**Kamil Skierski**

**Programowanie ekstremalne**

Ostatnio coraz głośniej jest wśród programistów o programowaniu ekstremalnym,

tajemniczej idei wywodzącej się z metod agile development znajdującej coraz szersze

zastosowanie w tworzeniu poważnego oprogramowania dla biznesu i sektora publicznego, a

ostatnio nieśmiało wkraczającej do studiów developerskich tworzących gry wideo. W tym

artykule postaram się przybliżyć i wyjaśnić to tajemnicze pojęcie i wskazać dla niego

zastosowania w procesie tworzenia gier komputerowych. Pokażę, gdzie dzisiejsze

kierownictwo większości projektów z branży najczęściej popełnia błędy i jak programowanie

ekstremalne może pomóc w ich wyeliminowaniu.

Podobnie jak sport ekstremalny jest skrajną formą aktywności fizycznej, tak

programowanie ekstremalne jest skrajnym podejściem do tworzenia oprogramowania od

strony technicznej, czyli od strony kodu. W przeciwieństwie jednak do sportu, gdzie ryzyko i

adrenalina są nieraz sensem jego uprawiania i rosną w miarę zdobywania doświadczenia, w

przypadku programowania celem jest właśnie minimalizacja ryzyka, bądź nawet jego

całkowite wyeliminowanie, co jak się okaże – wcale nie jest związane z redukcją liczebności

zespołu developerskiego, czy sztucznym ograniczaniem budżetu. Jednocześnie pozwala na

tworzenie oprogramowania, które będzie odzwierciedlało aktualne potrzeby klienta, bądź

rynku (nawet jeśli zmienią się one znacząco w trakcie prac), co w przypadku gier ma

kolosalne znaczenie – sprostanie stale rosnącym wymaganiom wydawców jest dzięki temu

łatwiejsze.

Zanim przejdziemy do omawiania samej teorii, należy wspomnieć kilka słów o historii

tej ideologii. Metodologia ta narodziła się w 1996 roku, a za jej autora uważany jest

powszechnie Beck Kent, który w tym czasie pracował dla koncernu Daimler-Chrysler. Do

spopularyzowania tej metodologii walnie przyczyniła się jego opublikowana w 1999 roku

książka zatytułowana „eXtreme Programming eXplained”, która doczekała się do dnia

dzisiejszego kilku wydań. Może być ona traktowana jako swoisty manifest tego ruchu.

Publikacja opisywała m.in. 12 głównych zasad (tzw. „najlepsze praktyki”), na których

programowanie ekstremalne jest oparte (dziś zasad tych w niektórych publikacjach jest

wydzielonych nawet 20 i więcej, choć wynika to jedynie z podziału głównych zasad na

bardziej szczegółowe). Każda z tych zasad dotyczy innego aspektu tworzenia kodu. Ze

względu na zastosowanie można je podzielić na 4 znaczące kategorie:

 Planowanie

 Projektowanie

 Implementacja

 Testowanie

Powyższa kolejność choć wydaje się być standardowa i powszechnie stosowana w

przemyśle gier wideo, jest w kontekście programowania ekstremalnego całkowicie

przypadkowa, a nawet – nie do końca prawidłowa.

**Sedno programowania ekstremalnego**

Programowanie ekstremalne jest przeznaczone dla niewielkich zespołów

programistycznych (od 2 do kilkunastu osób), czyli akurat dla zespołów odpowiedzialnych za

programowanie gier. To, że jest tak rzadko przez nie wykorzystywane wynika z błędnego

przekonania, że prowadzi do zwiększenia wydatków na produkcję, tym samym zwiększając

ryzyko. Ponadto wiele osób w naszej branży wychodzi z fatalnego założenia, że kod napisze

się sam, że rozwiązania problemów zostaną znalezione w odpowiednim czasie, tj. wtedy gdy

te problemy wystąpią i przez to nie potrafią przyjąć najważniejszych postulatów

programowania ekstremalnego.

Tym co odróżnia programowanie ekstremalne od innych metodologii jest nacisk, jaki

położono w nim na testowanie. Jest on tak duży, że wielu programistów nie jest w stanie z

tego powodu jej przyjąć i używać na co dzień. Na czym to polega? Otóż przed napisaniem

właściwego kodu aplikacji, czy w tym przypadku gry, należy zaimplementować dla niego

wszystkie testy, przetestować każdy jego fragment! Może się to wydawać niedorzeczne – jak

testować kod, którego jeszcze nie ma?! Ba, który nawet nie wiemy jak będzie wyglądał! I

właśnie w tym tkwi sens tworzenia testów. Tworząc je przed napisaniem kodu zmuszamy się

niejako do gruntownego zastanowienia nad jego przyszłym kształtem oraz nad

funkcjonalnością jaką chcemy mu nadać. Im testy są dokładniejsze, tym lepszą wizję

przyszłego kodu będziemy mieli, a tym samym jego jakość będzie wyższa. Nie mówiąc o

tym, że i implementacja będzie znacznie prostsza (wszystko będzie dokładnie przemyślane i

nie będzie miejsca na tzw. „on-the-fly-programming” powszechnie niestety spotykane w

gamedevie), a liczba błędów zostanie ograniczona do minimum.

Można jednak zadać pytanie, po co najpierw projektować grę w UMLu (tym samym

wyrabiając sobie zdanie na temat kształtu kodu), a potem ją jeszcze testować. Cóż, odpowiedź

jest prosta. Bardzo rzadko w produkcji gier (przynajmniej w Polsce) diagramy w UMLu

faktycznie powstają, a nawet jeśli ma to miejsce, to same w sobie nie gwarantują

niezawodności kodu, gdyż są jedynie pewną formą jego opisu, jego obrazem nie

uwzględniającym faktycznych kwestii implementacyjnych.

Kolejną ważną kwestią w XP jest bardzo dobry kontakt między developerami. Wiele

osób zarzuca programowaniu ekstremalnemu, że prowadzi do marginalizacji dokumentacji,

bądź jej całkowitego zlikwidowania. Nie jest to do końca prawda. Jednak faktem jest, że

metodologia ta przekłada kontakty między developerami nad papierową wersję dokumentacji.

Do tego stopnia, że wymaga ona, aby każdy dzień zaczynał się krótkim, obowiązkowym dla

wszystkich developerów spotkaniem w celu omówienia wątpliwości, napotkanych

problemów, czy znalezienia rozwiązań na nie. W ten sposób zmniejsza się potrzeba

organizowania w ciągu dnia dodatkowych spotkań (z osobami, które na spotkaniu się nie

pojawiły), zaburzających właściwy rytm pracy, a zatem i jej efektywność. Ponadto każdy z

programistów wie co i jak ma danego dnia robić. Developerzy muszą sobie nawzajem

pomagać, a ponadto każdy z nich jest równo odpowiedzialny za tworzony kod (co może się

nawet wiązać z brakiem konieczności istnienia głównego programisty).

Prawda, że trudno się tu z niektórymi ideami zgodzić? To właśnie dlatego tak wiele

zespołów nie decyduje się na postępowanie według reguł programowania ekstremalnego.

**Postulaty programowania ekstremalnego**

***Planowanie***

*Twórz w sposób iteracyjny*

Sensem iteracyjnego tworzenia kodu, jest podzielenie procesu produkcji

oprogramowania na fazy (iteracje, przyrosty) o równej długości (od 1 do 3 tygodni). Choć

podobne zwyczaje pozornie są spotykane w naszej branży, to mają one nieco odmienny

kształt.

Otóż bardzo często opracowuje się pobieżnie wszystko co musi zostać wykonane w

procesie produkcji, a następnie dzieli się to na kolejne wersje, czy też etapy. W ten sposób

powstaje swego rodzaju road-mapa. Choć bywa ona bardzo pożyteczna, pogoń za jej

realizacją prowadzi często do stosowania nadgodzin, tworzenia kodu niskiej jakości, niechęci

do refaktoringu i innych komplikacji oraz nadużyć.

Natomiast według XP na początku projektu wszystkie fazy są puste. Gdy dana iteracja

jest rozpoczynana ustalane jest na spotkaniu, co ma zostać w jej trakcie zrealizowane, czy

zaimplementowane. Developerzy mają prawo wyboru zadań spośród tych, które wspólną

decyzją zostały przyjęte do realizacji. Muszą też oszacować, ile czasu potrzebują na ich

implementację. To oszacowanie jest konieczne. Jedna z prawd dotyczących tworzenia

oprogramowania głosi, że jest to zawsze oszacowanie od dołu. Innymi słowy dane zadanie

nigdy nie zajmie mniej czasu niż to założono, lecz prawdopodobnie nawet więcej. Ponadto

jeśli czas potrzebny na realizację jakiegoś zadania oszacowano na więcej niż długość iteracji,

to znak, że zadanie jest zbyt złożone i trzeba je rozbić na mniejsze.

Po tym spotkaniu zaczyna się realizacja iteracji, trwająca właśnie te 1 do 3 tygodni.

Nic więcej niż zaplanowano nie może zostać zaimplementowane! Co więcej, gdy

widać, że czasu na zaimplementowanie wszystkich założeń jest zbyt mało, nie należy

zmuszać ludzi do nadgodzin, lecz odrzucić te nadmierne założenia (całkowicie lub do czasu

rozpoczęcia kolejnej iteracji). To, że nie udało się zrealizować pierwotnych założeń nie jest

bowiem winą developerów, lecz nierealnego time-line’u.

*Spraw by każdy wiedział, jak działa każdy moduł*

W naszej branży specjalizacje programistów są bardzo silne i równie powszechne.

Mamy programistów-ekspertów z dziedziny programowania:

 Grafiki

 Fizyki

 GUI

 AI

 Narzędzi

 Muzyki

 Mechaniki

Zwykle w ramach zespołu developerskiego jest tylko 1-2 specjalistów z danej

dziedziny (zwykle są to też główni programiści w tym konkretnym obszarze). Ich odejście

może oznaczać dla projektu katastrofę. A nawet jeśli nie, to i tak doprowadzi do zmarnowania

czasu, na nauczenie pozostałych programistów choćby elementarnej obsługi danej części

kodu.

W związku z tym należy zachęcać developerów, aby próbowali zmieniać kod z

dziedziny, w której się nie specjalizują, dzięki czemu będą mieli pewną świadomość jego

działania. Należy ich zachęcać do modyfikowania i oglądania każdego fragmentu programu.

*Często wypuszczaj release*

Wersje testowe, czy tech-dema powstają często, gdy projekt jest już w

zaawansowanym stadium, co niejednokrotnie oznacza, że gra jest już w produkcji od roku

albo dwóch lat. Przy tak zaawansowanej wersji nie jest już możliwe wprowadzenie dużych

zmian w rdzeniu gry bez zaburzenia jej struktury – bez kompletnej katastrofy.

XP wymaga natomiast, aby wersje release wypuszczać jak najczęściej. Dostarczając je

zespołowi testującemu już od początku produkcji, od samego początku otrzymujemy też jego

komentarze i sugestie. Możemy zmienić dowolny fragment gry na tyle wcześnie by nie

narażać projektu na straty w przyszłości, a jednocześnie zapewnić, że docelowy produkt

będzie maksymalnie grywalny i bezbłędny. Takie podejście sugerują też znani projektanci

gier, polecający tworzenie pierwszych prototypów już w fazie preprodukcji.

*Jeśli XP zaczyna zawodzić, zmień je!*

W niektórych projektach kierownictwo trzyma się ściśle wytyczonych reguł. Reguły

natomiast to tylko wskazówki jak prowadzić projekt. W szczególnych okolicznościach może

się okazać, że część reguł programowania ekstremalnego wcale nie pomaga projektowi, a

wręcz mu szkodzi. W takiej sytuacji należy mieć odwagę i reguły te zmienić, dostosowując

do potrzeb projektu! Wynika to z faktu, że jak na razie nie udało się stworzyć metodologii

uniwersalnej, sprawdzającej się w każdym rodzaju projektów.

***Projektowanie***

*Niech projekt będzie prosty*

Im prostszy projekt, tym lepiej. Nikogo nie trzeba przekonywać, że implementacja

prostego projektu wymaga mniej czasu niż implementacja czegoś, co jest skomplikowane.

Prosty projekt oznacza, że nie będzie w nim nadmiarowej funkcjonalności, zbędnych

„udogodnień” – jedynie to, co chcemy aby nasza gra zawierała. I nic więcej! Dodatkową

funkcjonalność będzie można dodać w przyszłości, jeśli będzie wymagał tego rynek bądź też

wydawca. Znacznie łatwiej jest rozwijać coś co jest proste, od samej konserwacji czegoś

bardzo złożonego.

Przykładem mogą być gry cRPG, w których za wszelką cenę dąży się do komplikacji,

stworzenia maksymalnie złożonego systemu gry, uzyskania pełnej nieliniowości. Zamiast 100

umiejętności pierwotnie lepiej zaimplementować 10, gruntownie przetestować, upewnić się,

że gra jest grywalna, a jeśli starczy czasu dodać pozostałe.

*Nie dodawaj nadmiernej funkcjonalności*

Powszechnym błędem jest pisanie kodu, który ma być maksymalnie funkcjonalny.

Okazuje się jednak, że większość tej funkcjonalności (niektóre statystyki mówią o 90%)

nigdy nie zostanie użyta, co oznacza, że czas przeznaczony na jej zaimplementowanie został

po prostu zmarnowany. Często implementowanie tej funkcjonalności oznacza też

ograniczenie czasu na inne, bardziej naglące sprawy, co tylko obniża ich jakość – a tym

samym jakość całej produkcji.

Innymi słowy należy się skupiać na tym co jest potrzebne w danej chwili.

Takie podejście jest powszechne przy tworzeniu silników gier. Zamiast myśleć o

funkcjonalności potrzebnej do tworzonej aktualnie gry, developerzy nieraz wybiegają daleko

w przyszłość i dodają do silnika FPP właściwości, które znajdą zastosowanie w przypadku

strategii z rzutu izometrycznego. Marnotrawstwo czasu!

*Dokonuj refaktoringu, wtedy gdy jest to konieczne*

Ta zasada dotyczy dostosowywania się do zmieniających się potrzeb. Błędem wielu

programistów i projektów jest to, że powstaje rozdmuchany kod, nad którym w pewnym

momencie ciężko zapanować. Wynika to z obaw developerów, że dany fragment kodu, choć

od dawna bezużyteczny (a być może od początku) przyda się w przyszłości. Często też boimy

się modyfikować kodu, który działa, ale można by go napisać lepiej, gdyby zmienić kilka

koncepcji w projekcie. Nie mówiąc o sytuacji, w której boimy się od nowa napisać kod, który

niby działa poprawnie, ale w sumie nikt nie ma pojęcia dlaczego. Zresztą ta ostatnia sytuacja

jest najgorszą z możliwych. Jeśli coś działa, a nie powinno to znaczy, że gdzieś jest błąd,

który ujawni się w najmniej oczekiwanym momencie.

***Programowanie***

*Stosuj ustaloną konwencję*

Bardzo istotne jest by tworzony kod był zgodny z przyjętą przez zespół (bądź firmę)

konwencję programistyczną. W ten sposób każdy znający konwencję programista nie będzie

miał problemów ze zrozumieniem czytanego kodu i jego ewentualnymi modyfikacjami w

przyszłości. Bardzo dobrym pomysłem jest dostarczenie elektronicznej lub papierowej wersji

konwencji. Ponadto nie powinna być ona zbyt długa (maksymalnie kilka stron), by jej

zapamiętanie nie było zbyt trudne dla programistów.

*Najpierw napisz testy, potem właściwy kod*

Zasadę tę zacząłem omawiać we wstępie. Teraz przyjrzę się jej bardziej szczegółowo.

Przy testowaniu bardzo ważnym pojęciem są tzw. „unit test”, czyli test sprawdzający

poprawność działania pewnej jednostki kodu (np. funkcji, modułu, klasy), innymi słowy

sprawdzający poprawność wyjścia pewnego fragmentu kodu. Zaimplementowanie testów

przed faktyczną implementacją kodu zapewnia, że wszystkie sytuacje wyjątkowe, które w

testach przewidziano nie spowodują zakłócenia poprawności działania kodu. Testem może

być np. sprawdzenie poprawności wyników obliczeń, ale i sprawdzenie, jak program poradzi

sobie z nietypowymi danymi wejściowymi (np. dzielenie przez zero, dodawanie liter).

W praktyce zasada ta może zostać sprowadzona do następującego postępowania:

1. Implementacja jednego testu, dotyczącego pewnego małego fragmentu kodu.

2. Implementacja maksymalnie prostego kodu, który zaliczy test.

Kroki te należy powtarzać do momentu aż kod, przybierze docelową postać. Zwróćmy

uwagę na wyrażenie „maksymalnie prostego kodu”, jako że odnosi się ono do innych zasad

XP.

Wielu developerów prawdopodobnie odrzuci ten postulat, gdyż wymaga on z pozoru

dodatkowego czasu, a tego przy produkcji gier i tak zawsze brakuje. Jednak takie myślenie

jest zwykle błędne. W rzeczywistości bowiem, tak jak już wspomniałem:

1. Implementacja właściwego kodu jest znacznie łatwiejsza, co zwykle oznacza

skrócenie czasu potrzebnego na jej wykonanie.

2. Wzrasta jakość kodu – co oznacza, że w późniejszej fazie produkcji będzie można

pominąć wiele miesięcy testowania kodu, szukania w nim potencjalnych błędów,

szukania ich przyczyny i ich poprawiania.

Poza testami jednostek kodu, należy też oczywiście testować współpracę między

poszczególnymi podzespołami (np. klasami, czy modułami).

*Programowanie w parach*

Dla wielu osób jak zauważyłem jest to esencja programowania ekstremalnego. Nie jest

to jednak prawda. Jest to jedynie jeden z postulatów, ale wcale nie najważniejszy. Można

jedynie rzec – nietypowy. Programowanie w parach oznacza, że zespół developerski

podzielony jest na 2-osobowe zespoły pracujące nad tym samym fragmentem kodu przy

jednym monitorze. Idealną jest sytuacja, gdy rzeczywiście mogą tak pracować. W ten sposób,

gdy praktycznie patrzą sobie na ręce, wzrasta jakość kodu. Wyłapywane są drobne błędy, czy

nawet literówki. Programiści uczą się od siebie nawzajem nowych technik i sztuczek. Zresztą

dwie osoby mogą wymyślić rozwiązania, których żadna z nich samodzielnie by nie

wypracowała.

W praktyce dość często spotyka się z sytuacją, że jeden z programistów odpowiada za

tworzenie testów, a drugi za tworzenie właściwego kodu dla tworzonego fragmentu

oprogramowania.

I tu znów wielu developerów powie, że reguła ta wiąże się z dodatkowym wydatkiem.

Ale też nie jest to prawda. Powstający kod jest lepszy, powstaje znacznie szybciej. Nawet jeśli

wiąże się większymi wydatkami na początku, to w dłuższym okresie okaże się trafioną

inwestycją.

*Często integruj kod*

W przypadku profesjonalnych zespołów bardziej niż prawdopodobne jest to, że

będziesz zmuszony do stosowania jakiegoś systemu kontroli wersji (Subversion, CVS). W

związku z tym, gdy wprowadzisz (a raczej twoja para) zmiany do kodu źródłowego musisz

dokonać jego integracji z kodem istniejącym w repozytorium. Nic trudnego. Z pozoru. XP

wymaga bowiem, aby (poza gruntownym przetestowaniem własnego kodu) po dokonaniu

integracji napisać testy sprawdzające poprawność działania połączonych w jedność modułów.

W ten sposób istnieje pewność, że zintegrowany kod jest poprawny. Aby zapewnić, że w czasie gdy jedna para integruje kod, inna go nie zmieniła (tym samym wymuszając na

pierwszej ściągnięcie z repozytorium aktualnej wersji, jej przetestowania, ściągnięcia

aktualnej wersji, i tak w kółko...) w danej chwili integracją może zajmować się tylko jedna

para (pozostałe pracują w tym czasie nad swym kodem).

*Optymalizuj na samym końcu*

Błędem wielu programistów jest według XP to, że pisząc wstępną wersję kodu, od

razu chcą, żeby była ona maksymalnie wydajna. Z pozoru nie ma w tym nic złego, ale takie

podejście zwykle prowadzi do dwóch rzeczy:

1. Kod staje się mniej czytelny, a zatem i cięższy do utrzymania, co ma też znaczenie w

przyszłości, gdy będą w nim potrzebne zmiany. A będą jeśli będziemy stosować się do

innych zasad programowania ekstremalnego.

2. Optymalizując kod już w trakcie jego tworzenia, możemy sprawić, że marnujemy

cenny czas na poprawianie fragmentu, który nie wpłynie znacząco na wydajność całej

aplikacji. Zamiast zgadywać co jest wąskim gardłem, musimy je znaleźć mierząc

wydajność aplikacji (np. profilując), czego możemy dokonać dopiero, gdy system jest

już kompletny. Przykładowo po co optymalizować mnożenie kwaternionów, jeśli

nasza gra wykorzystuje tylko macierze? A w tym samym czasie kod sieciowy nie

zapewnia płynnej rozgrywki wielu graczom.

*Brak nadgodzin!!!*

Paradoksalnie postulat ten jest jednym z ważniejszych. Jednocześnie jest to chyba

najczęściej łamana zasada w branży gier wideo. Siedzenie przy komputerze nawet po

kilkanaście godzin dziennie wpływa druzgocąco na morale i motywację zespołu. Nie mówiąc

już o tym, że kod tworzony w tym czasie jest wyjątkowo niskiej jakości - zmęczony

programista, bez względu na umiejętności, nie jest w stanie dobrze programować kilkanaście

godzin na dobę, popełnia nieraz szkolne błędy, których znalezienie później będzie niezwykle

trudne. Dlatego XP dopuszcza maksymalnie 8 godzin programowania dziennie i 40 godzin w

tygodniu.

Jeśli występuję konieczność stosowania nadgodzin, to wina nie leży po stronie

developerów lecz po stronie kierownictwa projektu, które stworzyło nierealny time-line.

Rozwiązaniem jest więc albo wydłużenie czasu projektu, albo zmniejszenie zakresu projektu.

Kolosalnym błędem jest też zatrudnianie w tym momencie nowych ludzi, gdyż ich

praca będzie nieefektywna, a ponadto inni programiści będą musieli poświęcać im swój czas,

aby nauczyć ich nowych technologii, stosowanych konwencji, itd.

Według Johna Brewera, możliwy jest w szczególnych przypadkach tydzień, w

którym nadgodziny występują. Jeśli jednak wystąpią dwa takie tygodnie pod rząd to oznaka,

że coś jest z projektem nie tak.

*Każdy jest równo odpowiedzialny za kod*

Ten postulat już zacząłem tłumaczyć. Idea polega na tym, że każdy z developerów ma

takie same prawa do kodu i odpowiada za niego w równym stopniu, co oznacza, że może

zmienić każdą linijkę, poprawić każdego buga, czy nawet dokonać refaktoryzacji kodu

napisanego przez innego programistę!

Wynika to z faktu, że ludzie nie są nieomylni. Często wychodzi się z błędnego

przekonania, że wszystko musi być tak jak powie główny programista. Niestety, nawet

główny programista (bez względu na doświadczenie i umiejętności) od czasu do czasu

popełni błąd, albo każe zrobić coś, co przeczy intuicji, czy nawet zdrowemu rozsądkowi.

Zwykle też ostatecznie cała odpowiedzialność i tak spadnie na wykonawcę.

Zresztą w rzeczywistości jest tak, że jeśli specjalista np. od AI zmieni coś w kodzie

renderującym, pewnie zostanie za to co najmniej skrytykowany. Jeśli chcemy pozostawać w

zgodzie z tą i innymi regułami programowania ekstremalnego, nie dość, że taka krytyka nie

może mieć miejsca, to zachowanie takie należy wręcz pochwalać.

***Testowanie***

*Każdy fragment kodu, musi mieć odpowiadające mu testy*

W zasadzie postulat jest dość oczywisty w kontekście naszych rozważań – dla

każdego fragmentu kodu muszą istnieć odpowiadające mu unit testy, sprawdzające

poprawność jego działania. Testy te muszą być napisane przed napisaniem właściwego kodu.

Szerzej o tym zagadnieniu pisałem już w poprzednich podrozdziałach.

*Kod jest umieszczany w repozytorium, dopiero po zaliczeniu wszystkich testów*

Również i ten postulat powinien wydawać się dość klarowny. Zanim zmodyfikowany i

zintegrowany kod zostanie wysłany do repozytorium, musi zwrócić oczekiwany wynik dla

wszystkich swoich testów. Co więcej testy te muszą zostać zaliczone także po dokonaniu

integracji z kodem już znajdującym się na repozytorium.

*Gdy znajdziesz błąd, stwórz dla niego testy*

Gdy w swym programie znajdziesz błąd (co jest więcej niż prawdopodobne), należy

dla niego napisać testy, które zapewnią, że po jego rozwiązaniu w przyszłości bug ten nie

powróci, co leży niestety w ich złośliwej naturze.

**Stosować? Nie stosować?**

To czy w naszych projektach powinniśmy stosować programowanie ekstremalne

wcale nie musi być oczywistością. Jak widać wiele reguł może być z różnych względów

trudnych do zaakceptowania, wiele może być też niezgodnych z polityką prowadzoną przez

nasz zespół. O ile koszty ich stosowania nie są w większości przypadków zbyt wysokie, tak

ich wprowadzenie może nastręczać wielu trudności. Ponadto programowanie ekstremalne

wymaga od programistów dużego doświadczenia i myślenia w odpowiednich kategoriach.

Ograniczenia jakie nakłada programowanie ekstremalne są zatem dosyć wysokie.

Jednocześnie XP oferuje bardzo wiele w zamian: wyższą jakość kodu, elastyczne

oprogramowanie, zadowolenie wydawców. Odpowiedź zatem na postawione w tytule

rozdziału pytanie zależy od tego co jest dla nas ważniejsze.

**Bibliografia**

[1] pl.wikipedia.org/wiki/**Programowanie**\_**ekstremalne**‎

[2] mfiles.pl/pl/index.php/Metodyka\_**Extreme**\_**Programming**‎

[3] www.governica.com/**Programowanie**\_**ekstremalne**‎

[4] wazniak.mimuw.edu.pl/images/3/31/Io-12-wyk-bw.pdf‎