مجموعه مقاله‌های دومین همایش ملی فناوری‌های نوین در مهندسی برق و کامپیوتر
۱۳ شهریور ۱۳۹۳، دانشگاه آزاد اسلامی واحد فسا،
ELECCOMP2-xxxx

بررسی مشکل شکنندگی نقاط قطع در جنبه گرایی

جلیل عظیم پور[۱*]، محمد اسفندیاری[۲]

[۱*] عضو هیات علمی دانشکده مهندسی و مدیر فناوری اطلاعات و ارتباطات دانشگاه آزاد اسلامی واحد بوشهر (نویسنده مسئول) azimpour@iaubushehr.ac.ir
[۲]مدرس و مدیر گروه رشته مهندسی تکنولوژی نرم افزار دانشگاه آزاد اسلامی واحد دلوار m.esfandiari@mail.sbu.ac.ir

چکیده

توسعه مبتنی بر جنبه در حال حاضر مورد توجه محققان زیادی به خصوص محققان مهندسی نرم‌افزار است. این مدل توسعه با استفاده از خاصیت پیمانه بندی قوی نسبت به کاهش پیچیدگی در سیستم های نرم افزاری عمل می کند. در این متد، دغدغه های سیستم به دو دسته دغدغه اصلی و مداخله ای تقسیم می شوند. شناسایی دغدغه ها را می توان با استفاده از راه حل هایی که تا کنون ارائه شده اند انجام داد. اما مشکلی که در این مقاله به آن می پردازیم مشکل شکنندگی نقاط قطع است. این مشکل به دلیل عدم اتصال صحیح این دو دغدغه به یکدیگر می باشد. این عدم اتصال صحیح بدلیل تعریف ناصحیح نقاط قطع می باشد. در صورت عدم اتصال صحیح این دو دغدغه به یکدیگر ممکن است نتیجه حاصل با نتیجه اصلی نرم افزار یکسان نباشد. همچنین درصورتیکه نقاط قطع صحیح تعریف نشوند نگهداشت سیستم بسیار مشکل و هزینه بر خواهد بود. لذا در این تحقیق نحوه تعریف نقاط قطع را بیان می کنیم و مزایا و معایب هر کدام را بیان می کنیم.

کلیدواژه‌ها: جنبه گرائی، برنامه نویسی جنبه گرا، توسعه نرم افزاری جنبه گرا، نقطه قطع ، دغدغه

مقدمه

شکنندگی Pc[۱] ها یک مشکل بسیار مهم و اساسی در برنامه‌نویسی جنبه‌گرا است. این مشکل بدینگونه است که با توجه به اینکه تغییرات یک امر غیر قابل انکار در توسعه نرم‌افزار است، اگر این تغییرات در کد پایه[۲] صورت گیرد، این تغییرات باعث می‌شود تا یک سری از Jp [۳] ها به اشتباه یا در حوزه یک Pc قرار گیرد و یا یک Jp که باید در این حوزه قرار گیرد، در نظر گرفته نشود[۱][۲][۳][۴]. پس می‌توان Pc صحیح را بدینگونه تعریف کرد:

Pc درست و صحیح، آن نقطه انقطاعی است که در نسخه غیر قابل تغییر آن، قادر باشد Jp های صحیحی را در نظر بگیرد[۱].

با توجه به اینکه تغییرات در نرم‌افزار امری غیر قابل انکار است، اگر Pc ها به گونه‌ای تعریف شوند که در صورتی که تغییری در کد پایه برنامه صورت گیرد، مجبور باشیم که در پی آن تغییرات، تعریف Pc ها را نیز دستکاری کنیم، اصطلاحا به این Pc ها، نقاط انقطاع شکننده گفته می‌شود. این دستکاری خود می‌تواند باعث تاثیرات موجی در نرم‌افزار خواهد شد[۱].

دلیل اینکه تغییرات در تعریف Pc امری بسیار جدی می‌باشد و تمامی سعی متولیان جنبه‌گرایی بر این است که در تعریف Pc ها تغییراتی ایجاد نشود این است که صحت تغییرات در تعریف Pc گارانتی شده نیست[۲]. بدین معنا که برای مثال اگر در زبان جاوا نام متدی تغییر کند، تمامی کلاس‌هایی که از آن متد استفاده کرده‌اند در زمان کامپایل خطا می‌دهند و برنامه نویس می‌تواند آنها را در زمان کامپایل مشاهده و تغییر دهد. اما در تغییر تعریف Pc هنوز هیچ ابزاری تعریف نشده است که بتواند صحت آن را گارانتی کند و یا اینکه تغییراتی که در کد پایه داده می‌شود بررسی کند که قبلا این Jp جز Pc یک بوده است اما با اعمال تغییر جدید دیگر جز آن نخواهد بود. پس نتیجه تغییرات در زمان اجرا مشخص می‌شود و پیدا کردن و تصحیح این جمله خطاها که در نتیجه برنامه تاثیر می‌گذارند بسیار کار مشکلی می‌باشد.

نکته قابل توجهی که در مورد Pc ها وجود دارد این است که Pc ها ذاتا شکننده هستند[۲]. به بیان بهتر همانگونه که قبلا گفته شد تغییرات در نرم‌افزار امری غیر قابل انکار است. حال اگر این تغییرات در هر یک از سه حوزه زیر باشد بر روی معنا و مفهوم تعریف Pc تاثیر بسزایی خواهد داشت و باعث می‌شود تا در تعریف Pc ها تجدید نظر کرد و آنها را تغییر داد. که این تغییرات خود می‌تواند باعث شکنندگی Pc شود. این تغییرات عبارتند از[۲][۳]:

- تغییر نام کلاس، متد یا متغیر: این نوع تغییرات بر روی call, execution,set/get تاثیر می‌گذارد.
- جابجایی متد و کلاس: با توجه به اینکه Pc ها مکان لغوی Jp ها را در نظر می‌گیرند، پس بر روی within, withinCode تاثیر می‌گذارد.

۱- Pointcut fragility
۲- Base code
۳- Join point

- اضافه و حذف کلاس، متد و متغیر: اضافه کردن موجودیت جدید ممکن است باعث شود تا این موجودیت با یکی یا چندین تعریف Pc همخوانی داشته باشد و به صورت ناخواسته کد Advice جنبه مربوطه برای آن موجودیت جدید اضافه شود و حذف موجودیت باعث مشکل گم شدن Jp خواهد شد.
- تغییر امضای متد: execution, call هر دو می‌توانند بر روی امضای متد کار کنند و در صورتی که امضای متدی تغییر کند می‌تواند بر روی این دو عملگر اثر گذارد.

به طور خلاصه سعی متدهای مختلف توسعه نرم‌افزار هموار سازی مسیر نگهداشت سیستم است. اما در متد جنبه‌گرایی این مسیر نه اینکه هموارتر نشده بلکه بسیار سخت و طاقت‌فرسا می‌باشد که دلیل اصلی این موضوع بررسی مداوم Pc ها می‌باشد[۴].

مشکل شکنندگی Pc کاملا به نحوه تعریف آنها برمی‌گردد. یعنی برنامه‌نویس با توجه به استفاده از تکنیک‌های اشتباهی ممکن است این مشکل را ایجاد کند که باعث می‌شود نرم‌افزار ایجاد شده، نتیجه مورد نظر را در بر نخواهد داشت. اما در بعضی از روش‌ها برنامه نویس می‌بایستی یک دید کامل و جامع نسبت به تمامی بخش‌های نرم‌افزار داشته باشد تا بتواند کدهای خود را بنویسد که این مسئله در بعضی از موارد امکان پذیر نخواهد بود و یا شاید مشکل ساز باشد[۲].

بررسی روش های تعریف نقاط قطع:

۱- رابط نشانگر[۱]

در ابتدا به این موضوع می‌پردازیم که رابط نشانگر چیست؟ و در این روش Pc ها چگونه تعریف می‌شوند؟ و درنهایت به نقاط قوت و ضعف آن بپردازیم.

رابط نشانگر یک رابطی[۲] می‌باشد که هیچگونه متدی ندارد. و اگر کلاسی فرزند این نوع از روابط شود تنها مانند این است که یک نشانه بر روی آن خورده است مانند رابط Serializable در جاوا. در این روش از این روابط استفاده می‌شود تا کلاس‌های مورد نظر را شناسایی کند و کدهای Advice را بر روی آنها اجرا کند.

نکته قابل ذکر در این روش این است که رابط مذکور می‌تواند در دو قسمت قرار گیرد:

- در قسمت کد پایه
- در قسمت جنبه

یکی از تفاوت‌هایی که بین این دو مورد وجود دارد این است که ممکن است آن رابطی که در کد پایه وجود داشته باشد، متهایی را نیز شامل شود که تمامی کلاس‌های فرزند آنها را پیاده سازی کرده باشند.

همانگونه که می‌دانیم موجودیت‌های شی‌گرا عبارتند از کلاس، متد، متغیر. حال اگر بخواهیم از این روش استفاده کنیم و اگر بخواهیم برای تمامی موجودیت‌های مختلف یک کلاس و یا کلاس‌های موجود را مدنظر گرفته و برای آنها کدهای Advice مختلفی را اختصاص دهیم، باعث پیچیدگی بیش از اندازه Pc شده که اصلا این عمل منطقی نیست.

مشکلات دیگر این روش عبارتند از:

۱- اگر دغدغه‌ای مداخله‌ای میان چندین جنبه پخش شده باشد. آنگاه هر جنبه می‌بایستی کلاس‌های مورد نظر را فرزندان روابط مختلف کنند که باعث پیچیدگی کد خواهد شد.

۲- مسئله استفاده مجدد: اگر کلاسی تعریف شد و این کلاس رابطی را به پدری پذیرفت، حال این کلاس می‌خواهد وارد سیستمی دیگر شود که این رابط نقشی دیگر را برعهده داشته باشد و مفهومی دیگر را در برداشته باشد، اینجاست که این وش جوابگو نخواهد بود و می‌بایستی با دستکاری در کد Pc آن را تغییر داده که همین تغییرات می توانند ایجاد کننده تاثیرات موجی بوده و Pc را بسیار شکننده کند.

۳- در این روش با توجه به یک سری if/else هایی مشخص کرده‌ایم که برای چه کلاسی چه اتفاقی بیافتد. زیرا ما نمی‌خواهیم برای همه کلاس‌ها به یک شکل عمل کنیم یعنی نمی‌خواهیم برای همه کلاس‌ها یک کد Advice خاص اجرا شود و عملا این عمل باعث می‌شود که برای هر کلاس جدیدی آن را دستکاری کرد و همچنین با این عمل تنها مسئله شکنندگی Pc را یک قدم به عقب انداخته‌ایم و آن را درون کدهای Advice قرار داده‌ایم.

اگر رابط در کد اصلی بیاید باز وضعیت کمی بهتر خواهد بود. زیرا در این حالت عموما آن رابط یک سری متدهایی را داراست و کلاس‌های فرزند یک اشتراکی را با یکدیگر دارند و این موضوع می‌تواند کمی پیچیدگی را کمتر کند[۱].

۲-استفاده از Wild-cards

یکی از روش‌های دیگری که برای تعربف Pcها معرفی شده است استفاده از Wild-cardها می‌باشد. این روش تا حدودی شبیه به استفاده از عبارات منظم[۳] می‌باشد. مزیتی که این روش دربردارد، این است که Pcها تا حد قابل توجهی عمومی شده و قابل حمل خواهند شد و همینطور باعث می‌شود تا به دلیل تغییراتی در کد پایه نیازی به

۱- Marker Interface
۲- Interface
۳- regular expression

دستکاری در تعریف Pc نداشته باشیم که این مسئله تا حدودی جلوی مشکل شکنندگی Pc ها را خواهد گرفت.

اما ایراد این روش را می‌توان اینگونه بیان کرد که با توجه به استفاده از Wild-card ها گستره بسیار زیادی از Jp ها را شامل خواهد شد و این امر این مسئله را در پی خواهد داشت که بعضی از Jpها ناخواسته در حوزه یک Pc قرار گیرند که این مسئله احتمال وجود شکنندگی Pc را افزایش خواهد داد[۱]. همچنین این روش بر پایه یک قانون نامگذاری[۱] استفاده می‌کند که این قانون توسط کامپایلر بررسی نمی‌شود و کاملا برعهده برنامه‌نویس می‌باشد و صحت آن گارانتی شده نمی‌باشد. این موضوع احتمال خطا و بروز مشکل شکنندگی Pc را بسیار افزایش می‌دهد[۲].

نمونه‌ای از این روش را در شکل ۱ می‌بینید.

execution(*m*(..))

شکل ۱: نمونه‌ای از استفاده از wild-card

در شکل ۱ گفته شده است که هر متدی که در آن حرف m وجود داشته باشد را شامل شود. خوب این مسئله گستره بسیار زیادی را از متدها را شامل خواهد شد و ممکن است متدهایی وجود داشته باشند که در آنها حرف m استفاده شده باشد اما نباید جز این Pc قرار گیرند.

۳- Pc با عبارت‌های استثناکننده[۲]

این روش تا حد بسیار زیادی شبیه روش قبل می‌باشد. یعنی در این روش نیز از Wild-card استفاده می‌شود. اما دلیل به وجود آمدن این روش برطرف کردن بعضی از ایرادات روش قبل می‌باشد.

همانگونه که در بخش قبل بیان شد، با توجه به اینکه استفاده از Wild-card گستره بسیار زیادی از Jp را شامل می‌شود و این امر باعث می‌شود تعدادی از Jpها ناخواسته در حوزه یک Pc قرار گیرند که باعث مشکل شکنندگی Pc خواهد شد.

در این روش می‌خواهیم با استفاده از یک‌سری عباراتی آن Jp هایی که نباید جز این Pc قرار گیرند را مشخص کنیم. به این عبارات، عبارت‌های (شرط‌های) استثنا کننده گفته می‌شود. با استفاده از این روش ایرادی که بر روش قبل یعنی استفاده از Wild-cardها بود برطرف می‌شود.

اما تا حدودی این روش شبیه روش اول یعنی استفاده از رابطه‌های نشانگر است. این شباهت را می‌توان اینگونه بیان کرد که در هر دو روش قوانین به صورت عمومی می‌باشد اما تا حدودی گستره آنها متفاوت است. بدین معنی که در روش رابطه‌های نشانگر هر کلاسی که رابط مورد نظر را به پدری بپذیرد شامل آن Pc می‌شود. اما در این روش قانونی که در نظر گرفته می‌شود، همه Jp ها را شامل نمی‌شود.

در این روش تمامی Jpهایی که می‌بایستی در Pc مورد نظر قرار گیرد را جمع آوری می‌کنیم و در میان آنها الگویی را پیدا می‌کنیم که برای اکثر آنها مناسب باشد. حال باید راهی را در نظر بگیریم تا بقیه Jp هایی که با آن الگو سازگار نبودند را نیز به گونه‌ای در آن Pc قرار دهیم. در همین‌جا است که باید از عبارات استثنا کننده استفاده کرد. همینطور برای جدا کردن Jp هایی که با الگو سازگار بودند اما در لیست مورد نظر ما نبودند، می‌بایستی از عبارت‌های استثنا کننده استفاده کرد[۱].

اگر سیستم ما کوچک باشد این روش می‌تواند مناسب باشد، اما اگر ما با یک سیستم بزرگ روبه‌رو باشیم این روش خود می‌تواند عاملی بر مشکل شکنندگی نقاط انقطاع باشد. چون در سیستم‌های کوچک حجم نقاط الحاق بسیار کم است پس حالات استثنا بسیار کم و یا حتی ممکن است وجود نداشته باشد. اما در سیستم‌های بزرگ با توجه به اینکه تعداد نقاط الحاق زیاد است و پیدا کردن الگوی مشترک میان آنها که بتواند همه آنها را شامل شود بسیار کار مشکل و شاید بتوان گفت امکان پذیر نیست. پس در سیستم‌های بزرگ حالات استثنا بسیار به وفور دیده می‌شود. پس به ازای همه این حالات استثنا باید در تعریف Pc ذکر شود. حال این موضوع را اینگونه در نظر بگیرید، اگر بخواهیم تغییراتی را در سیستم اعمال کنیم، با توجه به اینکه تعریف Pcها بسیار به کد پایه وابسته است، به نوعی باید تعریف Pc را نیز تغییر داد. همین تغییر دادن Pc سیستم را با مشکل شکنندگی Pc مواجه می‌سازد.

۴-دغدغه‌های مداخله‌ای همگن و ناهمگن

دغدغه‌های مداخله‌ای به دو نوع همگن و ناهمگن تقسیم می‌شود[۵]. دغدغه‌های مداخله‌ای همگن بدین صورت است که چندین Jp را با یک کد advice همگام می‌سازیم و نوع ناهمگن آن بدین صورت است که هر Jp با یک کد Advice همگام می‌شود[۱].

حال به بررسی این موضوع می‌پردازیم که کدام مدل برای جلوگیری از مشکل شکنندگی Pc بهتر عمل می‌کند. هرچند برای این مورد هنوز راه حل کلی بیان نشده است، اما دیدگاه‌های مختلفی این مسئله را بررسی کرده‌اند. از یک دیدگاه دغدغه‌های مداخله‌ای ناهمگن بسیار مستعد مشکل شکنندگی Pc می‌باشد. زیرا Pc این‌گونه دغدغه‌ها بسیار به کد پایه وابسته می‌باشد و در صورت اعمال تغییرات در کد پایه می‌بایستی در تعریف Pc ها نیز تغییراتی را نیز اعمال کرد. اما از دیدگاه دیگر با توجه به اینکه در دغدغه‌های مداخله‌ای همگن یک کد Advice به چندین Jp وصل می‌شود پس تعریف Pc آنها بسیار کلی بوده و ممکن است تعدادی Jp را ناخواسته شامل شود که این مورد همان شکنندگی Pc می‌باشد[۱].

۵-Pc های مبتنی بر مدل

۱- naming rules

۲- Exception Clause

Pc های مبتنی بر مدل، Pc هایی هستند که دیگر بر خلاف روش‌های ذکر شده در قبل به کد پایه وابسته نمی‌باشند، بلکه به مدل مفهومی برنامه وابسته می‌باشند و چون این مدل مفهومی در طول توسعه نرم‌افزار تقریبا ثابت باقی می‌ماند، پس این نوع Pc ها دچار تغییر کمتری می‌شوند و بدین گونه است که مشکل شکنندگی Pcها را کاهش می‌دهد[۴].

این روش بدین‌گونه عمل می‌کند که عملا مشکل شکنندگی Pcها را به مشکل دیگری که راه حل‌های فراوانی برای آن در نظر گرفته شده و اثبات شده تبدیل می‌کند. مشکل جدید، همگام‌سازی مدل مفهومی مذکور با برنامه مورد نظر می‌باشدکه با استفاده از مستند سازی همگام و قوی مدل مفهومی این امر میسر می‌گردد[۴].

همانگونه که در قبل گفته شد این روش به جای وابستگی به ساختار کد پایه، به مدل مفهومی برنامه وابسته می‌باشد. در روش‌های مرسوم، تعریف Pcها به اصل کد پایه وابسته می‌باشد و قوانینی که توسط Pc وضع می‌شود، می‌بایستی توسط برنامه نویسان کد پایه در نظر گرفته شود و از تمامی این قوانین با خبر باشند. همچنین برنامه نویسان کد پایه می‌بایستی بسیار با نظم بوده تا این قوانین را رعایت کنند چراکه این موارد از قوانین نحوی طبعیت نمی‌کنند و در صورت وجود خطا برنامه نویس متوجه آن نخواهد شد[۴]، لذا در روش مبتنی بر مدل، وابستگی Pcها را بر روی مدل مفهومی برنامه قرار داده‌اند. این مدل مفهومی برنامه شامل یک سری قوانینی است که در زمان مدل کردن سیستم وضع می‌شود و در طول توسعه نرم‌افزار ثابت می‌ماند[۴].

در این روش متد ها در مدل مفهومی کلاسه بندی می شوند و فقط بررسی می شود که آیا امضای متدی در کلاس مورد نظر وجود دارد یا خیر و چون مدل مفهومی تغییر ناپذیر است پس نیازی تعریف Pc تغییر نخواهد کرد و مشکل شکنندگی Pc در این حالت بسیار بهبود یافته است.

اولین ایرادی که بر این روش وارد است این است که این روش با اما و اگرهای زیادی روبروی می‌باشد. بدین معنا که استفاده از این روش همیشه امکان پذیر نیست. مزیت اصلی این دیدگاه بر روی این فرضیه قرار دارد که این مدل مفهومی به مدل اصلی برنامه نسبت به کد اصلی حساس‌تر است. این فرضیه در صورتی درست می‌باشد که بتوان سیستم نرم‌افزاری را بر اساس یک سری الگوهای طراحی توصیف کرد که این الگوها پایدارتر از خود سیستم نرم‌افزاری باشد. پس اگر این امر میسر نگردد این وابستگی به مدل مفهومی کاهش خواهد و به کد اصلی افزایش می‌یابد. مثالی دیگر در این مورد عبارتند از الگوهای طراحی وظیفه دسته بندی عناصر پایه یا همان بخش اصلی برنامه را بر عهده دارند که این دسته بندی بر اساس خصوصیات ساختاری و رفتاری آنها می باشند. نکته قابل توجه در این است که الگوهای طراحی به خصوصیات جزئی و دقیق نرم افزار بستگی ندارد. با توجه به این نکته می توان گفت که عناصر پایه نرم افزار یا همان بخش اصلی برنامه می تواند گسترش پیدا کند به شکلی که الگوی طراحی و کلاسه بندی که توسط الگوی طراحی صورت گرفته بدون تغییر می ماند. بنابراین اگر زبان تعریف Pcها اگر به جای اینکه به عناصر اصلی برنامه اشاره کند به عناصر الگوی طراحی اشاره کند مشکل شکنندگی Pc به حداقل می‌رسد. اما نکته‌ای که در اینجا حائز اهمیت است این است که این به حداقل رسیدن تا زمانی صورت می‌گیرد که مواردی که Pc با آنها درگیر است و مد نظر دارد در الگوی طراحی کمتر از تعداد مواردی باشد که در بخش اصلی برنامه وجود دارد. علاوه بر این، این دیدگاه که Pc ها به عناصر الگوی طراحی اشاره کند، قابلیت استفاده مجدد تعریف Pc ها را افزایش می دهد البته این افزایش قابلیت استفاده در صورتی بدست می‌آید که الگوهای مورد استفاده در نرم‌افزارهای زیادی مورد استفاده قرار گرفته باشند.

مشکل دیگری که این دیدگاه دارد را می‌توان اینگونه بیان کرد که این دیدگاه به زبانی که این مدل مفهومی را توصیف میکند، بالاخص طریقه اتصال بخش‌های کد اصلی و این مدل مفهومی، بستگی دارد. برای مثال اگر زبانی جهت اتصال از String Pattern Matching بر روی عناصر کد اصلی مانند متدها و متغیرها استفاده کند، توسعه و گسترش نرم‌افزار بسیار مشکل شده و استحکام این مدل مفهومی جهت توسعه نرم‌افزار به خطر می‌افتد.

مشکل دیگری که می‌توان برای این دیدگاه در نظر گرفت این است که پیروان این دیدگاه، نمی‌توانند یک تکنیک توسعه نرم‌افزار را ارائه دهند و تاکنون ارائه نداده‌اند.

۶- Pc مبتنی بر طراحی

در این دیدگاه قدمی بزرگ در برطرف کردن مشکل شکنندگی Pc ها برداشته شد. این روش برخلاف روش‌هایی که تاکنون مورد بررسی قرار گرفته است که بر اساس ساختار برنامه ارائه شده‌اند، بر اساس رفتار برنامه بنا شده است. در این روش مسئله را با تغییر روش بیان Jp حل کرده است[۶].

این روش را با یک مثال بیان می‌کنیم تا بتوانیم نقاط قوت این روش را هر چه بهتر بیان کنیم. فرض کنیم که می‌خواهیم یک بافر را پیاده سازی کنیم. فرض کنید این بافر دو متد get و put داشته باشد که به ترتیب مقدار بافر را بر می‌گرداند و بعدی مقداری را در بافر قرار می‌دهد. حال می‌خواهیم برای این کلاس یک جنبه ایجاد کنیم. دو جنبه که Pc های آن به دو روش مختلف سنتی تعریف شده است، پیاده سازی می کنیم. در جنبه اول Pc به روش ساده تعریف شده است و عینا نام متدها بیان شده است و در جنبه دوم با استفاده از Wild card نقاط انقطاع تعریف شده است.

حال می‌خواهیم دو تغییر را در کد اصلی یعنی در کلاس بافر اعمال کنیم و بعد آنها را بررسی کنیم. این دو تغییر بدین گونه است که:

۱. می‌خواهیم متدی به نام putAll(int[]) را به این کلاس اضافه کنیم. اگر تغییر توسط جنبه ۱ دیده نخواهد شد.چراکه در این جنبه عینا نام متد ذکر شده است و برای اینکه این متد نیز شناسایی شود کد این جنبه را دستکاری کنیم که این مسئله مشکل شکنندگی Pc را بوجود می‌آورد. اما توسط جنبه شماره ۲ به خوبی شناسایی می‌شود. چراکه با استفاده از Wild-card ها این امر میسر گردیده است. پس نیازی به دستکاری جنبه شماره ۲ نداریم و به نوبه خود کمی استوارتر از جنبه شماره ۱ می‌باشد. پس دیدیم که استفاده از روش اول بسیار ضعیف عمل کرده است و مشکل شکنندگی Pc را بسیار افزایش می دهد.

۲. در این تغییر می‌خواهیم متدی به نام returnElements(int) را به این کلاس بیافزاییم. این متد وظیفه برگرداندن مجموعه‌ای از داده‌ها با طول دریافتی در متد از داده‌های بافر را بر عهده دارد. این متد توسط جنبه اول شناسایی نمی‌شود چون در این جنبه عینا نام متدها ذکر شده است. در این تغییر جنبه شماره ۲ نیز ضعیف عمل کرده است و نمی‌تواند این متد جدید را شناسایی کند. چرا که با استفاده از Wild-card ها این امر را میسر نمی‌کند. دلیل این امر این است که این دو روش، بر روی ساختار کد اصلی استوار است.

اما روش Pc های مبتنی بر طراحی این مشکل را اینگونه حل می‌کند. در این روش Jp ها را به روش دیگری تعریف کرده و از این طریق مسئله شکنندگی Pc را مرتفع می‌کند. در این روش الگویی برای Jp ها در نظر می‌گیرد و Pc ها با این الگوها کار می‌کنند.

در این روش یک نمودار فعالیت است که الگوی Jp را بیان می‌کند و در آن از امضای متدی استفاده نشده است. مطلب بیان شده در این نمودار به صورت ساده است که با توجه به فیلد(متغیر) مورد نظر در سمت راست یک مقدار دهی می‌باشد و یا اینکه در سمت چپ و یا در دستور return قرار دارد. اگر در سمت چپ مساوی(مقدار دهی، علامت =) قرار گیرد بدین معنی است که قرار است مقداری برای آن در نظر گرفته شود و در الگوی Produce قرار می‌گیرد. اگر در سمت راست یا در دستور return باشد بدین معنی است که قرار است مقدار آن خوانده شود و در الگوی consume قرار می‌گیرد. برای توضیحات بیشتر میتوانید به منابع [۶][۷] مراجعه کنید.

حال که Jp را با تعریف کردیم می‌خواهیم به نحوه تعریف جنبه با استفاده از این الگوها بپردازیم.

در این روش Pc ها به این الگوها متصل هستند و نه از Wild-card استفاده شده و نه اینکه ساختار کد اصلی را می‌بیند بلکه رفتار کد اصلی مورد توجه می‌باشد. با این روش، دیگرمشکل آن متد قبل یعنی returnElements(int) که توسط جنبه های ذکر شده در دو روش بیان شده قبل، حل شده و هر متد دیگری با هر امضایی به صورت اتوماتیک و با استفاده از رفتار آن قابل کلاسه‌بندی می‌باشد. همانگونه که می‌بینید این روش بسیار استوارتر از روش‌های ذکر شده تا کنون می‌باشد.

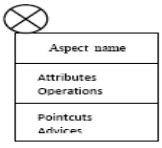
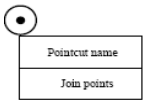
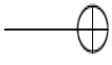
اما تاکنون هرچه که بیان شد مزایای این روش بوده است و هم اکنون می‌خواهیم به نقاط ضعف این روش بپردازیم. این روش در جاهایی کاربرد دارد که بتوان مکانیزم رفتاری آن را مشخص کرد و همچنین در مثال فوق اگر نام متغیر تغییر کرد و یا اینکه چندین متغیر با هم خواستند کار کنند مسئله پیچیده تر خواهد شد و تعریف Pc شاید به مشکل بر بخورد.

۷- استفاده از زبان طراحی جنبه‌گرایی

در این روش پس از طراحی نرم افزار پایه با استفاده از یک سری نمودارها و فازها به صورت گرافیکی جنبه‌ها، Pc و Jp را رسم می‌کنیم و مدیریت می‌کنیم. برای رسم نمودارهای این روش می‌بایستی از زبان طراحی جنبه‌گرایی یا AODL استفاده کرد. این زبان که بر پایه UML بنا شده است، این امکان را می‌دهد تا مفاهیم شی‌گرایی و جنبه‌گرایی را هر دو با هم در یکجا با نمودارها به نمایش در بیاوریم[۸].

این زبان در زمینه جنبه‌گرایی دارای مولفه‌های جدول ۱ می‌باشد. که برای اطلاعات کامل در مورد این مولفه‌های می‌توانید به منبع[۹] مراجعه کنید.

جدول ۱مولفه‌های زبان طراحی جنبه گرایی

جنبه		نقاط انقطاع	
نقاط الحاق		ارتباط دوختن	

در این روش طراحی بر پایه ۳ فاز بعد از طراحی نرم افزار پایه پیشنهاد شده است که عبارتند از:

۱. طراحی نقاط الحاق از دیدگاه ساختاری و رفتار با استفاده از دو نمودار Join point Identification Diagram و Join point Behavioural Diagram
۲. طراحی جنبه‌ها با استفاده از نمودار Aspect Design Diagram
۳. آمیخته شدن جنبه‌ها با مدل برنامه اصلی توسط دو نمودار Aspect-class Structure Diagram و Aspect-class Dynamic Diagram

علاوه بر این نمودارها، نمودار های دیگری نیز نیز وجود دارند که هرکدام به نحوی در این امر، طراح را یاری می‌کنند. مانند Pointcut-Advice Diagram که برای تعریف Pc و ارتباط آنها با Jpها مورد استفاده قرار می‌گیرد[۸].

اما اینکه این روش چگونه می‌تواند از مشکل شکنندگی Pc ها جلوگیری کند، بدینگونه است که با توجه به اینکه در این روش همه چیز به صورت گرافیکی صورت می‌گیرد مدیریت آن بسیار ساده تر می‌باشد.

نتیجه‌گیری

راه حل‌های متعددی برای شکنندگی Pc ها ارائه شدند که هرکدام نقاط ضعف و قوت خاص خود را دارند. راه حلی بهتر است که ارتباط میان کد اصلی و جنبه‌ها را به حداقل برساند. تمامی راه حل‌ها سعی بر کاهش این ارتباط دارند و هر کدام تا حدی موفق به کسب این مورد شوند. اما نکته قابل توجه این است که هیچکدام از روش‌های موجود به صورت صددرصد این مشکل را برطرف نمی‌کند. اما روش های مبتنی بر طراحی بهتر عمل کرده و آن روشی بهتر عمل کرده است که مبتنی بر ساختار کد پایه نباشد بلکه مبتنی بر رفتار کد اصلی باشد.

مراجع و منابع

[۱] P. Greenwood, A. Rashid and R. T. Khatchadourian, Contributing Factors to Pointcut Fragility, *OOPSLA '09 Proceedings of 3rd Workshop on Assessment of Contemporary modularization Techniques (ACoM*), pp. 19-24, Oct 2009,Orlando USA.

[۲] M. Störzer and C. Koppen, PCDiff: Attacking the Fragile Pointcut Problem, *European Interactive Workshop on Aspects in Software,* September 2004, Berlin, Germany.

[۳] M. Stoerzer and J. Graf, Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software, *ICSM '05 Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 653-656, 2005, Washington, DC, USA.

[۴] A. Kellens, K. Gybels, J. Brichau and K. Mens, A Model-driven Pointcut Language for More Robust Pointcuts, *Workshop on Software Engineering Properties of Languages and Aspect Technologies(SPLAT) collocated with AOSD*, March 2006, Bonn, Germany.

[۵] C. Zhang and H. Jacobsen, Efficiently mining crosscutting concerns through random walks, *Proceedings of the 6th international conference on Aspect-oriented software development (AOSD)*, pp. 226-238, 2007, New York.

[۶] W. Cazzola, S. Pini and M. Ancona, Design-Based Pointcuts Robustness Against Software Evolution, *Workshop on Reflection, AOP, and Meta-Data for Software Evolution(RAMSE)*, pp. 35-45, 2006, France.

[۷] W. Cazzola and S. Pini, "Join Point Patterns: a HighLevel Join Point Selection Mechanism," *Workshops and Symposia at MoDELS*, pp. 17-26, 2006, Genoa, Italy.

[۸] S. iqbal and G. Allen, Pointcut Design with AODL, *The Twenty-Fourth International Conference on Software Engineering and Knowledge Engineering(SEKE)*, pp. 418-421, 2012, California, USA.

[۹] S. Iqbal and G. Allen, Designing Aspects with AODL, *International Journal of Software Engineering*, Vol. 4, No. 2, 2011, pp. 3-18

↑ تا حد امکان دو ستون موجود در صفحه آخر را تراز کنید. ↑