**بسم الله الرحمن الرحیم**

**دانشگاه علوم وفنون مازندران**

**Aspect Oriented**

**ارائه دهندگان:**

**حسین اسحقی**

**احسان حقانی**

**رضا تنظیفی**

**فرشته کیایی**

**سید رضا حسینی**

**سید محمد حسینی**

**درس : مهندسی نرم افزار(2)**

**استاد راهنما : منهدس مفیدیان**

**تاریخ: 12/9/1391**

فهرست

[تاریخچه پیمانه بندی سیستم : 1](#_Toc344216603)

[دغدغه : 2](#_Toc344216604)

[جدا سازی دغدغه ها : 2](#_Toc344216605)

[دغدغه های مداخله ای Crosscutting Concern : 4](#_Toc344216606)

[درهم پیچیدگی : (Tangling) 5](#_Toc344216607)

[پراکندگی (Scattering ) 6](#_Toc344216608)

[جنبه ها (Aspect ): 8](#_Toc344216609)

[بخشهای مختلف جنبه 10](#_Toc344216610)

[بافتن**(weaving)** 11](#_Toc344216611)

[انواع بافنده ها**:** 12](#_Toc344216612)

[مهندسی نرم افزار جنبه گرا: 14](#_Toc344216613)

[مستند نیازها: 14](#_Toc344216614)

[مدل عمومی مهندسی نیازهای جنبه گرا 15](#_Toc344216615)

[مهندسی جنبه گرا 17](#_Toc344216616)

[طراحی و برنامه نویسی جنبه گرا: 19](#_Toc344216617)

[وارسی و معتبر سازی 21](#_Toc344216618)

# تاریخچه پیمانه بندی سیستم :

از سال 1968 در کنفرانس NATO توسعه ی سیستم های نرم افزاری به عنوان یک مشکل مهندسی مطرح شد. رفته رفته مهندسین نرم افزار به این نتیجه رسیدند که باید سیستم را به قسمت های کوچکتری شکاند.کوچک کردن مسئله ی نهایی می تواند کمک بسیار زیادی به حل مساله اصلی بکند. مهندسین نرم افرارجهت انجام این جداسازی ، سیستم را به پیمانه های مشخص تقسیم کردند. وظایف سیستم را به عملیات های مشخصی تقسیم کردند و در قالب توابع پیاده سازی کردند. مدل سازی سیستم و پیمانه بندی آن در قالب توابع که به آن برنامه نویسی ساخت یافته می گویند، راه حل مناسبی برای پیمانه بندی نبوده زیرا توابع به تنهایی نمی توانستند سیستم را به ماژول های مستقل از هم تقسیم کنند. علاوه بر آن برنامه نویسی ساخت یافته توانایی شبیه سازی و پیاده سازی نیازهای کاربران را نداشت. پس این بار برنامه نویسان سبک جدیدی به نام برنامه نویسی شی گرا را ارائه دادند. در برنامه نویسی شی گرا ماژول های سیستم در قالب اشیا پیاده سازی شدند. دیگر مهندسین نرم افزار به این نتیجه رسیدند که به خوبی می توانند با لستفاده از این سبک به دغدغه (نیازها، خواسته ها) پاسخ دهند. شی گرایی توانست نسبت به گذشته پیاده سازی نرم افزار را به دنیای واقعی نزدیک تر کند. همچنین نیاز کپسوله سازی توسعه دهندگان سیستم را هم برآورده کرد. برنامه نویسی شی گرا توانست تا حدودی نیاز جدا سازی و کپسوله سازی نیازها را برآورده کند. با گسترش نرم افزار ها و نیازها، نیازهایی به وجود آمدند که آن ها را نمی توان در یک شی پیاده سازی کرد بلکه به صورت مداخله ای[[1]](#footnote-2) در چندین شی درگیر است. در اینجا بود که شی گرایی نمی توانست به درستی به این نیازهای مداخله ای پاسخ دهد و آن ساختار کپسوله ای خود را حفظ کند. پس برنام نویسان شی گرا برای پاسخ به این نیاز ها سیستم خود را دچار مشکل درهم ریختگی و پراکندگی کردند.

# دغدغه :

دغدغه ها را در واقع می توان نیاز های کاربران به حساب آورد. اینکه جه مسائلی برای ذی نفعان دارای ارزش است و باید سیستم به آن ها پاسخ دهد. البته این به آن معنا نیست که صرفا نیاز های عملیاتی کاربر جز دغدغه ها باشد، بلکه نیاز های عملیاتی سیستم و نیازهای کیفی خود نیز شامل دغدغه ها است. گاهی ممکن است یک دغدغه شامل چندین نیاز شود.

به عنوان مثال امنیت می تواند به عنوان یک دغدغه مطرح شود. چون برای کاربر دارای اهمیت و ارزش است. در نتیجه سیستم باید آن را برآورده کند. و یا برای یک سیستم اطلاعاتی گزارش گیری از داده ها یک دغدغه است.

# جدا سازی دغدغه ها :

جدا سازی دغدغه ها یکی از روش های کلیدی برای طراحی سیستم ها و پیاده سازی آن ها است. در این روش نرم افزار خود را به گونه ایی تحلیل و تقسیم می کنیم که هر جز کاری فقط یک کار انجام داده و یا روی یک موضوع مانور دهد. این امر باعث شده خرابی و نقص در یک عنصر سبب خرابی در کل نشود. پیمانه کردن سیستم، طراحی ، فهم و مدیریت سیستم های پیجیده را آسان می گرداند. تیم توسعه می توانند به صورت جداگانه ای و بدون توجه فراوان به المان های دیگر صرفا بر روی جز مورد نظر خود تمرکز کند.

انواع دغدغه های مختلف از دیدگاه ذی نفعان:

1. "دغدغه های وظیفه : که مربوط به وظیفه خاصی است که سیستم باید آن را شامل شود. برای مثال در سیستم کنترل ترن، یکی از دغدغه های وظیفه ای ترمز قطار ها می باشد.
2. دغدغه های کیفیت سرویس : که مربوط به رفتار های غیر وظیفه ای سیستم است. که شامل شاخص هایی مانند کارایی، فابلیت اطمینان و در دسترس پذیر بودن سیستم است.
3. دغدغه های سیاستی یا مشی: مربوط به تمامی سیاست هایی است که ناظر بر استفاده از سیستم است. دغدغ ها در policy شامل امنیت و در امان بودن و همچنین نگرانی های شامل قوانین کسب و کا سازمان است.
4. دغدغها های سیستمی : که مربوط به خصوصیات کلی سیستم مانند قابلیت نگهداری و پیکربندی آن است.
5. دغدغه های سازمانی : که مربوط اهداف سازمانی و اولیت های سازمان در راستای هدف آن هاست. شامل تولید سیستم به اندازه بودجه ، استفاده از دارایی های نرم افزاری و نگهداری اعتبار سازمانی است."[[2]](#footnote-3)

# دغدغه های مداخله ای Crosscutting Concern :

در سیستم های شی گرا گاهی کاربران نیاز هایی دارند که نمی توان آن ها را در یک شی متمرکز کرد و برای پیاده سازی آن چندین عنصر از سیستم تاثیر می پذیرند.

به عنوان مثال می توان به واقعه نگاری[[3]](#footnote-4) اشاره کرد. فرض کنید در سیستم شما بعد از انجام هر تراکنش در پایگاه داده باید تاریخ و شخصی که آن عمل را انجام داد واقع نگاری شود. ممکن است چندین شی و تابع عمل تغییر بر روی پایگاه داده را انجام دهند و بعد از اجرای آن عملیات باید واقعه نگاری صورت پذیرد. لذا باید آن را در چندین شی بگنجانیم. پس نتوانستیم عمل واقع نگاری را در یک مولفه متمرکز کنیم در نتیجه واقعه نگاری یک دغدغه ی مداخله ای است.

دو ویژگی اصلی که می توان برای دغدغه های مداخله ای در نظر گرفت:

1- " عدم تجزیه واضح از دیگر بخش ها هم در طراحی هم در پیاده سازی

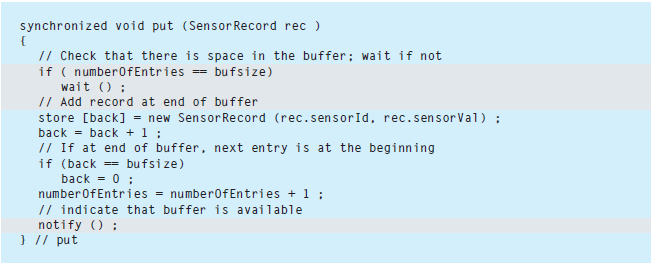
2- قرار گرفتن کد پیاده سازی در میان چندین مولفه "[[4]](#footnote-5)

می توان دغدغه های مداخله ای را به دو دسته تقسیم کرد:

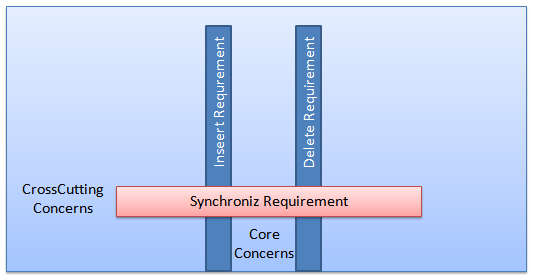
" نوع اول : شامل دغدغه هایی هستند که نیازهای غیر وظیفه مندی را برآورده می کنند که به آنها دغدغه های پایه گفته می شود. مانند امنیت.

نوع دوم : شامل دغدغه هایی هستند که با نیازهای وظیفه مندی سرکار دارند مانند کلیه موارد کاربردی."

برای مثال سیستمی را در نظر می گیریم که در آن نیاز به دسترسی هم زمان به یک بافر اشتراکی دارند. یک فرآیند داده را به بافر اضافه می کند و فرآیندی دیگر داده ها را از همان بافر برمی دارد. این بافر اشتراکی قسمتی از یه سیستم حافظه است که فرآیندی داده را در بافر گذاشته و مصرف کننده داده را از روی آن برمی دارد. دغدغه اصلی ما نگهداری یک بافر اشتراکی می باشد که مرتبط با وظایف اصلی شامل حذف کردن و اضافه کردن عناصر به بافر است.برای اطمینان از اینکه دو تابع بطور همزمان شروع به اضافه کردن و برداشت کردن از بافر نکنند نیاز دیگری به مانند همزمان سازی فرآیند ها به وجود می آید که خود می تواند منجر به ایجاد دغدغه دومی در سیستم شود. سیستم باید به گونه ای طراحی شود که تابع اضافه کننده داده یی را در بافر روی داده یی که هنوز خوانده نشده نریزد و همچنین فرآیند حذف خانه خالی بافر را نخواند.



نیاز به درج و حذف هر کدام به عنوان یک دغدغه وظیفه مند تعریف می شوند که در دو پیمانه (تابع) تقسیم بندی و پیاده سازی می شود و نیاز دیگر سیستم که به عنوان غیر همگام سازی فرآیند ها بود به عنوان دغدغه غیر وظیفه مند تعریف می شود. مشکل اینجاست که نمی توان این دغدغه را بصورت در یک پیمانه بصورت مستقل پیاده سازی کرد زیرا این دغدغه را نمی توان بصورت یک پیمانه در دیدگاه شی گرایی یا ساخت یافته در نظر گرفت و این نوعی از تداخل دغدغه ها است.



# درهم پیچیدگی : (Tangling)

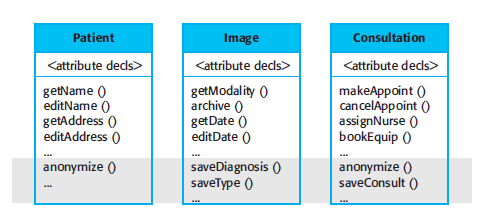
اگر مولفه ای به گونه یی پیاده سازی شود که در فرآیند توسعه آن دغدغه های مختلفی پیاده شده باشند یک درهم پیچیدگی اتفاق افتاده. از معایب این امر می توان به گذینه های زیر اشاره کرد:

* توسعه گر یک مولفه نیاز به این دارد که دغدغه های مختلف سیستم را تحلیل و درک نماید تا بتوان تنها یک مولفه را پیاده سازی کند. در اینجا مولفه بجای شامل بودن یک دغدغه درگیر چندین دغدغه می شود و از قابل فهم بودن جلوگیری می کند.
* خاصیت پیمانه ای بودن نرم افزار زیر سوال می رود زیرا هر پیمانه باید بیانگر مفهوم دسته ای از دغدغه های مرتیط از سیستم باشد.

# پراکندگی (Scattering )

در صورتی که دغدغه یی در پیمانه های مختلف دخالت داشته باشد مجبوریم آن دغدغه را در تمام پیمانه ها پیاده سازی کنیم که این امر موجب پراکندگی خواهد شد . اگر دغدغه یا نیاز ما دست خوش تغییرات شود جهت بروز رسانی مجبوریم دغدغه ها را در پیمانه های مختلف تغییر دهیم. دغدغه های مداخله ای این گونه اند زیرا کد آن باید به دلیل ماهیت این دغدغه یعنی درگیری با پیمانه های مختلف سیستم در میان مولفه های مختلف سیستم پخش شود.

برای مثال با توجه به دیاگرام شکل زیر ، این شکل 3 کلاس موجود در سیستم ذخیره بیماران بیمارستانی را نشان می دهد که شامل چند متد اصلی برای مدیریت داده های بیماران در پایگاه داده می باشد. توابع حاشور خورده عملیات مورد نیازی را نشان می دهند که به عنوان دغدغه ثانویه سیستم در نظر گرفته می شود. مشکل اصلی با پراکندگی و درهم پیچیدگی زمانی است که نیازهای اساسی سیستم تغییر کند برای مثال نیازی جدید مبنی بر ذخیرهی داده های آماری جدیدی در اطلاعات رکورد بیمار تعریف می شود. زمانی که دغدغه ای در سیستم تغییر می کند این تغییر مختص به پیمانه خاصی نمی شود و باید کل سیستم و مولفه ها را جست و جو کرد تا تغییرات را در همه جا اعمال کنیم. از معایب این کار پر هذینه بودن آن می باشد، زیرا هم باید زمانی صرف آنالیز کردن مولفه ها گذاشته شود و هم باید بعد از اعمال تغییرات، مرحله ی تست انجام گیرد. همچنین ممکن است که در پیاده سازی به دلیل پراکندگی کدها اشتباهات نحوی و منطقی در برنامه رخ دهد که موجب افزایش خطا و اشتباه در نرم افزار می شود.



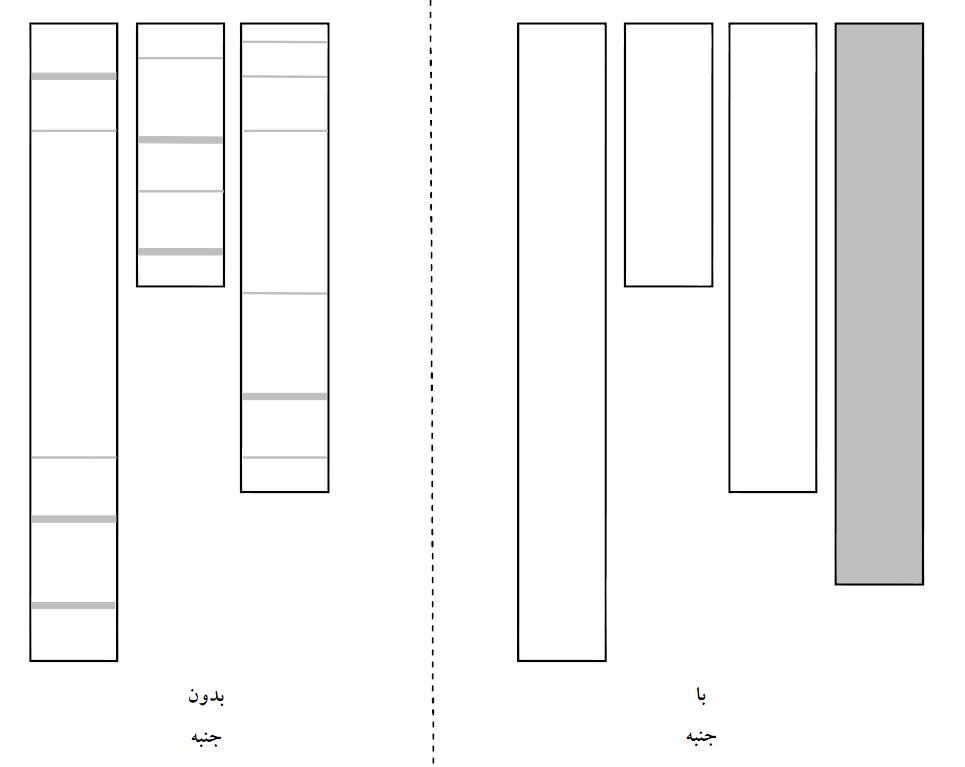
# جنبه ها (Aspect ):

مشکل پراکندگی و درهم ریختگی در برنامه توسط جنبه ها قابل حل است . در برنامه های نرم افزاری بخشی از دغدغه های کاربران با هم تداخل دارند و همانطور که اشاره شد این دغدغه ها در یک مولفه (شی) قابل پیاده سازی نیستند.

در یک سیستم جنبه گرا می توان دغدغه های مداخله ای را به صورت یک دغدغه جدا در نظر گرفت جدا سازی این دغدغه ها و پیاده سازی آن به وسیله ی جنبه مشخص می شوند.

اگر تا کنون از سبک برنامه نویسی شی گرا [[5]](#footnote-6) استفاده می کردید و با آن آشنا بودید. در برنامه نویسی جنبه گرا با مفهوم جدیدی به نام جنبه آشنا می شوید . پس باید به مفاهیمی از قبیل تابع ، Class و ... مفهومی به نام جنبه را نیز اضافه کنید. برنامه نویسی جنبه گرا جدا از برنامه نویسی شی گرا نیست ، بلکه در واقع قابلیت جدیدی است که به برنامه نویسی شی گرا اضافه شده است.

در برنامه نویسی شی گرا پیمانه بندی سیستم (که یکی از اهداف مهندسی نرم افزار است) در قالب اشیا انجام می شود ، اشیا می توانند به خوبی به نیازهای عملیاتی و غیر مداخله ای پاسخ دهند . اما در برنامه نویسی جنبه گرا علاوه بر کلاس ها که برای پاسخ به نیازهای ساختاری استفاده می شوند. جنبه ها هم قرار دارند و وظیفه ی پاسخ به نیازهای مداخله ای ( دغدغه های مداخله ای ). بنابراین در برنامه نویسی جنبه گرا دو بعد کلاس و جنبه وجود دارد.



در شکل بالا مستطیل ‌های سفید کلاس هستند. در قسمت  بدون جنبه که صرفا با کلاس پیاده سازی شده، شیارهای خاکستری کد‌هایی‌ در هر کلاس هستند  که برای پیاده سازی نیاز مداخله‌ای (مانند واقعه نگاری) نوشته شده است. چون این اعمال در چندین نیاز مورد استفاده قرار می‌گیرد پس کلاس‌های متعددی را درگیر کرده.در قسمت با جنبه کلاس ها بدون کد‌های پراکنده مربوط به دغدغه‌های مداخله‌ای پیاده سازی شده اند. کد های مربوط به پیاده سازی نیاز های مداخله ایی در قسمت جنبه (مستطیل خاکستری) پایده سازی شده اند. این امر به پیمان بندی هرچه کامل تر سیستم  کمک می‌کند چرا که در پیمان بندی کمترین تداخل بین پیمانه‌ها (به عنوانه مثال اشیا) باید وجود داشته باشد و هر بخش صرفا وظایف مربوط به خود را انجام می دهد.

بخشهای مختلف جنبه

یک جنبه بخش‌های مختلفی‌ را شامل می شود که به شرح زیر است:

Advice:

کدی که برای پیاده سازی یک دغدغه می‌نویسیم advice نام دارد. این بخش از جنبه در زمانی‌ که جنبه باید اجرا شود، اجرا می شود. کدهای جنبه میتواند شامل پیاده‌سازی دغدغه‌های وظیفه مندی و یا غیر وظیفه مندی باشد اما در اکثر موارد جنبه برای پیاده سازی نیازهای غیر وظیفه مندی (نیاز‌های کیفی‌) استفاده میشود.

در واقع هدف از به وجود آمدن جنبه پاسخ گوئی به نیازهای کیفی‌ بوده، چرا که این نیاز‌های کیفی‌ هستند که ناظر به کّل سیستم اند و تداخل در وظایف مختلف را ایجاد می‌کند.

Join point:

به رویداد‌هایی [[6]](#footnote-7)در زمان به وقوع پیوستن آن باید آن جنبه بافته[[7]](#footnote-8) یا اجرا شود

"مدل join point تعریفی‌ از مجموعه رخداد‌هایی‌ است که می‌توان در یک برنامه جنبه گرا مورد اشاره واقع شود. مدل join point حالت استانداردی ندارد و هر زبان برنامه‌نویسی جنبه گرا مدل join point  خود را دارد."[[8]](#footnote-9)

به عنوانه مثل در زبان aspectj موارد زیر را داریم:

۱.:Call event زمانی‌ که متد یا سازنده‌ای صدا زده میشود مثل توابعی که اول آنها نام update آماده است

۲. : Execution events زمانی‌ که متد یا سازنده‌ای در حل اجراست.

۳.: Initialization event هنگام ایجاد کلاس یا اشیا

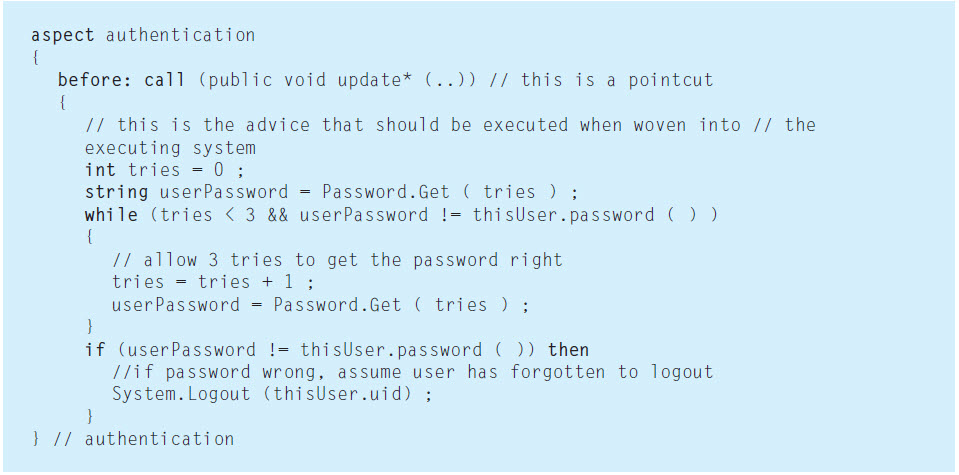
۴ .:Data eventهنگام دسترسی به یک فیلد و یا دست‌کاری آن (مثل get و ‌set در شی‌ گرایی (

۵exception event. : زمانی‌ که استثنای خاصی‌ در برنامه به وجود میاید.

Point cut:

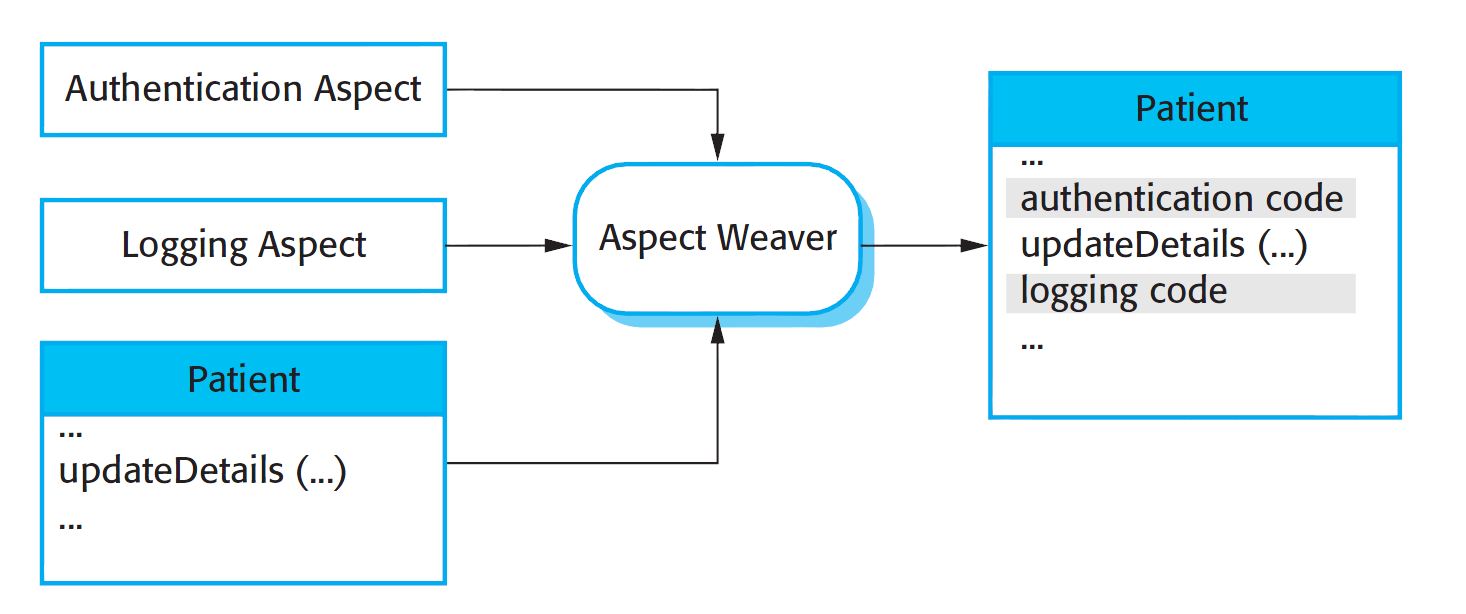
"رخداد‌هایی‌ را شناسایی می‌کند که با advice در ارتباط هستند این به آن معناست که شما میتوانید advice را در نقاط مختلفی‌ از برنامه بسته به مدل join point که پشتیبانی‌ می‌شود بیاورید"[[9]](#footnote-10)

شیک زیر مثالی از یک جنبه است.



بافتن**(weaving)**

بافنده در واقع در کنار کامپایلر ها قرار می گیرند. زمانی که کد های اصلی برنامه و همچنین کد های جنبه تولید شدند، بافنده کد های جنبه و برنامه اصلی را دریافت می کرده، سپس آنها را به ترتیب مورد نظر کنار هم قرار می دهد. این عمل را طبق نظر آقای کیزالس یکپارچه سازی[[10]](#footnote-11) می گویند. در نهایت کد های یک/ارچه شده در اختیار کامپایلر قرار می گیرند.



نباید این اشتباه را کرد که بافنده جنبه ها [[11]](#footnote-12)کامپایلر هستند.بلکه واسطی است بین کدهای برنامه و کامپایلر. همانطور که از توضیحات مربوط به بافنده مشخص است، جنبه ها همچون اشیا مفاهیم انتزایی هستند. یعنی ماشین آن کدی که توسط کامپایلر ها کامپایل می شود را دریافت می کند اما درکی از کلاس ها یا جنبه ها ندارد، و این عناصر برای ماشین تعریف نشده هستند. کامپایلر ها هستند که متوجه می شوند به عنوان مثال یک شی خاص در کجای برنامه اصلی قرار دارد.

انواع بافنده ها**:(Aspect Weaver)**

1. سورس کد قبل از پردازش[[12]](#footnote-13)

در این شیوه ابتدا بافنده کد برنامه را گرفته و جنبه ها را در جایی که باید اعمال شود قرار می دهد. و یک کد به زبان مورد نظر(مثلا جاوا یا C++) تولید کرده و آن کد تحویل کامپابلر داده می شود، کامپایلر نیز همچون گذشته همان کد برنامه را کامپایل می کند.

1. بافنده متصل زمانی [[13]](#footnote-14)

در این روش بافنده در کامپایلر قرار دارد و همزمان با کامپایل برنامه بافتن جنبه نیز انجام می شود. این روش مستلزم این است که کامپایلر دست کاری شود و بافنده به آن اضافه شود. این روش در AspectJ نیز استفاده شده است.

1. بافنده پویا در زمان اجرا [[14]](#footnote-15)

در این روش جنبه ها با کد برنامه ادغام نمی شود و در زمان اجرای برنامه در هرلحظه محل برش چک می شود و در صورت رخ دادن نقطه اتصال ، کدهای جنبه اجرا می شوند.

در بین این سه روش پرکاربردترین روش ، بافنده متصل زمانی است که سرباری اضافی زمان اجرا را بر خلاف روش بافندگی پویا در زمان اجرا ندارد اما با وجود سرباری زیاد پویایی و انعطاف پذیری زیادی دارد . روش اول نیز منسوخ شده است که به دلیل انعطاف پذیری بسیار پایین آن است

# مهندسی نرم افزار جنبه گرا:

زمانی که می خواهیم نرم افزاری را با مدل جنبه گرا پیاده سازی کنیم باید در مرحله طراحی معماری ، این مساله را لحاظ کنیم و در مرحله ای که شناسایی نیازها باید این مساله را در نظر گرفت که دغدغه ها به عنوان یک اصل و پایه برای بدست آوردن نیازمندی ها استفاده شودو بنابراین نیازهای سیستم را با دید جنبه گرایی بدست آورد. برای بدست آوردن دغدغه های کاربران ابتدا باید نیازهای آنان را شناخت و سپس اقدام به جداسازی نیازهای مداخله ای و پایه نمود .پس مبحث مهندسی جنبه گرا را باید از مهندسی نیازها شروع کرد.

# مستند نیازها:

در مهندسی نرم افزار، مستند نیازها بخش بسیار مهمی از فرآیند طراحی و تحلیل سیستم است. این مستند در واقع مشخص کننده ی ابعاد و خدماتی است که سیستم باید آن ها را ارائه کند.

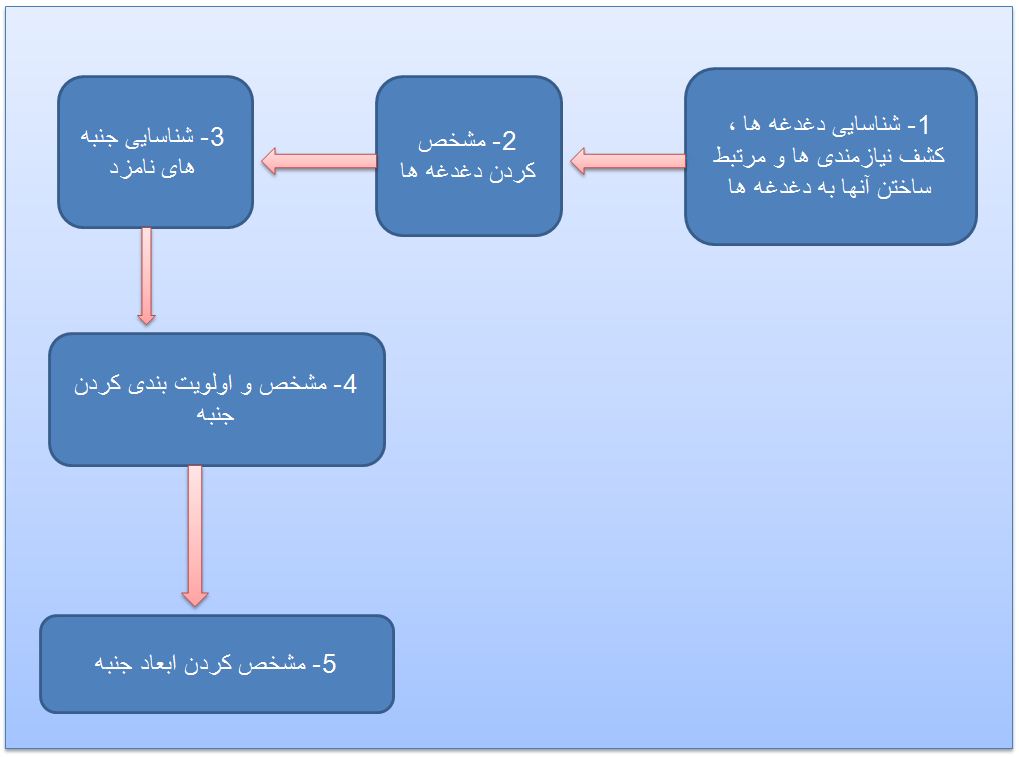
بعد از آن که نیازهای کاربران که همان انتظارات کاربر از سیستم است بدست آمد ،می توان آن ها را به دو دسته نیاز های عملیاتی (وظیفه مندی) و کیفی (غیر وظیفه مندی) تقسیم کرد.

"نیازهای عملیاتی ، قابلیت ها و سرویس هایی هستند که سیستم در اختیار کاربران قرار می دهد."[[15]](#footnote-16)

نیازهای کیفی نوعی از نیازمندی هاست که اغلب با نام های "نیازمندی های کارایی، کیفیت سرویس نیازمندی ها و محدودیت ها نیز مورد ارجاع قرار می گیرند و موضوعاتی در ارتباط با خود سیستم و چگونگی خوب انجام دادن وظایف توسط سیستم را مشخص می کنند. نمونه هایی از این نیازها عبارت است از: قابلیت دسترسی ، قابلیت آزمون پذیری، قابلیت نگهداری و استفاده آسان" [[16]](#footnote-17) جنبه ها اکثرا برای برطرف کردن این نوع نیازها آمده چرا که شی گرایی راه حل مناسبی برای برطرف کردن این نوع نیازها ارائه نداده است.

# مدل عمومی مهندسی نیازهای جنبه گرا

این مدل برای بررسی دغدغه های مداخله ای در مهندسی نیازمندی های جنبه گرا معرفی می شود و طبق شکل مراحل ان را توضیح خواهیم داد.



1- در این مرحله ابتدا بر اساس نیازهای ذینفعان ، دغدغه های ذینفعان را مشخص می کنیم و ارتباط بین نیازمندی ها و دغدغه ها را چک می کنیم.

2- این مرحله برای مشخص کردن جزئیات دغدغه هایی که شناسایی شده اند خواهد بود.

3- این فعالیت را به شناسایی جنبه های نامزد بر اساس دغدغه های مشخص شده در مرحله قبل می پردازیم. این عمل به این صورت است که اگر دغدغه ای با چندین نیازمندی تداخل داشته باشد به عنوان جنبه نامزد انتخاب می کنیم.

4- در این مرحله ابتدا جرئیات جنبه های نامزد را مشخص می کنیم. این عمل باعث می شود ارتباطات و تداخلات بین جنبه ها پدیدار شوند و در ادامه بتوان جنبه ها را اولویت بندی کرد.

5-در این مرحله تاثییرات جنبه ها به مرحله بعد از مهندسی نیازمندی ها بررسی می شود و ابعاد یک جنبه مشخص می شود. تاثییرات را در دو بعد بررسی می کنیم.

* بعد نگاشت : بعد نگاشت این مساله را بررسی می کند که یک جنبه بعد از مشخص شدن امکان دارد به صورت یک جنبه پیاده سازی نشود و به صورت یک متد یا یک تصمیم ( مانند یک تصمیم برای انتخاب معماری) در سیستم نگاشت شود. دلیل بررسی و تحلیل جنبه ها در مرحله ی مهندسی نیازمندی ها همین بعد از تاثیرات جنبه ها است که امکان دارد به صورت جنبه پیاده سازی نشود و تاثیری کلانتر داشته باشد.  
  - بعد تاثیر: تاثیر یک جنبه می تواند بر روی یک یا چند موضوع مختلف باشد. برای مثال قابلیت دسترسی که یک جنبه است ، معماری سیستم را تحت تاثیر قرار می دهد در حالی که جنبه زمان پاسخ، هم معماری و هم جرئیات طراحی را تحت تاثیر قرار می دهد.

در این مرحله اطلاعات تفکیک بین تداخل ها که در مرحله قبل بدست آمده مورد استفاده قرار می گیرد.

# مهندسی جنبه گرا از دید Jacobson & NG

Jacobson & NG پیشنهاد دادند که در زمان طراحی یک سیستم دغدغه ها (نیاز) ی ذینفعان مختلف را از دو دیدگاه مورد بررسی قرار دهیم. نیازهایی که به عنوان هسته سیستم هستند (عملیاتی) و نیازهای همراه[[17]](#footnote-18) .دلیل این همه تاکید برروی نیازها این است که پیدا کردن دغدغه ها و ابعاد آن همه از روی نیازها و دیدگاه های کاربر بدست می آید.

همانطور که در شکل دو صفحه بعد مشخص است " هسته سیستم مجموعه ای از ویژگی های سیستم است که هدف ضروری یک سیستم را پیاده سازی می کند"[[18]](#footnote-19)

به عنوان مثال در سیستم حفظ اطلاعات بیماران، هسته اصلی سیستم امکان ایجاد، ویرایش ، مدیریت و دسترسی به داده های بیماران در پایگاه داده است .

اما نیازهای همراه بخش دیگری از دیدگاه ها و نیازهای کاربران است که به هسته اصلی متصل است. این دسته از نیازها باید با هسته اصلی یکپارچه شوند.

در همان مثال اطلاعات بیمار وضعیت کنترل دسترسی کاربران به داده های پایگاه داده نیز مهم است و یا رمز نگاری آن داده ها. این ها نمونه هایی از نیازهای همراه است.

انواع نیاز های همراه 13:

1. عملکرد ثانویه : این Extension ها قابلیت ها و عملکردهای اضافی را به وظیفه های اصلی سیستم[[19]](#footnote-20) اضافه می کنند. برای مثال وظیفه گزارش گیری از داروها در سیستم مدیریت بیماران می توان به عنوان نیازو عملکرد ثانویه سیستم در نظر گرفت.
2. Policy Extension (سیاست سازمان): اگر سازمان یک سری نیاز خاص و قانون مند برای زیرسیستم های خود در نظر بگیرد و از یک سری توابع قانون مند استفاده کند، Policy ها تعریف می شوند. به عنوان مثال اضافه کردن ویژگی های امنیتی به سیستم به عنوان یک سیاست سازمانی در نظر گرفته شود.
3. Qos Extension (کیفیت سرویس): در واقع قابلیت عملکردی است که باعث می شود سطح انتظارات کیفی سیستم را بتوان کنترل کرد تا سیستم از لحاظ کیفی به سطح مطلوبی برسد. مثال های متفاوتی می توان در نظر گرفت :

-تعداد مراجعات هم زمان در یک سیستم پایگاه داده ای را در زمان Peak آن سیستم.

-پشتیبان گیری خودکار سیستم برای بهبود عملکرد در زمانی که سیستم با شکست مواجه می شود.

4. Infrastructure Extension (زیرساخت): قابلیت عملکردی به منظور حمایت از پیاده سازی یک سیستم در برخی از Platformهای خاص اضافه می کند.

Extensionها در واقع قابلیت ها و ویژگی های اضافی را به هسته سیستم یا عملکردهای اصلی سیستم اضافه می کنند و جنبه ها راهی برای اجرا این Extensionها هستند.

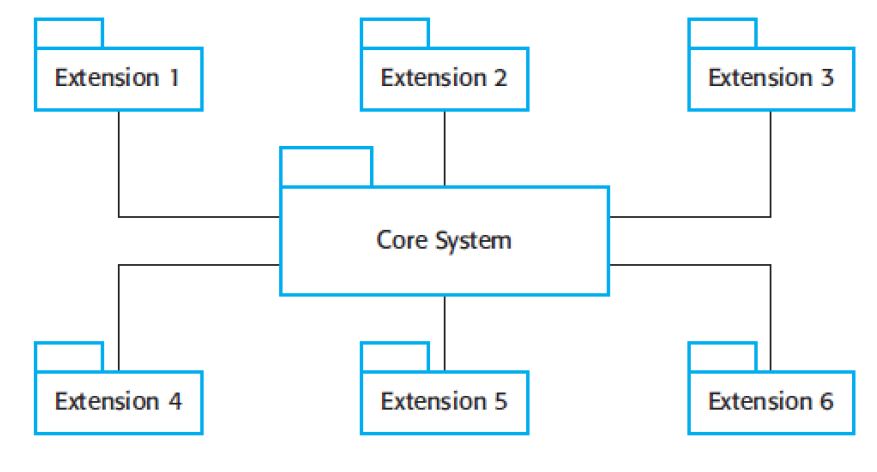
یک شیوه برای جداسازی دغدغه‌ها بررسی آنان از دیدگاه ذینفعان مختلف است. به این شکل که در هر رده از ذینفعان نیاز‌های آنان مشخص شود. و در این بین نیاز آن‌هایی‌ که نیازهای دیدگاه‌های دیگر را تحت تاثیر قرار میدهند به عنوان دغدغه‌های cross-cutting ( مانند مقررات ، امنیت) مطرح هستند این دغدغه‌ها ممکن است نیاز‌هایی‌ را ایجاد کند که تمام یا بخشی از دیدگاه‌های موجود در سیستم را تحت تاثیر قرار دهد.

برای توسعه ی یک سیستم به این روش زمانی‌ که نیازهای هسته ی  اصلی‌ 15شناسایی شد باید الزامات مورد نیاز برای آن را نیز تشخیص داد.

که به صورت بسته‌هایی‌ به سیستم اصلی‌[[20]](#footnote-21) متصل میشود. هر یک از این نیازها الزامات گروهای مرتبط با ذینفعان است.

"اگر نیاز مطابق با دیدگاه ذینفعان سازماندهی شود میتواند آنها را به وسیله ی کشف نیازهای وابسته که در تمام یا اکثر دیدگاه‌ها ظاهر شده اند تجزیه و تحلیل کرد." [[21]](#footnote-22)

با استفاده از این کار می‌توان قابلیت‌های اصلی‌ هسته سیستم را پیدا کرد و نمایش داد و در مورد دیگر نیازها می‌توان آنها را به صورت بسط هایی[[22]](#footnote-23) به قابلیت‌های هسته اضافه کرد.



طراحی و برنامه نویسی جنبه گرا:

"طراحی جنبه گرا فرآیندی از طراحی سیستم است که از جنبه برای پیاده سازی دغدغه‌های متقاطع[[23]](#footnote-24) و بسط هایی[[24]](#footnote-25)‌ که در فرایند مهندسی‌ نیازها تشخیص داده شده اند استفاده می‌کند" [[25]](#footnote-26)

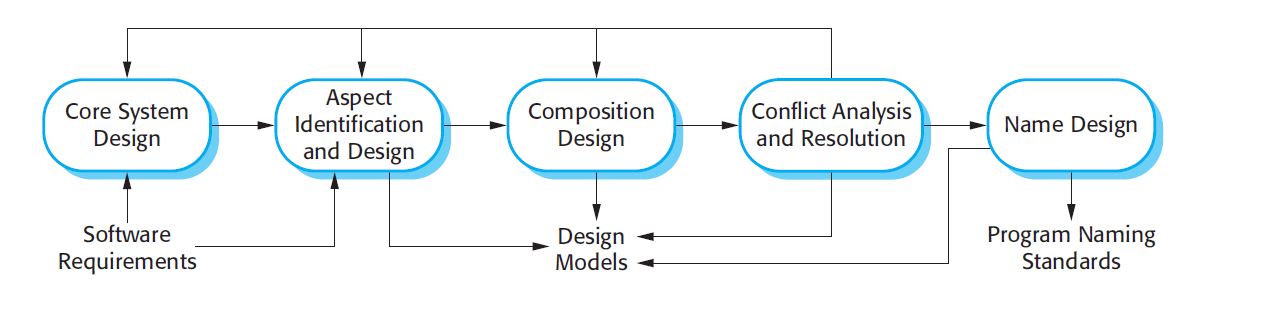
"شما همچنین نیاز دارید تا بدانید که چگونه این جنبه ها با دیگر مولفه های سیستم تر کیب خواهد شد و مطمئن شوید در ترکیب ابهامی به وجود نمیآید"[[26]](#footnote-27)

NG , Jacobsen در کتابی نحوه ی استفاده از موردهای استفاده[[27]](#footnote-28) در فرایند مهندسی جنبه گرا توضیح داده اند. دیدگاه کلی ان ها به این ترتیب است که هر مورد استفاده یک جنبه از برنامه را نمایش می دهد. و بسطهایی[[28]](#footnote-29) که برای پشتیبانی از دیگر موردهای استفاده آمده اند را میتوان به صورت یک جنبه در زبان جنبه گرا پیاده سازی کرد.

همانطور که میدانید در زبان UML مفهوم به نام Extension use case وجود دارد. این موردهای استفاده19 یک وظیفه ی دیگر را به سایر موردهای استفاده اضافه میکنند.

در زمان توسعه موردهای استفاده باید در ان ها به دنبال ویژگی هایی مشترکی گشت که میتوان ان ها را به عنوان بسط هایی به مورد های استفاده اصلی اضافه کرد . به عنوان مثال میتوان به واقعه نگاری اشاره کرد که به عنوان مورد استفاده ی بسط داده شده[[29]](#footnote-30) بر سایر موردهای استفاده بیان کرد.

بر اساس نظر سامرویل فرایند طراحی جنبه گرا شامل مراحل زیر است:



1. طراحی هسته سیستم: در این مرحله شما باید معماری سیستم را طراحی کنید تا از هسته وظیفه گرایی[[30]](#footnote-31) سیستم پشتیبانی کنید، همچنین معماری باید وارد معقوله ی کیفیت نیازمندیهای سیستم مانند احتیاجات قابلیت اعتبار[[31]](#footnote-32) و کارایی شود.
2. شناسایی جنبه و طراحی که با بسطهایی[[32]](#footnote-33) که در مرحله مهندسی نیاز شناسایی شد شروع می شود، شما باید اینها را تحلیل کنید تا ببینید که آیا آنها خودشان جنبه هستند یا باید به چندین جنبه شکسته شوند، وقتی آنها شناسایی شدند آنها می توانند به طور جداگانه طراحی شوند.
3. طراحی ترکیب[[33]](#footnote-34) در این مرحله شما باید هسته سیستم و طراحی جنبه را تحلیل کنید تا کشف کنید کجا جنبه باید با هسته سیستم ترکیب شود. در اصل شما نقاط اتصال برنامه که جنبه در آن بافته (ساخته) می شود را شناسایی می کنید.
4. تحلیل تضاد[[34]](#footnote-35) و resolution: یک مشکل در مورد جنبه این است که ممکن است زمانی که با هسته سیستم ترکیب می شوند در یکدیگر دخالت کنند تضاد زمانی رخ می دهد که یک point cut clash با جنبه های مختلف مشخص می کند که آنها باید در همان نقطه از برنامه ترکیب شوند، وجود داشته باشد. اگر چه ممکن است تضادهایی دقیق[[35]](#footnote-36) بیشتری وجود داشته باشند. وقتی جنبه ها به صورت مستقل طراحی می شوند ممکن است فرضیاتی در مورد وظیفه گرایی هسته سیستم که باید اصلاح شوند بسازد. اگرچه وقتی تعداد زیادی جنبه باهم ترکیب می شوند یک جنبه ممکن است وظیفه گرایی سیستم را تغییر دهد تا اندازه ای که دیگر جنبه ها آن را پیش بینی نمی کنند. رفتار کلی سیستم در آن زمان ممکن است غیر قابل انتظار باشد.

طراحی نام: این یک فعالیت طراحی مهم است که استانداردهایی برای نام گذاری موجودیت ها در برنامه را معین می کند. این مهم است که از مشکل تصادفی (اتفاقی) point cut اجتناب کنیم. این مشکل زمانی رخ می دهد که در تعداد کمی از نقاط های اتصال[[36]](#footnote-37) برنامه، نام به طور تصادفی با نام موجود در طرح point cut یکسان است. بنا براین ندانسته در آن نقطه به کار برده می شود. بدیهی است که این موضوع نا مطلوب است منجر به رفتار غیر منتظره برنامه شود، بنابراین شما باید یک برنامه نامگذاری طراحی کنید تا احتمال کل وقوع این اتفاقات را کمینه کند.

# وارسی و معتبر سازی

"وارسی و معتبر سازی فرایندی است که برنامه با مشخصات مطابقت می کند(وارسی[[37]](#footnote-38)) و نیاز های واقعی ذینفعان را محقق می سازد(معتبر سازی[[38]](#footnote-39))" [[39]](#footnote-40)

فرایند تست و اعتبار سنجی در سیستم هایی که از مدل جنبه گرا استفاده می کنند تفاوتی با دیگر سیستم ها نداردو همانند آن ها هم تست جعبه سیا و هم تست جعبه سفید.

در تست جعبه ی سیاه برنامه نهایی اجرایی به عنوان جعبه ی سیاه تلقی می شود و تست هایی برای ان پیش بینی شده که ان ها را اجرا میکنند و تشخیص داده میشود که ایا سیستم با نیازمندی های خود مطابقت دارد یا خیر. تست جعبه سفید نیز مانند سایر سیستم ها انجام میشود اما وجود جنبه ها مشکلاتی را برای تست با استفاده از کد منبع جهت یافتن عامل خطا ایجاد میکند. چرا که به عنوان مثال جنبه ها بصورت بسط هایی هستند که کد آن در جای دیگر قرار دارد و در زمان اجرا به محل برش[[40]](#footnote-41) بافته میشود. و این جا به جایی باعث سخت تر شدن debug مشود.

بازرسی برنامه توسط تیمی شامل خواننده های کد اجرا میشود که در کد برنامه به دنبال خطا ها هستند. این کد شیوه موثر برای کشف خطا میباشد. در برنامه های جنبه گرا به دلیل جا به جایی (که به آن اشاره شد) خواندن کد به شیوه ترتبیی ( از بالا به پایین) کمی سخت تر است.

یک راه حل این است ، خواننده در یک نگاه صرفا توجه خود را به کد های هر قسمت معطوف کند. و در نگاهی دیگر به این مساله توجه کند که جنبه ها به چه نقاطی باید بافته شود و آیا محل برش 31 به درستی برای جنبه مشخص شده یا خیر. این مساله مستلزم آن است که خواننده کاملا به آن زبان مسلط بوده و مدل join point آن را به خوبی بشناسد.

1. Crosscutting Concern [↑](#footnote-ref-2)
2. سامرویل- Software Engineering- ویرایش نهم - ص 568 [↑](#footnote-ref-3)
3. logging [↑](#footnote-ref-4)
4. **پایان نامه کارشناسی ارشد دانشگاه آزاد اسلامی - وحید عبدل زده -ساله ۸۸\_۸۹ -**  tesis\_abdelzad.pdf ص15 [↑](#footnote-ref-5)
5. object oriented [↑](#footnote-ref-6)
6. event [↑](#footnote-ref-7)
7. wave [↑](#footnote-ref-8)
8. سامرویل- Software Engineering- ویرایش نهم - ص 574 [↑](#footnote-ref-9)
9. سامرویل- Software Engineering- ویرایش نهم - ص 574 [↑](#footnote-ref-10)
10. integration [↑](#footnote-ref-11)
11. aspect weaver [↑](#footnote-ref-12)
12. source code per-process [↑](#footnote-ref-13)
13. Link-Time [↑](#footnote-ref-14)
14. Dynamic weaving at execution time [↑](#footnote-ref-15)
15. مهندسی نرم افزار-سید مصطفی مفیدیان-فصل 3- مهندسی نیازها-ص1 [↑](#footnote-ref-16)
16. **پایان نامه کارشناسی ارشد دانشگاه آزاد اسلامی - وحید عبدل زده -ساله ۸۸\_۸۹ -**  tesis\_abdelzad.pdf ص11 [↑](#footnote-ref-17)
17. Extensions [↑](#footnote-ref-18)
18. سامرویل- Software Engineering- ویرایش نهم - ص 593 [↑](#footnote-ref-19)
19. Core System [↑](#footnote-ref-20)
20. core system [↑](#footnote-ref-21)
21. سامرویل- Software Engineering- ویرایش نهم - ص 578 [↑](#footnote-ref-22)
22. extensions [↑](#footnote-ref-23)
23. cross-cutting conceits [↑](#footnote-ref-24)
24. extensions [↑](#footnote-ref-25)
25. سامرویل- Software Engineering- ویرایش نهم - ص 580 [↑](#footnote-ref-26)
26. سامرویل- Software Engineering- ویرایش نهم - ص 580 [↑](#footnote-ref-27)
27. use case [↑](#footnote-ref-28)
28. Extension [↑](#footnote-ref-29)
29. Extension use case [↑](#footnote-ref-30)
30. core functionality [↑](#footnote-ref-31)
31. detentions [↑](#footnote-ref-32)
32. extension [↑](#footnote-ref-33)
33. composition [↑](#footnote-ref-34)
34. confliction [↑](#footnote-ref-35)
35. subtle [↑](#footnote-ref-36)
36. join points [↑](#footnote-ref-37)
37. Verification [↑](#footnote-ref-38)
38. Validation [↑](#footnote-ref-39)
39. سامرویل- Software Engineering- ویرایش نهم - ص 584 [↑](#footnote-ref-40)
40. Point cut [↑](#footnote-ref-41)