بسم تعالی

موضوع : Software Process

دانشکده ملی و مهارت آیت الله خامنه ای

نام و نام خانوادگی : سهیل نعمت الهی

مهندسی نرم افزار

TDD

در **TDD** ، ابتدا تست‌های واحد برای یک ویژگی یا قابلیت جدید نوشته می‌شوند. سپس، کد نوشته می‌شود تا این تست‌ها با موفقیت اجرا شوند. پس از آن، در صورت نیاز، کد بهبود می‌یابد و تغییرات لازم اعمال می‌شود تا تست‌ها به طور پایدار پاس شوند. این چرخه معمولاً به صورت "نوشتن تست، کدنویسی، و اصلاح **(Refactor)"** تکرار می‌شود.

مزایا :

کاهش باگ‌ها و اشکالات: نوشتن تست‌ها قبل از کدنویسی به شناسایی سریع‌تر باگ‌ها کمک می‌کند و باعث کاهش خطاهای احتمالی می‌شود.

کد تمیزتر و ساخت‌یافته‌تر: با استفاده از **TDD**، برنامه‌نویسان بهینه‌تر و منظم‌تر کد می‌نویسند، زیرا هدف، تنها برآورده‌کردن تست‌هاست.

بهبود انعطاف‌پذیری در تغییرات : چون تست‌ها از قبل وجود دارند، اعمال تغییرات در کد آسان‌تر است و می‌توان بدون نگرانی از ایجاد باگ، تغییرات جدید را امتحان کرد .

افزایش قابلیت نگهداری : کدهای مبتنی بر **TDD** معمولاً دارای مستندسازی بهتری هستند که به فهم و نگهداری پروژه در آینده کمک می‌کند.

معایب :

افزایش زمان توسعه اولیه: نوشتن تست‌ها قبل از کد نهایی ممکن است زمان بیشتری ببرد، مخصوصاً برای پروژه‌های پیچیده .

نیاز به یادگیری و تجربه بیشتر: برنامه‌نویسان برای پیاده‌سازی TDD نیاز به مهارت و تجربه کافی دارند که در صورت عدم تجربه ممکن است روش صحیح را رعایت نکنند .

پوشش ناقص تست‌ها: اگر تست‌ها به درستی نوشته نشوند، ممکن است برخی بخش‌های کد بدون پوشش تست باقی بمانند که به باگ‌های غیرمنتظره منجر می‌شود .

کاربرد :

توسعه برنامه‌های بزرگ و پیچیده: TDD در پروژه‌هایی که نیاز به دقت بالا و تغییرات مداوم دارند، مانند سیستم‌های بانکی یا نرم‌افزارهای تجاری کاربرد دارد.

فریم‌ورک‌ها و کتابخانه‌های عمومی :در ساخت فریم‌ورک‌ها یا کتابخانه‌ها که توسط توسعه‌دهندگان مختلف استفاده می‌شود،TDD به بهبود کیفیت و پایداری کمک می‌کند.

محیط‌های توسعه چابک(Agile) : TDD به خوبی با فرآیندهای چابک هماهنگ است و به تیم‌های اسکرام یا XP کمک می‌کند که کدهای قابل اطمینان‌تری ارائه کنند.

FDD

در**FDD** ، پروژه به چندین **ویژگی (Feature)** تقسیم می‌شود که هر کدام نمایانگر یک عملکرد کوچک و قابل استفاده برای کاربر نهایی هستند. هر ویژگی شامل مجموعه‌ای از وظایف توسعه‌ای کوچک است که در طی فرآیندی پنج مرحله‌ای (مدل‌سازی کلی، ساخت لیست ویژگی‌ها، برنامه‌ریزی ویژگی، طراحی ویژگی و ساخت ویژگی) توسعه داده می‌شود.

مزایا :

تقسیم‌بندی پروژه به بخش‌های کوچک‌تر: به دلیل تقسیم پروژه به ویژگی‌های کوچک، مدیریت و پیگیری پیشرفت کار آسان‌تر است.

تمرکز بر ویژگی‌های کاربردی: FDD تمرکز خود را بر ویژگی‌هایی می‌گذارد که برای کاربر نهایی قابل استفاده و ارزشمند است.

افزایش بهره‌وری و انگیزه تیم: با مشاهده تحویل مداوم ویژگی‌های جدید، انگیزه و بهره‌وری تیم افزایش می‌یابد.

بهبود قابلیت ردیابی و مستندسازی: هر ویژگی به‌طور مستقل قابل پیگیری است و به بهبود مستندسازی فرآیند توسعه کمک می‌کند.

معایب :

محدودیت در پروژه‌های کوچک: FDD بیشتر مناسب پروژه‌های بزرگ و پیچیده است و در پروژه‌های کوچک ممکن است هزینه بیشتری به همراه داشته باشد.

نیاز به برنامه‌ریزی دقیق: درFDD ، تهیه و مدیریت لیست ویژگی‌ها و برنامه‌ریزی برای تکمیل آن‌ها نیاز به برنامه‌ریزی دقیق دارد.

تمرکز کمتر روی تست‌های مداوم: برخلافTDD ، FDD بر تست‌های مداوم و خودکار تأکید کمتری دارد و ممکن است نیاز به تست‌های دستی بیشتری داشته باشد.

کاربرد :

پروژه‌های بزرگ و سازمانی: **FDD** برای پروژه‌هایی با تیم‌های بزرگ که ویژگی‌های زیادی دارند مانند سیستم‌های **ERP** یا **CMS** بسیار مناسب است.

پروژه‌های مبتنی بر مشتری: این روش برای پروژه‌هایی که مشتری نیازهای مشخصی دارد و به‌دنبال مشاهده مداوم نتایج است، بسیار کارآمد است.

پروژه‌های با زمان‌بندی مشخص و سخت‌گیرانه: **FDD** به تیم‌ها کمک می‌کند ویژگی‌ها را بر اساس اولویت ارائه کنند و به این ترتیب می‌توان پروژه را بر اساس مهلت‌های زمانی بهتری مدیریت کرد.

BDD

روشی برای توسعه نرم‌افزار است که هدف آن، نوشتن کدی است که رفتار سیستم را به‌صورت شفاف و قابل فهم برای تمام اعضای تیم - از جمله افراد غیر فنی مانند تحلیل‌گران و مشتریان - مشخص کن. **BDD** ترکیبی از توسعه مبتنی بر تست **(TDD)** و تاکید بیشتر بر رفتار و عملکرد نهایی سیستم است.

در**BDD** ، تست‌ها و الزامات نرم‌افزار با استفاده از زبان ساده و طبیعی، معمولاً به زبان انگلیسی یا زبان خاصی به نام **Gherkin**، توصیف می‌شوند. این توصیفات معمولاً به‌صورت سناریوهایی در قالب الگوهایی مانند **Given-When-Then** نوشته می‌شوند که به توصیف شرایط اولیه، اقدامات و نتیجه نهایی می‌پردازند.

مزایا :

ارتباط بهتر با تیم و ذی‌نفعان: BDD باعث می‌شود تا تمامی اعضای تیم، از جمله افراد غیر فنی، به راحتی بتوانند رفتار سیستم را درک و مشارکت کنند.

کاهش سوءتفاهم‌ها: با استفاده از زبان ساده، BDD به کاهش سوءتفاهم‌ها در خصوص نیازهای نرم‌افزار کمک می‌کند و دقت اجرای نیازها را افزایش می‌دهد.

افزایش قابلیت نگهداری و خوانایی کد :توصیفات روشن و سناریوهای تست باعث می‌شود تا کد نوشته‌شده ساخت‌یافته و قابل فهم باشد.

تشویق به نوشتن تست‌های کاربردی: تمرکز BDD بر رفتار کاربر نهایی است، بنابراین تست‌ها و کدها بیشتر به سمت رفع نیازهای واقعی و رفتار مطلوب سیستم می‌روند.

معایب :

نیاز به زمان و منابع بیشتر: BDD نیاز به زمان بیشتری برای تعریف رفتارها و نگارش سناریوهای مختلف دارد، به‌خصوص در پروژه‌های بزرگ.

پیچیدگی در تغییرات زیاد :اگر نیازها یا رفتار سیستم به‌طور مکرر تغییر کنند، نگهداری از سناریوهای BDD دشوارتر خواهد بود.

نیاز به یادگیری ابزارهای خاص: BDD معمولاً با ابزارهایی مثل Cucumber و SpecFlow انجام می‌شود، که تیم باید با آن‌ها آشنایی پیدا کند.

کاربرد :

- **پروژه‌های مشتری محور :در پروژه‌هایی که تعامل با مشتری اهمیت بالایی دارد و نیاز به مشارکت و بازبینی مستمر رفتار سیستم توسط مشتری وجود دارد.**
- **پروژه‌های تیمی و چندبخشی: BDD برای پروژه‌هایی که اعضای تیم در زمینه‌های مختلف (توسعه‌دهندگان، طراحان، تحلیل‌گران، مشتریان) فعال هستند و نیاز به درک مشترک دارند، بسیار مناسب است.**
- **محیط‌های توسعه چابک(Agile) : BDD به‌خوبی با روش‌های چابک تطابق دارد و به توسعه‌دهندگان کمک می‌کند تا رفتارهای مورد انتظار از سیستم را بر اساس تعاملات و ورودی‌های کاربر درک و پیاده‌سازی کنند.**

BDD معمولاً در محیط‌هایی که تأکید بر رفتار و تجربه کاربری وجود دارد، به خوبی عمل می‌کند و به همگرایی تیم‌ها و درک بهتر رفتارهای سیستم کمک می‌کند.

CDD

یک روش توسعه نرم‌افزار است که بر تعریف و رعایت قراردادها یا واسط‌های صریح میان اجزای مختلف سیستم تأکید دارد. هدف **CDD**، بهبود هماهنگی و یکپارچگی بین سرویس‌ها یا ماژول‌های یک نرم‌افزار و جلوگیری از بروز خطاهای ارتباطی است.

در **CDD** ، **قرارداد** به مجموعه‌ای از مشخصات، شرایط و محدودیت‌ها گفته می‌شود که دو یا چند جزء از سیستم باید برای تعامل و تبادل داده‌ها به آن پایبند باشند. این قراردادها معمولاً شامل ورودی‌ها، خروجی‌ها، فرمت داده‌ها، و شرایط پیش‌نیاز و پس‌نیاز هستند. قراردادها به‌صورت صریح در قالب‌هایی مثل **JSON Schema**، **OpenAPI (Swagger)** یا گراف **QL** تعریف می‌شوند و اطمینان حاصل می‌شود که تمامی اجزا با این قراردادها هماهنگ باشند.

مزایا :

کاهش خطاهای ارتباطی: با استفاده از قراردادهای مشخص، احتمال خطاهای ارتباطی بین اجزای مختلف سیستم کاهش می‌یابد و تعامل بین آن‌ها پایدارتر می‌شود.

افزایش قابلیت اعتماد و پایداری سیستم: چون هر جزء دقیقاً می‌داند که از سایر اجزا چه انتظاری دارد و چه داده‌هایی دریافت یا ارسال می‌کند، این موضوع باعث افزایش اعتمادپذیری سیستم می‌شود.

توسعه موازی اجزا: با تعریف قراردادها، تیم‌ها می‌توانند به‌صورت مستقل و موازی بر روی ماژول‌ها کار کنند و با این حال، به یکپارچگی نهایی سیستم اطمینان داشته باشند.

بهبود تست‌پذیری و اطمینان از صحت سیستم: با تعریف قراردادها، تست‌ها بهبود می‌یابند و می‌توان تضمین کرد که هر جزء طبق قراردادهای مشخص عمل می‌کند.

معایب :

نیاز به زمان و منابع برای تعریف قراردادها: تعریف دقیق و کامل قراردادها ممکن است زمان‌بر باشد و نیاز به مشارکت افراد مختلف تیم دارد.

پیچیدگی در تغییر قراردادها: تغییر قراردادها در میانه پروژه ممکن است به مشکلات هماهنگی و بازنگری‌های متعدد در کدها و اجزای سیستم منجر شود.

محدودیت در انعطاف‌پذیری: اگر قراردادها به‌درستی تعریف نشوند یا بسیار سخت‌گیرانه باشند، ممکن است انعطاف‌پذیری تیم در تغییرات آینده کاهش یابد.

کاربرد :

- **سیستم‌های میکروسرویس: CDD** به‌ویژه در معماری‌های مبتنی بر میکروسرویس‌ها که اجزا نیاز به ارتباط دقیق و پایدار دارند، بسیار مفید است.
- **سرویس‌های API:** در پروژه‌هایی که بر روی توسعه **API** تمرکز دارند و اجزای خارجی یا مشتریان به **API** متصل می‌شوند، **CDD** به بهبود هماهنگی کمک می‌کند.
- **سیستم‌های توزیع‌شده**: در سیستم‌هایی که دارای اجزای توزیع‌شده هستند، **CDD** برای هماهنگ کردن رفتار بین ماژول‌ها و جلوگیری از بروز مشکلات ارتباطی مفید است.

CDD با تمرکز بر شفافیت در ارتباطات و رعایت قراردادها، به‌ویژه در پروژه‌های بزرگ و سیستم‌های پیچیده، به‌عنوان یک روش مؤثر برای اطمینان از عملکرد صحیح اجزای مختلف سیستم استفاده می‌شود.

D3

AUP (Agile Unified Process) روشی برای طراحی و توسعه نرم‌افزار است که به تمرکز روی منطق و ساختار حوزه کاری **(Domain)** و مدل‌سازی دقیق آن می‌پردازد.

در**D3** ، ابتدا دامنه کاری به صورت دقیق تحلیل و مدل‌سازی می‌شود. این مدل‌سازی شامل تعریف موجودیت‌ها، ارزش‌ها، و اجزای اصلی دامنه است که به نام ارزش‌ها و موجودیت‌های کلیدی شناخته می‌شوند. همچنین، دامنه به

زیردامنه‌ها تقسیم می‌شود تا اجزا و مسئولیت‌های مختلف به‌خوبی تفکیک شوند. هر زیردامنه در قالب یک **Bounded Context** (محدوده مشخص) قرار می‌گیرد و به عنوان مرزی برای مشخص‌کردن مفهوم و تعاریف خاص دامنه استفاده می‌شود.

مزایا :

هماهنگی بیشتر بین تیم‌های فنی و غیرفنی: **D3** با استفاده از زبان مشترک بین توسعه‌دهندگان و افراد دامنه، از سوءتفاهمات جلوگیری می‌کند.

ساختاردهی بهتر و تفکیک‌پذیری کد: مدل‌سازی مبتنی بر دامنه باعث می‌شود تا اجزای سیستم به‌طور منطقی و ساختاریافته تقسیم شوند.

قابلیت تغییر و توسعه آسان‌تر: به دلیل توجه به اصول دامنه و جداسازی زیردامنه‌ها، افزودن یا تغییر ویژگی‌ها در سیستم ساده‌تر و منظم‌تر انجام می‌شود.

مدیریت پیچیدگی‌های دامنه: **D3** **به‌خوبی از عهده پیچیدگی‌های موجود در دامنه‌های پیچیده و بزرگ برمی‌آید و آن‌ها را به واحدهای قابل کنترل تقسیم می‌کند.**

معایب :

نیاز به زمان و منابع بیشتر: مدل‌سازی و تحلیل دامنه نیاز به زمان و تلاش بیشتری دارد و در پروژه‌های ساده ممکن است پیچیدگی و هزینه را افزایش دهد.

نیاز به تخصص دامنه: تیم توسعه باید با حوزه کاری پروژه آشنایی کامل داشته باشد، و این ممکن است نیازمند آموزش و جلسات مشترک با متخصصان حوزه باشد.

پیچیدگی در پروژه‌های کوچک: استفاده از **D3** در پروژه‌های کوچک ممکن است مزیت زیادی نداشته باشد و تنها به بار اضافی منجر شود.

کاربرد :

پروژه‌های سازمانی و تجاری: D3 برای پروژه‌هایی که دامنه پیچیده و بزرگی دارند، مانند سیستم‌های بانکی، **ERP**، و مدیریت مشتری **(CRM)** بسیار مفید است.

سیستم‌های توزیع‌شده: به دلیل تفکیک‌پذیری Bounded Contextها و مدل‌سازی دقیق دامنه، D3 در سیستم‌های توزیع‌شده که نیاز به هماهنگی بین ماژول‌های مختلف دارند، بسیار کاربرد دارد.

محیط‌های توسعه چابک: **D3** به‌خوبی با روش‌های چابک مانند اسکرام و کانبان هماهنگ است و به تیم‌ها کمک می‌کند نیازهای پیچیده و در حال تغییر دامنه را به‌صورت ساختارمند پیاده‌سازی کنند.

UCD

روشی برای طراحی و توسعه محصولات و خدمات است که در آن نیازها، خواسته‌ها، و محدودیت‌های کاربران در مرکز فرآیند طراحی قرار می‌گیرد. هدف UCD ایجاد محصولاتی است که تجربه کاربری بهتری ارائه دهند و به‌راحتی قابل استفاده باشند.

در UCD ، فرآیند طراحی به‌صورت مداوم بر اساس بازخورد و نیازهای کاربران بهبود می‌یابد. این رویکرد شامل مراحل مختلفی از قبیل تحقیق درباره کاربران، طراحی و prototyping ، تست و ارزیابی است. در هر مرحله، نظرات و تجربیات کاربران جمع‌آوری و در فرآیند تصمیم‌گیری لحاظ می‌شود.

مزایا :

بهبود تجربه کاربری: با تمرکز بر نیازها و خواسته‌های کاربران، **UCD** به ارائه تجربه کاربری بهتر و کاربرپسندتر منجر می‌شود.

کاهش خطاها و نارضایتی‌ها: با انجام تست‌های کاربری و دریافت بازخورد، مشکلات و نواقص قبل از عرضه نهایی شناسایی و برطرف می‌شوند.

افزایش نرخ پذیرش: محصولاتی که با رویکرد **UCD** طراحی می‌شوند، معمولاً با استقبال بیشتری مواجه می‌شوند زیرا به‌خوبی نیازهای کاربران را برآورده می‌کنند.

توسعه مستمر و بهبود محصول: **UCD** به تیم‌ها این امکان را می‌دهد که با دریافت بازخورد مستمر، محصولات را به‌روز و بهبود دهند.

معایب :

زمان‌بر بودن: مراحل تحقیق، تست و دریافت بازخورد ممکن است زمان‌بر باشد و به تأخیر در روند توسعه منجر شود.

نیاز به منابع بیشتر: برای اجرای مؤثر **UCD**، نیاز به تیم‌های چندوظیفه‌ای و منابع اضافی وجود دارد که ممکن است هزینه‌بر باشد.

چالش در تحلیل داده‌ها: جمع‌آوری و تحلیل داده‌های کاربران به‌صورت کیفی ممکن است دشوار باشد و نیاز به مهارت‌های خاصی داشته باشد.

کاربرد :

طراحی وب‌سایت‌ها و برنامه‌های کاربردی: UCD در طراحی وب‌سایت‌ها و اپلیکیشن‌ها برای اطمینان از تجربه کاربری مناسب و کاربرپسند بسیار مؤثر است.

محصولات فیزیکی: در طراحی محصولات فیزیکی مانند دستگاه‌های الکترونیکی، **UCD** به بهبود ارگونومی و کارایی کمک می‌کند.

خدمات آنلاین و دیجیتال: در طراحی خدمات آنلاین، مانند سیستم‌های مدیریت محتوا یا نرم‌افزارهای ابری، **UCD** به بهبود تعامل و تجربه کاربران کمک می‌کند.

UDD

توسعه مبتنی بر استفاده (Use-Driven Development or UDD) رویکردی در فرآیند توسعه نرم‌افزار است که بر اساس نیازها و رفتارهای کاربران نهایی طراحی و پیاده‌سازی می‌شود. در این رویکرد، ابتدا تحلیل عمیقی از نیازهای کاربران انجام می‌شود و سپس ویژگی‌ها و قابلیت‌های نرم‌افزار بر اساس این تحلیل طراحی می‌گردند.

مزایا :

تمرکز بر نیاز کاربر: UDD به تأمین نیازهای واقعی کاربران کمک می‌کند و اطمینان می‌دهد که محصول نهایی به نیازهای بازار پاسخ می‌دهد.

کاهش ریسک: با درک بهتر از نیازهای کاربران، احتمال بروز مشکلات و نیاز به تغییرات در مراحل پایانی کاهش می‌یابد.

انعطاف‌پذیری: UDD به تیم‌های توسعه اجازه می‌دهد تا به سرعت به تغییرات نیاز کاربران پاسخ دهند و قابلیت‌های جدیدی را به نرم‌افزار اضافه کنند.

بهبود کیفیت محصول: با توجه به بازخورد مداوم از کاربران، کیفیت محصول نهایی بهبود می‌یابد.

معایب :

نیاز به زمان و منابع: تحلیل و جمع‌آوری نیازها ممکن است زمان‌بر و پرهزینه باشد.

چالش‌های ارتباطی: ممکن است ارتباط با کاربران برای فهم درست نیازها دشوار باشد و منجر به سوءتفاهم شود.

خطر تغییرات مکرر: به دلیل تأکید بر نیازهای کاربر، ممکن است نیازها در طول فرآیند توسعه تغییر کنند که ممکن است بر زمانبندی پروژه تأثیر بگذارد.

تکیه بر بازخورد: کیفیت نهایی محصول به میزان و دقت بازخورد کاربران بستگی دارد.

کاربرد :

توسعه نرم‌افزارهای وب و موبایل: UDD به ویژه در پروژه‌هایی که به نیازهای سریع کاربران پاسخ می‌دهند، مانند نرم‌افزارهای وب و اپلیکیشن‌های موبایل، کاربرد دارد.

محصولات با ویژگی‌های پیچیده: در پروژه‌هایی که ویژگی‌های پیچیده‌ای دارند و نیاز به تعامل نزدیک با کاربران دارند، **UDD** می‌تواند بسیار مفید باشد.

توسعه Agile : UDD به خوبی با روش‌های توسعه چابک **(Agile)** همخوانی دارد و می‌تواند به بهبود فرآیندهای توسعه کمک کند.

ویژگی	TDD (Test-Driven Development)	FDD (Feature-Driven Development)	BDD (Behavior-Driven Development)	CDD (Component-Driven Development)	D3 (Data-Driven Development)	UCD (User-Centered Design)	UDD (Use-Driven Development)
تعریف	توسعه بر اساس نوشتن تست‌ها	توسعه بر اساس ویژگی‌های مشخص	توسعه بر اساس رفتار و تعاملات	توسعه بر اساس مؤلفه‌ها	توسعه بر اساس داده‌ها	طراحی متمرکز بر کاربر	توسعه متمرکز بر نیازهای کاربر
تمرکز اصلی	تست نرم‌افزار	ویژگی‌ها و قابلیت‌ها	رفتار کاربران	مؤلفه‌ها و ماژول‌ها	داده‌ها و مدل‌های اطلاعاتی	نیازها و تجربه کاربر	نیازهای واقعی کاربران
مزایا	افزایش کیفیت و اطمینان از کارکرد	مدیریت بهتر ویژگی‌ها	بهبود ارتباطات با ذینفعان	ایجاد نرم‌افزارهای ماژولار	انطباق با نیازهای داده‌ای	بهبود تجربه کاربر	پاسخگویی به نیازهای بازار
معایب	نیاز به زمان و منابع بیشتر	ممکن است زمان‌بر باشد	نیاز به درک عمیق رفتار	پیچیدگی در مدیریت مؤلفه‌ها	پیچیدگی در مدیریت داده‌ها	نیاز به تحقیقات عمیق	ممکن است نیازها در حین توسعه تغییر کنند
توسعه تست‌ها	بله	خیر	بله	خیر	خیر	خیر	خیر
رویکرد	چابک و تکراری	تکراری و برنامه‌ریزی شده	چابک و تکراری	تکراری	تکراری و آزمون	طراحی متمرکز	تکراری و با توجه به بازخورد
مناسب برای	پروژه‌های با اهمیت کیفیت	پروژه‌های بزرگ با ویژگی‌های زیاد	پروژه‌هایی با نیازهای پیچیده	پروژه‌های ماژولار	پروژه‌های داده‌محور	پروژه‌های با تمرکز بر تجربه کاربر	پروژه‌هایی با نیازهای متنوع