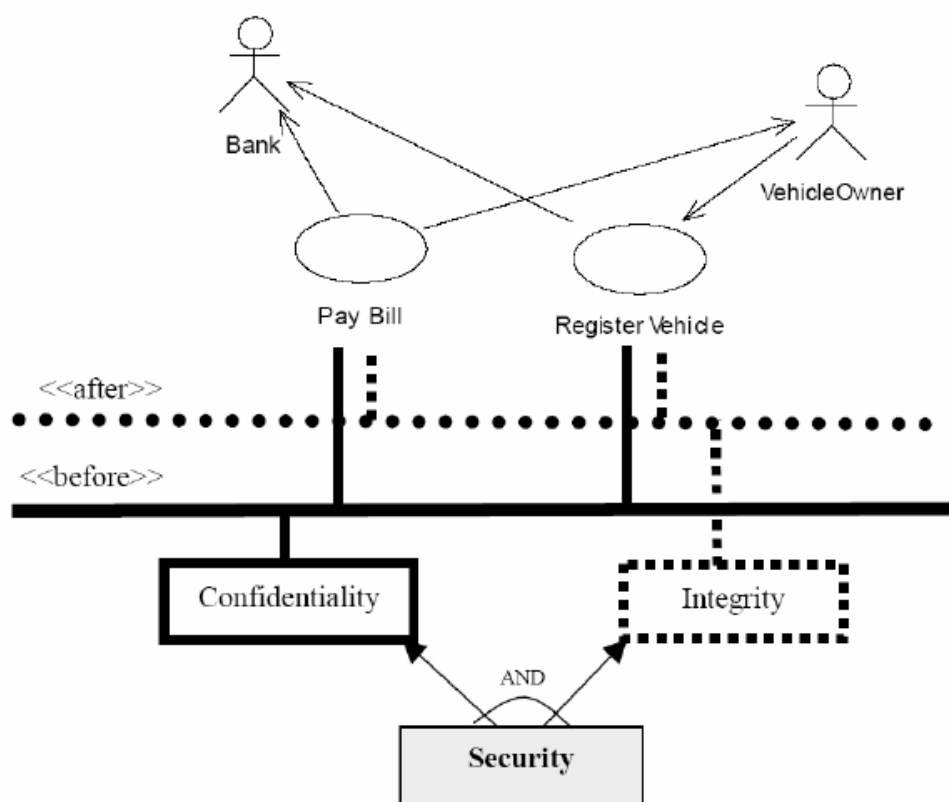
- هم‌پوشانی overlap
- بازنویسی override
- پوشاندگی wrapping

را برای نیازمندی‌های عملیاتی، توسط نیازمندی‌های غیرعملیاتی، معرفی می‌کند. همپوشانی یک نیازمندی غیرعملیاتی، به معنای آن است که: نیازمندی غیرعملیاتی، نیاز عملیاتی را توسط وارد شدن در قبل یا بعد از آن تغییر می‌دهد. بازنویسی به معنای آن است که رفتاری که صفت کیفی مشخص می‌سازد، جایگزین رفتار نیازمندی عملیاتی می‌شود. در نهایت، پوشاندگی به معنای پوشش نیازمندی عملیاتی توسط نیازمندی غیرعملیاتی است. دو شکل زیر مثال‌هایی از این نماد را مشخص می‌سازد.



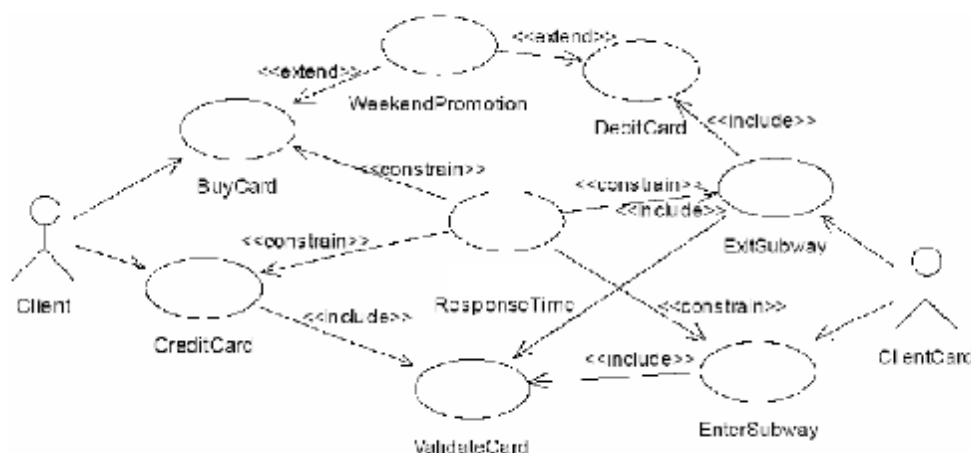
شکل 72- صفت کیفی زمان پاسخ، که نیازمندی‌های عملیاتی را پوشش می‌دهد

در شکل بالا، دیده می‌شود که چگونه صفت زمان پاسخ، عملیات خاصی از چهار کلاس بعدی را تحت پوشش قرار می‌دهد. این کلاس‌های سناریو را می‌توان در ردیف مکان الگوی صفت کیفی زمان پاسخ مشاهده نمود. نیازمندی‌های مربوط به این هم در ردیف نیازمندی‌ها مشخص می‌شود. در شکل زیر هم پوشانی پرداخت صورت حساب و ثبت نام وسیله را با نیازمندی غیرعملیاتی‌ای به نام امنیت، مشاهده می‌کنید. در این جا می‌توان از ورود محرمانه قبل از مورد کاربردها و یکپارچگی بعد از اجرای مورد کاربردها، انجام می‌شوند. بنابراین، خطوط پررنگ نمایانگر گزینه‌ی قبل از، و خطوط کمرنگ، نمایانگر گزینه‌ی «بعد از» هستند.



شکل 73- امنیت در بعضی از موردکاربردها، هم‌پوشانی دارند

در این متد، پیشنهاد شده است که علاوه بر روابط معمول بین موردکاربردها در یوامال (شمول، گسترش و ارث‌بری) از روابط جدیدی با نام‌های همکاری، خرابکاری و محدودیت استفاده نماییم. همکاری نمایانگر یک بده‌بستان مثبت بین دو مورد کاربرد است، خرابکاری نمایانگر بده‌بستان منفی است و محدودیت نمایانگر آن است که یک خصوصیت جهانی، دیگر موردکاربردها را محدودتر می‌سازد. یک نمونه از محدودیت را در شکل زیر مشاهده می‌کنید:



شکل 74- نمود صفت کیفی زمان پاسخ در دیاگرام مورد کاربرد، با استفاده از رابطه‌ی

محدودیت

یکی دیگر از مصنوعات جدیدی که در اینجا معرفی می‌شود، جزئیات الگوی مورد کاربرد (مورد کاربرد UCPS, Pattern Specification) است. این مستند، مورد کاربردهای نوعی را، که بعداً نمونه‌هایشان ساخته می‌شود، نمایش می‌دهد.

ایده‌ی جزئیات الگوی مورد کاربرد، همانند ایده‌ی جزئیات الگوی اتباطی در رهیافت پیشین است. نقش‌های مورد کاربرد، نمایانگر مباحث متغیر با زمان‌اند و با تغییر تنظیمات سیستم تغییر خواهند نمود. برای تشریح این مورد کاربردها در حالت نوعی، می‌توان نوع خاصی از دیاگرام فعالیت را استفاده نمود. خاص بودن این دیاگرام هم در شمول آن بر وجود نقش‌های مختلف و در تعریف آن است، و باعث می‌شود ما دارای مستندی به نام جزئیات الگوی فعالیت (APS)¹²⁹ گردیم. آی‌پی‌اس در زمان ترکیب مورد کاربردها با توجه به نقش‌ها و عملگرهای در نظر گرفته شده، نمونه‌سازی می‌شوند. یک مثال از این ترکیب را در شکل زیر مشاهده می‌کنیم:

¹²⁹ Activity Pattern Specification

```

Compose <use case A> with <use case B>
  <step #.> Replace |<modelElement A>
    with
      <modelElement B>
    []
    <modelElement N>

```

شکل 75- ترکیب مورد کاربرد A و مورد کاربرد B به همراه نمونه سازی از نقش A

5-6-3- فرایند رهیافت مورد کاربرد مبنای جنبه ای

مراحل این رهیافت را در شکل زیر مشاهده می کنید. همانند رهیافت مورد کاربرد کلاسیک، این رهیافت نیز با شناسایی نقش ها و مورد کاربردها آغاز می گردد. سپس برای جداسازی عملیات هایی که در چندین مورد کاربرد وجود دارند، این مورد کاربردهای شناسایی شده را بازبینی می کنیم. این عملیات های تقاطعی را به عنوان مورد کاربردهایی با روابط شمول یا گسترش با مورد کاربردهای اصلی نمایش می دهیم.

حال که مباحث عملیاتی را ماژوله نمودیم، فرایند را توسط تحلیل نیازمندی هایی که تاکنون جمع نموده و همچنین آنهایی را که از ذینفعان به طور مداوم جمع آوری می نماییم، برای شناسی نیازمندی های غیرعملیاتی، ادامه می دهیم. این نیازمندی های غیرعملیاتی را در الگوهایی، همانند شکل زیر نشان می دهیم:

Name	The name of the quality attribute
Description	Executive description
Focus	A quality attribute can affect the system (i.e. the end product) or the development process
Source	Source of information (e.g. stakeholders, documents)
Decomposition	Quality attributes can be decomposed into simpler ones. When all (sub) quality attributes are needed to achieve the quality attribute, we have an AND relationship. If not all the sub quality attributes are necessary to achieve the quality attribute, we have an OR relationship
Priority	Expresses the importance of the quality attribute for the stakeholders. A priority can be MAX, HIGH, LOW and MIN
Obligation	Can be optional or mandatory
Influence	Activities of the software process affected by the quality attribute
Where	List of the actors influenced by the quality attribute and also a list of models (e.g. use cases and sequence diagrams) requiring the quality attribute
Requirements	Requirements describing the quality attribute
Contribution	Represents how the quality attribute affects other quality attributes. This contribution can be positive (+) or negative (-)

شکل 76- جدول الگوی صفات کیفی

در مرحله‌ی بعدی، نیازمندی‌های غیرعملیاتی را در مدل مورد کاربردی که در مراحل قبلی توسعه داده بودیم، وارد می‌کنیم. در اینجا مورد کاربردهای غیرعملیاتی را در تقابل با مورد کاربردهای عملیاتی موجود، می‌توانیم رابطه‌دهی نماییم (این روابط می‌توانند: گسترش، شمول، اثرثبری، همکاری، خرابکاری و محدودیت باشند).

درنهایت، این فرآیند مورد کاربردهایی را که کاندیدای جنبه شدن هستند را خلاصه می‌کند. اگر مورد کاربردی با بیش از یک مورد کاربرد دیگر رابطه داشته باشد، آنگاه یک کاندید خوب برای جنبه را داریم.

5-6-4- شناسایی و بررسی مباحث تقاطعی با رهیافت مورد کاربرد مبنای جنبه‌ای

در این رهیافت، نیازمندی‌های عملیاتی را از طریق مورد کاربردها می‌شناسیم. سپس این موردکاربردها را برای رفتارهای تکراری و عملیات‌های تقاطعی تحلیل کرده و آنها را بوسیله‌ی شمول و گسترش جداسازی می‌کنیم. در اینجا، حتی یک عمل ساده هم اگر در بسیاری از موردکاربردها تکرار شود، می‌تواند بالقوه تقاطعی تلقی شود. بررسی نیازمندی‌های عملیاتی را می‌توان به صورت کلاسیک به همراه روابط جدید ادامه داد.

در مقابل، شناسایی نیازمندی‌های غیرعملیاتی، خیلی واضح نیست. در اینجا فقط گفته شده که با تحلیل و بررسی نیازمندی‌ها، آنها را شناسایی نماییم ولی بررسی آنها پس از شناسایی، با روش‌های خاصی انجام می‌پذیرد. (الگوها، صفات کیفی، در مدل مورد کاربرد و ...)

5-7- مقایسه‌ی بین رهیافت‌های مختلف

در اینجا، ما با استفاده از معیارهای ارائه شده در ابتدای گفتار و به صورت آوردن جداولی، بررسی می‌کنیم. هر رهیافت برای هر معیار ارائه شده، چه تمهیدی را پیش‌بینی نموده و اصولاً چگونه از آن پشتیبانی می‌کند.

• قابلیت ردیابی

• قابلیت ردیابی نیازمندی‌ها و تغییرات تا مبداء تولید

رهیافت	قابلیت‌های که ردیابی نیازمندی‌ها را تا مبدا تولیدشان میسر می‌سازد
پری‌ویو	قسمت‌های مبدا و تاریخچه‌ی تغییر در الگو؛ استفاده از تمرکز دیدگاه
مورد کاربرد	بازیگران دیاگرام مورد کاربردها
ضد مورد کاربرد	بازیگران دیاگرام مورد کاربردها به همراه ضد مورد کاربردها و راه حل‌شان
آرکید	دیدگاه نیازمندی‌ها
AOSD/UC	بازیگران در دیاگرام مورد کاربرد
SMA	بازیگران در دیاگرام مورد کاربرد؛ منبع، در الگوی نیازمندی‌های غیرعملیاتی
AUCDA	بازیگران در دیاگرام مورد کاربرد؛ منبع، در الگوی نیازمندی‌های غیرعملیاتی

شکل 77- جدول مقایسه‌ی قابلیت ردیابی نیازمندی‌ها و تغییرات، تا مبداء تولیدشان، بین

رهیافت‌های مختلف

• قابلیت ردیابی بین مصنوعات مختلف

رهیافت	قابلیت‌های که ردیابی نیازمندی‌ها را تا مبدا تولیدشان می‌سازد
پری‌ویو	قسمت‌های مبدا و تاریخچه‌ی تغییر در الگو؛ استفاده از تمرکز دیدگاه
مورد کاربرد	بازیگران دیاگرام موردکاربردها
ضد موردکاربرد	بازیگران دیاگرام مورد کاربرد‌ها به همراه ضد مورد کاربرد‌ها و راه حلشان
آرکید	دیدگاه نیازمندی‌ها
AOSD/UC	بازیگران در دیاگرام مورد کاربرد
SMA	بازیگران در دیاگرام موردکاربرد؛ منبع، درالگوی نیازمندی‌های غیرعملیاتی
AUCDA	بازیگران در دیاگرام موردکاربرد؛ منبع، در الگوی نیازمندی‌های غیرعملیاتی

شکل 78- جدول مقایسه‌ی قابلیت ردیابی بین مصنوعات مختلف

حال با توجه به مقایسات انجام شده و برحسب تحقیقاتی که در حال حاضر بیشتر مورد توجه قرار گرفته است، ما رهیافت «مدل کردن سناریوها بوسیله‌ی جنبه «SMA» را به عنوان متد برتر در حال حاضر انتخاب کرده و به توضیح جزئیات این روش به همراه یک مثال در طول بحث، می‌پردازیم. این روش در حال حاضر از طرف کسانی که در سال 98، UML و RUP را ارائه کرده‌اند، حمایت شده و پشتیبانی شرکت‌هایی همچون IBM و Rational را در اختیار دارد.

5-8- جزئیات روش SMA

در اینجا، سعی می‌کنیم که به تکرار مکررات نپرداخته و بیشتر بر روی نقاط گنگ این روش توجه نماییم. به هر حال توصیه می‌کنیم که پس از پایان این بخش، دوباره به مبحث SMA بازگشته، تا جریان کار را با دیدی بهتر ببینید.

5-8-1- مقدمه

این روش برای استفاده در سیستم‌های اثر متقابل - مبنا¹³⁰ مورد استفاده قرار می‌گیرد. اثر متقابل¹³¹، نمایانگر رفتار چندین مولفه‌ی سیستم در ارتباط با هم، برای رسیدن به یک هدف است. این اثرات متقابل برای مدل کردن نیازمندی‌های اولیه، بسیار مناسب است. این اثرات، نمایانگر مبادلات کلی در بین مولفه‌های سیستم، بدون توجه به جزئیات این مبادلات است. در این جا ما نشان می‌دهیم، چگونه می‌توان این اثرات را مدل نمود (با استفاده از جنبه‌ها)، به صورتی که بتوان آنها را به سرعت ارزیابی و صحت‌یابی نمود. این اثرات متقابل را بوسیله‌ی دیاگرام‌های ترتیب مدل می‌نماییم. اثرات جنبه‌ای، آن اثراتی هستند که اثرات معمولی را قطع می‌کنند. این اثرات جنبه‌ای را هم با استفاده از جزئیات الگوی اثرات مدل می‌کنیم. برای نمایش این تکنیک، از مثال «در دستگیری خطا» استفاده می‌کنیم. اثرات جنبه‌ای و غیر جنبه‌ای را بوسیله نمونه‌سازی از اثرات جنبه‌ای، با اثرات غیر جنبه‌ای ترکیب می‌کنیم. نتیجه‌ی مجموعه‌ی ارتباطات، به صورت اتوماتیک به مجموعه‌ای از دیاگرام‌های ماشین وضعیت یوامال تبدیل می‌شود. مثال ما در این بخش، یک سیستم پارکینگ ماشین، به همراه یک جنبه‌ی جدید است.

بخش‌های بعدی به ترتیب، مباحث زیر را توضیح می‌دهند: اثرات جنبه‌ای، کاربرد این رهیافت به همراه مثال، نشان دادن چگونگی تولید ماشین وضعیت، و در نهایت نتیجه‌گیری بحث.

¹³⁰ Interaction-Based

¹³¹ Interaction

5-8-2- اثرات متقابل جنبه‌ای

جزئیات الگو¹³²، یک روش برای رسمی کردن خصوصیات رفتاری و ساختاری یک الگوی تعریف شده است. ثبت جزئیات الگوها بر مبنای یوامال است. نحو تجریدی یوامال، بوسیله‌ی یک فرامدل¹³³، تعریف می‌شود. جزئیات الگوها، در اصل نوعی از این فرامدل‌ها هستند که المان‌های مدل، که در الگو با هم درگیرند را مشخص می‌سازند. هر المان در جزئیات الگو یک نقش¹³⁴، که در اصل، یک فراکلاس یوامال است، بوده و این فراکلاس دارای خصوصیات اضافی‌ای است که هر المان دیگری که بخواهد آن نقش را پر نماید، می‌بایست آن را دارا باشد. با توجه به این که هر نقش زیر مجموعه‌ای از نمونه‌های فراکلاس‌های یوامال را نمایش می‌دهد، هر جزئیات الگو می‌تواند با تخصیص المان‌های مدل یوامال به نقش‌های موجود در خود، نمونه سازی شود. یک مدل با یک جزئیات الگو مطابقت دارد، اگر و تنها اگر المان‌های مدل که نقش‌های جزئیات الگو را پر کرده‌اند، خصوصیات آن نقش را ارضاء نماید.

از این جزئیات الگوها در ساختار الگوها (بوسیله‌ی جزئیات الگوی استاتیک SPS)، اثرات متقابل (بوسیله‌ی جزئیات الگوی اثر متقابل، IPS) و رفتارهای حالت-مبنا (بوسیله‌ی جزئیات الگوی ماشین وضعیت SMPS) می‌توان استفاده نمود. ما در این روش از جزئیات الگوی اثر متقابل‌ها استفاده می‌کنیم. هر جزئیات الگوی اثر متقابل، یک الگوی اثرگذاری را بین اجزایش مشخص می‌کند. هر جزئیات الگوی اثر متقابل شامل نقش‌های خط‌عمر و یک نقش دسته‌بندی کننده¹³⁵ مشخص می‌شود، که نوع خاصی از دسته‌بندی کننده‌ی یوامال است. هر جزئیات الگوی اثر متقابل، در اصل حالت رسمی یک الگوی مشاهده‌گر¹³⁶ است. نام نقش را بوسیله‌ی یک خط عمودی پایان می‌دهیم. به این وسیله نشان می‌دهیم که آنها نقش‌اند. هر دیاگرام ترتیبی که با جزئیات الگوی اثر متقابل مطابقت دارد، می‌بایست تمام نقش‌های موجود در جزئیات الگوی اثر متقابل را ارضاء کند.

¹³² pattern specifications

¹³³ Meta Model

¹³⁴ Role

¹³⁵ Classifier role

¹³⁶ observer pattern

توجه داشته باشید که یک دیاگرام ترتیب مطابقت دار با یک جزئیات الگوی اثر متقابل، می‌تواند المان‌هایی افزون بر المان‌های جزئیات الگوی اثر متقابل را دارا باشد، فقط تنها شرط، ارضاء نمودن از لحاظ تعدد و دیگر محدودیت‌های نقش است.

در این جا ما هر جنبه را به صورت یک الگو نمایش می‌دهیم. در این حالت اثرات متقابل جنبه‌ای را بوسیله‌ی جزئیات الگوی اثر متقابل‌ها مدل‌سازی می‌کنیم. هر جزئیات الگوی اثر متقابل، علاوه بر نقش‌ها می‌تواند دارای اجزای غیر نقشی و نهایی باشد. به عنوان مثال در اثر متقابل امنیت، ما می‌خواهیم که هر کاربر جدید را با رمز عبورش مطابقت دهیم. این کار در تمامی مکان‌ها، به یک صورت انجام می‌پذیرد، پس ما یک کلاس و فراخوانی متد خاص را برایش فراهم می‌کنیم تا دیگر نیازی به نمونه‌سازی در آن قسمت، نداشته باشیم. در این حالت که ما علاوه بر نقش‌ها، دارای عناصر غیر نقشی و نهایی شده هستیم، هر دیاگرام ترتیب مطابقت دار با یک جزئیات الگوی اثر متقابل، می‌بایست فقط با نقش‌ها، مطابقت داشته باشد، یعنی نیازی نیست که در آن دیاگرام دوباره قسمت‌های نهایی شده را قرار دهیم. برای توضیح این بخش، یک سیستم پاسخ‌گوی خودکار در بانکها را در نظر می‌گیریم که دارای یک جنبه‌ی امنیت است. این سیستم رمز عبور را چک می‌کند. یک دیاگرام ترتیب مطابقت دار با این الگو، می‌تواند برداشت از حساب یک کاربر باشد. این جزئیات الگوی اثر متقابل، می‌تواند دارای نقش‌هایی برای کاربر و سیستم پاسخگویی خودکار و یک المان غیر نقشی، برای چک کردن رمز عبور باشد. دیاگرام ترتیب مطابقت‌دار می‌تواند دارای یک‌سری کاربرد سیستم پاسخگویی مشخص و نهایی بوده، که در جاهای مختلف می‌تواند از جنس‌های مختلف باشد، ولی نیازی نیست که دوباره شامل چک کردن رمز عبور گردد، زیرا آن را در جنبه، مجزا نموده‌ایم.

ضمیمه 1: تعاریف

- **نیازهای غیر عملیاتی:** این دسته از نیازها در اصل نیازهایی هستند که برای اجرای صحیح سیستم لازم می باشند ولی جزء عملیات های هسته ای سیستم نیستند. از جمله ی این نیازها، Logging است. Logging در سیستم ها به معنای ذخیره سازی ی اطلاعات اجرای عملیات ها و یا اصولا هر چیزی که برای دنبال کردن رفتار سیستم نیاز است، میباشد. سیستم ها می توانند log های خود را بر روی فایل ذخیره سازند و یا روی مانیتور نمایش دهند و یا حتی بر روی دیتا بیس بریزند. همان طور که دیده شد logging عملیات اصلی یک سیستم مانند تولید سند حسابداری نیست. ولی عملیاتی است که برای بررسی صحت و در ادامه برای نگهداری از سیستم بسیار لازم است. به عنوان مثال دیگر، می توان به اعمال امنیتی اشاره نمود. مانند سطوح دسترسی و غیره.
- **نگرانی های قطع کننده:** اینها در اصل همان نیازهای غیر عملیاتی می باشند. ولی با این نام به شیوه ای جدید به آنها نگریسته می شود. در اصل تمامی نیازهای غیر عملیاتی را می توان در اقصی نقاط سیستم مشاهده نمود که بصورتی کاملا غیر تفکیکی با آنها ادغام شده اند. بنابراین این نیازها در اصل کل سیستم را به صورت کامل قطع نموده و با هر بخشی به صورتی تقاطع ایجاد می نماید. به این ترتیب مشاهده می شود که در یک سیستم، نیازهایی مانند امنیت با تمامی اعمال، مانند ایجاد سند حسابداری، تغییر سند حسابداری و ... درگیر شود یا برای مثالی واضح تر logging میتواند قبل و بعد و در حین اجرای هر متدی استفاده شود. بنابراین مشاهده می شود که نگرانیهای قطع کننده در اصل دید بهتری را از نیازهای غیر عملیاتی به اشخاص مختلف می دهند.
- **نقطه تقاطع:** همان گونه که گفته شد نیازهای غیر عملیاتی میتوانند در مکانهای مختلف، سیستم را قطع نمایند. در اینجا به هر نقطه ای که در آن یک نیاز غیر عملیاتی، یک نیاز عملیاتی را قطع می کند به عنوان یک نقطه ی تقاطع شناخته می شود. شناسایی این نقاط

می تواند بسیار مشکل باشد. در اصل نقطه ی تقاطع قسمتی است که در آن یک متد فراخوانی می شود.

- **پیشنهاد:** هر پیشنهاد به همراه یک نقطه ی تقاطع معنا پیدا می کند و در اصل مشخص می سازد که نیازهای غیر عملیاتی قبل، بعد و یا در حین وقوع یک نقطه ی تقاطع دارای معنی است (به معنای دیگر اعمال مربوط به این نیاز در کدام حالت ذکر شده میبایست به وقوع بپیوندد).

- **جنبه:** در زبانهای برنامه سازی جنبه گرا هر نیاز غیر عملیاتی را می توان در مفهوم جدیدی به نام جنبه نهان سازی نمود. هر جنبه دارای یکسری نقاط تقاطع و همچنین پیشنهاداتی برای آن نقاط تقاطع است. در این حالت می توان هر جنبه را با کلاسها در زبانهای شیءگرا مقایسه نمود با این تفاوت که کلاسها داده ها و عملیاتها را نهان سازی می کنند ولی جنبه ها در اصل نیازهای غیرعملیاتی را به همراه نقاط تقاطع و پیشنهادات مربوط به هریک را نهان سازی میکنند.

- **دوختن:** دوختن در اصل فرایندی است که طی آن به ازای هر جنبه، به نقاط تقاطع و پیشنهادات آن توجه شده و عملیات درون جنبه را، به درون سیستم اصلی وارد می کنند. پس از دوختن است که سیستم به همراه جنبه ها شروع به عمل مینماید. این فرایند میتواند در زمان کامپایل رخ دهد (مانند AspectJ) یا در زمان اجراء رخ دهد (مانند AspectWerkz). هر دوی این دیدگاهها برای اعمال نیازها به سیستم اصلی دارای طرفداران خود است ولی به علت اینکه مباحث آن مربوط به تخصص کامپیوتر است از آوردن آنها در اینجا صرف نظر میکنیم. در اصل ما در اینجا به مسایل مربوط به شناسای جنبه ها و همچنین نقاط تقاطع و پیشنهادات بیشتر علاقه داریم.

روش جنبه گرا در توسعه ی سیستمها

با گسترش سیستمهای اطلاعاتی، هرروزه نه تنها این سیستمها بزرگتر می شوند بلکه مسایلی مانند قابلیت اطمینان و سرعت و راحتی استفاده که با نام فاکتورهای کیفی شناخته میشوند نیز به آنها افزوده می گردند. این فاکتورهای کیفی به صورت تقاطعی ساختار اصلی برنامه را قطع مینمایند. علاوه برآن فاکتورهای کیفی یکسری از عملیتهای سیستم، همانند توزیع شدگی، همزمانی و غیره هستند که به عنوان بخشی از سیستم، کل ساختار دیگر قسمتها را قطع می کنند. تمامی این مسائل تولید و توسعه ی سیستمها را بسیار مشکل ساز می سازد.

در پاسخ به نیاز روز افزون به تولید نرم افزار، دیسپلینهای مهندسی نرم افزار برای ارایه ی یکسری فرآیندهای ساخت یافته برای تولید آنها بوجود آمده است. فازهای اولیه ی تمامی این فرآیندها (البته غالباً) عبارتند از: مهندسی نیازمندیها، معماری و طراحی.

تکنیکهای تولید و توسعه ی جنبه گرا یک وسیله ی سیستماتیک برای شناسایی، مقیاس پذیری، نشان دادن و ترکیب این نگرانیهای متقاطع هستند. این تکنیکها بر روی روش شیء گرا سوار می شوند و خط بطلانی برای آنها به حساب نمی آیند. (همانگونه که روش شیء گرا بر روی روش ساخت یافته سوار شد. چون در انتها هنوز متدها در داخل کلاسها موجودند، تنها شیوه ی فراخوانی آنها تفاوت نموده است).

بیشترین کار بر روی تکنیکهای تولید و توسعه ی جنبه گرا، برای حمله به چنان نگرانیهایی بوجود آمدند، در ابتدا نیز بیشترین کار بر روی ساخت زبانهای برنامه سازی جنبه گرا معطوف بود. سپس فعالیتهایی بر روی جمع آوری نیازها و طراحی از دیدگاه جنبه ای بوجود آمد.

معیارهای مقایسه ی عمومی

از آنجایی که یک روش کاملاً مشخص برای تحلیل و طراحی سیستمها از دیدگاه جنبه گرا وجود ندارد، می بایست بین روشهای پیشنهادی موجود مقایساتی را انجام بدهیم. این مقایسات می تواند با توجه به معیارهایی انجام گیرد که در طی بهترین تجربیات گذشته در صنعت نرم افزار بوجود آمده

است. این معیارهای مقایسه در میان فازهای تحلیل و معماری و طراحی می‌تواند تغییر نماید و هر یک معیارهای مربوط به خود را داشته باشند ولی به صورت کلی یکسری از معیارها بین تمامی این فازها یکسان است، که ذیل آنها را بیان می‌داریم:

- قابلیت ردیابی در طول چرخه عمر نرم افزار: قابلیت ردیابی بین تولیدات چرخه‌ی نرم افزار، به عنوان یکی از فاکتورهای کیفیت نرم افزار مطرح می‌گردد. ردیابی این امکان را فراهم می‌آورد که ما بتوانیم محصول هر فاز را به همراه ارتباط آن با محصول پیشین و پسین و دلایل اتخاذ تصمیمات، را شناسایی نماییم. بنابراین چنانچه ما یک محصول (مانند یک مستند تحلیل) را شناسایی نمودیم، می‌توانیم آن را تا مرحله‌ی تولید ردیابی نموده و تغییرات هر مرحله را به همراه دلایل آنها بررسی نماییم. با توجه به آنچه گفته شد، ردیابی دارای دو وجه است:

0 ردیابی محصولات از زمان ساخت تا تغییرات آن

0 ردیابی بین محصولات چرخه‌ی عمر

اولین زیرمجموعه بیانگر این است که یک محصول یا تغییرات روی آن، از کجا ناشی می‌شود و دومین زیرمجموعه کمک می‌کند تا تبدیلات یک محصول را در طول فازهای یک پروژه بررسی نماییم.

- قابلیت ترکیب شدن: به این اشاره دارد که ما بتوانیم محصولات را کنار هم قرار دهیم و در نتیجه بتوانیم یک دید کلی از سیستم مورد نظر بدست آوریم. جزئیات ترکیب‌بندی به این می‌پردازد که چگونه مستندات تحلیل و طراحی در کنار هم قرار گیرند تا دید کلی سیستم را بدست دهند. در حقیقت قابلیت ترکیب شدن در نقطه‌ی مقابل تجزیه کردن سیستم قرار دارد. اگرچه تجزیه کردن یک سیستم از دید برنامه‌نویسان، برای تولید راحت‌تر سیستم، بسیار اساسی است، ولی این تجزیه تنها در حالتی مفید است که بتوان دوباره آنها را ترکیب نمود و سیستم را بدست

آورد. در مستند، معیار ترکیب شدن به معنای وجود معناشناسی کافی، (برای مثال، در تحلیل‌های دیدگاهی، ترکیب دیدگاه‌های مدیران و منشیان به چه معنایی است؟) همراه با پشتیبانی قواعد (چه عملگرهای ترکیبی‌ای را استفاده نماییم و خروجی یک ترکیب به چه معنایی است) است. چون متدهای جنبه‌گرا دارای اتمهای جدید تجزیه پذیر هستند -همان جنبه‌ها- می‌بایست مکانیزم‌های جدیدی برای ترکیب ارایه داده شود یا اینکه نشان داده شود که از تکنیک‌های موجود به چه صورتی استفاده نمود.

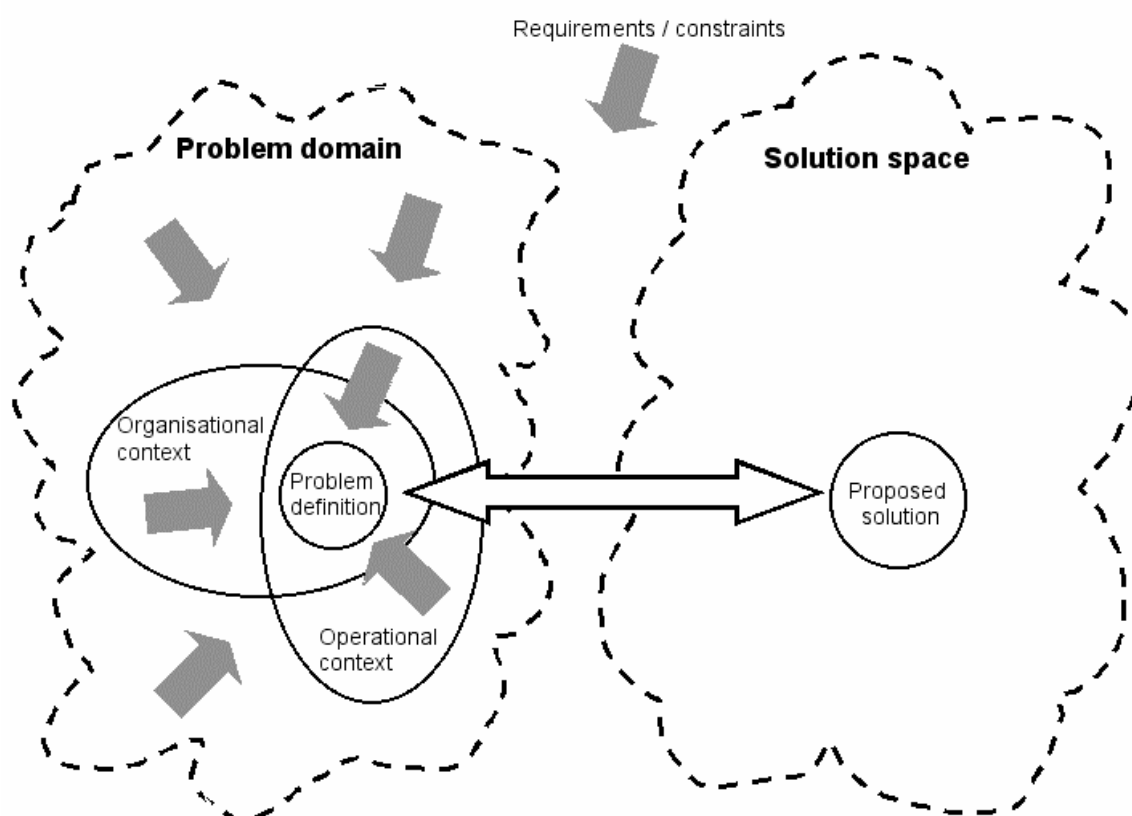
- قابلیت نمو کردن: این معیار برای ایجاد تغییر در محصولات است. این تغییر برای محصولات نیازمندی/معماری/طراحی موجود یا حذف/اضافه محصولات جدید است. این قابلیت تغییر، تضمین می‌کند که کلیت محصولات هر فاز در حین تغییر از دیدگاه مفهوم کلی و ساختار ثابت مانده ولی تغییرات را در جزییات امکان پذیر می‌سازد. اگر این ساز و کار در سیستم تولید اعمال نگردد، تغییرات در فازهای نهایی می‌تواند باعث از هم‌گسیختگی و ناهمگونی محصولات فازهای مختلف می‌شود.

- قابلیت مقیاس‌پذیری: این معیار بسیار مهم است زیرا یک رهیافت صحیح می‌بایست بتواند که برای پروژه‌های کوچک و بزرگ، هماهنگ شود. زیرا اکثر پروژه‌ها بصورت کوچک آغاز میشوند و سپس درطول زمان کم کم بزرگ میگردند. در بعضی موارد معیار مقیاس‌پذیری را با استفاده از ابزارها میتوان کنترل نمود، ولی مسایلی وجود دارد که نمیتوان آنها را با ابزار مقیاس‌پذیری حل نمود.

مهندسی نیازمندی‌ها

هر نیازمندی، دراصل از یک مشکل تجاری مشتق می‌شود. برای مثال بررسی‌ی مدارک پاسپورت یا اضافه نمودن امکانات به گوشی‌های موبایل. پروژه‌ها می‌توانند برای ایجاد محصولات جدید، تعریف

شوند، یا اینکه برای رشد محصولات موجود، ولی قدیمی انجام گردند. در تمام حالات، هر پروژه در محیطی نصب می‌گردد که دارای واسطه‌هایی با کاربران، المانهای فرآیند یا دیگر سیستم‌های نرم‌افزاری و سخت‌افزاری داشته باشند. همانند شکل زیر:

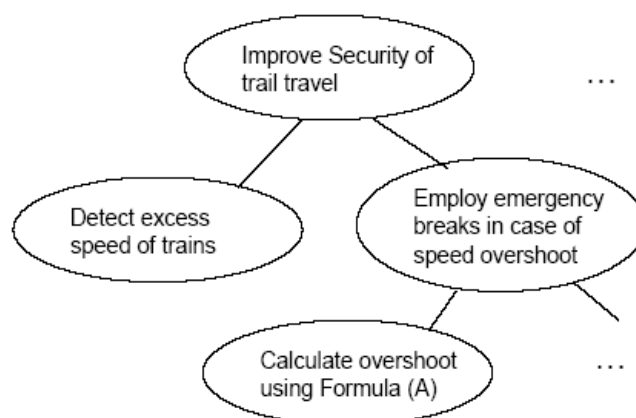


شکل 79- محیط مهندسی نیازمندی‌ها

نیازمندی، یک صفت خاص یا قابلیت را بیان می‌کند که سیستم می‌بایست برآورده ساخته، تا بتواند مشکل تجاری‌ای را حل نماید. راه حل کلاسیک برای شناسایی یک نیازمندی این بود که آنها را به دو گروه عملیاتی و غیر عملیاتی تقسیم بندی نمایند. نیازهای عملیاتی نیازهایی هستند که نرم افزار می‌بایست برآورده سازد تا بتوان گفت که صحیح عمل می‌کند. در مقابل نیازهای غیرعملیاتی بیانگر کیفیت نرم افزارند.

نیازمندی‌ها را می‌توان در طیف‌های متفاوتی از دیدگاه‌های تجاری تا دیدگاه‌های فنی بررسی نمود. دیدگاه‌های فنی در بررسی نیازمندی‌ها نمی‌بایست به منظور حکمی قطعی تلقی شوند، بلکه تنها

باید به صورت یک ابزار کمکی برای بیان نیازها در طیف تجاری استفاده شوند. برای راحتی ما به این طیف‌های متفاوت دیدگاه‌ها، به صورت لایه‌های مختلفی نگاه می‌کنیم. بالاترین لایه در اصل یک هدف از سیستم را از دیدگاه استراتژیک ارائه می‌دهد. در شکل زیر هدف سطح بالایی را برای یک شرکت ریلی در راستای ارتقاء امنیت مسافرت ریلی مشاهده می‌کنید:



شکل 80- سطوح نیازمندی‌ها برای سیستم امنیت ریلی

سطح بعدی در بررسی نیازمندی‌ها، به عنوان نیازمندی‌های کاربر شناخته می‌شود و به عنوان کارهایی است که توسط کاربران، سازمان و یا دیگر سیستم‌ها، از سیستم تولیدی انتظار می‌رود. برای مثال در سطح دوم شکل بالا نشان داده شده است که هرگاه سرعت قطار در مسیری خاص، از حد مجاز بیشتر گردید، می‌بایست ترمزهای اضطراری عمل نمایند. یکی از مهمترین وظایف یک تحلیلگر، شناسایی این نیازهای کاربری است. این کار شامل انشعابات بیشتر در سیستم است تا درک صحیحی از کارهایی که سیستم باید انجام دهد به دست آورده شود. محدودیت‌ها، برای تعیین حدود در راه‌حل‌های ممکن بکاربرده می‌شوند. این محدودیت‌ها می‌توانند تکنیکی بوده، (مانند پهنای باند در دسترس) یا مربوط به حوزه‌ی مشکل باشند. (مانند قوانین و استانداردها).

فرآیند تبدیل یک نیاز به نرم افزار، بسیار پیچیده است و هرچه فرآیند پیش می رود هزینه اعمال خطاهای موجود در تحلیل، به صورت فزاینده ای افزایش می یابد. برای همین هر فرایند تحلیلی که این خطاها را بتواند زودتر شناسایی نموده و یا تغییرات را بر روی نرم افزار ساده تر نماید، از درجه ی مقبولیت بیشتری برخوردار است.

هرچه سیستم دریافت نیازمندی، امکان تخصیص تنها یک نیاز را به یک مولفه نرم افزاری بدهد، سیستم بهتر تجزیه شده و می توان یک بخش از سیستم را موظف به انجام یک کار دانست. در این حالت می توان برای هر قسمت، یک واسط برنامه نویسی تهیه نمود، که معین کننده ی نقاط ارتباط با دنیای خارج باشد.

ولی دسته های بسیاری از نیازمندی ها وجود دارد که قرار دادن آنها در قسمت های مجزا، با روش های کلاسیک برنامه نویسی امکان پذیر نیست. بسیاری از نیازمندی های غیر عملیاتی در این دسته قرار می گیرند. برای مثال، کارایی، یکی از ملاکهای معماری و محیط اجرای برنامه است و هیچ کس نمی تواند این را به عنوان یک مولفه ی مجزا از ساختارهای دیگر برنامه، بررسی نماید. این مشکل که در سیستمها نیازمندی هایی وجود دارد که نمی توان آنها را به عنوان یک بخش مجزا در سیستم بررسی نمود، باعث شده است تا جریانی برای ساخت تحلیل نیازمندی های جنبه گرا بوجود بیاید.

برای بررسی نیازمندی ها هم اکنون مدل های مختلفی وجود دارد که از میان آنها، می توان به عنوان اصلی ترین ها، به مدلهای زیر اشاره کرد:

- دیدگاه ها
- موارد کاربردی
- اهداف

اگرچه هریک از این رهیافت ها، شیوه ی مناسبی برای شناسایی و تحلیل بسیاری از نیازمندی ها فراهم می آورد، ولی به صورت مجزا برای نگرانی های تقاطعی، اعمال خاصی را تعریف ننموده اند.

بنابراین رهیافت جنبه گرا سعی در ایجاد ابزارهایی برای هر یک از مدل‌های گفته شده دارد، تا بهتر بتواند به بررسی نگرانی‌های تقاطعی بپردازد.

هدف تحلیل نیازمندی‌ها نه تنها به منظور جداسازی نیازهای غیرعملیاتی و تقاطعی است، بلکه وسیله‌ای را فراهم می‌آورد که طی آن، بتوان بهتر تضادهای موجود، به علت درهم‌پیچیدگی را در بین آنها شناسایی نموده و نقطه‌ی تعادلی را برای آن در نظر بگیریم.

معیارهای خاص تحلیل نیازمندی‌ها

برای اینکه بتوانیم معیارهای اختصاصی مقایسه‌ی بین مدل‌های تحلیلی مختلف را بدست بیاوریم، به دوره‌ی عمر نیازمندی‌ها و کل محصول نرم افزاری نظری می‌افکنیم.

دوره‌ی عمر نیازمندی‌ها

در بخشهای قبلی اشاره کردیم که در یک تحلیل نیازمندی‌ها می‌بایست ویژگی‌ها و عملیات‌های یک سیستم را بدست آوریم. این جزییات، تحت عنوان نیازمندی‌ها، می‌بایست چندین خصوصیت را دارا باشند.

1. صحت

2. کامل بودن

3. امکان پذیری

رسیدن به این نقاط کیفی، مستلزم دستیابی به یک نقطه‌ی تعادل، از طریق مذاکره با کاربران و دیگر افراد ذینفع در پروژه است. این نیازمندی‌ها، می‌بایست به صورت صحیحی مستند شده و میان افرادی که قرار است آن را پیاده‌سازی کنند، یکی از مسائل ارتباطی باشد. این مستندات قبل از رسیدن به دست پیاده‌سازان می‌بایست، مورد تست صحت قرار گیرد. این تست صحت، بررسی می‌کند، آیا این نیازمندی‌ها، همان چیزی است که کاربر در نظر دارد یا خیر. در حین تولید، این مستندات، باید مدیریت شوند تا تغییرات را تحت کنترل درآوریم. (البته این موردی است که هم

اکنون بسیار جای بحث دارد. طرفداران متدولوژی‌هایی مانند RUP، از این نظریه دفاع می‌کنند، ولی طرفداران متدولوژی‌های سبک‌وزن، مانند متدولوژی‌های چابک بسیار با طرز تفکری که منجر به کنترل تغییر می‌شود مخالفند.)

عقیده‌ی مرسوم در بین طرفداران متدولوژی‌های سنگین وزن به این منوال است که نیازمندی‌های سیستم باید به یک نقطه‌ی مناسب از لحاظ تکامل برسند تا فازهای بعدی قابل اجرا باشد. این اتفاق برای این است که از تغییرات احتمالی که در آینده ممکن است اتفاق بیافتد جلوگیری شود و ریسک یک چنین تغییرات پایین بیاید. در حالیکه در بسیاری از حوزه‌ها، شناسایی کامل نیازمندی‌ها، امکان‌پذیر نیست و حتی اگر قابل شناسایی باشند، احتمال تغییر، اضافه و یا کم شدن نیازمندی‌های جدید، بسیار زیاد است. اگر محیط یک محصول متغیر باشد، نیازمندی‌های آن محصول نیز متغیر خواهند بود. بنابراین از عهده‌ی تغییرات برآمدن، یکی از مهمترین ملزومات متدولوژی‌های نیازمندیست.

حتی اگر فرایند را به صورت کامل بهینه نمایید، مقداری تغییر، بعد از طراحی و حتی پیاده‌سازی سیستم، مورد نیاز خواهد بود.

همانطور که گفته شد، نیازمندی‌های عملیاتی، توسط متدهای کلاسیک به خوبی شناسایی می‌شوند و مشکل تنها برسر نیازمندی‌های غیرعملیاتی است. اما هر مدلی که برای شناسایی این نیازمندی‌ها ارائه شود، لااقل می‌بایست به خوبی روش‌های گذشته باشد تا بتواند مورد قبول واقع شود. بنابراین، ما اولین معیار را به صورت زیر ارایه می‌دهیم:

• شناسایی و رسیدگی نیازهای عملیاتی و غیرعملیاتی به صورت همزمان؛ که خود این

بخش سه مبحث زیر را برای ما باز می‌کند:

0 هر رهیافت می‌بایست بتواند که نیازهای غیرعملیاتی را شناسایی کند و از این

نظر می‌بایست لااقل همانند متدهای قدیمی عمل نماید

0 رهیافت پیشنهادی می‌بایست که نیازهای عملیاتی و غیزعملیاتی‌ای، که به صورت تقاطعی عمل می‌نمایند را شناسایی نموده و آنها را تحت لوای جنبه‌ها از باقی سیستم مجزا سازد.

0 این رهیافت‌ها می‌بایست هم شناسایی وهم رسیدگی را پشتیبانی نمایند. زیرا نیازمندی‌ها به صورت آماده موجود نیستند و پس از شناسایی نیز نمی‌توان آنها را بدون هیچ بررسی‌ای رها نمود.

البته شناسایی نیازمندی‌ها، دارای اختلافات مختصری به شرح زیر هستند:

- مهندس نیازمندی‌ها می‌بایست که بداند ذینفعان پروژه به چه چیزی احتیاج دارند. این مورد نمی‌تواند اتفاق بیافتد مگر اینکه طراحان و معماران سیستم نیز در فرایند شناسایی نیازمندی‌ها، شرکت کنند. بنابراین، شناسایی به همراه مباحث کنترلی مطرح می‌شود، آنها نیز از طریق معیارهای تصدیق و صحت سنجی قابل دستیابی است.
 - درهنگام کار برروی شناخت نیازمندی‌ها، نیازمند آنیم که با مسائلی مانند منبع نیازمندی‌ها، تاثیرات دوطرفه و تضادهای بین نیازها، اهمیت آنها برای ذینفعان و چگونگی همکاری بین آنها در سیستم نهایی پردازیم. از بین این مسائل، منبع نیازمندی‌ها را می‌توان به وسیله‌ی معیار ردیابی جواب داد. در مورد تضاد، قابلیت ترکیب شدن به ما اطلاعات می‌دهد. در اینجا ما از معیار نقطه‌ی تعادل و پشتیبانی تصمیم برای حل تعارضات شناخته شده در مرحله‌ی ترکیب استفاده می‌کنیم.
- بنابر آنچه گفته شد، معیار صحت و اعتبارسنجی، به شناسایی و کنترل مرتبط می‌شود. این معیارها در دو حوزه‌ی خاص عمل می‌کنند:
- باید جزییات نیازمندی‌ها، قابل اعتبارسنجی توسط ذینفعان پروژه باشد.

- محصولات فازهای معماری و طراحی می‌بایست از طریق مستندات نیازمندی، قابل صحت‌سنجی باشند، به این ترتیب می‌توان اطمینان داشت که کل سیستم توسط تمامی افراد درگیر شناخته شده است.

- شناسایی نقطه‌ی تعادل و پشتیبانی تصمیم، برای حل تعارضات در نیازمندی‌ها بوجود می‌آید. تضاد در بین نیازمندی‌های تقاطعی و غیر تقاطعی غیرقابل جلوگیری‌ست. این معیار با این سروکار دارد که بتواند این تعارضات را رفع نماید. در این معیار برای مذاکره بر سر نیازمندی‌ها، باید امکاناتی را دید و تصمیمات نقطه‌ی تعادل می‌بایست ثبت شده و با دیگر افراد درگیر در تولید درمیان گذاشته شود.

مباحث ثبت و مذاکره بر سر نیازمندی‌ها را می‌توان با معیار ردیابی نیز بررسی نمود. در اصل معیار ردیابی به ثبت و مذاکره برسر این مورد می‌پردازد که یک مستند از کجا آمده و چگونه می‌توان تبدیل یافته‌ی آن را در فازهای بعدی مشاهده نمود. اما معیار ردیابی توضیح نمی‌دهد که یک مستند تحلیل چرا و با چه تصمیماتی به یکسری تصمیمات معماری و طراحی تبدیل گردیده است. این مورد را می‌توان با معیار زیر ارزیابی نمود:

- نگاشت: پشتیبانی از نگاشت نیازها به مراحل بعدی تولید تا حدودی برای نیازمندی‌های تقاطعی معنا پیدا می‌کند، زیرا تمامی این نیازمندی‌های تقاطعی به سمت مستندات معمولی طراحی و معماری نمی‌روند؛ بعضی از آنها تنها به تصمیمات و یا نقاط تعادلی ختم می‌شوند. به همین علت ما نیاز داریم تا بدانیم که هر نیازمندی چگونه به ایجاد تصمیمات می‌انجامد و آیا ابزارهای مناسبی برای یک چنین نگاشتی فراهم می‌کند یا خیر.

مباحث مربوط به شناخت و ذخیره و ایجاد ارتباط توسط شیوه‌ی برخورد با نیازمندی‌های متفاوت (مانند نیازمندی‌های عملیاتی، غیرعملیاتی و تقاطعی)، تحت تاثیر قرار می‌گیرد. اگر یک شیوه‌ی مشخص برای تعامل با این نیازمندی‌های مختلف وجود داشته باشد، هر سه مورد ذکر شده در ابتدا

درست انجام می‌پذیرد. در نقطه‌ی مقابل، وجود تفاوت در برخورد با نیازمندی‌هاست. برای همین، معیار زیر را ارائه داده‌ایم:

- یکسانی برخورد با نیازمندی‌ها: به این می‌پردازد که آیا تمامی موارد به یک شیوه برخورد می‌شود یا نه. این معیار به این معنا نیست که تمامی انواع نیازمندی‌ها دقیقاً یکسان برخورد شود، ولی این هم باعث نمی‌شود که سازماندهی نیازمندی‌ها به صورت‌های متفاوتی انجام گیرد. بلکه می‌بایست تا حدود زیادی مشابهت در برخورد با نیازها وجود داشته باشد به این ترتیب تعداد زیادی مجموعه‌های از هم مجزا در تحلیل نیازمندی‌ها بوجود نخواهد آمد.

نتیجه‌ی معیارها:

در آخر برای روشن‌تر شدن مطلب کلیه‌ی معیارهای نیازمندی‌ها را به صورت زیر می‌آوریم:

1. شناسایی و بررسی تمامی نیازهای تقاطعی و غیر تقاطعی
2. شناسایی نقطه‌ی تعادل و پشتیبانی تصمیم
3. نگاشت نیازها، به مراحل بعدی توسعه
4. صحت و اعتبارسنجی
5. یکسانی در برخورد با نیازمندی‌ها

بخش 6- روش‌های چابک در توسعه نرم افزار

6-1- مقدمه ای بر متدلوژی چابک توسعه نرم افزار:

(Nandhakumar and Avison 1999) در بررسی متدلوژی های سیستم های اطلاعاتی اولیه بیان می دارند که این روشها برای ارائه تصاویر و مدلهای بصری به منظور کنترل سیستم های در حال توسعه به کار می رفتند. پیش تر نیز در این راستا مطالعات دیگری صورت پذیرفته بود که عمده آنها بیانگر این مطلب می باشند که روشهای سنتی توسعه سیستم های اطلاعاتی در عمل بسیار قاعده مند و غیر منعطف می باشند. از جمله این تحقیقات Parnas and Clement 1986 را می توان نام برد که در این زمینه مشکلات روشهای توسعه سنتی را بررسی کردند. این دست مشکلات توسعه دهندگان سیستم های اطلاعاتی را بر آن داشت که روشهایی منعطف در برابر تغییر نیازمندیها و با توانایی توسعه سریع سیستم های اطلاعاتی را ابداع نمایند. مجموعه این روشها را روشهای چابک توسعه نرم افزار می نامند.

روش XP - یکی از روشهای چابک توسعه نرم افزار - به عنوان نقطه شروع روشهای چابک به شمار می رود. پس از توسعه این روش و با تاثیر پذیری از آن، روشهای دیگر از قبیل Feature Driven Development، Crystal، Adaptive software Development، و غیره توسعه یافتند که مجموعه این روشها را روشهای چابک توسعه نرم افزار می نامند.

از انجائیکه میزان بازگشت سرمایه واقعی به سبب استفاده از فرآیند به کار گرفته شده در سازمان های توسعه دهنده نرم افزار مشخص نمی باشند بالتبع از تاثیر گذاری استفاده از روشهای چابک - به عنوان فرآیندهای توسعه در سازمان ها شناخت اندکی وجود دارد. این نکته که یک روش مشخص برای همه سازمانها نتایج مشابهی را به همراه ندارد نیز تاثیر به سزائی بر عدم شناخت میزان بازگشت سرمایه با استفاده از روشهای چابک دارد.

در این گزارش بر آنیم که با بررسی سیستماتیک متدلوژی های چابک توسعه نرم افزار چهارچوبی برای انتخاب مناسب ترین روش ارائه دهیم.

6-1-1- تاریخچه متدلوژی چابک توسعه نرم افزار (سایت www.wikipedia.org)

تعاریف جدید از توسعه چابک نرم افزار در نیمه اول قرن 1990 میلادی به عنوان روشی در مقابل روشهای سخت و مشکل ارائه شد.

فرآیند مدل‌های آشنایی که پیش از آن استفاده می شد به نظر، کند، تشریفاتی، و دارای تناقضاتی در رابطه با عملکرد واقعی مهندسين نرم افزار بود.

درمقابل روشهای چابک روشهای روان و اسان می باشند که در سال 2001 میلادی، اعضای برجسته انجمنی در اسنوبرد در طی ملاقاتی که با یکدیگر داشتند بر روی نام "روشهای چابک" توافق کردند. تا اینکه بعدها بسیاری از افراد این انجمن، اتحادیه چابک را که یک سازمان غیر انتفاعی می باشد را برای ارتقای توسعه چابک، تشکیل دادند.¹³⁷ روشهای چابک پیش از سال 2000 پدید آمدند که تا امروز کاربردی ترین آنها عبارتند از:

1. Extreme Programming (1996)
2. Adaptive Software Development
3. Feature Driven Development
4. DSDM(1995)
5. Scrum (1986)
6. Crystal Clear

از میان این روشها روش XP به عنوان نقطه اصلی شروع متدلوژی های چابک به حساب می آید. در بخش 3 تاریخچه تشکیل هر یک از متدلوژی های بالا به تفکیک بیان شده است.

¹³⁷ Agile alliance

6-1-2- بررسی ویژگیها و خصوصیات متدلوژی های چابک توسعه نرم افزار

سازمانها در حوزه مهندسی نرم افزار همواره یک سوال اساسی را پیش روی خود داشته اند و ان این است که چگونه می توان با در نظر گرفتن کیفیت مطلوب و مورد انتظار در نرم افزار ،سرعت تولید آن را افزایش داد و به صورت چابکتری به سمت دستیابی به نرم افزار نهایی گام برداشت. در سال 2000 میلادی، Truex و همکارانش سوالی را مطرح نمودند که در زمان بندی برنامه ریزی و اجرای برنامه به منظور توسعه نرم افزار به واقع می بایست از چه نوع روش و متدهایی جهت توسعه نرم افزار استفاده نمود؟ در این بررسی ،محققین دو نوع کلی از متدها و روشهای توسعه نرم افزار را معرفی کردند ،که این دو روش کلی عبارتند از:

1.توسعه نرم افزار به کمک متدهای رایج و مطرح

2.توسعه نرم افزار به کمک متدهای خاص و حاشیه ای

در این بررسی خروجی تحقیق عبارت بود از یک جدول مقایسه ای بین دو نوع متد های ارائه شده که در جدول شماره 1 نشان داده شده است.

ویژگی های فرآیند توسعه نرم افزار	
توسعه نرم افزار به کمک متدهای رایج و مطرح	توسعه نرم افزار به کمک متدهای خاص و حاشیه ای
فرآیند توسعه کاملاً مدیریت می شود و فرآیندهای کنترلی بر کل سیستم وجود دارد.	کنترل فعالیت ها به صورت تصادفی انجام می گیرد.
فرآیند تولید خطی است و به صورت تدریجی تکمیل می گردد.	فرآیندها به صورت همزمان شروع می شوند و همپوشانی آنها و مشکلات ناشی از ان در زمان تولید حل می شود.

در توسعه از فعالیتهای یکتا و مخصوص سازمانی که نرم افزار را توسعه می دهد، استفاده می شود.	در توسعه از فعالیت های استاندارد و شناخته شده استفاده می شود.
فرآیند توسعه کاملاً بحث پذیر و قابل انعطاف و تغییر است.	فرآیند توسعه نرم افزار امری کاملاً منطقی، با هدف و فعالیتهای کاملاً مشخص است.

شکل 81- جدول مقایسه بین متدهای رایج و خاص

با انجام این مقایسه تفاوت های موجود بین متدهای رایج و متدهای خاص مشخص شد. اما نکته قابل توجه آن است که می توان از متدهای خاص به عنوان مبنای فکری توسعه متدهای چابک نام برد.

❖ بررسی کلی و معرفی

تعریف اصلی از متدولوژی های چابک اولین بار توسط گروه مشاورین نرم افزار در سال 2001 میلادی به وسیله اعلانیه رسمی¹³⁸ ارائه گردید. در زمینه چابکی در این اعلانیه 4 رویکرد اصلی مورد توجه قرار گرفته بود که این 4 رویکرد عبارتند از :

1. فعالیتهای شخصی و تعاملات بین افراد

در طی فرآیند توسعه نرم افزار در این حوزه هدف افزایش روحیه کار تیمی بین افراد است. بدین ترتیب که این حوزه بر ایجاد ارتباط مثبت بین اعضای گروه توسعه نرم افزار و تعیین نقش های افراد برای پیشبرد کار تیمی تاکید فراوان دارد.

2. توسعه نرم افزاری کارا مبتنی بر مستندات

در این رویکرد هدف آن است که در فرآیند توسعه نرم افزار مستندات دقیق و کاملی تولید شود تا افراد تیم توسعه دهنده نرم افزار خود پس از پایان فعالیت هایشان بتوانند کار خود را بر مبنی

² Agile Manifesto

مستندات به صورت کامل تحت بررسی قرار دهند و به سرعت ایرادات به وجود آمده را بر طرف سازند.

3. برقراری یک تعامل مثبت با مشتریان

در این حوزه نیز رویکرد مطرح آن است که می بایست بین اعضای تیم توسعه نرم افزار و مشتری خاص نرم افزار تعامل سازنده ای در جهت رفع کامل نیازمندیهای مشتری به وجود آید. بدین ترتیب با وجود چنین تعاملی نیازمندیهای مشتری به سرعت به گروه توسعه دهنده نرم افزار منتقل می شود و به سرعت این نیازها مرتفع می گردد. لازم به ذکر است که محدوده اینگونه برقراری تعاملات و برطرف سازی نیازمندیها باید کاملاً مبتنی بر قرارداد باشد.

4. پاسخگویی در برابر تغییرات در جریان برنامه کاری

بدین معنی که در طول عمر فرآیند توسعه نرم افزار هر دو طرف توسعه دهندگان و مشتری باید در سطحی مناسب از تصمیم گیری باشند که بتوانند به راحتی نیازمندیهای خود را به یکدیگر انتقال دهند و برطرف سازی نیازمندیها به سرعت انجام پذیرد.

Cockbun(2002a) هسته اصلی فرآیند توسعه نرم افزار چابک عنوان می دارد که این متد در عین سادگی بسیار پیچیده است و لازم است که در این نوع پروژه های نرم افزاری تعاملات و ارتباطات بین افراد به درستی مشخص گردد. در رویکرد سهل، وی اذعان می دارد که استفاده از این متد حوزه انعطاف و مانور توسعه دهندگان سیستم را زیاد می کند. اما در رویکرد مشکل، وی می گوید که باقی ماندن در این نوع بازی که به مقدار زیادی متکی بر روابط انسانی است بسیار سخت و مشکل است.

وی برخی نکات را برای دستیابی به موفقیت در این نوع از پروژه ها مطرح کند که عبارتند از:

- در اتاقی که به منظور توسعه نرم افزار اختصاص داده می شود باید بین 2 تا 8 نفر حضور داشته باشند.

§ هدف از انجام چنین کاری ایجاد امکان بهبود تعاملات بین افراد است.

- از افراد متخصص در محل مشتری در زمان راه اندازی نرم افزار استفاده شود.
- § هدف از انجام این کار کوتاه کردن زمان بین ارائه بازخورد تا برطرف سازی مشکل و کسب رضایت مشتری است.
- در نظر گرفتن بازه‌های زمانی کوتاه برای انجام کار
- § این بازه باید بین 1 تا 3 ماه باشد. هدف از انجام این کار تست سریع نرم افزار و آماده برای کار کردن نرم افزار است.
- ایجاد روند اتوماتیک برای تست
- § تسریع روند تست نرم افزار با استفاده از تست مکانیزه کدهای نرم افزار
- استفاده از توسعه دهندگان مجرب
- § استفاده از اینگونه افراد سرعت توسعه نرم افزار را بین 2 تا 10 برابر افزایش می دهد.

Ø ویژگیها:

- (Miller 2001) ویژگیهای فرآیند چابک توسعه نرم افزار را برای تحویل سریع ارائه داد به طوری که اجازه کوتاه کردن چرخه حیات پروژه ها را به صورت:
1. پیمانه ای بودن در سطوح فرآیند توسعه
 2. تکرار های با چرخه حیات کوتاه که قادر به رسیدگی و اصلاح سریع می باشد.
 3. زمان مشخص با چرخه های تکرار از 1 تا 6 هفته ای
 4. حذف همه فعالیتهای غیر ضروری در فرآیند توسعه
 5. انطباق با اثرات ریسکهای جدید
 6. نگرش فرآیند افزایشی که اجازه می دهد ساخت برنامه های کاربردی در مراحل کوچکتر انجام شود.

7. نگرش همگرایی و افزایشی که در طول زمان، مدیریت تغییرات را آسانتر نموده و در نتیجه مخاطرات (ریسک) را کاهش می دهد .

8. دیدگاه مشتری محوری که در آن توجه به افراد بیش از میزان توجه به فرآیند و تکنولوژی است.

9. کار مشارکتی و تعاملی

فراهم کرد.

آقای فیورو¹³⁹ استراتژی های فوری را برای انطباق یک فرآیند مبهم با تغییرات نیازمندیها توضیح می دهد.

وی بیان می کند که نیازمندیها ممکن است در یک تکرار موفق، معرفی، بازبینی و یا حذف شوند. با این تفصیل روش چابک برای:

- فراهم کردن اولین تحویل در هفته، دستیابی سریع به نتایج مطلوب و بازخورد سریع

- ایجاد راه حل های ساده که کمتر قابل تغییر و نیز تغییراتشان آسانتر می باشد.

- بهبود کیفیت طراحی، ایجاد تکرار نهایی کم هزینه تر برای پیاده سازی

- تست مستمر برای تسریع فرآیند توسعه، هزینه پائین تر و کشف نقایص

طراحی شده است .

یک مجموعه از نگرش های مشترک در فرآیند چابک توسعه نرم افزار توسط آمبلر¹⁴⁰ به صورت زیر

پیشنهاد شده است: (ambler 2002)

- اهمیت دادن به افراد

- مستند سازی کمتر در حد امکان

- ابزار مدلسازی به اندازه ای که فکر می شود سودمند و راهگشا نمی باشند.

- نیازمند حجم طراحی زیادی نمی باشد.

¹³⁹ Favaro

¹⁴⁰ ambler

آقای بوهم¹⁴¹، متدلوژی چابک و متدلوژی فرآیندگرا را در جدول شماره 2 به صورت زیر تجزیه تحلیل کرده است. (Boehm 2002)

فضا	agile	فرآیند گرا
توسعه دهندگان	چابک، مطلع، مشارکتی، مرتب شده	فرآیند گرا، دسترسی به دانش خارجی، مهارت کافی
مشتریان	اختصاصی، مطلع، مشارکتی، نمایان، مشتری محور	دسترسی به دانش، مشارکتی نمایان، مشتری محور
نیازمندیها	تغییرات سریع	ثابت، قابل فهم
معماری	برای نیازمندیهای موجود طراحی شده	طراحی برای نیازمندیهای موجود و قابل پیش بینی
باز سازی	ارزان	گران
اندازه	تیمها و محصولات کوچکتر	تیمها و محصولات بزرگتر
اهداف اولیه	دستیابی سریع به ارزش	دستیابی به ضمانت محکم

شکل 82- جدول فضای بین روشهای چابک و فرآیند گرا (بوهم 2002)

Ø عوامل موفقیت روشهای چابک

اگر چه روشهای چابک در عمل با هم متفاوت می باشند،

¹⁴¹ Boehm

اما در تعدادی از ویژگیها شامل، توسعه تکرار پذیر، تمرکز بر روابط متقابل، ارتباطات، و کاهش منابع مصنوعی واسطه ای متمرکز مشترک می باشند.

از دید تولید، روشهای چابک زمانی که نیازمندیها در حال تغییر می باشند مناسب می باشند و برای سیستم هایی که دارای اهمیت در زمینه اعتبار بالا و ایمنی نیازمندی ها می باشند نا مناسب می باشند.

از دید سازمانی، مناسب بودن می تواند با تخمین سه بعد یک سازمان مانند فرهنگ، افراد و ارتباطات سنجیده شود.

در ارتباط با این فضا یک تعداد از فاکتورهای کلیدی شناسایی شدند:

- فرهنگ سازمانی باید پشتیبان مذاکرات باشد.
- افراد باید دلگرم باشند.
- افراد اندک اما شایسته تر
- سازمانها نیازمند داشتن محیطی هستند که سرعت ارتباطات را بین اعضای تیم تسهیل کنند.
- فاکتور مهم دیگر، اندازه پروژه می باشد. هنگامی که اندازه پروژه رشد می کند ارتباطات رو در رو مشکل تر می شود.

بنابراین روشهای چابک، مناسب برای پروژه هایی باتیم های کوچک، بین 20 تا 40 نفر می باشد. همچنین خانواده crystal معیاری را فراهم می کند که چطور روشی را برای پروژه انتخاب کنیم. انتخاب بر اساس اندازه پروژه، بحرانی بودن و اولویت شان می باشد. دیگر روش ها، تضمین واضحی برای تعیین مناسب بودن را فراهم نمی کنند.

توسعه چابک به طور گسترده ای برای تیم های کوچک (توسعه دهندگان کمتر از 10 نفر) و تیم هایی که در یک مکان قرار دارند، مستندسازی شده است. در ضمن توسعه چابک برای تیم هایی که در آن نیازمندی ها در حال تغییر سریع و غیر قابل پیش بینی می باشند، مناسب می باشد.

Ø تعاریف (Highsmith and Cockburn 2001):

• تامین کیفیت مورد نظر در روشهای چابک

استراتژی روشهای چابک، به دنبال کاهش هزینه تغییرات در سراسر پروژه می باشد. توسعه نرم افزار چابک کیفیت را در فرآیند تولید محصول طراحی می کند. به عبارتی روشهای چابک با استاندارد کردن فرآیند تولید و لحاظ نکات کیفی، به گونه ای روش ها و فرآیندهای خود را طراحی می کند که ما حاصل این فرآیندها محصولی با کیفیت باشد. به دیگر سخن روشهای چابک معتقد به طراحی کیفیت در فرآیند تولید محصول به جای بازرسی نهایی و کنترل کیفیت می باشد.

• اصول اساسی روشهای چابک

روشهای چابک دو مفهوم کلی را بیان می کنند که عبارتند از:

1. اعتماد بر قوانین کارکردن

2. افزایش کارایی افراد به صورت مشارکتی.

قوانین کاری بیانگر این است که چطور توسعه دهندگان و حامیان نرم افزار به وعده های خود در ارتباط با وظایفشان عمل می کنند.

قوانین می توانند منتقل، بازبینی و یا حتی لغو شوند اما همیشه واقعی می باشند و در زمینه افزایش کارایی افراد به صورت مشارکتی روشهای چابک به این صورت عمل می کنند که افراد به طور کارآمد در جهت افزایش سرعت و کاهش هزینه فعالیت نمایند. همچنین نظرات خود را با صحبت کردن رو در رو (انتقال مستقیم اطلاعات) به جای نوشتن با سرعت بیشتری انتقال دهند. کار مشارکتی همیشه نتایجی بهتر از کار انفرادی را حاصل می کند. از این رو مجموعه طراحان، طرح بهتر و کارآمدتری را نسبت به یک طراح آماده می کنند که روشهای چابک با توجه به خصوصیت

مشارکتی‌شان طرح‌های بهتری را ارائه می‌دهند. همچنین مشارکت بین افراد پروژه، مشتریان و حمایت کنندگان موجب تنظیم دقیق‌تر اولویت‌ها و بررسی کارآتر گزینه‌های ممکن می‌شود.

• باز خورد و تغییرات در روشهای چابک

به این علت که روش‌های توسعه چابک نرم‌افزار بیشتر در محیط آشفته و با تغییرات بالا کاربرد دارند، روش‌های چابک تنوع استفاده از چندین روش مختلف را برای بازخورد ثابت در تصمیمات تکنیکی، نیازمندی‌های مشتریان و محدودیتهای مدیریت پیشنهاد می‌کند. به عنوان مثال XP طرفدار برنامه‌نویسی دوتایی برای بازخورد؛ DSDM از چرخه کوتاه خصوصیات با تشکیل نمونه و SCRUM و ASD فرآیند تکرار پذیر با بازبینی تیمی را پیشنهاد می‌دهند. در واقع بازنگری تکرارها را توسط گروه نظارتی مشتریان انجام می‌دهد.

• تمرکز بر کار تیمی در روشهای چابک:

نزدیکی گروه‌ها و روابط متقابل قوی بین اعضای گروه یک نقطه قوت برای همه روش‌های چابک می‌باشد. استفاده از روشهای چابک نیازمند همکاری محکم مشتری می‌باشد. اگر مشتریان هر یک از نمایندگان دپارتمان داخلی یا مدیران بازاریابی محصولات احساس خوبی از مسیر نداشته باشند و نسبت به طرح‌ها سرگردان باشند، توسعه دهندگان چابک از آنها پیروی می‌کنند. در واقع مشتریان ضعیف عامل ایجاد سیستم‌های ضعیف می‌شوند.

6-2- مقایسه روش چابک با دیگر روشها (سایت www.wikipedia.org)

روش‌های چابک مقابل متدلوژی‌های برنامه محور و نظام یافته بوجود آمدند. این مقایسه در ذهن این تصور را پدید می‌آورد که متدلوژی‌های چابک، بی‌برنامه و نظام نیافته‌اند. در واقع تفاوت درست و دقیق روش‌های مورد نظر در تقابل انطباق‌پذیری و پیش‌بینی است به گونه‌ای که روشهای چابک متمرکز بر انطباق‌پذیری هستند و به جای پیش‌بینی دقیق مراحل آتی به یک پیش‌بینی کلی اکتفا کرده و توانایی انطباق‌پذیری خود را در مقابل تغییرات توسعه داده‌اند. این در حالی است که

روش‌های سنتی با پیش‌بینی دقیق و جزئی آینده با تدوین برنامه و نظام مشخص در مراحل آینده متمرکز بر پیش‌بینی می‌باشند. روش‌های متمرکز بر انطباق‌پذیری با تغییر شرایط به سرعت منطبق می‌شوند که این خصوصیت در شرایطی که سیستم مورد توسعه، متغیر و سرعت و میزان تغییرات بالا است، بسیار کارآمد می‌باشد. تیم‌هایی که بر اساس روش‌های انطباق‌پذیر فعالیت می‌کنند، می‌توانند به دقت وظایف هفته آینده خود را گزارش کنند و اما در مورد وظایف ماه آینده تنها مأموریت‌های کلی مشخص می‌باشد. این عدم قطعیت در برنامه‌ریزی اجرایی تا حدی است که این تیم‌ها در مورد برنامه 6 ماه آینده خود تنها از شرح کلی مأموریت‌ها یا ارزش‌هایی را که باید حاصل کنند، اطلاع دارند.

در نقطه مقابل، روش‌های "برنامه محور" به تفصیل بر برنامه‌ریزی آینده تمرکز دارند به گونه‌ای که آینده از نظر آنها پیش‌بینی شده و برنامه عملیاتی کاملاً مشخص می‌باشد. این تیم‌ها می‌توانند به دقت وظایف کل دوره فرآیند توسعه را گزارش کنند. این تیم‌ها مدیریت تغییرات را به سختی انجام می‌دهند. برنامه‌های تدوین شده برای اهداف اصلی پروژه در نظر گرفته شده‌اند ولی برای مدیریت تغییرات باید راهکاری غیر از این برنامه در نظر گرفت. معمولاً تیم‌های برنامه محور در درون خود گروهی را برای کنترل تغییرات در نظر می‌گیرند. "گروه کنترل تغییرات" در مراحل مختلف با توجه به تغییرات ایجاد شده برنامه تدوین شده را بازنگری و اصلاح می‌نمایند.

6-2-1- روش‌های چابک در مقایسه با توسعه تکرار پذیر¹⁴²

روش‌های چابک در درون خود به صورت تکرارپذیر عمل می‌نمایند به عبارتی روش‌های چابک بر پایه روش‌های توسعه تکرارپذیر بنا شده‌اند. تفاوت اصلی این روش‌ها با روش‌های مرسوم توسعه تکرارپذیر در دوره زمانی در نظر گرفته شده برای تکرارهاست. روش‌های چابک دارای دوره‌های

¹⁴² iterative development

زمانی کوتاه‌تر (در حد هفته نسبت به ماه) می‌باشند. در نتیجه با توجه به کوتاهی تکرارها توسعه نرم‌افزار در روش‌های چابک مشارکتی‌تر از روش‌های توسعه تکرار پذیر است.

6-2-2-2- روش‌های چابک در مقایسه با مدل‌های آبشاری¹⁴³

روش‌های چابک با مدل‌های آبشاری دارای نقاط اشتراک بسیار اندکی می‌باشند. در نظر بسیاری مدل‌های آبشاری دیگر منسوخ شده‌اند. اما امروزه نیز استفاده از این روش‌ها در برخی از گروه‌های توسعه نرم‌افزار مرسوم می‌باشد. از آنجائیکه مدل‌های آبشاری از گروه روش‌های توسعه برنامه محور و پیش‌بینی کننده می‌باشند، کلیه مراحل توسعه نرم‌افزار شامل مدیریت نیازمندی‌ها، تحلیل، طراحی، پیاده‌سازی و تست، کاملاً از پیش برنامه‌ریزی شده‌اند. عموماً در مدل‌های آبشاری پیشرفت پروژه را با دست‌آوردهای مراحل مختلف پروژه اندازه‌گیری می‌کنند. مثلاً اینکه پروژه در مرحله تشریح نیازمندی‌ها، مستندات طراحی، برنامه تست، بازنگری کدها و یا چیزهایی از این قبیل است. این روش‌ها معمولاً با یکپارچه‌سازی و تست پایان می‌پذیرد. زمانبری و دشواری عملیات یکپارچه‌سازی و تست یکی از عمده دلایل شکست پروژه‌هاست.

روش‌های چابک همانند سایر روش‌های توسعه تکرارپذیر هر چند هفته یا چند ماه خصوصیات کامل محصول توسعه یافته را یکپارچه سازی و تست می‌کند. در این روش تاکید بر مدیریت بدست آوردن خروجی‌های تا حد ممکن کوچک و سریع الوصول می‌باشد که در ادامه می‌توان با بهبود مستمر و اضافه کردن مجدد محصولات کوچک دیگر به آن تا انتهای چرخه پروژه، محصول نهایی را تولید نمود. اکثر تیم‌های چابک از مدل آبشاری در مقیاس کوچک به معنای به کارگیری کل چرخه آبشاری در هر تکرار استفاده می‌کنند.

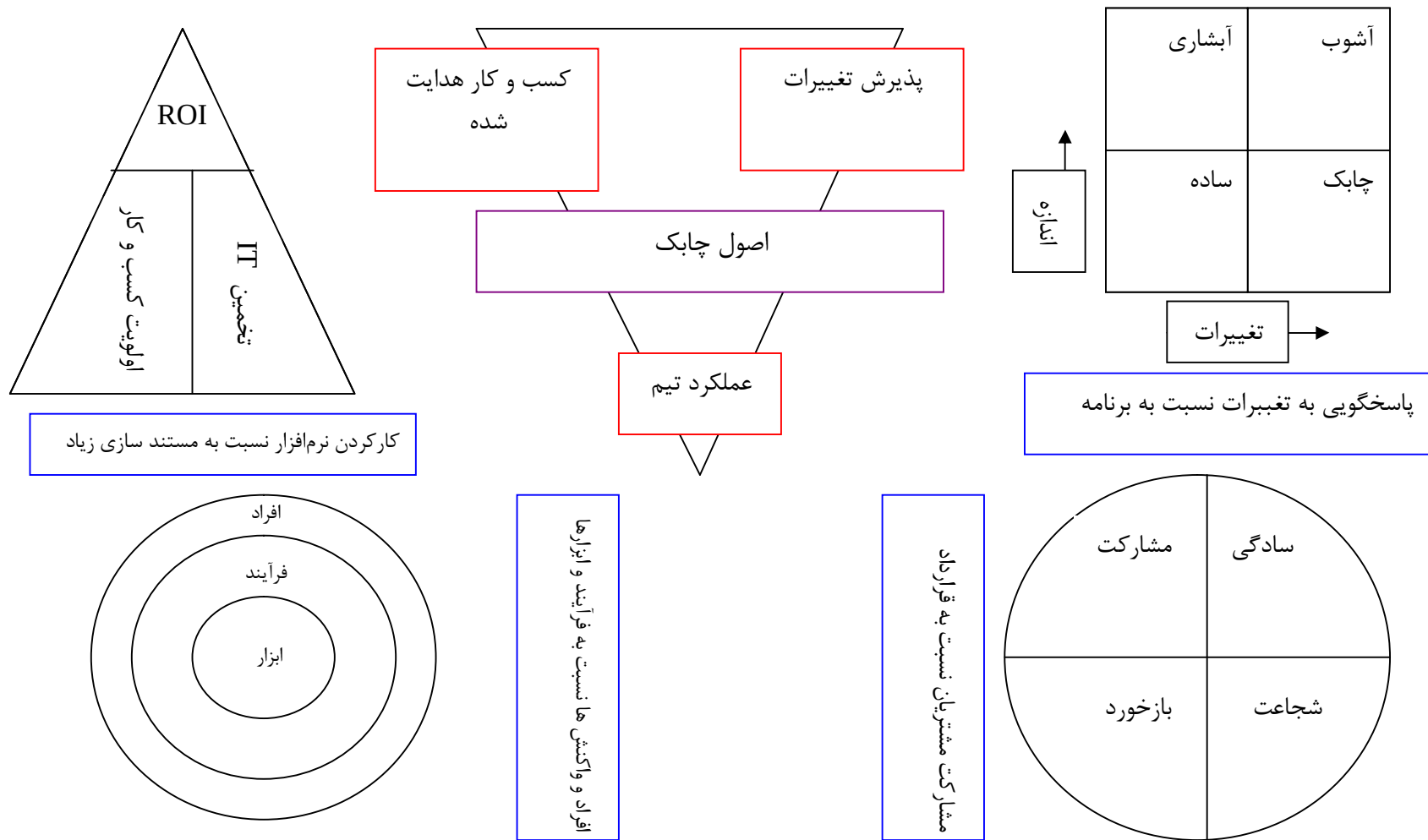
¹⁴³ waterfall model

6-2-3- روشهای چابک در مقایسه با کد نویسی⁵ Cowboy

کدنویسی با استفاده از روش cowboy به معنی نداشتن روش مدون برای توسعه نرم افزار می باشد. در این روش اعضای تیم هر آنچه که را احساس می کنند درست می باشد انجام می دهند. در مقایسه با روش cowboy، در روشهای چابک، ارزیابی مجدد برنامه بر اساس ارتباطات رو در رو با تاکید بر کمترین مستندسازی ممکن بنا شده است. بسیاری از افراد در فرآیند کد نویسی cowboy سردرگم می شوند ولی تیمهای توسعه چابک فرآیندهای از پیش تعریف شده و گاه بسیار نظام مند را انجام می دهند.

پس از توضیح مختصری که در مورد متدولوژی چابک انجام شد، اکنون هر یک از روشهای مطرح در این متدولوژی معرفی خواهند شد.

⁵ cowboy coding



شکل 83- جایگاه و اصول چابک

6-3- مروری بر روشهای چابک موجود:

در مطالعات منابع موجود در زمینه روشهای چابک، متدلوژیهای مختلفی را مورد بررسی قرار گرفته است، اما در اغلب منابع مورد بررسی 7 متدلوژی زیر، متدهای مشترکی هستند که در تمام منابع به آنها اشاره شده است:

- 1) OSS(O'Reilly 1999)
- 2) ASD(Highsmith 2000)
- 3) DSDM(Stapleton 1997)
- 4) FDD(Palmer and Felsing 2002)
- 5) Crystal(Cockburn 2002a)
- 6) Scrum(Schwaber 1995.Schwaber and Beedle 2002)
- 7) Xp(Beck 1999)

در این بخش به معرفی و مرور متدلوژیهای معرفی شده پرداخته خواهد شد. موارد عمده مطرح در بررسی متدلوژیها عبارتند از:

1. **تاریخچه:** که در آن زمان پیدایش هر یک از متدلوژیها و منابع منتشر شده در این زمینه مطرح خواهد شد.

2. **معرفی ساختار فرآیند:** در این قسمت به توضیح جامعی از فازهای موجود در زیست چرخ تولید هر یک از متدلوژیها اشاره می شود.

3. **کاربردها:** در این بخش موارد کاربرد هر کدام از متدلوژیها در انواع پروژهها بیان می شود.

4. **نقاط کلیدی و خصوصیات ویژه** که آنها را از دیگر روشها متمایز میکند .

5. **در نهایت مزایا و محدودیت های** هر یک از متدها بیان شده است.

Extreme Programming -1-3-6

Ø تاریخچه:

متدلوژی XP که مخفف واژه extreme programming است توسط ران جفریز¹⁴⁴، وارد کانینگام¹⁴⁵، کنت بک¹⁴⁶ در مسیر انجام پروژه C3 (لیست حقوقی)¹⁴⁷ در کشور امریکا توسعه در مسیر انجام این پروژه آقای "کنت بک" به مطالعه، بررسی و بازبینی متدلوژی‌های مطرح مهندسی نرم‌افزار در آن زمان پرداخت و نتایج تحقیقات خود را به صورت کتابی تحت عنوان **extreme programming explaine** در سال 1999 میلادی منتشر نمود. لازم به ذکر است که پروژه C3 در نهایت با شکست روبرو گردید اما متدلوژی معرفی شده توسط "کنت بک" به عنوان یکی از متدلوژی‌های علم مهندسی نرم‌افزار مطرح شد و مورد استفاده در دیگر پروژه‌ها قرار گرفت به نوعی که بررسی‌ها نشان می‌دهد، در سال 2006 میلادی حجم زیادی از پروژه‌های توسعه نرم‌افزار از این متدلوژی برای توسعه استفاده نمودند و این روند تا به امروز در حال رشد بوده است.

Ø فرآیند:

فرآیند زیست چرخ حیات متدلوژی XP از 6 فاز تشکیل شده است که عبارتند از: (beck 1999)

- (1) اکتشاف¹⁴⁸
- (2) برنامه‌ریزی¹⁴⁹
- (3) تولید نمونه اولیه به صورت تکرارپذیر¹⁵⁰
- (4) تولید¹⁵¹
- (5) تعمیر و نگهداری¹⁵²

¹⁴⁴ Ron Gefries

¹⁴⁵ Ward Cunningham

¹⁴⁶ Kent Beck

¹⁴⁷ Chrysler Comperhensive Compensation System

¹⁴⁸ exploration

¹⁴⁹ planning

¹⁵⁰ Iterative to release

¹⁵¹ productionizing

روند انجام هر یک از این فازها در این متدولوژی با استناد نتایج حاصله از مطالعات آقای "بیک" در پروژه C3، به صورت زیر می‌باشد. که در ادامه به معرفی هرفاز پرداخته خواهد شد.

• فاز اکتشاف:

در این فاز، مشتریان کارتهای مربوط به معرفی نیازمندی‌هایشان را تکمیل می‌نمایند. در هرکارت شرح نیازمندی‌های که مورد نظر کاربران نهایی نرم‌افزار بوده و لازم است نرم‌افزار این سطوح مختلف نیازمندی‌ها را پوشش دهد، ارائه می‌گردد. به عبارت دیگر با پرشدن این کارتها توسط کاربران نهایی نرم‌افزار محدوده پروژه تا حد مطلوبی قابل تخمین می‌باشد و تیم مجری پروژه با استفاده از این اطلاعات می‌توانند ابزارها و تکنولوژیها، را که لازم است در طول زمان توسعه نرم‌افزار مورد استفاده قرار گیرد را مشخص می‌نمایند. همچنین تیم مجری پروژه با استفاده از این اطلاعات می‌توانند فرآیندها و فعالیت‌هایی را که لازم است نرم‌افزار آنها را تحت پوشش خود قرار دهد مشخص می‌نمایند.

در گام بعد به منظور صحه گذاری بر روی نیازمندی‌های شناسایی شده نمونه سریعی¹⁵⁴ از نرم‌افزار مورد نظر تولید شده و توسط کاربران نهایی سیستم مورد بررسی قرار می‌گیرد. به این ترتیب تیم مجری پروژه نرم‌افزار می‌تواند نسبت به شناخت دقیق نیازمندی‌های سیستم از دید کاربران نهایی اطمینان لازم را کسب نماید. این مرحله از چرخه حیات متدولوژی بین چند هفته تا چند ماه به طول می‌انجامد که به میزان یادگیری و آشنایی برنامه نویسان با تکنولوژی و ابزارهای مورد نیاز و میزان همکاری کاربران نهایی نسبت به اعلام و صحه گذاری نیازمندی‌ها بستگی دارد.

• فاز برنامه نویسی:

¹⁵² maintance

¹⁵³ death

¹⁵⁴ prototype

در گام دوم این متدلوژی نیازمندی‌های کاربران نهایی اولویت بندی می‌گردند و بر اساس این اولویت بندی تیم مجری پروژه و کارفرما نسبت به نوع و روند انجام پروژه توافقاتی را حاصل می‌نمایند.

در ابتدا این فاز توسعه‌دهندگان نرم‌افزار با توجه به شرح نیازمندی‌های استخراج شده و محدوده تعیین شده پروژه برآوردی از میزان نفرساعت مورد نیاز پروژه و نوع مهارت و تخصص‌های آنها را انجام می‌دهند. هدف از انجام این فعالیت را می‌توان توسعه نرم‌افزار با استفاده از حداکثر کارایی پرسنل تیم توسعه دهنده در کمترین زمان ممکن با بیشترین کیفیت دانست. در این فاز حداکثر زمان مورد نیاز در جهت زمانبندی نخستین تکرار برای تولید نمونه اولیه 2 ماه می‌باشد.

• فاز تولید نمونه اولیه به صورت تکرار پذیر:

این مرحله دارای چندین چرخه تکرار برای توسعه اولین نسخه نرم‌افزار که حالت پروتوتایپ دارد، می‌باشد .

زمان در نظر گرفته شده در این فاز بین چرخه‌های تکرار شکسته می‌شود به طوری که هر کدام از تکرارهای تولید نرم‌افزار بین 1 تا 4 هفته توسعه یافته و پیاده‌سازی می‌شوند.

در اولین تکرار، نمونه اولیه‌ای از نرم‌افزار با در نظر گرفتن معماری کل سیستم از دو دید تحلیل فرآیندی و تحلیل سیستمی توسعه می‌یابد. این محصول نشان دهنده نمایی از کل نرم‌افزار نهایی است. در این میان مشتریان برای هر کدام از چرخه‌های تکرار بازخوردهایی را به تیم توسعه دهنده ارائه می‌نمایند. همچنین هر یک از کاربران نهایی در انتهای هر یک از تکرارها بر روی بخشی از محصول بدست آمده که مرتبط با حوزه کارکردی ایشان است، تست عملیاتی انجام می‌دهند و در نهایت در آخرین تکرار، طراحی نرم‌افزار تکمیل گشته و برای تولید نهایی آماده می‌گردد.

• فاز تولید:

همانطور که بیان شد پس از تکرار نهایی در فاز قبل، طراحی نرم‌افزار تکمیل گشته و برای تولید آماده می‌باشد. در فاز تولید، محصول برای کاربر نهایی با توجه به نیازمندی‌های ارائه شده وی،

توسعه می‌یابد، لازم به ذکر است در این فاز نیز همچون فاز پیشین لازم است نرم‌افزار در هر مرحله از تولید به دقت و حتی چندین مرتبه تست شود و عملکردش قبل از تهیه نسخه نهایی مورد بررسی و صحت‌گذاری قرار گیرد و در صورت نیاز به اعمال تغییر در طراحی نرم‌افزار در این فاز تغییرات مورد نیاز نهایی شده و در روند توسعه نرم‌افزار اعمال می‌گردد. در آخرین گام از این فاز به منظور پیاده‌سازی نرم‌افزار تمام مراحل انجام کار و پیشنهادات در جهت بهبود عملکرد سیستم مستند سازی می‌شود.

• فاز تعمیر و نگهداری:

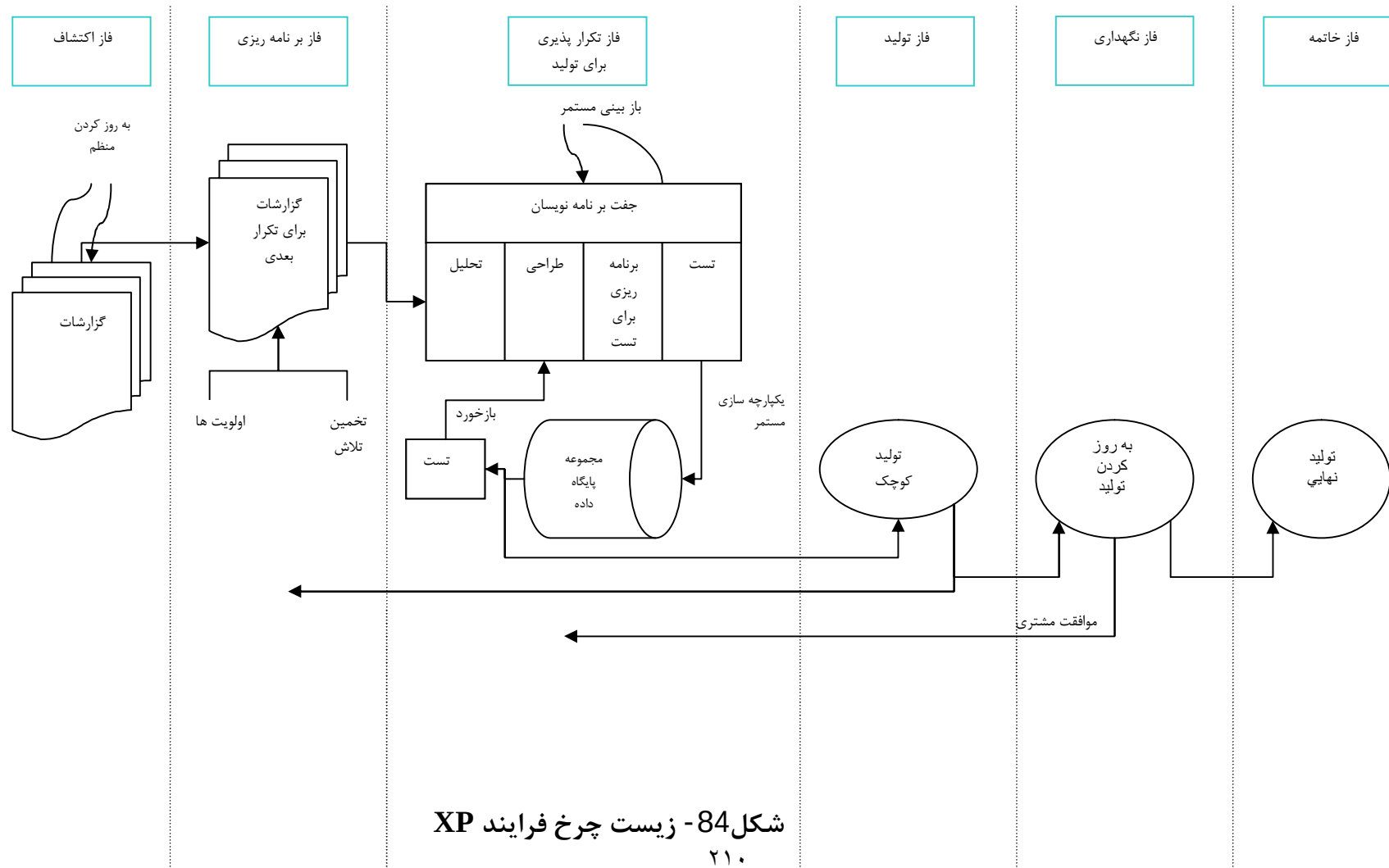
هدف اصلی این فاز تامین خواسته‌های کاربران نهایی پس از تحویل گرفتن نسخه نهایی نرم‌افزار توسط ایشان می‌باشد به طوری که در صورت پدید آمدن مشکل و یا وجود نواقص در نرم‌افزار، این مشکلات از سوی تیم مجری برطرف می‌گردند.

در واقع این فاز تضمینی است برای کاربران نهایی نسبت به اینکه در صورت بروز مشکل در نرم‌افزار مورد استفاده، این مشکلات از سوی تیم مجری بدون نیاز به دوباره‌کاری در حداقل زمان برطرف خواهد شد.

• فاز ختم پروژه:

در آخرین مرحله زمانی که مشتریان هیچ تقاضایی برای بهبود و اصلاح نرم‌افزار در مرحله پیاده سازی نداشته باشند و یا اینکه نرم‌افزار توسعه یافته نیازمندی‌های فعلی سازمان را تامین ننماید و یا در مسیر توسعه آینده سازمان در حوزه فناوری اطلاعات استفاده از این نرم‌افزار هزینه بر تخمین زده شود، چرخه حیات این مدل‌سازی را می‌توان پایان پذیرفته دانست.

شکل زیر نشان دهنده فرآیند این متدولوژی می باشد.



Ø کاربرد:

موارد کاربرد مطرح در این متدلوژی عبارتند از:

(1) **پاسخگویی سریع:** در واقع این متد برای پاسخگویی سریع به مسایل در مورد تغییر

نیازمندی‌ها به کار برده می‌شود. با استفاده از این متدلوژی تیم توسعه نرم‌افزار به سرعت

می‌تواند تغییرات مورد نیاز کاربر نهایی را در نرم‌افزار اعمال نماید.

(2) **کاهش ریسک:** اگر بخواهیم ریسک را کاهش و احتمال موفقیت را در پروژه‌ای با استفاده

از متدلوژی افزایش دهیم، بهتر است از متدلوژی XP استفاده شود. واضح است که ریسک

انجام پروژه با اعلام نیاز مشتری نسبت به تغییر نرم‌افزار در زمان معین افزایش می‌یابد.

همچنین در صورتی در تیم توسعه نرم‌افزار فناوری جدید توسعه مطرح شود که تیم پروژه

هیچ سابقه اجرایی بر روی آن نداشته باشد، ریسک پروژه افزایش می‌یابد. از این رو می‌توان

گفت که پروژه‌هایی با نیازمندی‌های پویا و ریسک بالا همراه با تیم برنامه نویسی کوچک، در

صورتی که از این متدلوژی استفاده نمایند، کارآمدتر و موفق‌تر عمل می‌کنند. (سایت

(Extremeprogramming)

(3) **حجم اندک مستندسازی:** در این متد به دلیل خاصیت چابکی آن تیم مجری پروژه

نمی‌تواند زمان زیادی را به تولید مستندات پروژه نرم‌افزار که در اغلب دیگر متدلوژی‌های

غیر چابک وجود دارد، تخصیص دهد. به این منظور این متد از مستندسازی به صورت

کمتر استفاده می‌نماید.

Ø مشخصات تیم توسعه دهنده نرم‌افزار

از آنجائیکه تیم توسعه دهنده نرم‌افزار در متدلوژی XP باید در یک محیط فیزیکی با یکدیگر

همکاری کنند در نتیجه اندازه تیم بین 3-10 نفر محدود می‌شود (سایت

(www.dacs.dtic.mil/techs/agile).

همچنین لازم نیست که توسعه‌دهندگان دارای مدارک بالا در برنامه‌نویسی و توسعه نرم‌افزار باشند بلکه افراد تیم می‌توانند افراد معمولی بوده که صرفاً دارای توانمندی برنامه‌نویسی در حد معمول باشند.

به طور معمول پرسنلی که در اینگونه پروژه‌ها درگیر می‌شوند لازم است کاربران فعال و پرنرزی باشند.

Ø نقاط کلیدی:

- (1) **مشتری مداری:** از آنجائیکه توسعه نرم‌افزار توسط متدولوژی XP بر اساس نیاز و درخواست کاربر نهایی است در واقع می‌توان گفت که این متد مشتری مدار می‌باشد.
- (2) **یکپارچه سازی مستمر:** در این متدولوژی به دلیل اینکه در چرخه توسعه نرم‌افزار تولید کد جدید بسیار سریع بوده و بخش‌های مختلف محصول نهایی به صورت روزانه یا ساعتی تولید می‌گردند لازم است بین این بخش‌های پیکره‌بندی و یکپارچگی مناسبی ایجاد شود به این معنی که می‌بایست کدهای جدید در سیستم تا حد امکان یکپارچه شوند و تمام تست‌های عملیاتی، بعد از یکپارچه سازی انجام گیرد.

Ø خصوصیات ویژه:

- (1) **بازطراحی:** از خصوصیات بارز XP بازطراحی فرآیندهای موجود در سیستمی که قرار است مکانیزه شود، است. این فعالیت شامل حذف دوباره کاری‌ها، بهبود ارتباطات، ساده کردن روند توسعه نرم‌افزار و ایجاد انعطاف‌پذیری در سیستم می‌باشد.
- (2) **توسعه نرم‌افزار همراه با ابهام:** روش XP توسعه نرم افزار موفق را با وجود ابهامات یا تغییرات دائمی نیازمندیها در تیمهای کوچک تا متوسط فراهم می‌سازد .
- (3) **تکرار پذیری کوتاه:** به منظور تولید نرم‌افزار، کل فرآیند تولید به تکرارهای متعدد با زمانهای کوتاه تقسیم می‌شود. هدف از انجام این کار شکستن محدوده پروژه به فعالیت‌های کوچکتر و قابل کنترل تر می‌باشد.

(4) **تولید کوچک:** به این معنی که یک نسخه اولیه از سیستم بعد از اولین تکرار تولید می‌شود و سپس این قابلیت در نرم‌افزار به وجود می‌آید که نسخه‌های دیگر در فاصله‌ای از چند روز تا چند هفته تولید می‌شوند.

(5) **تغییرات کد نرم‌افزار در هر زمانی و توسط هر فردی:** در این متدولوژی به دلیل خاصیت چابک بودن آن در تغییر کد نرم‌افزار بسیار سریع صورت می‌گیرد و چون کدهای تولید شده بر مبنا استاندارد مورد قبول تمامی تیم توسعه دهنده تولید شده‌اند، مجموعه افراد تیم می‌توانند روند برنامه‌نویسی افراد دیگر را متوجه شده و تغییرات مورد نیاز را در آنها اعمال نمایند. این تغییرات در واقع مطالبی حاصل از تست نرم‌افزار به وسیله کاربر نهایی می‌باشند.

(6) **محیط فیزیکی:** محیط فیزیکی در این روش دارای اهمیت می‌باشد. زیراتوسعه دهندگان باید به صورت face_to_face با یکدیگر همکاری کنند.

(7) **کاهش هزینه:** کاهش هزینه تغییرات از اهداف این روش می‌باشد. زیرا در روشهای سنتی توسعه سیستم مانند متدولوژی SSADM نیازمندی‌ها برای سیستم در آغاز پروژه مشخص می‌شوند و اغلب تا انتهای پروژه ثابت باقی می‌ماند. در این روش نیازمندی‌ها در هر مرحله به صورت تکراری مشخص شده و نیازمندی‌های تعیین شده قبلی مورد بازنگری قرار می‌گیرند. با انجام این کار تیم توسعه دهنده نرم‌افزار سعی می‌کند که هزینه تغییرات نیازمندی‌ها را که در مراحل انتهایی توسعه نرم‌افزار بالا خواهد بود، کاهش دهد.

(8) **زمانبندی بر اساس نوع درخواست کاربر نهایی:** در این متدولوژی تولید کد نرم‌افزار به صورت روزانه و بر اساس درخواست‌های کاربران نهایی است. در واقع می‌توان گفت این درخواست‌ها تعیین کننده عملیات های روزانه برای مدیران و توسعه‌دهندگان را تعیین می‌کند. بنابراین با توجه به این حالت کاربران نهایی سیستم را قبل از هر تولید، تست می‌کنند تا در صورت بروز مشکل در جهت اصلاح آن اقدامات لازم را انجام دهند.

Ø نقاط قوت:

1. برنامه‌نویسی به صورت دوتایی صورت می‌گیرد به این معنی که دو برنامه‌نویس بر روی هربخش از نرم‌افزار کار می‌کنند تا در صورت خارج شدن یکی از تیم، نیاز به مطالعه مجدد کدهای تولید شده نباشد و کار بی‌وقفه ادامه یابد.
2. این روش از نظر تمرکز بر جنبه‌های مختلف یک پروژه از قبیل جنبه‌های اقتصادی، رضایت کاربران نهایی، کیفیت طراحی محصول، بازیابی مستمر، حل مسائل، یادگیری، مدیریت پروژه و مدیریت پرسنل درگیر در پروژه نسبت به دیگر روش‌ها قابل توجه می‌باشد (Cockburn.2000).
3. تولید محصول یکپارچه تست شده در جهت یافتن نقایص و اشکالات محصول.
4. بازخورد سریع در این روش یکی از ویژگی‌های مهم متد XP است که با ابعاد مختلف توسعه سیستم در ارتباط می‌باشد ابعاد مختلف آن شامل سه نوع بازخورد زیر می‌باشند: (سایت wikipedia)

§ بازخورد سیستمی برای توسعه دهندگان: در این نوع از بازخورد

توسعه دهندگان سیستم با استفاده از test case های از پیش طراحی شده به بررسی سیستم می‌پردازند و از صحت عملکرد نرم‌افزار اطمینان حاصل می‌نمایند.

§ بازخورد برای مشتری: این بازخورد عبارت است از تست عملیاتی توسط

مشتری در زمان تحویل سیستم و همزمان با توسعه نرم‌افزار انجام می‌شود همچنین در این بازخورد تست‌کنندگان سیستم که معمولاً از تیم توسعه دهنده می‌باشند نیز به تست عملیاتی سیستم با استفاده از test case های از پیش نوشته شده می‌پردازند.

§ دریافت بازخورد از تیم: این بازخورد زمانی ارائه می‌شود که مشتری

نیاز جدیدی را برای توسعه به تیم توسعه‌دهنده ارائه می‌نماید. در این

حالت تیم توسعه دهنده تخمین زمانی را که برای توسعه و پیاده‌سازی

سیستم لازم است را ارائه می‌نماید.

5. همکاری نزدیک کاربران نهایی و داشتن ارتباطات و مشارکت آنها با تیم توسعه در

جهت پاسخگویی به سوالاتشان به منظور تفهیم روند عملکرد تیم و تعیین رویه

تست نرم‌افزار و اطمینان از پیشرفت توسعه.

6. از دیگر مزایای این متد مستند سازی محدود آن است که موجب صرفه جویی در

زمان و هزینه و کاهش فضا برای نگهداری مستندات می‌شود.

7. کاهش ریسک در پروژه و کاهش محصولات کاری واسطه از دیگر مزایای این روش

می‌باشد. این کاهش زمانی صورت می‌گیرد که بازخوردهای اشاره شده به درستی

به تیم توسعه پروژه ارائه شود.

8. طراحی ساده از خصوصیات مثبت این متد بوده که موجب سریع‌تر و کم هزینه‌تر

بودن آن نسبت به دیگر متدها که دارای طراحی پیچیده هستند، است.

Ø نقاط ضعف:

1. این روش به مدیریت عملی (برنامه عملیاتی) کمتر توجه دارد و تاثیر فرهنگ و

تکنولوژی بر توسعه در آن بسیار بالا است یعنی هر گونه مقاومت از طرف اعضای

پروژه، مدیریت و مشتریان در مقابل اصول XP موجب شکست پروژه می‌شود

(beck 1999b).

2. تیم‌ها باید حتما ارتباط face-to-face داشته باشند (هم مکان بودن اعضای

تیم) که در واقع وابستگی به محیط فیزیکی از معایب این روش می‌باشد.

3. در این روش زمان‌بندی بر اساس درخواست مشتری صورت می‌گیرد و به این دلیل

هزینه توسعه افزایش می‌یابد.

Scrum - 2-3-6

دیدگاه Scrum برای مدیریت فرآیند توسعه نرم‌افزار شکل گرفته است. همچنین این روش از تئوری فرآیند کنترل صنعتی برای توسعه سیستم‌ها که باعث انعطاف پذیری، سازگاری و بهره‌وری بیشتر نرم‌افزار توسعه یافته می‌شود، استفاده می‌کند. همچنین Scrum به بهبود فعالیت‌های مهندسی موجود همچون عمل تست کردن نرم‌افزار در یک سازمان کمک می‌کند، که شامل فعالیت‌های مکرر مدیریت در شناسایی نقایص فرآیند توسعه می‌باشد.

عقیده اساسی Scrum توسعه سیستم در محیط‌هایی است که متغیرهای تکنیکی همچون نیازمندی‌ها، چهارچوب زمانی، منابع و تکنولوژی که در طول پروژه تغییر می‌کند، بروز این تغییرات زمانی اتفاق می‌افتد که محیط توسعه، محیطی غیر قابل پیش بینی و پیچیده است. استفاده از این متد به تیم توسعه دهنده کمک می‌کند تا انعطاف پذیری در مقابل تغییرات را افزایش داده و در پاسخگویی به تغییرات مورد نیاز تسریع ایجاد کند.

Ø تاریخچه

SCRUM یک روش چابک در توسعه نرم‌افزار می‌باشد که بر اساس بازی راگبی نام گذاری شد که در واقع راهی است برای شروع دوباره بازی پس از بروز تخلف و مشکل در مسیر بازی.

در عمل اولین بار SCRUM در سال 1993 میلادی در کمپانی ایسل¹⁵⁵ توسط جف سوترلند¹⁵⁶

، جان اسکومنیوتیل¹⁵⁷، جف مکنا¹⁵⁸، زمانی که فاز تجزیه و طراحی ابزار شی گرا¹⁵⁹ را در پروژه trip

¹⁵⁵ Easel corporation

¹⁵⁶ jeff Sutherland

¹⁵⁷ John Scumniotales

¹⁵⁸ Jeff mekna

¹⁵⁹ OOAD

Round_engineering در کشور امریکا انجام می‌دادند، به کار برده شد. در آن زمان این افراد به روش‌های توسعه نرم‌افزار در جهت توسعه کاربردی سریع¹⁶⁰ و تبدیل آسان نیازمندی‌های محصول به قوانین کاری، احتیاج پیدا کردند.

در همین زمان آقای "کن سچوبر" از این نوع متدلوژی برای مدیریت توسعه نرم‌افزار موسسه‌اش (موسسه ADM) استفاده می‌نمود. این روش به عنوان بهترین گزینه در بسیاری از صنایع شناخته و معرفی شد که بعد ها این اصول به عنوان پایه‌ای برای SCRUM معرفی شدند. تعدادی از شرکت های معروف که از این متدلوژی در پروژه‌هایشان استفاده کردند عبارتند از :

1. Fuji- Xerox

2. Canon

3. Honda

4. NEC

5. Epson

6. Brother

7. 3M

8. Xerox

9. Hewlett- Packard

نتایج این پروژه‌ها توسط "تاکوچی" و "نوناکا" که مشخص کننده ویژگی‌های این متدلوژی به صورت حرفه‌ای بود در مجله "Harvard Business review" در سال 1986 میلادی تحت عنوان متدلوژی SCRUM منتشر شد.

Ø فرآیند

مراحل انجام شده در متدلوژی Scrum شامل 3 فاز است که عبارتند از:

(1) پیش بازی¹⁶¹

¹⁶⁰ prototype
¹⁶¹ Pre-game

(2) توسعه¹⁶²

(3) پس بازی¹⁶³

فارهای سه گانه این متدولوژی توسط schwaber (1995 و schwaber and beedle 2002) به صورت زیر بیان گردید :

• فاز پیش بازی

این فاز شامل دو زیر فاز به نام‌های برنامه‌ریزی و معماری/طراحی سطح بالا می‌باشد .

در بخش برنامه‌ریزی نخست تعاریفی از سیستمی که توسعه یافته است بیان می‌گردد. در این میان لیستی به نام "پس افت محصول"¹⁶⁴ که حاوی همه نیازمندی‌های شناخته شده است فراهم می‌شود. این نیازمندی‌ها ممکن است متعلق به مشتری، بخش فروش، بازاریابی، پشتیبانی مشتری و یا توسعه نرم افزار باشد. در این لیست همه نیازمندی‌ها بر اساس اولویت‌هایشان طبقه‌بندی می‌شوند و میزان تلاش مورد نیاز برای پیاده سازی محصول نهایی تخمین زده می‌شود. اگر درخواست جدیدی از سوی مشتری مطرح شود لیست نیازمندی‌ها به طور مداوم با اقلام جدید و جزئی تر به روز خواهد شد

در این فاز تعاریفی از تیم پروژه، ابزارهای مورد نیاز برای توسعه نرم افزار، ارزیابی ریسک و چگونگی کنترل آنها، درخواست‌های مورد نیاز برای یادگیری و بازبینی‌های مدیریتی، ارائه می‌گردد. در هر مرحله از تکرار، لیستی مطابق با نیازمندی‌های جدید کاربران نهایی توسط تیم توسعه Scrum برای استفاده در تکرار بعدی بازنگری می‌شود.

در فاز معماری و طراحی سطح بالای نرم‌افزار براساس اقلام جاری که در لیست پس افت محصول قراردارند، انجام می‌شود و همچنین بر اساس این داده‌ها،

¹⁶² develope

¹⁶³ Post-game

¹⁶⁴ Product backlog list

تصمیمات کلان برای معماری نرم افزار نیز اتخاذ می شود، برای ارتقای سیستم موجود، تغییرات مورد نیاز در پیاده سازی اقلام پست افت و همچنین مسائلی که در این میان ممکن است رخ دهد شناسایی می شوند و در جلسه بررسی طراحی سیستم، طرح پیشنهادی برای تصمیم گیری و پیاده سازی ارائه می شود.

• فاز توسعه

این فاز یکی از بخش های چابک دیدگاه Scrum می باشد که مانند جعبه سیاه عمل می کند. در این فاز فرآیند انجام غیر قابل پیش بینی است. در فاز توسعه متدولوژی Scrum، تفاوت های محیطی و متغیرهای تکنیکی از جمله چهارچوب زمان، کیفیت، منابع، پیاده سازی تکنولوژی و ابزار و حتی روش های توسعه شناسایی می شوند که امکان دارد این متغیرها در طول انجام فرآیند تغییر کنند. در واقع این گونه متغیرها در بخش اسپرینت مشاهده و کنترل می شوند.

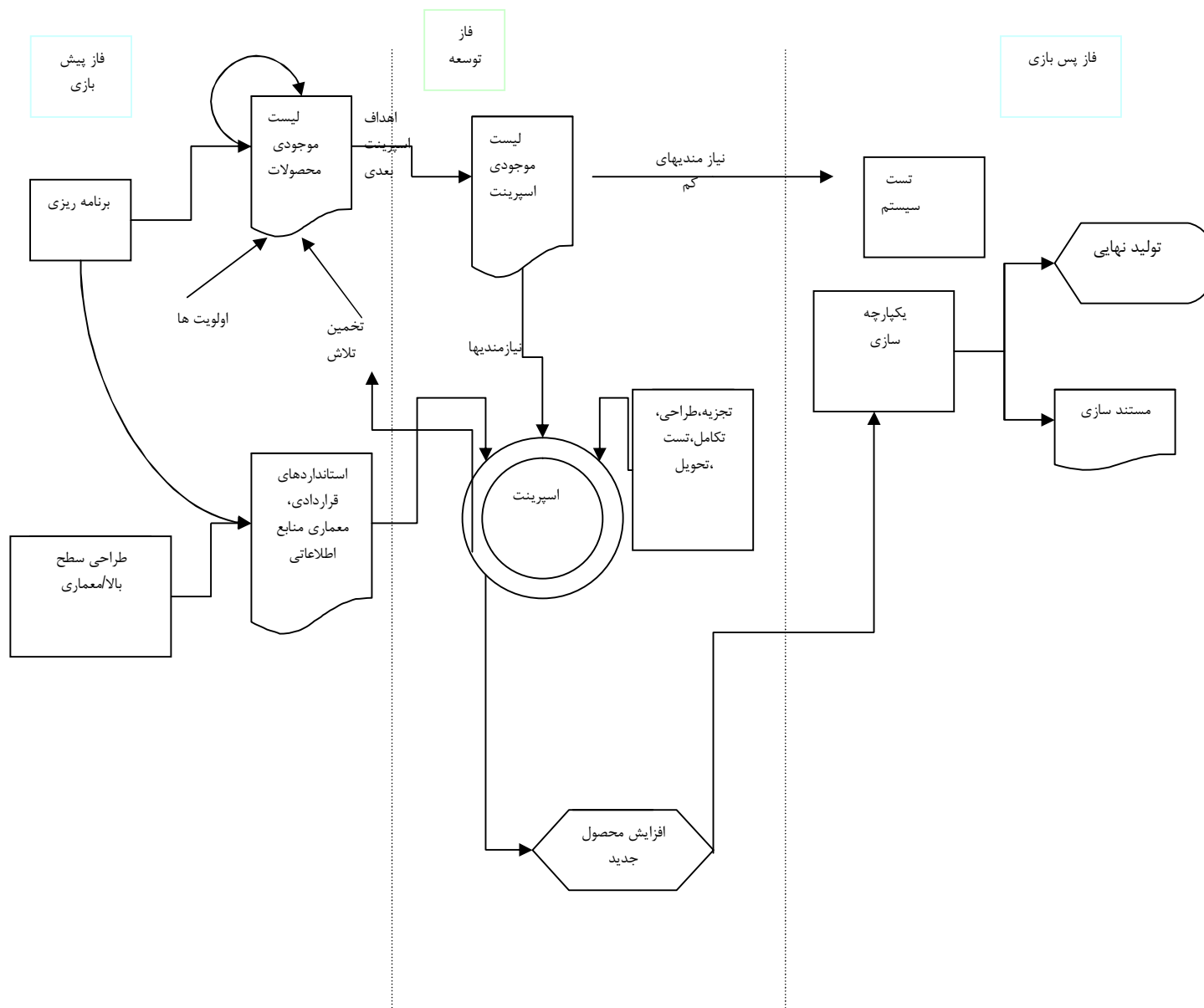
برای برطرف نمودن مشکل غیر قابل پیش بینی بودن روند توسعه متدولوژی Scrum با استفاده از ابزارهای کنترل پروژه، در صورت پدید آمدن تغییرات حالتی منعطف تر به تیم پروژه القا می نماید تا در پاسخگویی به تغییرات تسریع حاصل گردد.

در فاز توسعه، توسعه نرم افزار در چرخه اسپرینت انجام می گیرد. اسپرینت، در واقع چرخه تکرار پذیری است که به طور عملیاتی به وجود آمده و برای تولید سود بیشتر، ارتقاء یافتند و هر چرخه اسپرینت از فازهای قدیمی تولید نرم افزار مانند: فازهای شناسایی نیازمندی ها، تحلیل، طراحی، تکامل و تحویل تشکیل شده است.

همان طور که اشاره شد در فاز پیش بازی تصمیمات در جهت معماری و طراحی سیستم اتخاذ می شود و در نهایت در طول چرخه توسعه اسپرینت، معماری نرم افزار به مرحله اجرا می رسد. هر چرخه اسپرینت بین یک هفته تا یکماه به طول می انجامد.

• فاز پس بازی

مرحله نهایی در متدولوژی scrum بسته تولید نرم افزار می باشد. این فاز زمانی به مرحله اجرا می رسد که تکمیل ارائه متغیرهای محیطی مانند شناسایی نیازمندی ها تایید شوند. پس از تایید کامل شدن نیازمندیهای کاربران نهایی، نرم افزار تولید، یکپارچه سازی و تست می شود و در نهایت تمامی فعالیتهای صورت گرفته مستندسازی می شود.



شکل 85- زیست چرخ فرایند SCRUM

Ø نقاط کلیدی

- 1- متدلوژی scrum از تیم‌های توسعه کوچک خود سامانده¹⁷⁷ تشکیل شده است به طوری که ارتباطات بین افراد درون تیم و دانش ضمنی و غیر رسمی را نیز در تعامل مستقیم با یکدیگر افزایش می‌یابد.
- 2- نظارت سرپرست را بر روی تیم توسعه دهنده نرم افزار کاهش می‌دهد. (سایت dacs.dtic.mil/techs/agile)
- 3- چرخه تولید متدلوژی scrum غالباً 30 روزه می‌باشد.

Ø خصوصیات ویژه

- 1- در این متدلوژی چهارچوب فکری از دیدگاه مشخص و قابل تکرار در روشهای غیر چابک به دیدگاه جدید متدلوژی SCRUM برای توسعه محصول جدید تغییر یافته است.
- 2- این متدلوژی یک روش هدف‌گرا است، به این معنی که برای هر تیم اهداف خاصی تعیین می‌شود و تیم در مسیر انجام پروژه به دنبال محقق نمودن آن اهداف می‌باشد. این سطح از هدف‌گذاری برای تیم‌های توسعه نرم‌افزار خصوصیات محصول را در حد مطلوبی مشخص می‌نماید و محیط قابل کنترلی را برای فعالیت ایجاد می‌نماید.
- 3- به علت انعطاف پذیری بالای این متدلوژی در تحویل محصول، این متد قادر به تغییر برنامه پروژه و دست آوردهای پروژه در هر زمانی با توجه به نیازمندی‌های جدید می‌باشد.
- 4- این متد با استفاده از ابزارهای کنترلی که دارد به کاهش گسیختگی در زمان بندی¹⁷⁸ کل پروژه که باعث به وجود آمدن سطحی مناسب از انعطاف پذیری را فراهم می‌آورد موجب کاهش اتلاف زمان و تحویل به موقع محصول می‌شود.

¹⁷⁷ Self-organizing

¹⁷⁸ time fragmentation

5- دلیل تاکید SCRUM بر مدیریت پروژه عبارت است از جلوگیری از آشوب ایجاد شده در شرایط غیر قابل پیش بینی و پیچیده. این روش همانند XP زمان بندی مراحل انجام کار بر اساس درخواست مشتری طراحی می نماید که خود نیز دارای مزایا و معایبی از جمله رضایت مشتریان و افزایش هزینه توسعه می باشد. (Schwaber 1995)

Ø کاربرد

1- این متدولوژی قابل استفاده در محیط های غیر قابل پیش بینی می باشد. دیدگاه scrum بر پایه توسعه سیستم هایی است که دارای چندین متغیر محیطی و تکنیکی مانند نیازمندیها، چهارچوب زمانی، منابع و تکنولوژی است که در طول فرآیند امکان تغییر هر کدام از این متغیرها وجود دارد. این موارد در واقع فرآیند توسعه را غیر قابل پیش بینی و پیچیده می سازند که نیازمند انعطاف پذیری در پاسخگویی به تغییرات می باشند.

2- اساساً ابزارهای موجود در این متد در فعالیت های مدیریت پروژه های نرم افزاری مورد استفاده قرار می گیرند. اما در موارد خاص برخی پروژه های غیر نرم افزاری نیز با SCRUM قابل مدیریت کردن می باشند.

3- متدولوژی SCRUM برای دو نوع پروژه های کوچک و تقریباً بزرگ کاربرد دارد. تیم پروژه ای که با استفاده از این متد به توسعه نرم افزار می پردازند به طور معمول بین 5 تا 9 نفر و به کار برده می شود و اگر به اعضای بیشتری در جهت توسعه نیاز باشد باید چندین تیم شکل در محیط توسعه نرم افزار شکل بگیرد. (schwaber and beedle(2000)

Ø نقاط قوت

متدولوژی های توسعه سنتی فقط برای پاسخگویی به محیط های توسعه قابل پیش بینی کاربرد دارند و شاهد این نکته آن است که حتی نگرش های جدیدتر مانند متد حلزونی "بوهم" نیز هنوز در پاسخگویی به تغییرات یکباره نیازمندی ها در ابتدای پروژه دارای محدودیت عملکرد می باشند. اما متدولوژی scrum برای ایجاد انعطاف پذیری در مواجهه با ایجاد تغییرات طراحی شده است. در

واقع مکانیزم کنترلی را برای برنامه‌ریزی تولید محصول و سپس مدیریت متغیرها در زمان پیشرفت پروژه فراهم می‌آورد. این متد سازمانها را برای تغییرات پروژه و تحویل محصول در هر زمانی توانا می‌سازد. برای ایجاد سطح مطلوبی از انعطاف پذیری و تسریع در پاسخگویی به تغییرات در این متد تیم‌های توسعه مشارکتی تشکیل می‌شوند که قادر به اشتراک‌گذاری دانش نانوشته و ضمنی خود در مورد توسعه فرآیند می‌باشند.

Ø نقاط ضعف

- 1- این متد برنامه عملیاتی مدونی برای یکپارچه‌سازی و تست نرم‌افزار ندارد.
- 2- این روش به غیر از موارد خاص کمتر در مدیریت پروژه‌های غیر نرم‌افزاری مورد استفاده قرار می‌گیرد و موارد کاربرد آن بیشتر در پروژه‌های نرم‌افزاری می‌باشد.
- 3- هزینه توسعه به علت زمان بندی بر اساس نیاز و درخواست مشتری افزایش می‌یابد.

Crystal - 3-3-6

متدلوژی Crystal مجموعه‌ای از متدلوژی‌ها می‌باشد. خانواده Crystal حاوی تعداد متنوعی از متدلوژی‌هاست که می‌توان با بررسی متدلوژی‌ها، کارآترین متدلوژی را با توجه به ماهیت هر یک از پروژه‌ها انتخاب نمود. روش CRYSTAL از اصولی برای بومی سازی متدلوژی‌ها استفاده می‌کند تا بتواند در پروژه‌های مختلف انعطاف پذیرتر عمل کنند. هر عضو از خانواده Crystal با یک رنگ که نشان دهنده وزن و اهمیت آن متدلوژی است نام‌گذاری شده است. به عنوان مثال هر چه به سمت رنگ‌های تیره‌تر پیش رود، متدلوژی مورد استفاده سنگین‌تر می‌باشد. و این در حالی است که این روش انتخاب مناسب ترین رنگ از متدلوژی را برای یک پروژه براساس اندازه و اهمیتش پیشنهاد می‌کند. در خصوص این مطلب می‌توان گفت که پروژه‌های بزرگ‌تر نیازمند مشارکت بیشتر و متدلوژی سنگین تر نسبت به پروژه‌های کوچک‌ترها می‌باشند .

پروژه‌هایی که از این نوع متدلوژی استفاده می‌کنند همیشه از چرخه توسعه افزایشی که بین 1 تا 3 ماه می‌باشد بهره می‌گیرند. (Cockburn 2002a).

و در همه متدلوژی‌های crystal، ارتباطات محکم و سبکی در محصولات به عنوان یک ارزش مشترک مطرح است که تمامی آنها بر اساس این اصول عمل می‌کنند.

لازم به ذکر است که حجم محصولات جانبی در نظر گرفته شده در متدلوژی همچون برخی از مستندات یا برخی اقدامات در خصوص مدیریت پروژه‌های نرم‌افزار می‌توانند برای تحویل مستمر نرم‌افزار کاهش یابند و همچنین کانال‌های ارتباطی درونی سازمان، در جهت تولید سریع و تحویل به موقع نرم‌افزار برقرار شوند. تعداد کارکنان و مکانهای مورد استفاده برای مذاکرات در جهت توسعه نرم‌افزار نبایستی دارای محدودیت باشد و افراد باید بتوانند در هر ناحیه جغرافیایی با یکدیگر تبادل اطلاعات داشته باشند.

در خانواده متدلوژی crystal پروژه‌ها به دو شکل بحرانی- حیاتی و غیر بحرانی- حیاتی تقسیم می‌شوند. نوع استفاده از این دو شکل دسته‌بندی با توجه به ویژگی انواع پروژه‌ها در دو بعد میزان راحتی تیم توسعه دهنده نرم‌افزار و میزان هزینه‌کرد مالی مورد نیاز برای هر پروژه، تعیین می‌گردد. متدلوژی‌های مورد استفاده در خانواده Crystal با رنگ‌های مختلف کدبندی می‌شوند. در زمانی که حجم پروژه بزرگ نیست و باید از متدلوژی‌های کوچک و سبک استفاده شود از کد clae استفاده می‌شود و این اولویت‌بندی دسته‌ها به همین ترتیب ادامه دارد و دسته بندی‌های بعدی عبارتند از:

Yellow ü

Orange ü

Red ü

Blue ü

Maroon ü

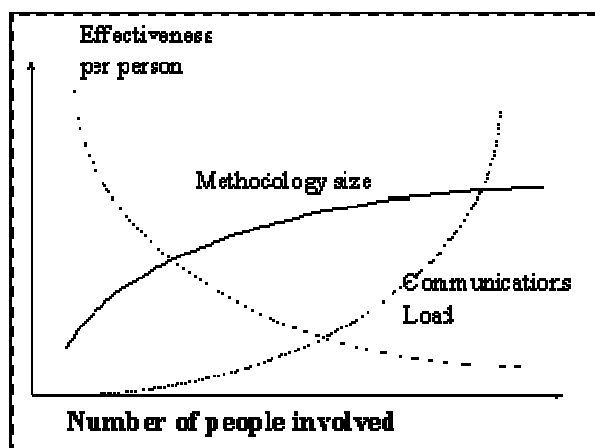
Violet ü

دسته آخر این متدلوژی‌ها برای گروه‌هایی از متدلوژی‌های بزرگ‌تر استفاده می‌شود.

Ø اصول مطرح در پروژه‌های Crystal: (Alistair A.R. Cockburn)

مطالعات بر روی خانواده Crystal حاکی از آن است که 4 اصل در ارتباط با پروژه‌هایی که با استفاده از متدولوژی Crystal انجام می‌شوند وجود دارد که عبارتند از:

اصل 1. در پروژه‌هایی که با استفاده از متد Crystal اجرایی می‌شوند، هرچه اندازه پروژه بزرگتر شود، تعداد افرادی که در پروژه به کار گرفته می‌شوند، افزایش می‌یابد. افزایش سایز پروژه به معنی افزایش نیاز به کنترل و هماهنگی است. در اینگونه پروژه‌های بزرگ ارتباطات بسیار افزایش می‌یابد و پیچیدگی ارتباطات نیز افزایش می‌یابد. همچنین زمانی که تعداد افراد درون یک پروژه افزایش می‌یابد با توجه به اینکه ارتباطات افزایش می‌یابد کارآمدی فردی هرکدام از افراد کمتر می‌شود. شکل 4 به صورت گرافیکی نشان دهنده این اصل است. مهمترین نکته در این اصل عبارت است از اینکه نباید انتظار داشت که متدولوژی که برای تیم‌های کوچک کارآمد است برای پروژه‌هایی که نیاز به تیم بزرگ و یا بالعکس کارآ باشد.



شکل 86- تاثیر اندازه پروژه

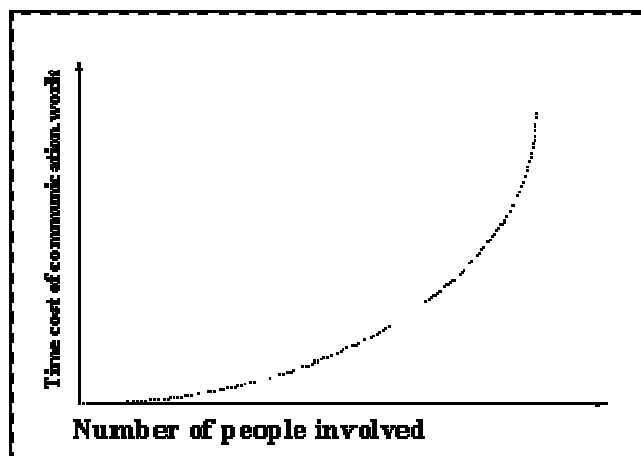
اصل 2. وزن متدولوژی مورد استفاده و مد رنگ بندی آن برای پروژه با اهمیت بالاتر بیشتر می‌شود. در واقع این اصل بیان کننده مقدار ظرفیت متدولوژی در حالت‌های مختلف اندازه پروژه است. اگر هزینه توسعه افزایش یابد، ریسک پروژه افزایش می‌یابد و خطر جدی در استفاده از متدولوژی به

وجود می‌آید در پروژه‌هایی که حالت بحرانی - حیاتی دارند به طور معمول این ریسک به وجود می‌آید.

در این اصل در زمان استفاده از متدلوژی 4 عامل کلیدی زیر دارای به منظور انتخاب تعداد افراد مورد نیاز برای انجام پروژه دارای اهمیت ویژه می‌باشند که عبارتند از:

- فقدان شرایط مطلوب کارکنان جهت توسعه نرم‌افزار
 - فقدان یا کمبود اندوخته احتمالی تیم توسعه نرم‌افزار به منظور پشتیبانی از فرآیند تولید
 - فقدان اندوخته ضروری و مورد نیاز سازمان برای تامین هزینه کرد پروژه
 - فقدان عوامل کنترلی حیاتی به منظور نگه‌داشت وضعیت پروژه
- این اصل بیان می‌کند که تیم توسعه می‌تواند هزینه اضافی توسعه را برای جلوگیری از خطای موجود بین تبادل اطلاعات بین بخش‌های مختلف را تعدیل بخشد. همچنین این اصل با چگالی متدلوژی در اهمیت سیستم‌های نرم‌افزاری ارتباط مستقیم دارد. **چگالی** به این معنی است که دقت در محصولات تولید شده با بازیابی محکم‌تر و انعطاف‌پذیری کمتر، بالاتر می‌باشد.

اصل 3. افزایش وزن متدلوژی مورد استفاده به معنای افزایش هزینه توسعه می‌باشد. به این معنی که افزایش مختصری در اندازه متدلوژی یا چگالی آن، مقدار زیادی هزینه پروژه را افزایش می‌دهد. تمرکز اصلی این اصل بر هزینه‌هایی است که به ازاء انتخاب متدلوژی‌های مختلف بر روند انجام کار تحمیل می‌شود. در واقع می‌توان گفت این اصل توجه زیادی به نوع فعالیت‌های انجام شده در پروژه و نوع همکاری‌های که لازم است در متدلوژی‌های مختلف به وجود آید، ندارد و صرفاً به سود، هزینه و خطرات پروژه تمرکز می‌کند. شکل زیر نرخ رشد هزینه‌ها، زمان و کارهای پروژه را به ازای اضافه شدن نیروی انسانی در پروژه نشان می‌دهد.

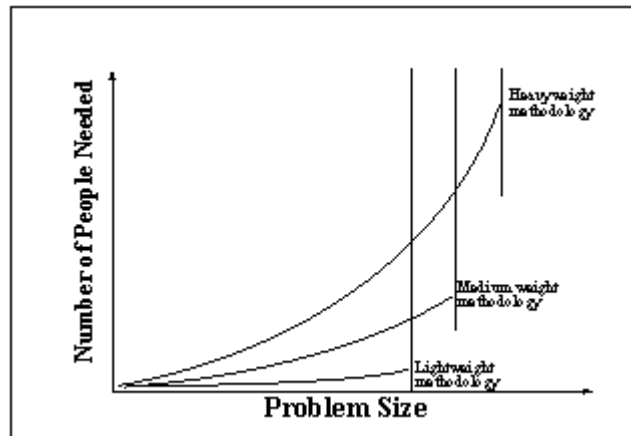


شکل 87- نرخ رشد هزینه زمانی که اندازه متدلوژی رشد می کند

در این میان می توانیم از اصول 1 تا 3 برای سنجش ارتباط بین اندازه متدلوژی و اندازه مسائل استفاده کنیم. در واقع این اصول بیانگر این نکته هستند که مسائل بزرگتر با تعداد افراد بیشتر حل می شود (هر چه اندازه مسائل افزایش یابد اندازه متدلوژی نیز افزایش می یابد).

اندازه پروژه و اندازه متدلوژی یک حلقه بازخورد مثبت ایجاد می کند، به این معنی که با تعداد افراد کمتر به متدلوژی سبکتر احتیاج داریم و وقتی افراد بیشتری در یک پروژه همکاری می کنند لازم است از متدلوژی سنگین تر استفاده شود اما نقطه منفی این استفاده آن است که متدلوژی سنگین بهره وری کار را کاهش می دهد.

از آنجاییکه اندازه پروژه کمتر از اندازه متدلوژی رشد می کند این نکته قابل توجه است که در کجا می توان چگونه مسائلی را حل نمود و فعالیتهای گروهی را مدیریت نمود. این بدان معنی نیست که تیم های کوچک همانند به تیم های بزرگ می توانند مسائل بزرگ را حل کنند بلکه یک فضای همپوشانی وجود دارد که تیم های کوچک با متدلوژی سبکتر می توانند مسائل مشابه را همانند تیم های بزرگتر با متدلوژی سنگین تر حل کنند. شکل زیر در واقع نشانگر نرخ رشد مشابه حل مسائل توسط تیم های با اندازه مختلف پروژه می باشد.



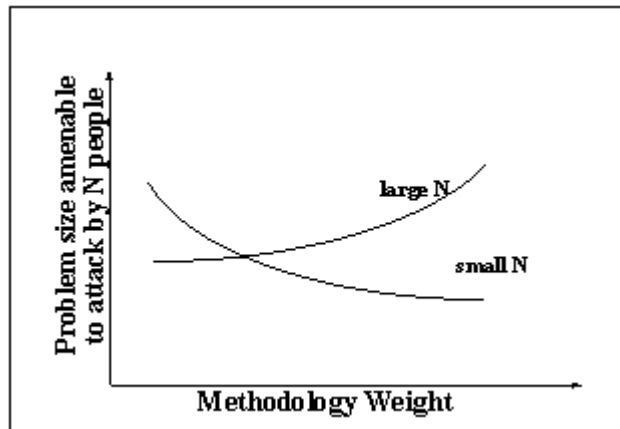
شکل 88- اندازه مسائل و متدلوژی موثر از کارمندان

نکته مهمی که می‌توان از این بررسی‌ها نتیجه‌گیری کرد آن است که در صورتی که تیم‌های کوچک و بزرگ بخواهند از یک متدلوژی با سنگینی مساوی استفاده نمایند، باز اندازه پروژه‌ای که می‌توانند آنرا با استفاده از همین متدلوژی حل نمایند متفاوت خواهد بود شکل 7 به تشریح این موضوع به صورت گرافیکی پرداخته است.

A) در سمت چپ نمودار، تیم کوچک از یک متدلوژی سبک استفاده می‌کند که قادر خواهد بود مسائل بزرگتر را حل کند نسبت به تیم بزرگتر که از همین متدلوژی سبک استفاده می‌کند.

B) در سمت راست نمودار، تیم بزرگتر که از متدلوژی سنگین استفاده می‌کند قادر است مسائل بزرگتر را حل کند نسبت به تیم کوچک تر که از همین متدلوژی سنگین استفاده می‌کند.

با توجه به این مطالب می‌توان گفت در صورتی که یک متدلوژی مطلوب را برای توسعه نرم‌افزار وجود داشته باشد در انواع مسائل ارزش آن متد با رشد تیم کاهش می‌یابد.

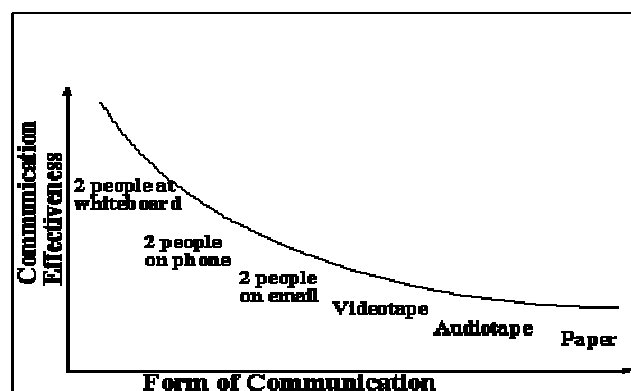


شکل 89- وزن متدلوژی و اندازه مسائل

اصل 4. بهترین حالت برای برقراری ارتباط بین اعضای تیم پروژه به منظور انتقال عقاید و نظرات، ارتباط به دو صورت ارتباط رو در رو و متقابل است. که ارتباطات متقابل از درجه اهمیت بالاتری نسبت به روابط رو در رو برخوردار است.

همانطور که در شکل 8 مشخص شده است این اصل به این نکته اشاره دارد که نزدیکی ارتباط افراد و نشستن آنها در کنار یکدیگر و به صورت مکرر همراه موجب بهتر شدن ارتباطات بین افراد و متقابلاً تسهیل فرآیند توسعه نرم افزار می شود. یکی از شاخص های تسهیل این فرآیند کاهش هزینه توسعه نرم افزار می باشد.

همچنین این اصل به این نکته تاکید دارد که بسته به اندازه پروژه، ایجاد ارتباطات متقابل و رودرو سخت و هزینه بر می باشد.

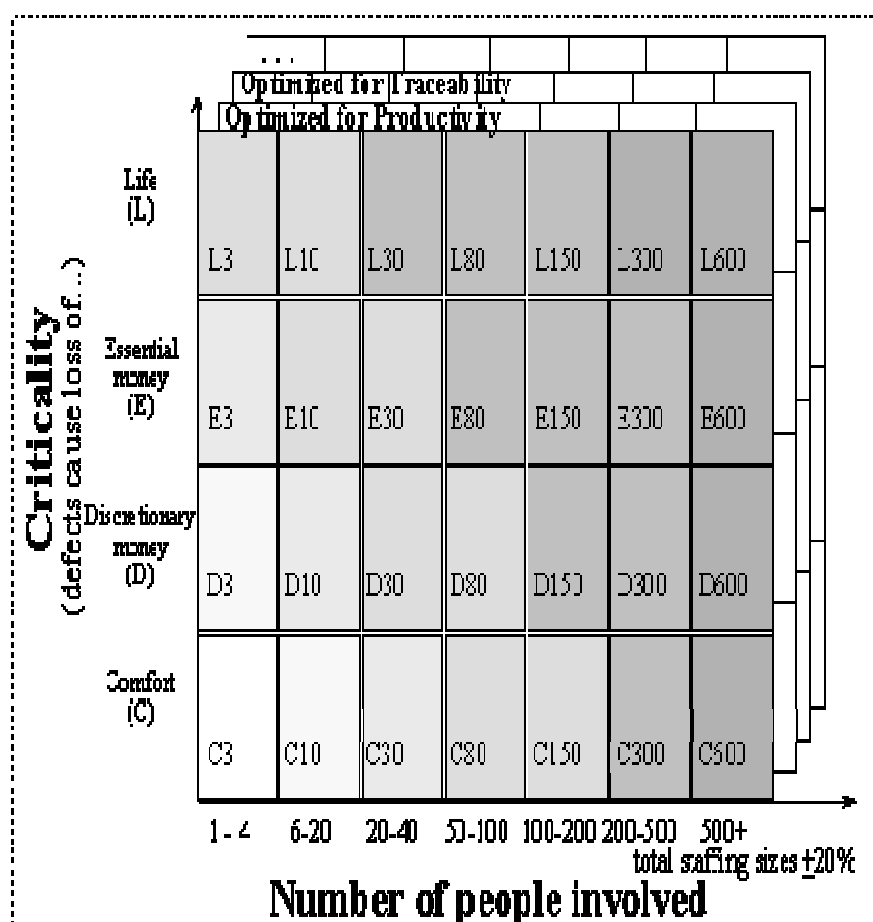


شکل 90- کارآمدی ارتباطات

فرآیند

به طور کلی متدلوژی Crystal بر دو بخش اندازه پروژه و اهمیت پروژه تاکید دارد و خانواده این متدلوژی بر مبنای این دو بعد اصلی دسته بندی می‌شوند. بر مبنای ارتباط بین این دو بعد ماتریسی تشکیل می‌شود که درون هرکدام از سلول‌های این ماتریس، متدلوژی‌های مختلفی قرار می‌گیرند که بسته به اینکه مدیران پروژه توسعه نرم‌افزار به دنبال چه سطحی از بهره‌وری، دقت، قابلیت تکرار و میدان دید باشد از یکی از این متدلوژی‌ها استفاده می‌نماید.

همانطور که در شکل مشخص شده است نمودار CRYSTAL به 7 بخش اندازه پروژه و 4 بخش در مورد اهمیت پروژه تقسیم می‌شود. در این شکل متدلوژی‌های بزرگتر که دارای ارتباطات بیشتر هستند به سمت راست و بالا با کنترل بیشتر رشد می‌کنند و متدهای سبکتر به پایین و چپ حرکت می‌نمایند. لازم به ذکر است حرکت به سمت بالا و راست هزینه توسعه را افزایش می‌دهد.



شکل 91- خانواده متدلوژی CRYSTAL (Cockburn (2002a)

Ø معرفی متدلوژی‌های مطرح در خانواده Crystal

§ متدلوژی Crystal clear:

این متدلوژی برای پروژه خیلی کوچک که حداکثر تا 6 توسعه دهنده را داراست استفاده می‌شود. در این متدلوژی توسعه‌دهندگان باید در یک فضای اداری با ساختار ارتباطی مستقیم مستقر شوند. در این متد با این تعداد توسعه دهنده احتیاج به همکاری تیم‌های فرعی نمی‌باشد. (Cockburn 2002a)

§ متدلوژی Crystal orange:

برای پروژه‌های متوسط با تعداد توسعه‌دهندگان بین 10 تا 40 نفر مورد استفاده قرار می‌گیرد. در این متدلوژی به صورت معمول زمان پروژه بین یک تا دو سال مد نظر قرار می‌گیرد. لازم به ذکر است که در این روش زمان مربوط به بازاریابی و فروش محصول¹⁶⁷ بسیار اهمیت دارد. لازم به ذکر است در این روش نکاتی از قبیل نگاه‌داشت ارتباطات با توسعه‌دهندگان موجود، آتی و پایینی (در صورت تغییر افراد)، تحت کنترل داشتن هزینه و زمان توسعه نرم‌افزار از موارد مورد توجه این روش می‌باشد.

تفاوت عمده بین این دو نوع متدلوژی در آن است که در متد Clear، فقط یک تیم در پروژه وجود دارد و در متد orange چندین تیم‌کاری که ممکن است نوع ماموریت متفاوتی داشته باشند، همکاری می‌کنند. وجه تشابه بین این دو متدلوژی نیز آن است که در هر دو متدلوژی یک شغل ممکن است شامل چندین نقش باشد.

همانطور که در شکل زیر نمایش داده شده است، اندازه تیم و تعداد افرادی که در توسعه نرم‌افزار با یکدیگر مشارکت می‌کنند بر روی بردار X نشان داده شده است. دومین بعد از این نمودار میزان بحرانی

¹⁶⁷ Time_to_Market

بودن پروژه را مورد بررسی قرار می‌دهد که در واقع پتانسیل خطر ایجاد شده توسط نقایص کشف نشده می‌باشد.

در این متدلوژی انتظار می‌رود که ارتباطات و همکاری اجزای متدلوژی‌ها با تغییر شرایط تغییر کنند. همچنین این انتظار نیز وجود دارد که اعتبار و پایایی اجزا تغییر کند.

Ø نکات کلیدی

این متد خانواده‌ای از روش‌هاست که هر کدام دارای یک دسته اصول و ارزش‌های مشابه می‌باشند. همچنین در متدلوژی CRYSTAL انواع مختلف تکنیک‌ها، ارزش‌ها، ابزارها، استانداردها نیز وجود دارد (Abrahmsson 2002).

Ø خصوصیات ویژه

ویژگی عمده این متدلوژی عبارت است از توانایی این متد در انتخاب بهترین گزینه برای تعیین روند انجام یک پروژه توسعه نرم‌افزار با توجه به متغیرهای مختلف در مسیر انجام پروژه.

Ø کاربرد

وقتی یک تیم در یک فضای اداری مشغول به فعالیت می‌باشد باید از متد CLEAR استفاده نماید و در غیر این صورت اگر چندین تیم در یک ساختمان اداری فعالیت می‌کنند از متد ORANGE استفاده می‌شود. در این متدلوژی با توجه به نوع کارکردش می‌توان از کاربران ماهر استفاده نمود زیرا دسترسی به آنها آسان است.

Ø نقاط قوت:

1. همانطور که گفته شد این روش در انتخاب بهینه‌ترین متدلوژی با توجه به شرایط پروژه بسیار توانا می‌باشد.

2. در این متد ارتباطات بهتر و تولید به صورت تکرار پذیر می باشد. همچنین در این متد افزایش بازخورد، نیاز به محصول کاری واسطه را کاهش می دهد و پروژه را نسبت به تغییرات، دارای مطابقت و سازگاری بیشتر می نماید.
3. این روش به ایمنی خروجی پروژه و کارمندان توجه زیادی دارد.

Ø نقاط ضعف

1. فقط 2 روش از 4 روش پیشنهاد شده (YELLOW.RED.ORANGE.CLEAR) در متدلوژی به صورت مستند و دقیق موجود است و دیگر روش ها دارای روند مشخصی نیستند (Abrahmsson 2002).
2. صرفاً تیمهایی که در یک مکان متمرکز باشند می توانند از این روش استفاده کنند و آزاد نبودن تعداد کارکنان و مکان برای مذاکرات از محدودیت های این روش است.
3. این روش فاقد عناصر ارزیابی در متد CLEAR و همچنین فاقد ساختاری مشخص برای خرده تیم ها در متد ORANGE می باشد.

Feature Driven Development - 4-3-6

متدلوژی FDD یک نگرش چابک و سازگار برای توسعه سیستم ها می باشد. این متدلوژی همه فرآیند توسعه نرم افزار را پوشش نمی دهد اما تمرکز زیادی بر فازهای طراحی و ساخت محصول نرم افزار دارد (Palmer and Felsing 2002). همچنین این متد برای کار با دیگر فعالیت های توسعه نرم افزار طراحی شده است. نگرش FDD بر توسعه تکرارپذیر با بهترین عمل در صنعت تاکید دارد. Palmer (and Felsing 2002)

Ø تاریخچه

FDD در ابتدا توسط Jeff De Luca برای بدست آوردن نیازهای مشخص یک پروژه توسعه نرم‌افزاری 15 روزه به همراه تیم توسعه دهنده 50 نفره در سال 1997 میلادی در بانک بزرگ سنگاپور اختراع شد. "جف دو لوکا" در توسعه این متد، مجموعه‌ای از پنج فرآیند را که شامل توسعه یک مدل کلی¹⁶⁸، لیست کردن¹⁶⁹، برنامه‌ریزی کردن¹⁷⁰، طراحی¹⁷¹ و ساختن خصوصیات¹⁷² می‌باشد را تدوین نمود. اولین فرآیند از این مجموعه یعنی توسعه مدل کلی از نگرش "پیتر کود" برای مدلسازی اهداف، اقتباس گردید. وی فرآیند دوم را از نظریه "پیتر کود" در استفاده نمودن از لیست خصوصیات مدیریت نیازمندی‌های عملیاتی و وظایف توسعه، اقتباس نمود. 3 فرآیند های دیگر در واقع نتایج تجارب جف دی لوکا در پروژه‌های توسعه نرم‌افزاری وی می‌باشند.

متدلوژی FDD اولین بار در به جهانیان در کتاب Java Modeling in Color with UML توسط پیتر کود، اریک لیفیر و جف دی لوکا در سال 1999 میلادی معرفی گردید.

Ø فرآیند

متدلوژی FDD از 5 فرآیند متوالی در طول طراحی و ساخت نرم‌افزار تشکیل شده است. که عبارتند از:

1- توسعه یک مدل کلی

2- لیست کردن خصوصیات

3- برنامه‌ریزی کردن

4- طراحی

¹⁶⁸ Develop an overall model

¹⁶⁹ Build a feature list

¹⁷⁰ Plan by feature

¹⁷¹ Design by feature

¹⁷² Build by feature

5- ساختن خصوصیات

در ادامه به بررسی هر کدام از نکات زیر پرداخته خواهد شد:

• فاز توسعه مدل کلی

زمانی که فاز توسعه مدل کلی شروع می شود خبرگان محیطی سازمان¹⁷³ که می توانند کاربر، مشتری، حامیان، تحلیل گران کسب و کار یا ترکیبی از آنها باشند از اهداف، مفاهیم و نیازمندی های سیستم برای ساخت آگاهی پیدا می کنند. هر چند که متدلوژی FDD به طور واضح جمع آوری و مدیریت نیازمندی ها را بیان نمی کند، احتمالاً نیازمندی های مستند شده مانند موارد کاربرد¹⁷⁴ و یا خصوصیات عملیاتی در این فاز قرار دارند. خبرگان محلی یک مسیر انجام کار¹⁷⁵ را ارائه می دهند. این مسیر ها در واقع بخش بندی های مختلف توسعه نرم افزار است که در هر پروژه چندین مسیر توسط خبرگان تعیین می شود و با پایان یکی دیگری آغاز می گردد. با استفاده از این سند اعضای تیم و معمار نرم افزار¹⁷⁶ که تصمیمات نهایی را در مورد طراحی محصول نهایی می گیرد از توضیحات سطح بالا و فرآیندی نرم افزار آگاه می شوند. قلمرو کلی به فضاهای مختلف تقسیم می شود و جزییات این مسیر توسط هر یک از اعضای قلمرو نگهداری می شود بعد از تکمیل سند هر مسیر کاری، یک تیم توسعه در گروه های کوچک برای تولید مدل های اهداف در فضای قلمرو تحت کنترل شان فعالیت می کنند. سپس تیم توسعه نرم افزار مدل های تدوین شده را مورد بررسی مجدد قرار می دهند و در نهایت بر روی مدل های مناسبی از اهداف تصمیم گیری نهایی را انجام می دهند.

• فاز ساختن لیستی از خصوصیات

¹⁷³ Domain expert

¹⁷⁴ use case

¹⁷⁵ walk through

¹⁷⁶ Chief architect

سه ابزار مسیر کاری‌های مختلف، مدل‌های تولید شده بر اساس اهداف و مستندات نیازمندی‌های موجود، منابع مناسبی برای ساختن لیستی جامع از خصوصیات نرم‌افزار توسعه یافته می‌باشند.

در این لیست تیم توسعه نرم‌افزار "فعالیت‌های دارای ارزش در نظر مشتری"¹⁷⁷ را تشریح می‌نماید. به منظور ایجاد لیست نیازمندی‌ها، ابتدا قلمروهای مطرح در پروژه تعیین می‌شود سپس برای هر قلمرو تیم پروژه تعدادی زیر قلمرو تعریف می‌نماید و برای هر کدام از زیر قلمرو مجموعه‌ای از خصوصیات اصلی¹⁷⁸ تعریف می‌شود و که همین مجموعه به مجموعه‌ای از خصوصیات جزئی‌تر تقسیم بندی می‌شوند.

این زیر مجموعه‌ها انواع فعالیت‌های درون فضای قلمرو پروژه را تشریح می‌کنند. لیست خصوصیات توسط کاربرنهایی و حامیان نرم‌افزار برای اطمینان از اعتبار و صحت‌گذاری مورد باز بینی قرار می‌گیرند.

• فاز برنامه ریزی توسط خصوصیات

فاز برنامه ریزی توسط خصوصیات شامل ایجاد کردن یک برنامه سطح بالا برای مجموعه خصوصیات شناسایی شده می‌باشد که توسط رئیس بخش برنامه‌نویسی¹⁷⁹ با توجه به اهمیت و وابستگی‌شان اولویت بندی می‌شوند. بنابراین گروه‌های خصوصیات شناسایی شده در فرآیند توسعه مدل کلی به هر یک از توسعه‌دهندگان نرم‌افزار جهت توسعه گان بخش واگذار می‌شود. به عنوان مثال در فرآیند توسعه نرم‌افزار این گروه‌های خصوصیات به class owner واگذار می‌شود که این افراد تحت سرپرستی رئیس بخش برنامه نویسان می‌باشند و وظایفشان عبارت است از طراحی، کدنویسی، تست و مستند سازی. این افراد توسعه Class هایی را بر عهده دارند که مالکشان می‌باشند. palmer and felsing (2002)

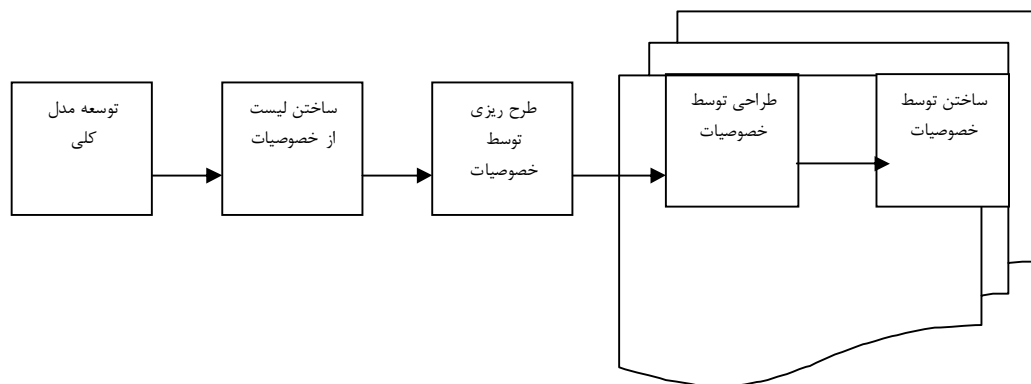
¹⁷⁷ Client valued functions

¹⁷⁸ major feature sets

¹⁷⁹ chief programmer

• طراحی توسط خصوصیات و ساخت توسط خصوصیات

در فاز طراحی و ساخت توسط خصوصیات، گروه کوچکی از خصوصیات از مجموعه خصوصیات انتخاب شده و تیم‌هایی برای توسعه این خصوصیات class ownerها تشکیل می‌شوند. این فاز یک فرآیند تکرار پذیر بوده و چرخه تکرار بین چند روز تا 2 هفته به طور میانگین به طول می‌انجامد. لازم به ذکر است که چندین تیم از خصوصیات می‌توانند به طور همزمان مجموعه‌ای از خصوصیات‌شان را طراحی و سپس توسعه دهند. فرآیند تکرار شامل وظایفی همچون بازرسی، کد نویسی، تست، یکپارچه‌سازی و بازرسی کدهای تولید شده می‌باشد. در این گام پس از یک تکرار موفق که خصوصیات گروه‌بندی



شکل 92- فرایند FDD
(Palmer and felsing 2002)

تکمیل شده و توسعه یافته‌اند در صورتی که پروژه تکمیل نشده باشد، چرخه تکرار و ساخت جدیدی با یک گروه جدید از خصوصیات به منظور تکمیل پروژه آغاز می‌گردد.

Ø نکات کلیدی

1. متدولوژی FDD متدی هدفگرا است به این معنی که در این متد توسعه اجزاء بر اساس اهدافی است که در پروژه در نظر گرفته شده است.

2. تکرارپذیری متد؛ در این متد از چرخه تکرار برای طراحی و ساخت خصوصیات استفاده می‌شود به طوری که در هر چرخه عملیات‌های بازرسی، کدنویسی، تست، یکپارچه‌سازی و بازنگری کدهای تولید شده صورت می‌گیرد و در نهایت محصول اصلی با استفاده از حلقه‌های تکرار مختلف توسعه می‌یابد.

Ø خصوصیات ویژه

1. نگرش FDD کل فرآیند توسعه نرم افزار را پوشش نمی‌دهد بلکه بر فازهای طراحی و ساخت تمرکز دارد.
2. در این متد مدلسازی بر اساس اهداف صورت می‌گیرد.
3. توسعه نرم افزار توسط متدلوژی FDD به سادگی صورت می‌گیرد.
4. زمان استفاده از چرخه متدلوژی FDD بین 2 روز تا 2 هفته می‌باشد (brooks 1995 and davis 1998).
5. این روش بر کیفیت در سرتاسر فرآیند توسعه تاکید دارد. به این صورت که در بازبینی‌های تکرارپذیر در بخش‌های مختلف تولید نرم‌افزار میزان کیفیت پروژه به دقت تخمین زده می‌شود.
6. برنامه‌ریزی مراحل انجام کار همچون متدلوژی XP و Scrum بر اساس درخواست مشتری و بر اساس فناوری است.
7. بخش تکرارپذیر فرآیندهای FDD یعنی فاز طراحی و ساخت، توسعه چابک را با انطباق پذیری سریع در مقابل تغییرات نیازمندی‌های کاربران نهایی و نیازمندی‌های کسب و کار پشتیبانی می‌شود.

Ø کاربرد

- 1- وجود کنترل دقیق: زمانی که در پروژه‌های با بازه زمانی کوتاه مدت احتیاج است کیفیت در کلیه مراحل به صورت دقیق و تحت کنترل باشد، می‌توان از متد FDD استفاده نمود.

2- FDD برای شروع پروژه‌های جدید، ارتقای پروژه‌ها، بهبود کیفیت کدهای تولید شده موجود و

وظایف پروژه با ایجاد یک نسخه دوم از درخواست موجود مناسب می باشند. Palmer and

Felsing(2002)

3- قالب استفاده در تیم‌های بزرگ توسعه نرم‌افزار می باشد.

Ø نقاط قوت

1- بازبینی دقیق فرآیند پروژه

2- انطباق پذیری با تغییرات

3- تاکید بر کیفیت در سراسر فرآیند

4- اطمینان از اینکه سیستم همیشه و در همه حال اجرا و در دسترس می باشد و در این متد

همواره اطلاعات مربوط به خصوصیات ایجاد شده جدید در دسترس می باشد. Abrahamsson

(2002)

Ø نقاط ضعف

1- روش FDD صرفاً بر فاز طراحی و پیاده سازی نرم افزار تاکید دارد و همه فعالیت‌های توسعه

نرم افزار را پوشش نمی دهد.

2- چون برنامه ریزی پیشرفت پروژه و تحویل محصول بر اساس درخواست مشتری است، در این

متد هزینه توسعه نرم افزار بالا می باشد.

Dynamic Systems Development Method -5-3-6

روش توسعه سیستم‌های پویا¹⁸⁰ بهترین چهارچوب اختصاصی و غیرانتفاعی برای توسعه کاربردی سریع است این متد اولین بار در انگلستان توسط کنسرسیوم DSDM توسعه یافته و مورد استفاده قرار گرفته است. (Stapleton 1997)

Ø تاریخچه

متدولوژی DSDM در کشور انگلستان در سال 1990 میلادی توسط کنسرسیوم DSDM که شامل تعدادی از فروشندگان و خبرگان با تجربه در توسعه سیستم‌های اطلاعاتی بود توسعه یافت. اولین نسخه این متد در ژانویه 1995 کامل گردید و سپس در فوریه سال 1995 انتشار یافت.

Ø فرآیند

متدولوژی DSDM شامل 5 فاز اصلی می‌باشد که عبارتند از:

- 1- فاز مطالعه امکان‌سنجی¹⁸¹
- 2- فاز مطالعه کسب و کار¹⁸²
- 3- فاز تکرار پذیری مدل عملیاتی¹⁸³
- 4- فاز تکرارپذیری طراحی و ساخت¹⁸⁴
- 5- فاز پیاده‌سازی¹⁸⁵

¹⁸⁰ DSDM

¹⁸¹ feasibility

¹⁸² Business study

¹⁸³ Functional model iteration

¹⁸⁴ Design and build iteration

¹⁸⁵ implementation

که دو فاز نخست آن به صورت متوالی عمل می‌کنند و فقط یکبار انجام می‌شوند به این معنی که این فازها در مسیر انجام پروژه تکرارپذیر نمی‌باشند و سه فاز دیگر تکرارپذیر و افزایشی می‌باشند. در ادامه به بررسی این 5 فاز پرداخته خواهد شد.

• فاز مطالعه امکان سنجی

در فاز امکان سنجی مناسب بودن متدولوژی DSDM برای پروژه توسعه نرم‌افزار تعیین می‌گردد و درمورد انواع پروژه‌های توسعه نرم‌افزار، مجموعه سازمان و افراد مرتبط با نرم‌افزار مورد بررسی قرار می‌گیرد و در نهایت در مورد اینکه آیا از متدولوژی DSDM استفاده شود یاخیر تصمیمات لازم اتخاذ می‌شود.

در این فاز دو نوع محصول کاری که عبارتند از گزارش امکان سنجی و طرح کلی برای توسعه ایجاد می‌شود.

در ضمن در این فاز ممکن است یک نمونه سریع اولیه، ایجاد شود که اگر شرایط کسب و کار و یا تکنولوژی مورد استفاده به درستی شناخته نشده باشد و یا مطابق با نیازمندی‌های کاربران نهایی نباشد، بتوان در محدوده پروژه را تعیین نمود و در مورد فعالیت‌های مطرح در فازهای بعدی تصمیمات لازم را اتخاذ نمود. این فاز بیشتر از چند هفته به طول نمی‌انجامد.

• فاز مطالعه کسب و کار

در فاز مطالعه کسب و کار ویژگی‌های ضروری برای کسب و کار و تکنولوژی، تجزیه و تحلیل می‌شوند. سازماندهی کارگاه‌های کاری در این فاز صورت می‌پذیرد. کارگاه‌های کاری مکان‌هایی هستند که تعدادی از مشتریان به منظور بررسی رویه‌های مرتبط با نرم‌افزار جمع می‌شوند و نظرات خود را نسبت به

محصول تولید شده و و بر روی اولویت بندی نیازمندی‌ها ابراز می‌نمایند. در این مکانها فرآیندهای کسب و کار و دسته‌بندی کاربران در "محدوده تعریف فضای کسب و کار"¹⁸⁶ توضیح داده می‌شوند. در این فاز دو نوع خروجی حاصل می‌شود که عبارتند از:

ü مفاهیم معماری سیستم: در این بخش معماری کلان نرم‌افزار تدوین می‌شود. این سند معماری در متدلوژی DSDM این قابلیت را دارد که بر اساس تغییرات محیط تا حد امکان نیازمندی‌هایش تغییر نماید.

ü طرح توسعه یک نمونه سریع: در این بخش استراتژی‌های توسعه پروتوتایپ و روند مدیریت تغییرات در این نمونه تدوین می‌گردد.

• فاز تکرار پذیری مدل عملیاتی

این فاز اولین فاز تکرار پذیر و افزایشی می‌باشد. در هر تکرار محتویات و نگرش برای تکرار پذیری برنامه‌ریزی می‌شود و نتیجه در چرخه تکرار بعدی تجزیه و تحلیل می‌شود. در این فاز هم تجزیه و هم کدنویسی انجام می‌گیرد و پروتوتایپ ساخته می‌شود و بازخوردهای بدست آمده از این نمونه در بهبود تجزیه و تحلیل مدل‌ها استفاده می‌شود.

لازم به ذکر است در این فاز یکی از مواردی که بر روی آن تاکید زیادی صورت می‌گیرد، تست کردن مستمر نرم‌افزار می‌باشد.

در این فاز برای مدل عملیاتی 4 نوع خروجی حاصل می‌شود که عبارتند از:

ü فعالیت‌های اولویت‌بندی شده: در این بخش لیستی از عملیات‌های اولویت بندی شده فراهم می‌گردد که در انتهای هر چرخه تکرار تحویل داده می‌شوند.

¹⁸⁶ Business Area Definition

ü مستندات بازبینی نمونه‌های عملیاتی: توضیحات و نظرات مشتریان را درمورد وضع موجود نرم‌افزار جمع‌آوری می‌شود و به عنوان ورودی در تکرارهای بعدی متوالی مورد استفاده قرار می‌گیرد.

ü نیازمندی‌های غیر عملیاتی: در این فاز نیازمندی‌های غیر عملیاتی لیست می‌شوند و به عنوان اطلاعات ورودی فازهای بعد مورد استفاده قرار می‌گیرند.

ü تحلیل ریسک توسعه آتی: در طول پروژه مستنداتی که در بخش‌های اولیه فاز تکرارپذیری مدل عملیاتی تولید می‌شوند دارای اهمیت فراوانی هستند چراکه از این مستند در فازهای بعدی همچون فازهای طراحی و ساخت مورد استفاده قرار می‌گیرند. بدین منظور لازم است در ابتدای پروژه بر روی مستندات تولید شده آنالیز ریسک صورت پذیرد تا در بخش‌های مختلفی که قرار است از نتایج این مستندات استفاده شود ابهامات استفاده کاهش یابد. چراکه هرچه خطاهای مربوط به تحلیل‌های اولیه در مستندات دیرتر در مسیر پروژه خود را نمایان سازند هزینه‌های تحمیل شده به پروژه افزایش می‌یابد.

• فاز تکرارپذیری طراحی و ساخت

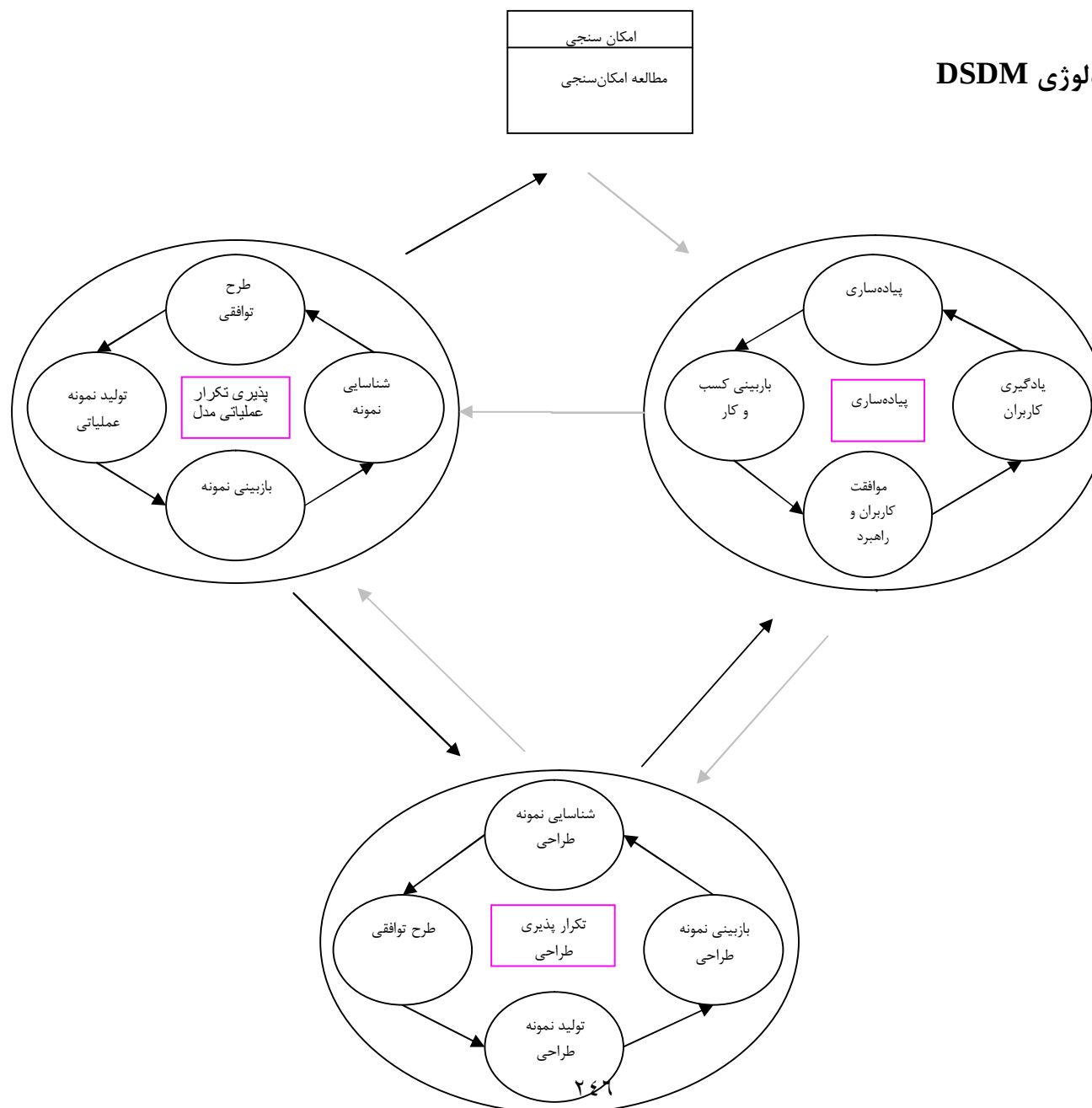
در واقع این فاز گام اصلی توسعه نرم‌افزار در این متدلوژی می باشد. در این فاز نرم‌افزار به طور کامل توسعه می یابد و تست می شود. همچنین در این مرحله طراحی‌های ایجاد شده و پروتوتایپ‌های تولید شده قبلی مجدداً توسط کاربران نهایی مورد بررسی و بازنگری قرار می‌گیرد و نتایج بررسی و اصلاحات آنها به عنوان ورودی توسعه آتی محصول براساس درخواست و نیازمندی‌های کاربران نهایی مورد استفاده قرار می‌گیرد.

• فاز پیاده سازی

در این فاز نرم افزار تولیدی از محیط توسعه به محیط واقعی انتقال داده می شود و در محیط کار کاربران نهایی به صورت عملیاتی راه اندازی می شود. پیش از راه اندازی کامل نرم افزار به کاربران نهایی چگونگی استفاده از نرم افزار آموزش داده می شود و در نهایت نرم افزار به آنها تحویل داده می شود. خروجی فاز پیاده سازی علاوه بر تحویل نرم افزار، شامل راهنمای کاربری استفاده از نرم افزار و گزارشی از بازبینی پروژه می باشد.

شکل 93- زیست چرخ متدلوژی DSDM

(Stapleton 1997)



Ø نکات کلیدی

1. در متدولوژی DSDM زمان عملیات‌های توسعه نرم‌افزار متغیر می‌باشند و کنترل آنها توسط چهارچوب زمانی صورت می‌پذیرد.
2. در این متد تیم‌های توسعه نرم‌افزار دارای اختیار عمل بالایی می‌باشند.
3. متدولوژی DSDM توسط اعضای کنسرسیوم که همان اعضای شرکت می‌باشند توسعه و نگهداری می‌شود و فقط اعضای هیئت فعال به نظام نامه DSDM¹⁸⁷ دسترسی دارند.

Ø خصوصیات ویژه

- تاکید متدولوژی DSDM بر توسعه نمونه اولیه سریع می‌باشد به طوری که بر خلاف گذشته که نمونه‌ها را دور می‌ریختند، امروزه تا تکمیل محصول نهایی از آن نگهداری می‌شود تا بتوان از آن در مواردی دیگری از کسب و کاری که قرار است مکانیزه گردد، استفاده شود.
- دیدگاه بنیادی متدولوژی DSDM بر خلاف ثابت نگه داشتن عملیات محصول و سپس تنظیم و تعدیل زمان و منابع بر این اساس، ترجیح می‌دهد که زمان و منابع را ثابت و سپس عملیات را بر اساس آن تعدیل و تنظیم نماید. طراحی، تست، یکپارچه‌سازی و تحویل نرم‌افزار در چهارچوب زمانی ثابت و کوتاه به طور مطلوب صورت می‌پذیرد.
- این روش دارای چرخه‌های تکرار پذیر است. یکی از مزایای تکرارپذیری ایجاد ارتباطات با مشتریان/کاربران نهایی، توسعه دهندگان و افراد درون سیستم می‌باشد و نرم‌افزار در تکرارهای آتی بهبود می‌یابد.
- توسعه نرم‌افزار در این متد به صورت افزایشی است؛ به طوری که سیستم به زیر سیستم‌های کوچکتر عملیاتی تقسیم‌بندی می‌شود و عملیات‌های جدید در هر تکرار جدید اضافه می‌شوند.

¹⁸⁷ White paper

- اساس متدلوژی DSDM ایجاد چهارچوبی برای گروه توسعه می‌باشد که چه کاری را باید انجام دهند و چرا موضوعات مورد بررسی اهمیت دارند. اما نسبت به این نکته که چطور باید کار را انجام دهند تاکید ندارد.

Ø کاربرد

- 1- نوع عملکرد متدلوژی DSDM در پروژه‌های مکانیزه نمودن یک کسب و کار نسبت به پروژه های مهندسی و علمی ساده تر است (stapleton 1997).
- 2- اگر محیطی دارای کاربران فعال باشد می توان از این متد استفاده نمود.
- 3- این متدلوژی چهارچوبی برای کنترل و توسعه مشارکتی (JAD) است. در واقع JAD درروش DSDM برای ایجاد درک درستی از سیستم جدید در شروع هر پروژه تشکیل می‌شود. JAD همکاری، کار گروهی و درک را در میان گروه‌های مختلف شرکت کننده افزایش می‌دهد. شرکت کنندگان با دیدگاه‌ها و اطلاعات مختلف، انواع بخش‌های سیستم جدید را ملاحظه و نیازمندی‌های اساسی آن را فراهم می‌کنند و کاربران را به شرکت کردن دعوت و اعتمادشان را جلب می‌نمایند.
- 4- DSDM در پروژه های بزرگ و کوچک به کار برده می‌شود و با استفاده از این متدلوژی سیستم‌های بزرگ می‌تواند به اجزای کوچک‌تر تقسیم شود به طوری که توسط تیم های کوچک تر توسعه یابد. لازم به ذکر است که اگر بخواهیم از DSDM در پروژه های زیر ساختاری بزرگ مانند یکپارچه‌سازی کاربردی سازمانها و میان افزارها مورد استفاده قرار گیرد نیازمند بومی سازی بیشتری می باشد (Abrahmsson 2002).

Ø مشخصات تیم توسعه دهنده نرم افزار

اندازه تیم در این روش بین 2 تا 6 نفر متغیر می باشد و ممکن است چندین تیم در یک پروژه مشغول به کار باشند. در هر تیم کاربر و توسعه دهنده از اعضای ثابت تیم توسعه هستند. به طور معمول پرسنلی که در اینگونه پروژه ها درگیر می شوند لازم است کاربران فعال و پر انرژی باشند.

Ø نقاط قوت

- 1- در متد DSDM عملیات تست در کل زیست چرخ حیات نرم افزار انجام می شود که باعث می شود نقایص و مشکلات کشف شده و در صورت نیاز تغییراتی بر روی آنها اعمال شود.
- 2- ارتباطات مشارکتی اعضای تیم در جلسات JAD در جهت فرهنگ سازی و افزایش انگیزش و تسریع در روند پروژه در این متد حائز اهمیت است.
- 3- در روش های سنتی، طراحی حجیم بوده و از بالا به پایین صورت می گیرد در صورتی که در متدولوژی DSDM به منافع حاصل از توسعه نرم افزار در کسب و کار، جلوگیری از دوباره کاری و یکپارچه سازی توجه می شود و در نتیجه هزینه ها کاهش یافته و سود افزایش می یابد.
- 4- این متد بر خلاف روش های سنتی در مورد زمان تحویل نرم افزار با کاربران نهایی همکاری می کند.
- 5- متدولوژی DSDM، مستند سازی دقیق در مورد محصول را تجویز نمی کنند در صورتی که در آن لیستی از اهداف و معیارهای کیفیت دارای اهمیت است و از توجه اکید به جزئیات قرارداد خودداری می شود.

Ø نقاط ضعف

- 1- متدولوژی DSDM برای سیستم های مهندسی و علمی توصیه نمی شود و عموماً در پروژه های مکانیزه نمودن روال های یک کسب و کار کاربرد دارد.

2- همانطور که در بالا اشاره شد در این روش فقط اعضای کنسرسیوم به نظام نامه DSDM

دسترسی دارند (Abrahmsson 2002).

Adaptive Software Development - 6-3-6

متدلوژی ASD اساساً بر توسعه سیستم‌های بزرگ و پیچیده تمرکز دارد. این متد بر توسعه تکرارپذیر افزایشی با ایجاد نمونه ثابت تأکید دارد و آشوب حاصل از محیط‌های پیچیده را تعدیل می‌نماید و چهارچوبی با هدایت کامل در جهت جلوگیری از شکست پروژه را فراهم می‌آورد

تاریخچه Ø

متدلوژی ASD در سال 1992 میلادی توسط جیمز های اسمیت¹⁸⁸ و بایر¹⁸⁹ در کمپانی Wall Street brokerage house to airline to telecommunication توسعه یافت.

فرآیند Ø

همانطور که در شکل 94 مشاهده می‌شود متدلوژی ASD از 3 فاز اصلی تشکیل یافته است که عبارتند از:

(1) تدبیر¹⁹⁰

(2) مشارکت¹⁹¹

(3) یادگیری¹⁹²

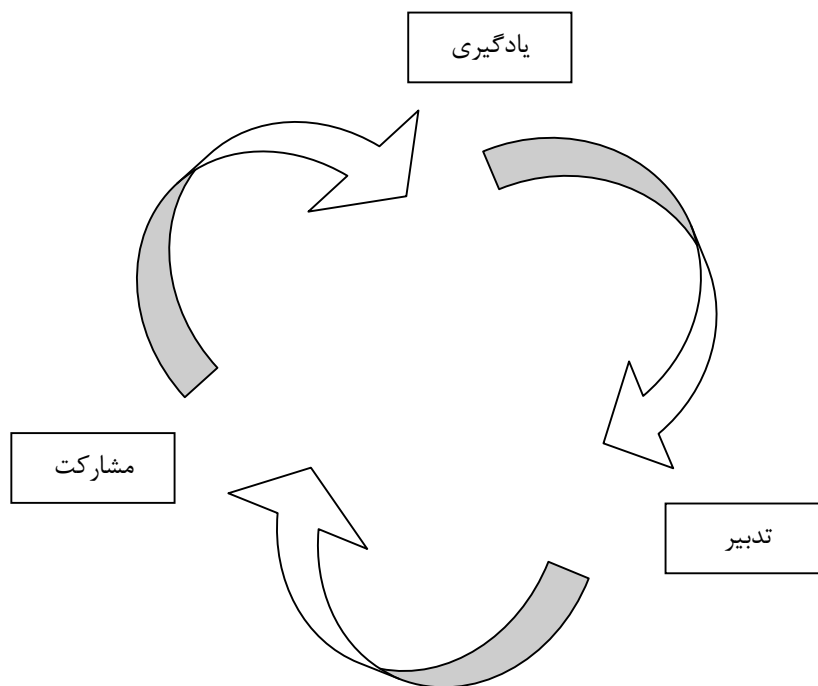
¹⁸⁸ James A. Highsmith

¹⁸⁹ Bayer

¹⁹⁰ speculate

¹⁹¹ collaborate

¹⁹² learn



شکل 94- چرخه Highsmith 2000 ASD، (p.18)

در ادامه به بررسی هرکدام از این فازها پرداخته خواهد شد.

• فاز تدبیر

این فاز در متدولوژی ASD به جای فاز برنامه‌ریزی که در دیگر متدولوژی‌ها کاربرد دارد، مورد استفاده قرار می‌گیرد. زمانی از فاز تدبیر استفاده می‌شود که عموماً ابهامات به عنوان یک ضعف در پروژه تلقی گردند و انحرافات نشان دهنده شکست پروژه باشند. در این فاز مسائل و مشکلات در ارتباط با نرم‌افزار بیان می‌گردد و برای حل و برطرف نمودن آنها برنامه‌ریزی انجام می‌گردد.

• فاز مشارکت

در فاز مشارکت تاکید زیادی بر کار تیمی برای توسعه نرم افزار با تغییرات بالا وجود دارد.

• فاز یادگیری

در این فاز اطلاعات جمع آوری شده بررسی می شود و سپس خطاهای موجود در آنها شناسایی و تایید می گردد و سپس در جهت رفع خطاها اقدام صورت می گیرد. این گونه اشتباهات سبب می شود که تجارب توسعه دهندگان افزوده گردد.

شکل 95 چرخه توسعه انطباقی را به طور جامع نشان داده است که در زیر به تفصیل عملکرد هر یک از فازها شرح داده خواهد شد.

فاز شروع پروژه اساس و بنیاد پروژه را تعریف می نماید و با تعاریفی از ماموریت های پروژه آغاز می شود. در واقع این ماموریت ها اساسا چهارچوبی برای دستیابی به محصول نهایی می باشند. نکته مهم در تعاریف ماموریت های پروژه، جمع آوری و بدست آوردن اطلاعاتی است که برای انجام پروژه مورد نیاز می باشند .

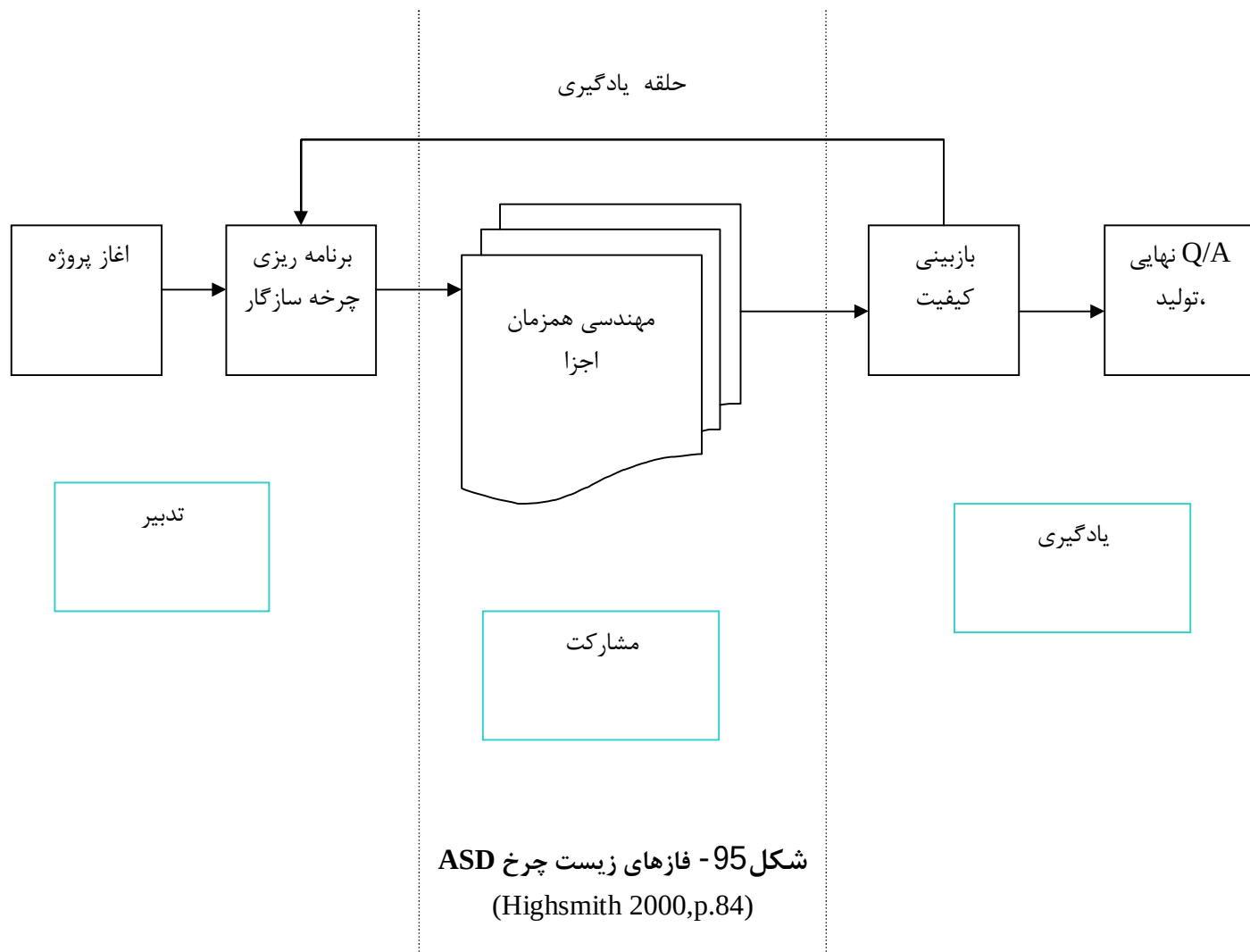
در فاز ابتدایی پروژه، اهداف و مجموعه زمان بندی مورد نیاز پروژه به تفصیل تعیین می گردد. در پروژه هایی که با استفاده از این نوع متدلوژی انجام می شوند، چرخه توسعه نرم افزار نوعاً بین 4 تا 8 هفته به طول می انجامد.

لازم به ذکر است که ASD جزء نگراست تا وظیفه نگر. در عمل این مفهوم به این معنی است که این متد بر نتایج و کیفیت نسبت به وظایف یا فرآیند طی شده در رسیدن به نتایج تمرکز بیشتری دارد . فاز دوم در متدلوژی ASD، فاز مشارکت است که چندین جزء از نرم افزار تحت توسعه همزمان قرار می گیرند. برنامه ریزی چرخه، یکی از بخش های فرآیند تکرار است که تعریف اجزاء به صورت مستمر مورد بازبینی و اصلاح قرار می گیرد تا بتوان هر اطلاعات جدیدی را در توسعه اجزاء شناسایی شده، انعکاس داد و تغییرات در نیازمندی های اجزاء را تصفیه نمود.

چرخه بعدی یعنی حلقه یادگیری که در شکل 95 مشاهده می‌شود بر بازبینی کیفیت فعالیت‌های توسعه نرم‌افزار به صورت مستمر تمرکز دارد. یک فاکتور مهم در بازبینی، حضور مشتری به عنوان یک گروه خبره به منظور بررسی دقیق و کاربردی بر روی نرم‌افزار می‌باشد.

از آنجاییکه در مسیر اجرایی نمودن هر چرخه، بازبینی کیفیت به ندرت اتفاق می‌افتد، لذا در انتهای هر چرخه بازبینی به صورت دقیق و کامل انجام می‌شود. در این متد بازبینی محصولات نهایی با استفاده از مشتریان و کاربران نهایی نرم‌افزار با استفاده از ابزارهای موجود در متد JAD پشتیبانی می‌شود. جلسات JAD ضرورتاً کارگاه‌های کاری هستند که توسعه‌دهندگان و نمایندگان مشتریان برای توضیح در مورد خصوصیات محصولات به صورت مطلوب و بالا بردن سطح ارتباطات، یکدیگر را ملاقات می‌نمایند. در نهایت نتایج حاصله از جلسه مستندسازی می‌شود تا در صورت پیش آمدن هرگونه سوال نتایج مذاکرات در دسترس باشد. متد ASD زمان بندی برای اداره جلسات JAD را پیشنهاد نمی‌کند بلکه آنرا به عنوان یک نکته مهم در آغاز پروژه مورد توجه قرار می‌دهد.

مرحله نهایی در متدلوژی Q/A ASD، نهایی و مرحله تولید می باشد. ASD توجهی ندارد که چطور فازها باید انجام شوند بلکه تاکید اصلی این متد بر ذخیره کردن مطالب یاد گرفته شده می‌باشد.



شکل 95- فازهای زیست چرخ ASD
(Highsmith 2000,p.84)

Ø نکات کلیدی

1. متدلوژی ASD مانند دیگر متدهای چابک توسعه نرم افزار دارای فرهنگ انطباق پذیری

نسبت به تغییرات می باشد.

2. در این متد بین توسعه دهندگان نرم افزار و کاربران نهایی همکاری زیادی وجود دارد.

همکاری نزدیک کاربران نهایی و داشتن ارتباطات و مشارکت آنها با تیم توسعه در جهت

پاسخگویی به سوالاتشان به منظور تفهیم روند عملکرد تیم و تعیین رویه تست نرم افزار و

اطمینان از پیشرفت توسعه انجام می گیرد.

3. متدلوژی ASD از چرخه های تکراری در جهت توسعه نرم افزار استفاده می نمایند.

Ø خصوصیات ویژه

- 1- **ماموریت محور:** به طوری که فعالیت‌ها در هر چرخه توسعه در مقابل ماموریت‌های پروژه تایید می‌شوند.
- 2- **جزء نگر:** فعالیت‌های توسعه بر توسعه نرم افزارهای کاری مانند ساختن قطعه کوچکی از نرم افزار در زمانی مشخص استوار می‌باشد.
- 3- **تکرار پذیر:** یک سری از روشهای آبخاری فقط در محیط‌های شناخته شده و تعریف شده عمل می‌کنند و دارای برنامه از پیش تعیین شده هستند، در صورتی که در روشهای چابک با وجود محیط آشفته تلاش بر توسعه تکراری و رفع نواقص کار در هر تکرار تا تکمیل نهایی خطاهای موجود به جای درست انجام دادن فعالیت توسعه آن هم صرفا در یک تکرار می‌باشند.
- 4- **ریسک پذیر:** یعنی در حد امکان توسعه با درجه ریسک بالا انجام می‌پذیرد که ریسک ممکن است به علت شناخت ناکافی و یا تغییرات بحرانی در سیستم رخ دهد.
- 5- **دارای چهار چوب زمانی مشخص است:** با تعیین زمان پایان پروژه با یک قاعده اساسی و مشخص، ابهام در پروژه‌های نرم‌افزاری پیچیده را کاهش می‌دهد .
- 6- **سازمانها به عنوان سیستم های انطباق پذیر در نظر گرفته می شوند:** در توسعه نرم‌افزار تغییرات در نیازمندی‌ها به طور مکرر صورت می‌گیرد بنابراین انطباق‌پذیری سازمان نسبت به این تغییرات در مقابل کنترل صرف این تغییرات دارای اهمیت است. برای ایجاد یک سیستم با ظرفیت انطباق بالا در مقابل تغییرات، توسعه‌دهندگان مستمرا تغییرات اجزایی را که می‌سازند ارزیابی می‌کنند.
- 7- **مشتری محوری:** مانند دیگر روشها، زمان بندی انجام پروژه و تحویل نرم افزار بر اساس درخواست مشتری (customer –scheduling) انجام می‌گردد.

Ø کاربرد

- 1- متد ASD در توسعه نرم افزارهای بزرگ و پیچیده استفاده می شود.
- 2- در شرایطی که ریسک پروژه توسعه نرم افزار بالاست می توان از متدلوژی ASD استفاده نمود.

Ø نقاط قوت

- 1- همانطور که در ابتدای توضیح این متد اشاره شد ASD تعادلی را در آشوب حاصل شده در چرخه توسعه فراهم می نماید. هدف این متدلوژی در واقع فراهم نمودن چهارچوبی با هدایت کافی در جهت جلوگیری پروژه ها از شکست می باشد.
- 2- در این روش نیازی نیست که تیم های توسعه نرم افزار در یک مکان با یکدیگر همکاری کنند، زیرا ساختن نرم افزارهای بزرگ و پیچیده نیاز به دانش گسترده تیم ها و استفاده از انواع تکنولوژی ها دارد .
- 3- مشارکت و کار تیمی در این متد دارای اهمیت زیادی می باشد و همچنین متدلوژی ASD تکنیک هایی را برای بالا بردن سطح مشارکت در کار تیمی با پیشنهادهای استراتژیکی همچون اشتراک گذاری اطلاعات، استفاده از ابزار ارتباطی و روشهای دیگر به پشتیبانی توسعه توزیع شده به کار می برد.

Ø نقاط ضعف

- 1- در این متد هزینه توسعه نرم افزار به علت توجه به درخواست مشتری در زمان بندی روند توسعه بالا می باشد.
- 2- متد ASD بیشتر بر فرهنگ و مفاهیم توسعه نرم افزار تمرکز دارد و کمتر به فعالیتهای توسعه نرم افزار توجه می کند .

Open source software development - 7-3-6

متدلوژی OSS، منابعی از کد نرم افزار را فراهم می نمایند که توسعه دهندگان نهایی نرم افزار می توانند آزادانه و بدون هزینه برای باز بینی عملکرد نرم افزار و توزیع مجدد آن از کدهای تولید شده استفاده نمایند.

فیلر و فیزگرالد¹⁹³ در بررسی های خود به عوامل انگیزشی و محرک هایی را شناسایی نمودند که توسعه دهندگان نرم افزار را تشویق به استفاده از متدلوژی OSS می نماید. که این 3 عامل عبارتند از Feller and Fitzgerald (2000):

- **عامل تکنولوژی:** به معنی نیاز به استفاده از کدهای تمیز و مرتب و منبع دار، چرخه های توسعه سریع تر، کیفیت استاندارد بالاتر، اعتبار و پایایی نرم افزار های توسعه یافته و در نهایت استانداردهای باز.
- **عامل اقتصادی:** به معنی نیاز مشارکتی برای تقسیم بندی در هزینه های توسعه نرم افزار و همچنین هزینه های حاصله از ریسک توسعه نرم افزار.
- **عامل سیاسی - اجتماعی:** به معنی تمایل به انجام فعالیت های معنادار و نتیجه گرا و ارتباطات مناسب با دیگر توسعه دهندگان نرم افزار.

Ø تاریخچه

متدلوژی OSS در سال 1998 میلادی توسط آقای "تیم اریلی"¹⁹⁴ توسعه یافت. در همین سال Tim O'Reilly به نمایندگی گروهی از افراد توسعه دهنده اعلام داشتند که OSS جایگزین Free software شده است که OSS نسبت به Free software ها دارای ابهامات کمتر و راحتی استفاده برای شرکت های جهان می باشند. البته در این میان بعضی از افراد معتقدند که با تولد اینترنت در سال 1996 میلادی OSS ها روی کار آمدند.

¹⁹³ Feller and Fitzgerald

¹⁹⁴ Time o'reilly

متدلوژی OSS اجازه می‌دهد که هر فردی نسخه جدیدی از نرم‌افزار را ساخته و آن را به سیستم عملیاتی جدید منتقل نماید و معمار توسعه سیستم آن را تکمیل نموده و سپس آن بخش توسعه یافته را با دیگر بازارهای توسعه نرم‌افزار به اشتراک گذارد.

Ø فرآیند:

فرایند توسعه نرم‌افزار توسط متدلوژی OSS شامل فازهای زیر می‌باشد:

(Sharma et al.2002)

- 1- کشف مسائل¹⁹⁵
 - 2- پیدا کردن داوطلب¹⁹⁶
 - 3- شناسایی راه حل¹⁹⁷
 - 4- توسعه کد و تست¹⁹⁸
 - 5- بازبینی تغییرات کد¹⁹⁹
 - 6- به کار گیری کد و مستند سازی²⁰⁰
- در هر فاز فعالیتهای زیر صورت می‌گیرد:

• فاز کشف مسائل

در این فاز بخش‌هایی از سیستم که قرار است توسعه یابند، برای توسعه شناسائی می‌شود.

• فاز پیدا کردن داوطلب

توسعه‌دهندگان در خصوص ماژول‌هایی که باید توسعه یابد داوطلب می‌شوند.

¹⁹⁵ Problem discovery

¹⁹⁶ Finding volunteers

¹⁹⁷ Solution identification

¹⁹⁸ Code development and testing

¹⁹⁹ Code change review

²⁰⁰ Code commit and documentation

- فاز شناسایی راه حل

در این فاز افراد توسعه‌دهنده نرم‌افزار در جهت توسعه به دنبال راه کارهایی جهت تسهیل فرآیند توسعه نرم‌افزار می‌باشند و سرانجام مسیر فرآیند توسعه را شناسایی به صورت بهینه مشخص می‌نمایند.

- فاز توسعه کد و تست:

کدهای نرم‌افزار توسعه یافته و نتیجه عملیات تست می‌شود.

- فاز بازبینی تغییرات کد:

پس از توسعه کد نرم‌افزار، تغییرات موجود در کدهای نرم‌افزاری توسط توسعه‌دهندگان بازبینی می‌شود.

- به کار گیری کد و مستند سازی:

در این فاز از کدهای توسعه یافته استفاده می‌شود و سپس نتایج حاصله مستندسازی می‌شود.

متدلوژی OSS دارای ویژگی‌هایی است که عبارتند از: (Mockus et al. 2000)

§ نرم‌افزارهایی که با استفاده از این متدلوژی توسعه می‌یابند در عمل با تعداد

زیادی از داوطلبان توسعه نرم‌افزار ساخته می‌شوند.

§ در این متدلوژی کار مشخص و از پیش تعیین شده‌ای برای توسعه‌دهندگان

نرم‌افزار تعیین نشده است و افراد بر اساس تمایلشان بخشی از نرم‌افزار را انتخاب

نموده و آنرا توسعه می‌دهند.

§ سطح طراحی نرم‌افزار در استفاده از این متدلوژی واضح نیست.

§ در این متدلوژی مستند از پیش تدوین شده‌ای به عنوان طرح پروژه، زمان‌بندی

برای انجام پروژه و یا لیست تحویل نرم‌افزار وجود ندارد.

§ در این متد نرم‌افزار با افزایش‌های کوچک تکمیل می‌شود.

§ برنامه توسعه نرم‌افزار بصورت تکراری تست می‌شود.

فرآیند توسعه OSS به عنوان فرآیند توسعه موازی و تلاشی برای اشکال‌زدایی از نرم‌افزار سازماندهی شده است که مکان و گروه‌های توسعه نرم‌افزار غیر متمرکز و به صورت همکاری بین گروه‌های توسعه نرم‌افزار می‌باشد و هزینه برای هر سهم توسعه دهندگان آزاد می‌باشد.

Ø نکات کلیدی:

1. در مدل‌های OSS توسعه‌دهندگان کسانی هستند که در توسعه قسمتی و یا کل نرم‌افزار داوطلب می‌شوند و در مناطق مختلف جغرافیایی تمرکز یافته‌اند و نیازی به ارتباطات رو در روی²⁰¹ توسعه دهندگان وجود ندارد. همچنین در این مدل منابع کد آزادانه و نامتمرکز در دسترس همه بخش‌ها می‌باشد.

2. فرآیند توسعه توسط مدل‌های OSS دارای مستندسازی مرسوم از پیش تعیین شده نمی‌باشد. بلکه شامل مراحل است که از طریق تجربه توسعه دهندگان نرم‌افزار بدست آمده است. (سایت open source.org)

3. توسعه نرم‌افزار توسط مدل‌های OSS به صورت تکرار پذیر افزایشی است.

Ø خصوصیات ویژه: (سایت www.wikipedia.org)

1. کاربران نهایی نرم‌افزار باید به عنوان گروه توسعه دهنده رفتار کنند به طوری که کاربران نهایی مانند تیم توسعه به منابع کد نرم‌افزار دسترسی دارند. به علاوه کاربران برای ارائه نرم‌افزارهای اضافی، نصب کدهای نرم‌افزار، گزارشات اشکال‌زدایی و مستند سازی و غیره تقویت شده‌اند. داشتن تعدادی گروه توسعه دهنده سرعت اینکه کدام نرم‌افزار باید باز شود را افزایش می‌دهد.

Linus'law اظهار می‌دارد که تعدادی از کاربران نهایی منابع کد را ملاحظه می‌کنند و همه اشکالات را یافت خواهند کرد و پیشنهاد می‌دهند که چطور آنها را رفع کنند. توجه داشته باشید که بعضی از کاربران نهایی دارای مهارت‌های برنامه نویسی پیشرفته می‌باشند که هر ماشین‌کاربر

²⁰¹ Face _To_ Face

یک عملیات تست مازاد را انجام می دهد. این عملیات تست جدید قابلیت یافتن و کنار گذاشتن اشکال را فراهم می کند.

2. **تولید سریع** یکی از ویژگیهایی است که نسخه اول نرم افزار باید سریعاً تولید شود تا شانس یافتن تیم توسعه نرم افزار را افزایش دهد.

3. **یکپارچه سازی تکراری** که کد جدید باید یکپارچه سازی شود. اغلب برای جلوگیری از کنار گذاشتن تعداد زیادی اشکال در انتهای زیست چرخ پروژه به کار برده می شود. و پروژه های OSS در جاییکه یکپارچه سازی اتوماتیک به صورت روزانه صورت می گیرد، انجام می شود.

4. **چند نسخه ای** که حداقل باید دو نسخه از نرم افزار باید موجود باشد. یک نسخه پست تر با خصوصیات بیشتر و چندین نسخه پایدار با خصوصیات کمتر. نسخه پست تر (که اغلب نسخه توسعه نامیده می شود) برای کاربرانی است که می خواهند فوراً از آخرین خصوصیات استفاده کنند و ریسک کد تست نشده را می پذیرند.

5. **پیمانه ای بودن بالا** به صورتی که ساختار عمومی نرم افزار برای توسعه موازی پیمانه ای باشد.

6. **ساختار تصمیم گیری پویا** به طوری که ساختار تصمیم گیری رسمی یا غیر رسمی مورد نیاز می باشد و استراتژی تصمیمات به تغییرات نیازمندیهای کاربران نهایی و دیگر فاکتورها بستگی دارد.

Ø کاربرد

1- این متدولوژی باعث می شود که محصولات نرم افزاری قابل فهم تر، اصلاح پذیر تر، قابل تکثیر، معتبر و قابل دسترس شوند.

2- OSS ها در حل چالش هایی مانند مسائل علمی، سازمانی و غیره استفاده می شوند و کمتر در مسائل تجاری کاربرد دارند.

Ø نقاط قوت

1. سیستم‌هایی که با متدلوژی OSS توسعه می‌یابند، نرم‌افزارهای کامپیوتری هستند که منبع کد آنها به سادگی در دسترس می‌باشند و توسعه دهندگان نرم‌افزار باید در نرم‌افزار خود این قابلیت را به وجود آورند که و دیگر توسعه‌دهندگان می‌توانند کدهای تولید شده را مطالعه نمایند و در آنها تغییر ایجاد کرده و آنها را بهبود دهند و دوباره آنها را در شکل اصلاح شده یا نشده توزیع نمایند.

2. با استفاده از این متد برای توسعه نرم‌افزار توزیع نرم‌افزار به صورت اصلاح شده و یا نشده معمولاً فاقد هزینه می‌باشد.

3. نرم‌افزارهایی که با استفاده از متدلوژی‌های OSS توسعه می‌یابند، اغلب در حوزه مشارکتی و عمومی توسعه می‌یابند.

Ø نقاط ضعف:

1. متدلوژی OSS وابستگی شدید به شبکه توسعه دهندگان نرم‌افزار دارد.
2. این متد صرفاً در حل مسائل و چالش‌های علمی و سازمانی کاربرد دارد.
3. OSS ها فاقد موعده زمانی مشخص برای تحویل محصول و یا نرم‌افزار به کاربران نهایی است.

6-4- نتیجه‌گیری

مطالعه نشان می‌دهد که در روند توسعه نرم‌افزار متدلوژی‌های سنتی که در حوزه علم مهندسی نرم‌افزار وجود دارند در عمل به منظور توسعه سریع یک نرم‌افزار دارای کارایی کمی هستند چرا که استدلال می‌شود که متدلوژی‌های سنتی در جزئیات ساختارمند عمل می‌نمایند که چنین سطحی از توسعه زمانی که توسعه‌دهندگان نرم‌افزار قصد دارند به صورت سریع نرم‌افزار مورد نظر را تولید کنند، آنها را با مشکل مواجه می‌کند.

با توجه به مطالب عنوان شده توسعه‌دهندگان به این نتیجه رسیدند که لازم است روش‌های جدیدی برای توسعه نرم‌افزارها ارائه گردد و به همین منظور روش‌های توسعه چابک، انتشار یافتند که درواقع معرفی این نوع از متدلوژی‌های توسعه نرم‌افزار حرکتی نوین در عرصه علوم مهندسی نرم افزار می‌باشد.

رویکرد توسعه نرم‌افزار به صورت چابک تاکید بر افراد، تبادل، کارکردن نرم‌افزار، مشارکت مشتریان و انعطاف‌پذیری نسبت به تغییرات فرایند، ابزار، قرارداد و طرح‌ها دارد. نتایج بررسی‌ها نشان می‌دهد که توسعه نرم افزار با استفاده از روش‌های چابک از ویژگی‌های زیر برخوردار هستند:

§ توسعه نرم‌افزار به صورت افزایشی: به این معنی که در اینگونه از روش‌ها

تولیدات کوچک نرم‌افزار با چرخه‌های سریع انجام می‌پذیرد.

§ ایجاد سطح مطلوبی از همکاری: به این معنی که در این روش‌ها مشتریان و

توسعه‌دهندگان پیوسته با یکدیگر با ارتباطات نزدیک دارند.

§ ایجاد سازگاری: به این معنی که نرم‌افزارهایی که با استفاده از این متد توسعه

می‌یابند قادر به ایجاد تطبیق با تغییرات آتی می‌باشند.

هدف از استفاده از این نوع متدلوژی در واقع قادر ساختن افراد حرفه‌ای در نرم‌افزار، پروژه‌ها و سازمان‌ها برای انتخاب و به کار بردن روش درست و مناسب در زمان مناسب می‌باشد. در نتیجه تفکر چابک، یک نگرش فرد محور برای توسعه نرم‌افزار می‌باشد.

بررسی‌ها نشان دهنده این نکته است که محققین و متخصصین حوزه مهندسی نرم‌افزار بر این باورند که روش‌های توسعه چابک نرم‌افزار یک راه بدیع از مسائل مهندسی نرم افزار می‌باشد در حالیکه این روش‌ها قادر به حل همه مسائل نمی‌باشند.

در زیر به چند نمونه از مطالب مهم و ضروری در ارتباط به روش‌های چابک اشاره داریم:

§ برای پیشرفت در پروژه‌های چابک تجربه دارای اهمیت بسیار زیادی می‌باشد، اما تجربه واقعی از تجربه ای که از روش‌های چابک بدست می‌آید مهم‌تر می‌باشد. طی تخمینی 25% تا 33% از پروژه‌های شخصی باید از لحاظ شایستگی و استفاده از تجارب دیگران بررسی شود ولی این مقدار تا 10% می‌تواند کاهش یابد.

§ روش‌های چابک به آموزش رسمی کمتر نسبت به روش‌های سنتی نیاز دارند. استفاده از یک جفت برنامه نویس برای توسعه هر بخش نیاز به آموزش را کاهش می‌دهد چراکه هرکدام از اعضای این تیم دونفره می‌توانند در موارد خاص نقش مربی نفر دوم را ایفا کنند.

§ اعتبار و ایمنی با استفاده از روش‌های چابک در پروژه‌های بحرانی فراهم می‌شود. در این نوع از متدها نیازمندی‌های عملکرد به صورت واضح تعیین می‌شوند و برنامه‌ریزی تست به صورت دقیق تشریح می‌شود.

§ فاکتور مهم در موفقیت پروژه‌هایی که از این نوع از متدولوژی‌ها به منظور توسعه استفاده می‌کنند عبارتند از: فرهنگ، افراد و ارتباطات. تاثیر این 3 فاکتور به این ترتیب است که روش‌های چابک برای دستیابی به موفقیت در استفاده نیاز برای پشتیبانی فرهنگ دارند. همچنین تیم‌هایی که در این روش‌ها مورد استفاده قرار می‌گیرند لازم است دارای صلاحیت و انسجام لازم برای توسعه باشند. برای ایجاد تیم‌های کاری در این روش‌های چابک از تعداد افراد کمتر و با شایستگی بیشتر استفاده می‌شود. تیم‌ها مورد استفاده باید به صورتی آرایش یابند که در میان ارتباطات به سادگی ایجاد شود و مجموعه تیم آنرا را پشتیبانی کنند.

§ بازسازی و بازنگری بر روی نرم‌افزار باید مکرراً و به صورت تکرارپذیر انجام شود بازسازی با مقیاس بالا مشکل نمی‌باشد و با استفاده از روش‌های چابک امکان پذیر می‌باشد.

§ در این روش‌ها لازم است پیش از توسعه نرم‌افزار تیم توسعه دهنده هزینه مد نظر خود را برای مستندسازی تعیین کند. بسیاری از سازمانها بیشتر از سطحی که مستندات نیاز دارند از تیم مجری تولید مستندات را مطالبه می‌نمایند در صورتی در توسعه نرم‌افزار با استفاده از این متدولوژی لازم است که تیم هدف خود را بر ایجاد ارتباطات موثر قرار دهد و مستند سازی به عنوان آخرین گزینه مطرح باشد.

بخش 7 - نتیجه گیری و ارایه چارچوب توسعه نرم افزار در پژوهشگاه اطلاعات و

مدارک علمی ایران

در بخش‌های پیشین به بررسی رویکردهای مختلف توسعه نرم‌افزار پرداختیم. آنچه از رهگذر این مرور ادبیات و بررسی (هر چند اجمالی) به دست می‌آید، چارچوبه‌ای برای مقایسه‌ی معماری، کاربردها، نقاط قوت و ضعف و سایر موارد مقایسه‌ای میان رویکردهای توسعه نرم‌افزاری است. در ادامه، ماحصل این پژوهش در قالب جداولی که نشان‌دهنده‌ی مقایسه‌ی رویکردهای توسعه در هر یک از زمینه‌ها به صورت مجزا می‌باشد ارایه می‌گردد. پیش از پرداختن به مبحث مقایسه، چند نکته مهم را که برای فهم بهتر مقایسه‌ها ضروری هستند و از بررسی رویکردهای مختلف توسعه نرم افزار بدست می‌آیند را ذکر می‌کنیم:

p رویکردهای توسعه نرم افزار بر خلاف نگرش‌های رایج در مقابل یکدیگر مطرح نشده‌اند بلکه در امتداد هم توسعه یافته‌اند. برای مثال چنانچه در قسمت‌های قبلی هم اشاره شد رویکردهای جنبه‌گرا و سرویس‌گرا در امتداد شی‌گرایی مطرح شده‌اند و در نقطه مقابل شی‌گرایی نمی‌باشند، حتی در لایه‌های زیرین خود از روش‌های توسعه شی‌گرا بهره می‌گیرند. همین نکته در مورد شی‌گرایی و رویکرد ساخت یافته نیز صادق است؛ به عبارتی رویکرد شی‌گرا در لایه‌های زیرین خود در درون متدهای کلاس‌ها از نگرش ساخت‌یافته و رویکرد الگوریتمی بهره‌می‌گیرد.

p RUP به عنوان برجسته ترین روش شی‌گرا تنها به روشی شی‌گرا برای توسعه نرم‌افزارها محدود نشده‌است، بلکه به مدد ابزارهای Rational (محصول شرکت IBM) چارچوبه‌ای را ارایه می‌دهد که با توجه به پایه‌ی شی‌گرایی نهفته در آن قابلیت استفاده از رویکردهای سرویس‌گرا و جنبه‌گرا را نیز برای توسعه دهندگان

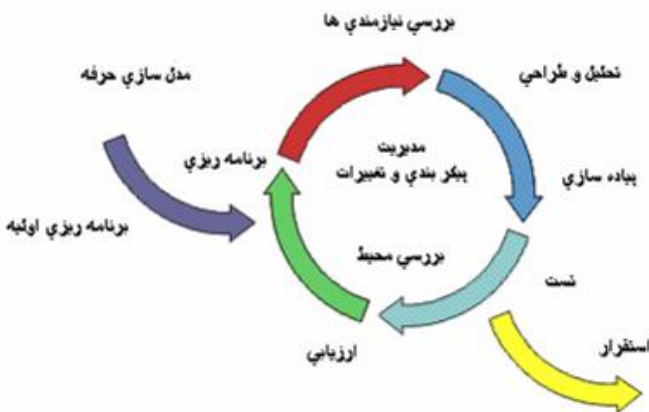
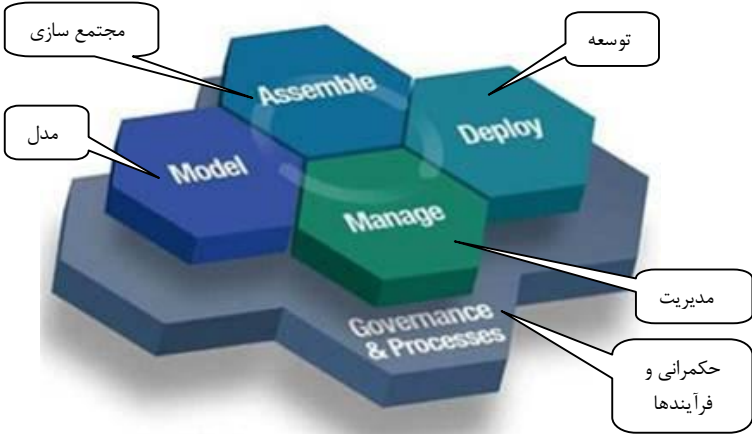
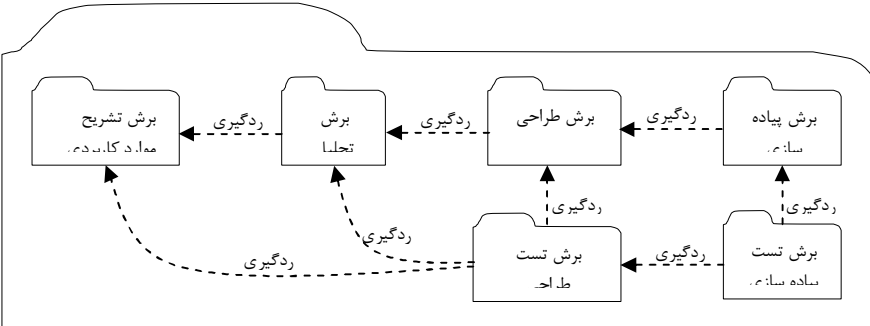
فراهم می‌سازد. گزافه نیست اگر RUP را فراتر از رویکرد شیء‌گرا چارچوبه‌ای برای مدیریت توسعه نرم‌افزار با بهره‌گیری از رویکردهای مختلف توسعه نرم‌افزار بدانیم.

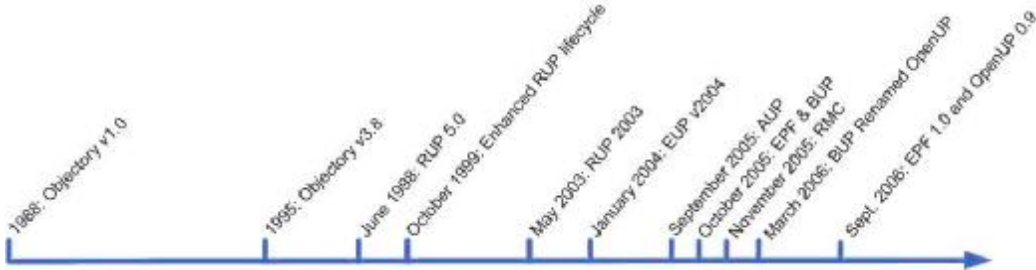
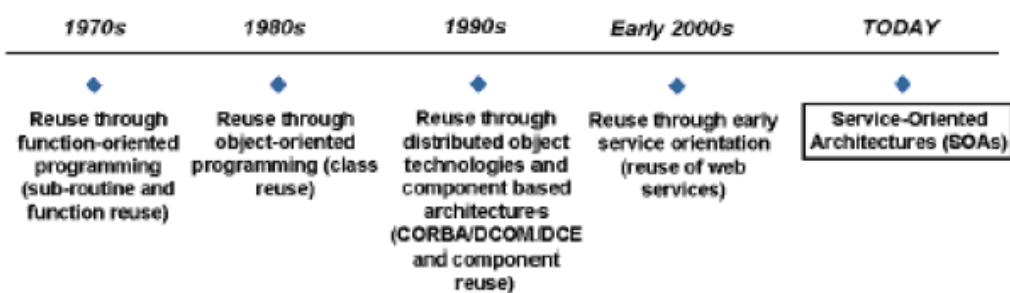
b لازم به ذکر است؛ چنانچه گفتیم، روش‌های چابک رویکرد توسعه نرم‌افزار به شمار نمی‌روند، بلکه مجموعه روش‌های اجرایی برای مدیریت توسعه نرم‌افزار می‌باشند. به عبارتی روش‌های چابک، مجموعه روش‌های مدیریت اجرایی توسعه نرم‌افزار با قابلیت بهره‌گیری از رویکردهای مختلف توسعه نرم‌افزار می‌باشند.

b با توجه به نکات فوق‌الذکر مقایسه رویکردهای توسعه باید جداگانه، و مقایسه روش‌های اجرایی توسعه (روش‌های چابک) نیز در بخشی مجزا باید بررسی شوند.

با توجه به موارد مطروحه در ادامه در دو بخش به مقایسه رویکردها و روش‌های اجرایی توسعه نرم‌افزار می‌پردازیم.

7-1- مقایسه رویکردهای توسعه

ردیف	رویکرد توسعه نرم افزار	فرآیند-معماری
1	شی گرا	
2	سرویس گرا	
3	جنبه گرا	

ردیف	رویکرد توسعه نرم افزار	تاریخچه
1	شی گرا	RUP برای اولین بار توسط شرکت IBM به وسیله بخش IBM's Rational division ارائه شد و برای توسعه این متد نرم افزار Rational به بازار عرضه شد. این نرم افزار به وسیله زبان مستند سازی UML پشتیبانی می شود.  Timeline of Rational Unified Process (RUP) versions: 1988: Objectory v1.0 1995: Objectory v3.8 - June 1998: RUP 5.0 - October 1999: Enhanced RUP lifecycle - May 2003: RUP 2003 - January 2004: EUP v2.04 - September 2005: AUP - October 2005: EPF & BUP - November 2005: RMC - March 2006: BUP Renamed OpenUP - Sept. 2006: EPF 1.0 and OpenUP 0.9
2	سرویس گرا	در دهه 1980 برای استفاده مجدد از معماری های شی گرا استفاده شده سپس در دهه 1990 برای پیاده سازی سیستم های توزیع شده پیچیده تر و بزرگتر حرکت به سمت معماری های اشیاء توزیع شده و توسعه بر موفه COM/DCCOM/CORBA آغاز شد و در اوایل دهه 2000 استفاده مجدد از نرم افزار در قالب سرویس های وب پدیدار گشت و امروزه معماری های سرویس گرا ظاهر شده اند  Timeline of reuse architectures: 1970s: Reuse through function-oriented programming (sub-routine and function reuse) 1980s: Reuse through object-oriented programming (class reuse) 1990s: Reuse through distributed object technologies and component based architectures (CORBA/DCCOM/DCE and component reuse) Early 2000s: Reuse through early service orientation (reuse of web services) TODAY: Service-Oriented Architectures (SOAs)

3	جنبه گرا	رویکرد جنبه گرا نیز همانند معماری سرویس گرا از رویکردهای جدیدی است که برای پاسخ گویی رفع مشکلات روش های ساخت یافته و شیء گرا توسعه یافته است. به علت تازگی مطالب ایر حوزه هنوز یک اجماع جامع و مانع از این مفاهیم در بین صاحب نظران بوجود نیامده است
---	----------	--

نکات کلیدی	رویکرد توسعه نرم افزار	ردیف
❖ این مدل‌لوی این اجازه را به کاربر می‌دهد تا فرآیند تولید نرم‌افزار را بر مبنای نیازهایش تغییر دهد. ❖ به مهندس نرم‌افزار این اجازه را می‌دهد که ساختار پروژه را بر مبنای مدل‌لوی و نیازهایش پیکره بندی نماید.	شی گرا	1
❖ این رویکرد روشی برای ساخت سیستم‌های توزیع شده ارایه می‌دهد که در آن‌ها عملکرد سیستم بصورت سرویس در اختیار کاربران و یا سایر سرویس‌ها قرار می‌گیرد ❖ سرویس؛ رفتار قراردادی تعریف شده‌ایست که هر قطعه‌ای می‌تواند آن را جهت استفاده سایر قطعات در سیستم تهیه و پیاده‌سازی نماید.	سرویس گرا	2
❖ رویکردهای جنبه گرا در فرآیند توسعه حاوی نکات اساسی زیر می‌باشند: ❖ رسیدگی، بررسی، ترکیب‌کردن و درنهایت ردیابی سیستماتیک نیازمندی‌های عملیاتی و غیرعملیاتی تقاطعی و همچنین نیازمندی‌های غیر تقاطعی ❖ به کار گیری مکانیزم‌های تجزیه، نمایش و ترکیب در زمینه‌ی مهندسی نیازمندی‌ها ❖ به کار گیری زبان‌های برنامه نویسی جنبه گرا برای پیاده سازی	جنبه گرا	3

ردیف	رویکرد توسعه نرم افزار	خصوصیات ویژه	کاربردها
1	شی گرا	✓ زمانی در توسعه نرم افزار از متدولوژی RUP استفاده می شود که پیچیدگی مدیریتی و پیچیدگی فنی هر دو در سطح بالایی قرار دارند	✓ این متدولوژی زمانی مورد استفاده قرار می گیرد که از نظر سایز نرم افزار بزرگ می باشد.
2	سرویس گرا	✓ با بکارگیری این رویکرد در توسعه نرم افزار می توان سیستم های جدید را بصورت یکپارچه و در کنار سیستم های قبلی راه اندازی نمود. ✓ سیستم های توسعه یافته بر مبنای این روش عملکرد خود را از طریق مجموعه سرویس های مجزا و در عین حال مربوط به هم در اختیار سایر درخواست کنندگان و یا دیگر سرویس ها قرار می دهند.	✓ سایز نرم افزارها کوچک و متوسط و تعداد نرم افزارهای مرتبط زیاد (که مجموعاً تشکیل سیستم بطرگ یکپارچه می دهند). ✓ توسعه سیستم های جدید به صورت یکپارچه با سیستم های قدیمی موجود ✓ یکپارچه سازی مجموعه نرم افزارهای کاربردی (Application Integration) توسط میان افزارها ✓ توسعه سیستم های تحت وب با بهره گیری از سرویس های وب موجود ✓ پر کاربرد برای سازمان های توسعه دهنده نرم افزار که استفاده مجدد از محصولات و سرویس های ولید شده برای

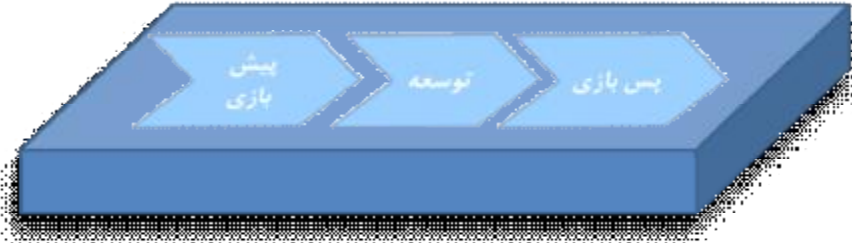
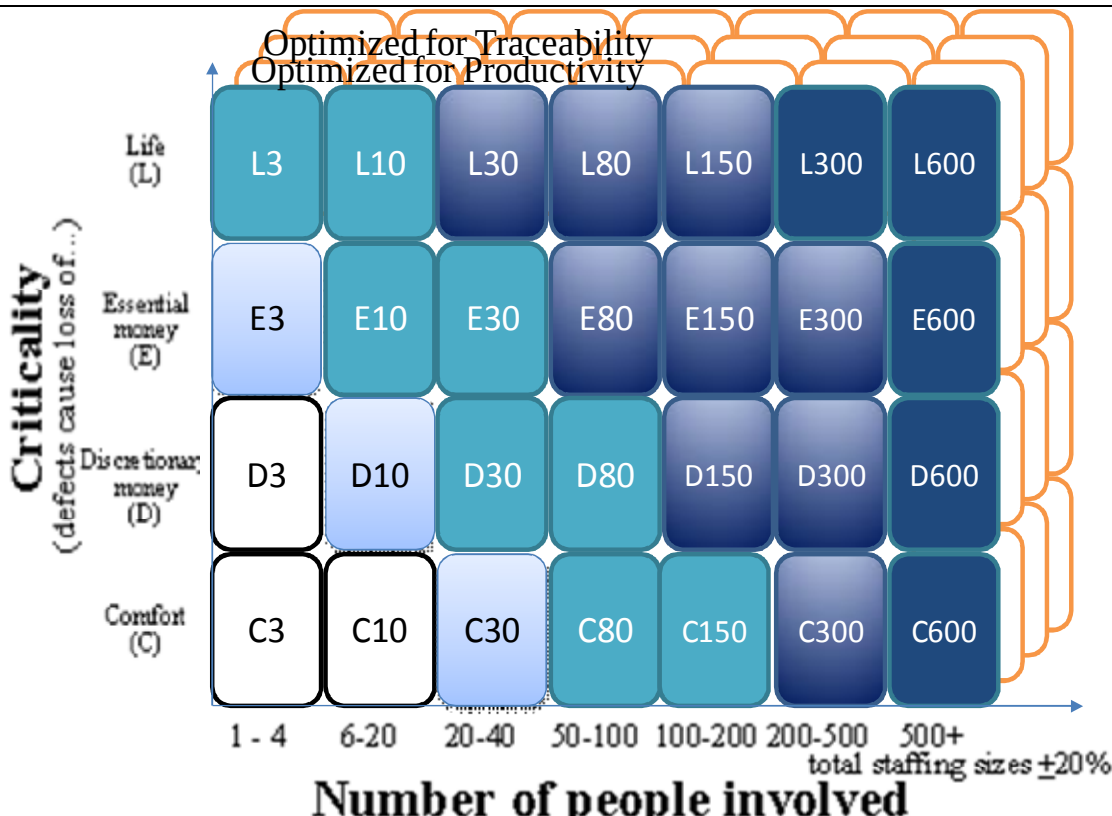
			آنها از اهمیت زیادی برخوردار است.
✓ پاسخگو بودن در زمینه نیاز به ترکیب ✓ پاسخگو بودن در زمینه نیاز به ردیابی ✓ به کار گیری تکنولوژی‌های جدید در فرآیند توسعه (استفاده زبان‌های جنبه گرا) ✓ زبان‌های جنبه گرا: § مکانیزم‌های ترکیب (نقطه‌ی تقاطع و نقطه‌ی اتصال) برای ماژولار نمودن مفاهیم توزیع شده و به هم تنیده (تقاطع‌ی) را به کار می‌گیرند • ایجاد یکریختی در تمام مراحل تولید نرم‌افزار در اثر شناسایی و رسیدگی به خصوصیات تقاطعی، در مراحل اولیه	جنبه گرا	3	✓ کاربرد در پروژه‌هایی که نیازمندی‌ها دارای پیچیدگی‌های زیاد و پاسخگویی به نیازمندی‌های متقاطع مورد نیاز است. ✓ پیچیدگی‌های نرم افزاری بالا و نیاز به ردگیری در مراحل مختلف تولید نرم افزار

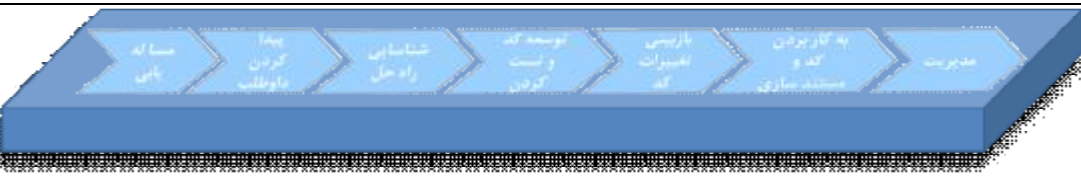
ردیف	رویکرد توسعه نرم افزار	نقاط قوت	نقاط ضعف
1	شی گرا	❖ ریسک ها سریعتر کاهش می یابند زیرا که عناصر بصورت تدریجی یکپارچه می شوند. ❖ درخواست های تغییر و راهکارهای مورد نیاز مورد توافق قرار می گیرند. ❖ توسعه و خالص سازی محصول راحت تر می باشد در نتیجه محصول کاملتری ایجاد می شود. ❖ سازمان ها می توانند در طی این روش اقدام به یادگیری نموده و فرآیند های خود را بهبود دهند. ❖ قابلیت استفاده مجدد افزایش می یابد.	❖ تعدا محصولات تولید شده در این متدلوژی زیاد است. ❖ پیچیدگی تولید نرم افزار زمان استفاده از متدلوژی زیاد است. ❖ انجام کار به وسیله این متدلوژی زمان بر است و نمی توان به صورت چابک نرم افزار را تولید نمود.
2	سرویس گرا	❖ امکان یکپارچه سازی با نرم افزارهای قدیمی و یا توسعه یافته با سایر رویکردها وجود دارد ❖ امکان استفاده از سرویس های وب توسعه یافته موجود زمان توسعه را کاهش	❖ پیچیدگی یکپارچه سازی نرم افزارهای مختلف در طی زمان گاه از تولید نرم افزار جامع یکپارچه بیشتر است ❖ سرویس های موجود تحت وب هنوز در بسیاری از زمینه ها توسعه نیافته و ناکارآمد

		می‌دهد	هستند
		❖ امکان استفاده مجدد در ابعاد دانه درشت‌تر امکان پذیر است ❖ سرویس‌ها و اجزای نرم‌افزار در عین یکپارچگی کمترین وابستگی را به یکدیگر دارند(مستقل هستند) و امکان تعویض و جایگزینی هر یک بدون نیاز به تغییر عمده در سایرین ممکن است. ❖ نگرش فرآیندی و الگوریتمی این رویکرد در پاسخگویی به نیازها با بهره‌گیری از اصول شی‌گرایی در لایه‌های زیرین موجب پاسخگویی به‌ترو دقیق‌تر به نیازمندی‌های موجود می‌شود.	❖ میان‌افزارها گاه خود نیازمند مدیریت و نگهداشت پیچیده‌ای هستند که وجود نرم‌افزار جامع یکپارچه چنین پیچیدگی‌ای را ندارد. ❖ روش‌های شناخت و تحلیل و طراحی بر اساس این رویکرد چندان توسعه یافته نیست و عموماً در حوزه پیاده‌سازی متمرکز است.
3	جنبه گرا	❖ شناخت نیازمندی‌ها به صورت جنبه‌گرا کامل‌تر و دقیق‌تر از رویکرد شی‌گرا و در امتداد آن سرویس‌گراست. ❖ پاسخگویی به برخی نیازمندی‌های تقاطعی در سطح پیاده‌سازی توسط برنامه‌نویسی جنبه‌گرا بسیار سهل‌الوصول‌تر است.	❖ ابزارهای برنامه‌نویسی جنبه‌گرا نوپا و کم‌کاربرد هستند و این مسأله برای شرکت‌های توسعه دهنده نرم‌افزار بسیار نگران‌کننده است. ❖ نکات یکپارچه سازی نرم‌افزارهای موجود و در حال توسعه مورد دقت قرار

نگرفته‌است. ۷ حجم زیاد مستندات امکان تولید چاپک را از تیم توسعه دهنده می‌گیرد.	۷ امکان ردگیری نیازمندی‌ها در مراحل مختلف تولید نرم‌افزار وجود دارد. (در حالت کلی رویکردهای جنبه‌گرا از مدیریت نیازمندی‌های قوی‌تری برخوردار هستند).	
---	---	--

7-2- مقایسه روش‌های اجرایی توسعه نرم‌افزار

فرآیند	نام روش
	Extreme Programming
	Scrum
 Optimized for Traceability Optimized for Productivity Life (L) Essential money (E) Discretionary money (D) Comfort (C) 1 - 4 6-20 20-40 50-100 100-200 200-500 500+ total staffing sizes $\pm 20\%$ Number of people involved	Crystal

فرآیند	نام روش
	Feature Driven Development
	Dynamic Systems Development Method
	Adaptive Software Development
	Open source software development

نام روش	تاریخچه	نکات کلیدی
Extreme Programming	✓ محصول کشور امریکا ✓ توسعه در سال 1996 میلادی ✓ اشخاص توسعه دهنده: KENT WARD, BECK, RON, CUNNINGHAM, GEFRIERES ✓ اولین بار در پروژه CHYSLER COMPERHENSIVE COMPENSATION SYSTEM(C3) با 8 توسعه دهنده استفاده شد	✓ توسعه مشتری مدار ✓ تیمهای کوچک ✓ یکپارچه سازی مستمر (ساخت روزانه یا ساعتی)
Scrum	✓ محصول کشور امریکا ✓ توسعه در سال 1993 میلادی صورت گرفته است ✓ اشخاص توسعه دهنده: JEFF JOHN, SUTHERLAND, JEFF, SCUMNIOTALES, MEKNNAL ✓ اولین بار در پروژه EASEL COPORATION با 8 توسعه دهنده	✓ مستقل می باشند ✓ تیم های کوچک ✓ دارای چرخه تولید 30 روزه ✓ تیم های توسعه، خود سامانده هستند.

نام روش	تاریخچه	نکات کلیدی
	استفاده شد.	
Crystal	✓ محصول کشور امریکا ✓ توسعه در سالهای قبل از 1988 میلادی ✓ اشخاص توسعه دهنده: ALISTAR COCKBURN ✓ اولین بار در پروژه WINIFRED و با تیمهای 40-20 نفره استفاده شد	✓ خانواده ای از روشها ✓ هر کدام دارای یک دسته اصول و ارزش های مشابه می باشند ✓ تکنیک ها، ارزشها، ابزارها، استانداردها متنوع هستند.
Feature Driven Development	✓ توسعه در سال 1997 میلادی ✓ اشخاص توسعه دهنده: JEFF DE LUCA ✓ اولین بار در توسعه پروژه بزرگ و پیچیده بانکداری سنگاپور استفاده شد	✓ دارای فرایند 5 مرحله ای ✓ تکرار خیلی کوتاه ✓ توسعه اجزای بر اساس اهداف
Dynamic Systems Development Method	✓ محصول کشور انگلستان ✓ توسعه در سال 1990 میلادی ✓ اشخاص توسعه دهنده: کنسرسیوم DSDM ✓ اولین بار در DEPTH OF .ORANGE.BT	✓ کنترلی برای توسعه کاربردی سریع (RAD) ✓ استفاده از چهار ر چوب زمانی ✓ تیمهای مختار DSDM ✓ کنسرسیوم فعال برای توسعه روشها

نام روش	تاریخچه	نکات کلیدی
	SEMA ,HEALTH LOGICA ,GROUP &BRITHISH MIDLAND، استفاده شد	
Adaptive Software Development	✓ توسعه در سال 1992 ✓ اشخاص توسعه دهنده: JEMES A.HIGHSMITH و BAYER ✓ اولین بار در کمپانی wall street brokerage house to airline to telecomminucation firms	✓ دارای فرهنگ انطباق پذیری با تغییرات ✓ مشارکتی ✓ توسعه تکرار پذیر بر اساس اجزای ماموریت محور
Open source software development	✓ توسعه در سال 1998 ✓ اشخاص توسعه دهنده: TIM O REILLY	✓ بر اساس داوطلب هاست ✓ توسعه توزیع شده ✓ اغلب برای حل چالش هاست (علمی، سازمانی، ...) تا مسائل تجاری

نام روش	خصوصیات ویژه	کاربردها
Extreme Programming	- بازسازی - ایجاد توسعه نرم افزار موفق با وجود ابهامات یا تغییرات دائمی - نیازمندیها در تیمهای کوچک تا متوسط - تکرار پذیری کوتاه با تولید کوچک - تغییر هر بخش از کد توسط هر فرد در هر زمانی - اهمیت داشتن محیط فیزیکی - کاهش هزینه تغییرات - تعیین مجموعه‌ای از عملیات‌های روزانه برای مدیران و توسعه دهندگان - زمان‌بندی بر اساس درخواست مشتری - تست محور	- مورد استفاده در تیمهای 3-10 نفره - مستند سازی محدود - مناسب برای محیطی که دارای کاربران فعال می باشد
	- دگرگونی چهارچوب فکری از دیدگاه مشخص و قابل تکرار به دیدگاه SCRUM برای توسعه محصول جدید. - هدف گرا هستند	- قابل استفاده در محیط های غیر قابل پیش بینی (شامل محیط پیچیده و متغیر) می باشد - تیم‌های کوچک و ایزوله با کمتر

نام روش	خصوصیات ویژه	کاربردها
	✓ توانا در تغییر برنامه پروژه و دست اوردهای پروژه در هر زمانی ✓ تاکید بر مدیریت پروژه ✓ کاهش گسیختگی زمان بندی (time fragmentation) ✓ زمان بندی بر اساس درخواست مشتری	از 10 نفر مهندس در پروژه های بزرگ می توانند از SCRUM استفاده کنند. ✓ اساسا در مدیریت پروژه های نرم افزاری استفاده می شود
Crystal	✓ توانایی برای انتخاب روش مناسب بر اساس اندازه و اهمیت ✓ در clear مستند سازی به صورت غیر رسمی می باشد	✓ در CLEAR فقط یک تیم در یک فضای اداری قرار دارد ✓ در ORANGE چندین تیم در یک ساختمان اداری موجود است ✓ اهمیت time-to-market در orange ✓ دسترسی آسان به کابران ماهر
Feature Driven Development	✓ طراحی و پیاده سازی بر اساس خصوصیات ✓ مدلسازی اهداف ✓ سادگی روش ✓ فاقد احتیاج به مدل فرایند ویژه ✓ تاکید بر کیفیت در سراسر فرایند	✓ مناسب برای پروژه هایی که نیازمند همکاری با دیگر پروژه هاست ✓ مناسب برای شروع پروژه های جدید ✓ مناسب برای زمانی که احتیاج است کیفیت در کلیه مراحل تحت

نام روش	خصوصیات ویژه	کاربردها
	✓ تکرار پذیری کوتاه ✓ دارای خصوصیات کوچک ✓ برنامه ریزی بر اساس درخواست مشتری و تکنیک (technical-schedual,customer-schedual)	کنترل باشد ✓ مناسب در تیم های بزرگ
Dynamic Systems Development Method	✓ استفاده کردن از PROTOTYPE ✓ ثابت و مشخص کردن زمان و منابع و سپس تعدیل و تنظیم کردن عملیات مطابق انها ✓ تکرار پذیری افزایشی ✓ داشتن نقشهای کاربردی مختلف ✓ دارا بودن چهار چوبی برای کنترل RAD	✓ مناسب برای سیستمهای کسب و کار نسبت به سیستمهای مهندسی و علمی ✓ مناسب برای محیطی که دارای کاربران فعال می باشد ✓ قابل استفاده در تیمهای 2-6 نفره ✓ چهارچوبی برای کنترل و توسعه مشارکتی (JAD) ✓ هم در پروژه های جدید و هم در توسعه سیستمهای موجود استفاده شود
Adaptive Software Development	✓ ماموریت محور ✓ جزئی مبنا ✓ تکرار پذیر	✓ در شرایطی که ریسک بالا می باشد ✓ مورد استفاده در سیستم های

نام روش	خصوصیات ویژه	کاربردها
	✓ ریسک پذیر ✓ دارای چهار چوب زمانی مشخص ✓ سازمانها به عنوان سیستم های انطباق پذیر در نظر گرفته می شوند ✓ به وجود آمدن سفارشهای خارج از قاعده شبکه ارتباطی افراد ✓ زمان بندی بر اساس درخواست مشتری (customer –scheduling)	بزرگ و پیچیده
Open source software development	✓ منابع کد آزادانه در دسترس همه بخش ها است ✓ تکرار پذیری افزایشی ✓ مسئله محور ✓ نامتمرکز ✓ نیازی به ارتباطات FACE TO FACE توسعه دهندگان نمی باشد ✓ تاکید بر توسعه ابزار یا طرح های دیگر که توسط افراد حرفه ای استفاده می شود	✓ مورد استفاده در حل چالش ها ✓ استفاده از کپی، بازبینی و توزیع منبع کد بدون پرداخت هزینه

نام روش	نقاط قوت	نقاط ضعف
Extreme Programming	✓ تولید محصول یکپارچه تست شده ✓ بازخورد سریع ✓ شرکت کردن مشتریان ✓ ارتباطات و مشارکت ✓ مستند سازی محدود ✓ کاهش ریسک در پروژه ✓ کاهش دادن محصول کاری واسطه	✓ به مدیریت عملی (برنامه عملیاتی) کمتر توجه داشته است ✓ تاثیر فرهنگ و تکنولوژی بر توسعه بسیار بالا است ✓ تیم ها باید حتما ارتباط face-to-face داشته باشند (هم مکان بودن اعضای تیم) ✓ هزینه توسعه به علت زمان بندی بر اساس درخواست مشتری افزایش می یابد.
	✓ انعطاف در پاسخ گوئی با مکانیزم های کنترلی برای طراحی نسخه محصول و میریت متغیرها در طی پیشروی پروژه ✓ توانائی به اشتراک گذاری دانش نانوشته و ضمنی افراد پروژه	✓ نداشتن برنامه عملیاتی مدون برای یکپارچه سازی ✓ نداشتن برنامه عملیاتی مدون در تست ✓ کمتر در مدیریت پروژه های غیر نرم افزاری استفاده می شود. ✓ افزایش هزینه توسعه به علت زمان بندی بر اساس درخواست مشتری
Crystal	✓ توانایی انتخاب متدلوژی مناسب تر	✓ فقط 2 تا از 4 روش پیشنهاد شده (YELLOW.RED.ORANGE.CLEAR) موجو

نام روش	نقاط قوت	نقاط ضعف
	ارتباطات بهتر و تولید به صورت تکرار پذیر، نیاز به محصول کاری واسطه را کاهش می دهد. مطابقت و سازگاری با تغییرات در clear مستند سازی به صورت غیر رسمی می باشد محصول کاری سبک دارای قابلیت سکنی در جهت ایمنی پروژه ایمنی در خروجی پروژه دسترسی آسان به کاربران ماهر یکپارچه سازی مکرر توجه به ایمنی کارمندان	د است صرفاً تیمهایی که در یک مکان متمرکز باشند می توانند از این روش استفاده کنند فاقد عناصر ارزیابی در CLEAR و همچنین فاقد ساختاری برای خرده تیم ها در ORANGE، در نتیجه نامناسب برای پروژه های بحرانی می باشد آزاد نبودن تعداد کارکنان برای مذاکرات آزاد نبودن مکان برای مذاکرات
Feature Driven Development	بازبینی دقیق فرایند	تاکید صرف بر طراحی و پیاده سازی

نام روش	نقاط قوت	نقاط ضعف
	پروژه ✓ انطباق پذیری با تغییرات ✓ تاکید بر کیفیت در سراسر فرایند	✓ برنامه ریزی بر اساس مشتری هزینه بر است
Dynamic Systems Development Method	✓ تست در سراسر چرخه حیات ✓ همکاری و مشارکت بالا ✓ یکپارچه سازی ✓ کاهش هزینه ✓ افزایش سود ✓ کاهش دوباره کاری ✓ خود هدایت بودن اعضای تیم ✓ مستند سازی محدود ✓ مشارکت مشتری	✓ نامناسب برای محیطی که فاقد کاربران فعال است ✓ نامناسب برای سیستمهای مهندسی و علمی ✓ فقط اعضای کنسرسیوم به اساسنامه دسترسی دارند
Adaptive Software Development	✓ ایجاد تعادل در اشوب حاصل شده ✓ مشارکت و کار تیمی	✓ بیشتر در مورد فرهنگ ها و مفاهیم می باشد تا فعالیت های نرم افزاری ✓ افزایش هزینه توسعه به علت زمان بندی بر اساس

نام روش	نقاط قوت	نقاط ضعف
	✓ نیازی به تیمهای هم مکان نمی باشد	در خواست مشتری
Open source software development	✓ در دسترس بودن منابع ✓ کد ✓ در زمان توزیع مجدد و باز بینی فاقد هزینه می باشد ✓ مشارکتی ✓ نیازمندیها به دقت شرح داده می شود	✓ وابستگی شدید به شبکه توسعه دهنده ✓ صرفا برای حل مسائل و چالشها کاربرد دارد نه برای توسعه نرم افزارهای کاربردی-تجاری ✓ روش نمی باشد بلکه در واقع توانایی در انتقال اصول ارتباطات OSS به توسعه نرم افزار تجاری است ✓ عدم وجود موعد زمانی مشخص

3-7- چارچوبی پیشنهادی توسعه نرم‌افزار در پژوهشگاه اطلاعات و مدارک علمی ایران

با توجه به مقایسه‌های انجام گرفته در رویکردها و روش‌های توسعه نرم‌افزار و همچنین بررسی نرم‌افزارهای موجود پژوهشگاه اطلاعات و مدارک علمی ایران، چارچوبی زیر به عنوان پیشنهادی برای توسعه نرم‌افزار در این مرکز ارائه می‌گردد:

روش پیشنهادی		مرحله	ردیف
RUP		چارچوبی توسعه (Framework)	1
جنبه گرا	سرویس گرا	تحلیل و طراحی	2
مواردی که پاسخ‌گویی به نیازمندی‌ها بسیار متأثر از تکنولوژی، سیستم‌های نرم‌افزاری موجود و شبکه‌ی ارتباطی نرم‌افزاری و سخت‌افزاریست به ویژه موارد تحت وب.	مواردی که پاسخ‌گویی به نیازمندی‌ها بسیار متأثر از مواردی که پاسخ‌گویی به نیازمندی‌ها بسیار متأثر از موضوع سیستم و گردش کار آن می‌باشد.		
متناسب با روش تحلیل و طراحی ولی با تمرکز بر روش‌های شی گرا و RUP		پیاده سازی	3
به کار گیری روش‌های چابک به‌ویژه XP		مدیریت اجرایی و تیم توسعه	4

بخش 8 - منابع و مآخذ

- 1) Alistair, A. and R. Cockburn (2004). "Humans and Technology".
- 2) Ambler, s. (2002). "Lesson in Agility from Internet-Based Development." IEEE Software 19(2): 66-73.
- 3) AndreasKuehlmann and A. Bergamaschi (2004). High-Level State Machine Specification and Synthesis. Yorktown Heights, N.Y., U.S.A, IBM Research Division , Thomas J. Watson Research Center.
- 4) Araújo, João, et al. (2004). Modeling and Composing Scenario-Based Requirements with Aspects. Colorado "clorado State University, USA."
- 5) Arsanjani, A. (2004). "Service-oriented modeling and architecture." IBM white papers.
- 6) Beck, Kent, et al. (Oct 13, 2000). Planning Extreme Programming, Addison-Wesley Professional.
- 7) Beck, k. (1999). Extreme Programming explained:Embrace change Addison-Wesley.
- 8) Bieber, g. and J. Carpenter (2003). "Introduction to Service-Oriented Programming." Motorola ISD.
- 9) Boehm, B. (2002). "Get Ready for the Agile Methods." Cra.computer 35(1): 64-69.
- 10) Booch and Grady (April 2003). Object-Oriented Analysis and Design with Applications, Adison Wesley.

- 11) Breu, Silvia, et al. (April 26, 2004). Case Studies in Aspect Mining, Extended Abstract. Softwaresysteme, Universität Passau.
- 12) Channabasavaiah, Kishore, et al. (2003). "Migrating to a service-oriented architecture." IBM official web site.
- 13) Clarke, S. and E. Baniassad (2005). Aspect-Oriented Analysis and Design: The Theme Approach
- 14) Addison-Wesley
- 15) Cockburn, A. (2000). Writing Effective Use Cases. The Crystal Collection for Software Professional
- 16) Addison-Wesley Professional.
- 17) Cockburn, A. (2002). Agile software development. Boston, Addison-Wesley.
- 18) Cockburn.acm.org, A.
- 19) Colan, M. (2004). "Service-Oriented Architecture Expands the Vision of Web Services " IBM official web site.
- 20) Colyer, A. and A. Clement (2005). "Aspect-oriented programming with AspectJ: An article from." IBM Systems Journal-Digital Jun 1.
- 21) Committee, A. a. I. (2006). Services and Components Based Architectures A Strategic Guide for Implementing Distributed and Reusable Components and Services in the Federal Government. F. C. I. O. Council.
- 22) corporation, A. (2005). "Service-Oriented Architecture at a glance".
- 23) Ding, H. (2004). Exploiting Extended Service-Oriented Architecture for Federated Digital Libraries. Norwegian Information Management Group, Norwegian Univ. of Sci.& Tech.

- 24) Favaro, j. (2002). "Managing requirement for business values." IEEE software 19: 15-17.
- 25) Feller, J. and B. Fitzgerald (2000). A framework Analysis of the open source software development paradigm. 21 st Annual International Conference on Information System, Brisbane,Australia.
- 26) Highsmith and Cockburn (2001). "Agile Software development:The business of Innovation".
- 27) Highsmith, J. A. (2000). Adaptive software development : A collaborative approach to managing complex system. New York, dorset House Publishing.
- 28) <http://eclipse.org/aspectj>.
- 29) http://en.wikipedia.org/wiki/Agile_software_development.
- 30) <http://en.wikipedia.org/wiki/AspectJ>.
- 31) http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1341253.
- 32) <http://research.ibm.com/cme/>
- 33) <http://webcourse.cs.technion.ac.il/236800/Winter2004-2005/ho/WCFiles/11a-ScenarioBasedAspects.ppt>.
- 34) <http://www.extremeprogramming.org>.
- 35) <http://www.ibm.com>.
- 36) <http://www.ibm.com/developerworks/rational/library/2782.html>.

- 37) Huhns, M. and M. P. Singh (2005). "Service-Oriented Computing: Key Concepts and Principles
IEEE Computer Society journal.
- 38) Jaakkola and T. i. (2005). Applying Service-Oriented Architecture to Geographically Distributed Industrial Information Systems. Computer Science, HELSINKI UNIVERSITY OF TECHNOLOGY.
- 39) Jacobson, I. and N. Pan-Wei (Dec 30, 2004). Aspect-Oriented Software Development with Use Cases, Addison-Wesley.
- 40) Jiang, M. and A. Willey (2005). Architecting Systems with Components and Services, IEEE
- 41) Kiselev, I. (2002). "Aspect-Oriented Programming with AspectJ." International journal of Information Technology.
- 42) lehrstuhl (2005). "Introduction to Service-Oriented Architecture." Executive Strategy.
- 43) Linthicum, D. S. (2003). Next Generation Application Integration: From Simple Information to Web Services, Addison Wesley.
- 44) Lublinsky, D. A. M. B. (2006). "SOA Enterprise Patterns: Services, Orchestration and Beyond".
- 45) Martin, C. and R. , , (June 98). "UML Tutorial: Finite State Machines." Engineering Notebook Column.
- 46) Mike, P. (2003). Service-Oriented Computing Concepts, . Characteristics and Directions Proceeding of the forth international conference on web information system engineering Tilburg University
- 47) Tilburg University

- 49) Mike, P. and W.-J. v. d. Heuvel (2005). "Service Oriented Architectures: Approaches, Technologies and Research Issues." VLDB Journal.
- 50) Miller, G. (2001). The characteristics of Agile Software Process. The International Conference of object-oriented language and system(TOOLS 39), santa Barbara,CA.
- 51) Mockus, A., R. Fielding, et al. (2000). A case study of open source software development. 22nd International conference software engineering, Limerick,Ireland.
- 52) Palmer, S. R. and J. M. Felsing (2002). A practical Guide to feature-driven development. NJ, Prentice-Hall.
- 53) Pekka, A., J. S. Ronkainen, et al. (2002). Review and analysis: Agile software development methods, VTT publication.
- 54) Schwaber, k. (1995). Scrum development process. Workshop on Business Object Design and Implementation.
- 55) Schwaber, k. and M. Beedle (2002). Agile Software Development with Scrum. NJ, prentice-Hall.
- 56) Sharma, S., v. Sugumaran, et al. (2002). "A framework for creating hybrid-open source software communities." Information system journal 12(1): 7-25.
- 57) Wendy, C., X. Wang, et al. (2004). "Developing Web services using the Microsoft.Net platform: technical and business challenges." Journal of Enterprise Information Management 17 (5).
- 58) whitepaper, w. (2005). Making the Transformation to Service- Oriented Architecture Capitalizing on the Next Revolution in IT.
- 59) Whitten, J. L., L. D. Bentley, et al. (2004). System Analysis and Design methods McGrawHill.

- 60) www.agilealliance.org.
- 61) www.agilemanifesto.org.
- 62) www.dacs.dtic.mil/techs/agile.
- 63) www.opensource.org.
- 64) Yang, A. (2004). "Critical Infrastructure for Service-Oriented Architecture." Westbridge Technology.
- 65) Zhu (2004). From OOP to SOP: What Improved, IEEE ,available at www.ieeexplore.org.
- 66) Zimmermann, O. and e. al (2 June 2004). "Elements of Service Oriented Analysis and Design." IBM official web site.
- 67) توده‌فلاح، س. (1383). "بررسی معماری سرویس گرا".
- 68) "<http://en.wikipedia.org/wiki/artifact>".
- 69) "<http://en.wikipedia.org/wiki/software-componentry>".
- 70) "<http://en.wikipedia.org/wiki/workflow>".
- 71) "<http://www.awprofessional.com/articles/article.asp?p=24671&seqNum=3&rl=1>".
- 72) "<http://www.enterpriseunifiedprocess.com/essays/history.html>".
- 73) "<http://www.enterpriseunifiedprocess.com/essays/history.html>".
- 74) Rational Unified Process, Best Practices for Software Development Teams. Rational Software White Paper TP026B, Rational Software Development Company.
- 75) June 1999). "OMG Unified Modeling Language." Version 1.3.

76) جزوه آموزشی کارگاه آموزشی متدولوژی شیء گرای USDP، شرکت ایزایران، مرکز تولید نرم افزار، مدیریت استانداردها و متدولوژی.

77) (1379). طرح ملی استاندارد مستند سازی، گزارش شناخت استانداردهای ساخت و مستندسازی محصولات نرم افزاری.

78) سرخانلی، س. کتاب آموزشی RUP (rational unified process) همراه با دو مثال جامع، نشر ناقوس

۷۹) فخرایی، س. و. "راهنمای متدولوژی RUP، فایل راهنمای نرم افزار Rational Rose".