МЕТОДИ ЗА ДИЗАЙН НА СОФТУЕРНАТА АРХИТЕКТУРА

Тактики за удовлетворяване на качествените изисквания

Преговор

- В досегашните лекции разгледахме
  - Понятието за софтуерна архитектура и архитектурни изгледи (структури, перспективи)
  - Най-често изискваните от една софтуерна система *качества* (нефункционални изисквания)
  - Как те се формализират, така че дефинициите им да не се припокриват и разделихме качествата на системни (технологични), бизнес и архитектурни.
  - Набор от подходящи архитектурни решения, доказали се в практиката, които решават конкретни проблеми
- В следващите лекции ще обобщим шаблоните като разгледаме на абстрактно ниво конкретни техники (тактики) за удовлетворяване на качествените изисквания.

Въведение

- Как става така, че един дизайн притежава висока надеждност, друг висока производителност, а трети – висока сигурност?
- Постигането на тези качества е въпрос на фундаментални архитектурни решения – тактики.
- Тактиката е архитектурно решение, чрез което се контролира ***резултата*** на даден сценарий за качество.
- Наборът от конкретни тактики се нарича архитектурна стратегия.

На днешната лекция

- Тактики за постигане на
  - Изправност/наличност (dependability/availability)
  - Производителност (performance)

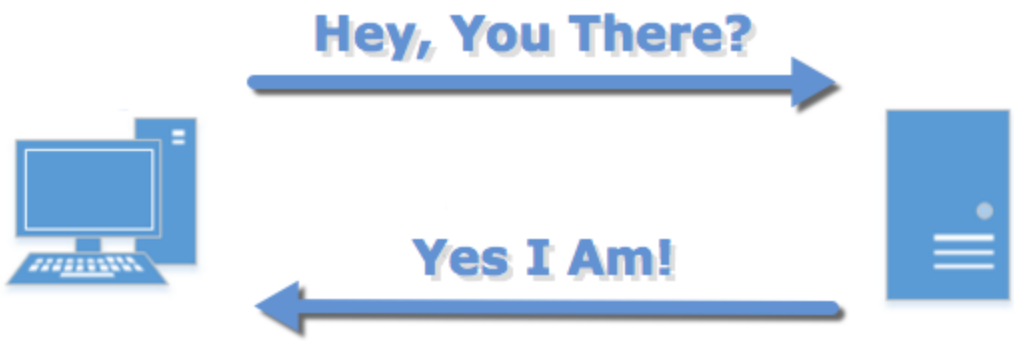
Тактики за изправност

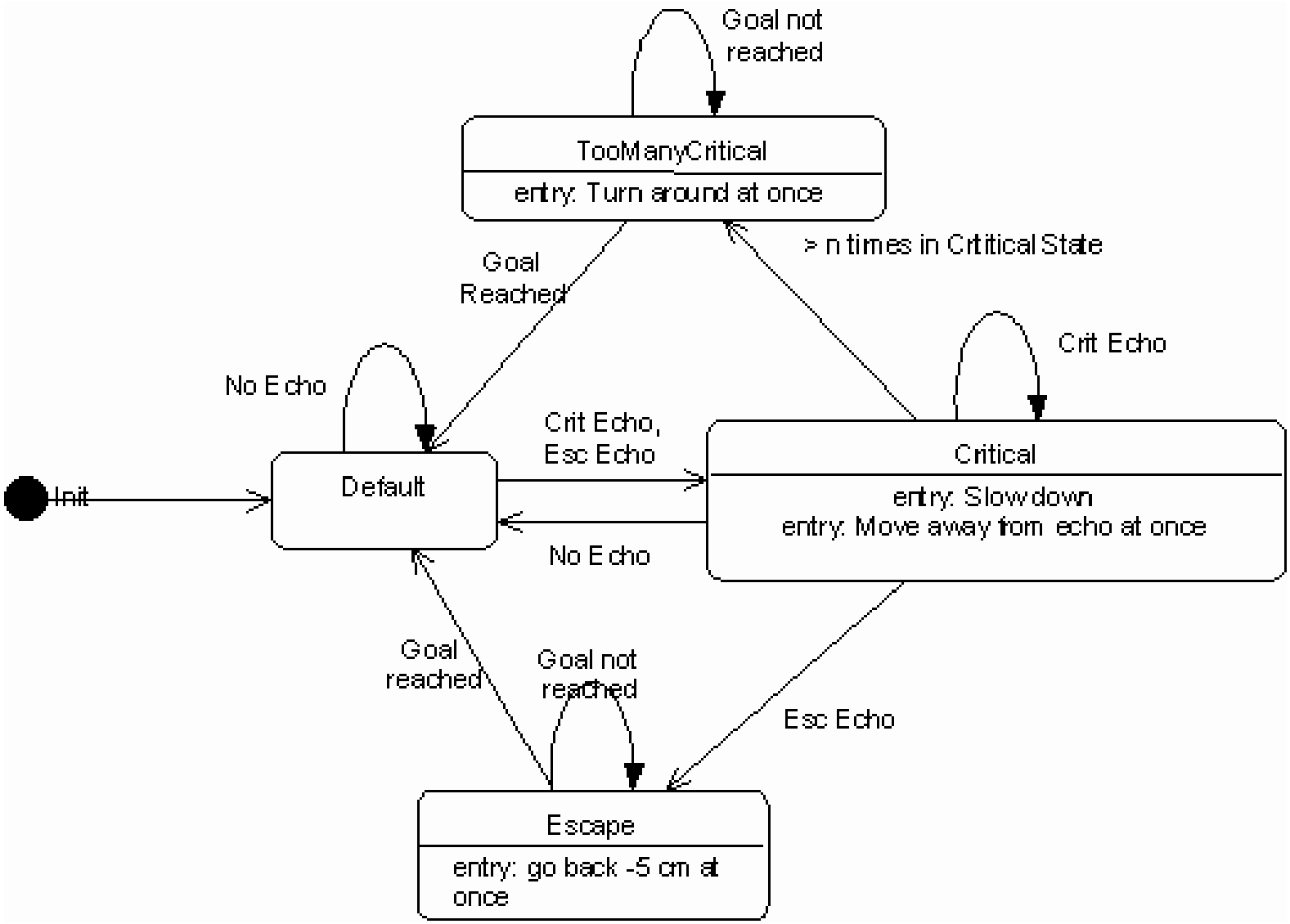
- Откриване и предпазване от откази
- Отстраняване на откази
- Повторно въвеждане в употреба

ТАКТИКИ ЗА ИЗПРАВНОСТ – ОТКРИВАНЕ НА ОТКАЗИ

Откриване на откази

- ***Exo (Ping/echo)*** – компонент А пуска сигнал до компонент Б и очаква да получи отговор в рамките на **определен интервал от време**. Ако отговорът не се получи навреме, се предполага, че в компонент Б (или в комуникационния канал до там) е настъпила повреда и се задейства процедурата за отстраняване на повредата.




Goal not reached
TooManyCritical
entry: Turn around at once
> n times in Critical State
Goal Reached
Crit Echo
No Echo
Crit Echo, Esc Echo
Init
Default
Critical
entry: Slow down
entry: Move away from echo at once
No Echo
Goal reached
Goal not reached
Esc Echo
Escape
entry: go back -5 cm at once

Откриване на откази

- ***Heartbeat, Keepalive*** – даден компонент периодично излъчва сигнал, който друг компонент очаква. Ако сигналът не се получи, се предполага, че в компонент А е настъпила повреда и се задейства процедура за отстраняване на повредата.
- Сигналът може да носи и полезни данни – напр., банкоматът може да изпраща журнала от последната транзакция на даден сървър. Сигналът не само действа като heartbeat, но и служи за лог-ване на извършените транзакции;

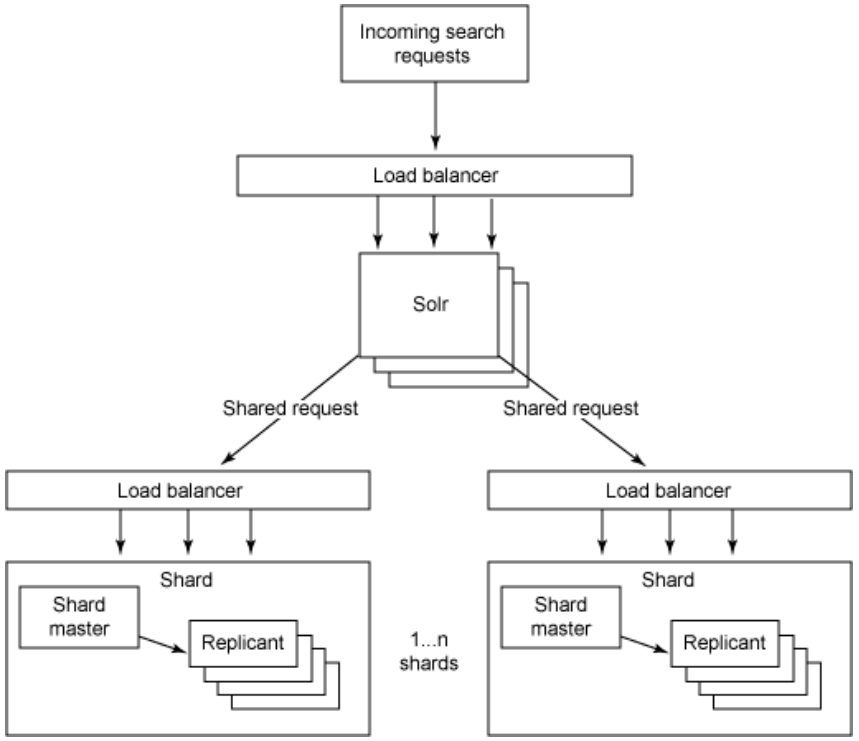
Откриване на откази

- ***Изключения (Exceptions)*** – обработват се изключения, които се генерират, когато се стигне до определено състояние на отказ.
- Обикновено процедурата за обработка на изключения се намира в процеса, който генерира самото изключение.

ТАКТИКИ ЗА ИЗПРАВНОСТ – ОСТСТРАНЯВАНЕ НА ОТКАЗИ

Какъв е основният подход за предотвратяване на откази?

- Репликация (Redundancy, Излишък)
- Стандартен подход
- От интерес са т.нар. гранични случаи



Отстраняване на откази

- ***Активен излишък (Active redundancy, hot restart)*** – важните компоненти в системата са дублирани (вкл. многократно). Дублираните компоненти се поддържат в едно и също състояние (чрез подходяща синхронизация, ако се налага това). Използва се резултатът само от единния от компонентите (т.н. активен);
- Обикновено се използва в клиент/сървър конфигурация, като напр. СУБД, където се налага бърз отговор дори при срив.
- Освен излишък в изчислителните звена се практикува и излишък в комуникациите, в поддържащият хардуер и т.н.
- Downtime-ът обикновено се свежда до няколко милисекунди, тъй като резервният компонент е готов за действие и единственото, което трябва да се направи е той да се направи активен.

Отстраняване на откази

- ***Пасивен излишък (Passive redundancy, warm restart)*** – Един от компонентите (основният) реагира на събитията и информира останалите (резервните) за промяната на състоянието.
- При откриване на отказ, преди да се направи превключването на активния компонент, системата трябва да се увери, че новият активен компонент е в достатъчно осъвременено състояние.
- Обикновено се практикува периодично форсиране на превключването с цел повишаване на надеждността
- Обикновено downtime-ът е от няколко секунди до няколко часа.
- Синхронизацията се реализира от активния компонент.

Основни предизвикателства

- Синхронизация на състоянието на отделните дублирани модули
- Данните трябва да са консистентни във всеки един момент
- Има ли *отстраняване на откази* при копиране на един и същи код?

Разнородност

- отказите в софтуера обикновено се предизвикват от грешки при проектирането
- Мултиплицирането на грешка в проектирането чрез репликация не е добра идея
- Просто увеличаване на броя идентични копия на програмата (подобно на техниките в хардуера) не е решение
- Трябва да се въведе разнородност в копията на програмата

Разнородност в проектирането

- Различни програмни езици
- Различни компилатори
- Различни алгоритми
- Ограничен/липса на контакт между отделните екипи
- Модул за избор на резултат от изпълнението на отделните копия

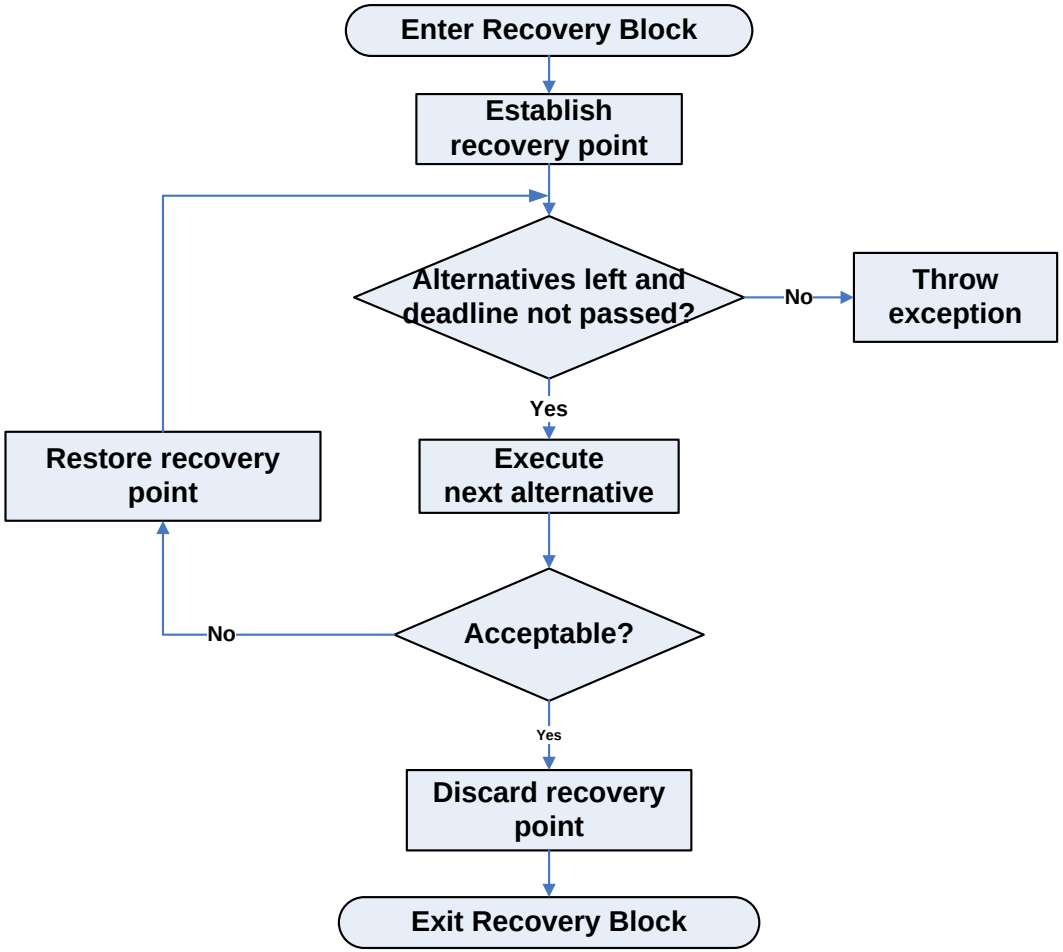
Техники за постигане на разнородност в проектирането

- Recovery Blocks
- Програмиране на N на брой версии (N-version programming)
- И др.

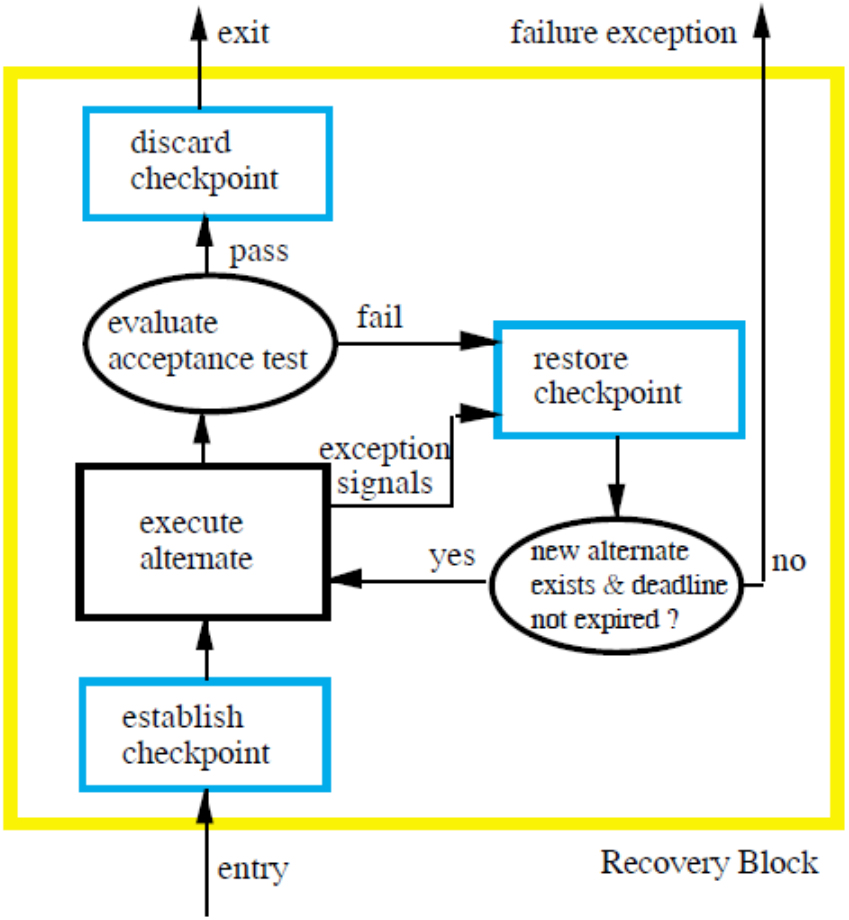
Recovery Blocks

- Разработват се няколко алтернативни модула на програмата
- Извършват се тестове за одобрение, за да се определи дали получения резултат е приемлив

Алгоритъм за recovery blocks

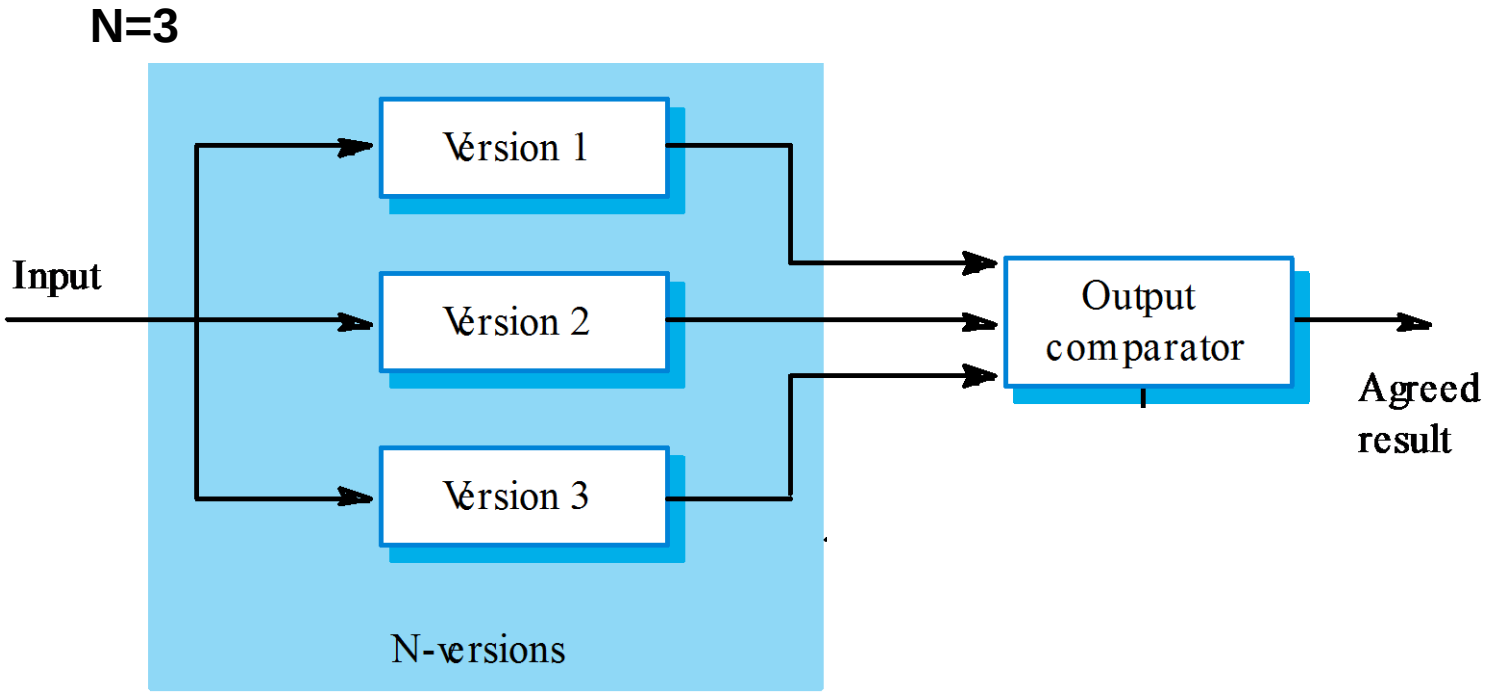


Принцип на Recoverv blocks



Източник: Randell, Brian, and Jie Xu. "The evolution of the recovery block concept.“ , 1995.

N-version programming



Източник: Sommervile, I. “Software Engineering”

N-version programming

- ***Voting*** - На различни процесори работят еквивалентни процеси, като всички те получават един и същ вход и генерират един и същ резултат, който се изпраща на т.н. voter (output comparator), който решава крайния резултат от изчислението. Ако някои от процесите произведе изход, който се различава значително от останалите, voter-ът решава да го изключи от обработката.
- Алгоритъмът за изключване на процес от обработката може да бъде различен и изменян, напр. отхвърляне чрез мнозинство, предпочитан резултат и т.н.

Пример за програмиране на N на брой версии

- Система за контрол на полета в самолетите Boeing 777
  - Използване е един език за програмиране - ADA
  - 3 различни среди за разработка
  - 3 различни компилатора
  - 3 различни процесора

Разнородност по време

- Предполага възникването на определени събития, които касаят работата на програмата, по различно време
- Методи за реализация на разнородност по време
  - Чрез стартиране на изпълнението в различни моменти от време
  - Чрез подаване на данни, които се използват или четат в различни моменти от времето

Тактики за Изправност – отстраняване на откази

- ***Извеждане от употреба (Removal from service)*** – премахва се даден компонент от системата, за да се избегнат очаквани сривове.
- Типичен пример – периодичен reboot на сървърите за да не се получават memory leaks и така да се стигне до срив.
- Извеждането от употреба може да става автоматично и ръчно, като и в двата случая това следва да е предвидено в системата на ниво архитектура.
- Предполага наличието на модул за наблюдение (monitoring)

ТАКТИКИ ЗА ИЗПРАВНОСТ – ПРИ ПОВТОРНО ВЪВЕЖДАНЕ В УПОТРЕБА

Повторно въвеждане в употреба

- ***Паралелна работа (shadow mode)*** – преди да се въведе в употреба компонент, който е бил повреден, известно време се оставя той да работи в паралел в системата, за да се уверим, че се държи коректно, точно както работещите компоненти.

Тактики за Изправност – повторно въвеждане в употреба

- ***Контролни точки и rollback (Checkpoint/rollback)*** – Контролната точка е запис на консистентно състояние, създаван периодично или в резултат на определени събития.
- Понякога системата се разваля по необичаен начин и изпада в не-консистентно състояние. В тези случаи, системата се възстановява (rollback) в последното консистентно състояние (съгласно последната контролна точка) и журнала на транзакциите, които са се случили след това.

ТАКТИКИ ЗА ПРОИЗВОДИТЕЛНОСТ

Тактики за производителност

- Целта на тактиките за производителност е да се постигне реакция от страна на системата на зададено събитие в рамките на определени времеви изисквания.
- За да реагира системата е нужно време, защото:
  - Ресурсите, заети в обработката го консумират;
  - Защото работата на системата е блокирана поради съревнование за ресурсите, не-наличието на такива, или поради изчакване на друго изчисление;
- Тактиките за производителност са разделени в три групи:
  - Намаляване на изискванията;
  - Управление на ресурсите;
  - Арбитраж на ресурсите;

Тактики за производителност – намаляване на изискванията

- Увеличаване на производителността на изчисленията – подобряване на алгоритмите, замяна на един вид ресурси с друг (напр. кеширане) и др.
- Намаляване на режийните (overhead) – не-извършване на всякакви изчисления, които не са свързани конкретно с конкретното събитие (което веднага изключва употребата на посредници)
- Промяна на периода – при периодични събития, колкото по-рядко идват, толкова по-малки са изискванията към ресурсите.
- Промяна на тактовата честота – ако върху периода, през който идват събитията нямаме контрол, тогава можем да пропускаме някои от тях (естествено, с цената на загубата им)
- Ограничаване на времето за изпълнение – напр. при итеративни алгоритми.
- Опашка с краен размер – заявките, които не могат да се обработят веднага, се поставят в опашка; когато се освободи ресурс, се обработва следващата заявка; когато се напълни опашката, заявките се отказват.

Тактики за производителност – управление на ресурсите

- ***Паралелна обработка*** – ако заявките могат да се обработват паралелно, това може да доведе до оптимизация на времето, което системата прекарва в състояние на изчакване;
- ***Излишък на данни/процеси*** – cache, load-balancing, клиентите в c/s и т.н.
- ***Включване на допълнителни ресурси*** – повече (и по-бързи) процесори, памет, диск, мрежа и т.н.

Тактики за производителност – арбитраж на ресурсите

- Когато има *недостиг* на ресурси (т.е. спор за тях), трябва да има *някой (напр. точно определен модул)*, която да решава (т.е. да извършва арбитраж) кое събитие да се обработи *с предимство*. Това се нарича **scheduling**.
- В scheduling-а се включват два основни аспекта – как се приоритизират събитията и как се предава управлението на избраното високо-приоритетно събитие.

Тактики за производителност – арбитраж на ресурсите

- Някои от основните scheduling алгоритми са:
  - FIFO – всички заявки са равноправни и те се обработват подред;
  - Фиксиран приоритет – на различните заявки се присвоява различен фиксиран приоритет; пристигащите заявки се обработват по реда на техния приоритет. Присвояването става съгласно:
    - Семантичната важност;
    - Изискванията за навременност;
    - Изискванията за честота;
  - Динамичен приоритет:
    - Последователно;
    - На следващото събитие, изискващо навременност;
  - Статичен scheduling – времената за прекъсване и реда за получаване на ресурси е предварително дефиниран.

МЕТОДИ ЗА ДИЗАЙН НА СОФТУЕРНАТА АРХИТЕКТУРА

Тактики за удовлетворяване на качествените изисквания

Част 2

Тактики за изменяемост (*modifiability*)

- Тактиките за постигане на изменяемост също се разделят на няколко групи, в зависимост от техните цели
  - ***Локализиране на промените*** – целта е да се намали броят на модулите, които са директно засегнати от дадена промяна
  - ***Предотвратяване на ефекта на вълната*** – целта е модификациите, необходими за постигането на дадена промяна, да бъдат ограничени само до директно засегнатите модули
  - ***Отлагане на свързването*** – целта е да се контролира времето за внедряване и себестойността на промяната

ТАКТИКИ ЗА ИЗМЕНЯЕМОСТ – ЛОКАЛИЗИРАНЕ НА ПРОМЕНИТЕ

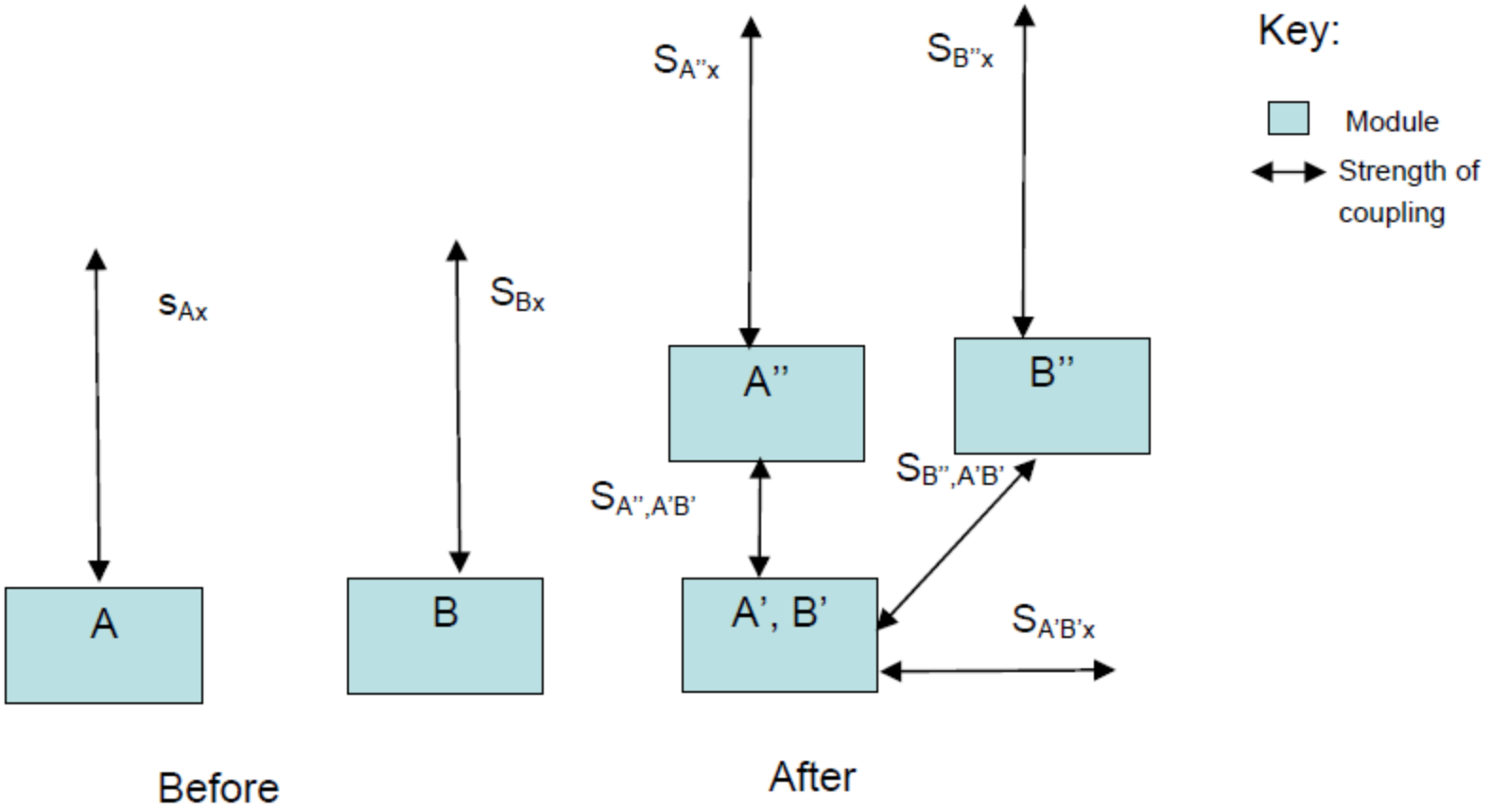
Локализиране на промените

- Въпреки, че няма пряка връзка между броя на модулите, които биват засегнати от дадена промяна и себестойността на извършване на промените, е ясно, че ако промените се ограничат във възможно най-малък брой модули, цената ще намалее.
- Целта на тази група тактики е отговорностите и задачите да бъдат така разпределени между модулите, че обхватът на очакваните промени да бъде ограничен.

Локализиране на промените

- **Поддръжка на семантична свързаност** – семантичната свързаност (semantic coherence) се отнася до отношенията между отговорностите в рамките на даден модул.
- Целта е задачите да се разпределят така, че тяхното изпълнение и реализация да не зависят прекалено много от други модули.
- Постигането на тази цел става като се обединят в рамките на един и същ модул функционалности, които са семантично свързани, при това разглеждани в контекста на очакваните промени.
- Пример за подход в тази насока е и използването на общи услуги (напр. чрез използването на стандартизирани application frameworks или middleware);

Семантична свързаност



Bachmann, F., L. Bass and R. Nord. Modifiability Tactics. SEI Technical Report. September 2007

Локализиране на промените

- **Очакване на промените –** прави се списък на най-вероятните промени (това е трудната част). След което, за всяка промяна се задава въпроса “Помага ли така направената декомпозиция да бъдат локализирани необходимите модификации за постигане на промяната?”.
- Друг въпрос, свързан с първия е “Случва ли се така, че фундаментално различни промени да засягат един и същ модул?”
- За разлика от поддръжката на семантична свързаност, където се очаква промените да са семантично свързани, тук се набляга на конкретните най-вероятни промени и ефектите от тях.
- На практика двете тактики се използват заедно, тъй като списъкът с най-вероятни промени никога не е пълен; доброто обмисляне и съставянето на модули на принципа на семантичната свързаност в много от случаите допълва така направения списък;

Локализиране на промените

- **Ограничаване на възможните опции** – промените, могат да варират в голяма степен и следователно да засягат много модули.
- Ограничаването на възможните опции е вариант за намаляване на този ефект.
  - Напр., вариационна точка в дадена фамилия архитектури (продуктова линия – product line) може да бъде конкретното CPU. Ограничаването на смяната на процесори до тези от една и съща фамилия е възможна тактика за ограничаване на опциите.

ТАКТИКИ ЗА ИЗМЕНЯЕМОСТ – ПРЕДОТВРАТЯВАНЕ НА ЕФЕКТА НА ВЪЛНАТА

Предотвратяване на ефекта на вълната

- Ефект на вълната има тогава, когато се налагат модификации в модули, които не са директно засегнати от дадена промяна.
- Напр., ако *модул А* се модифицира, за да се реализира някаква промяна и се налага модификацията на *модул Б* само защото *модул А* е променен. В този смисъл *Б* зависи от *А*.

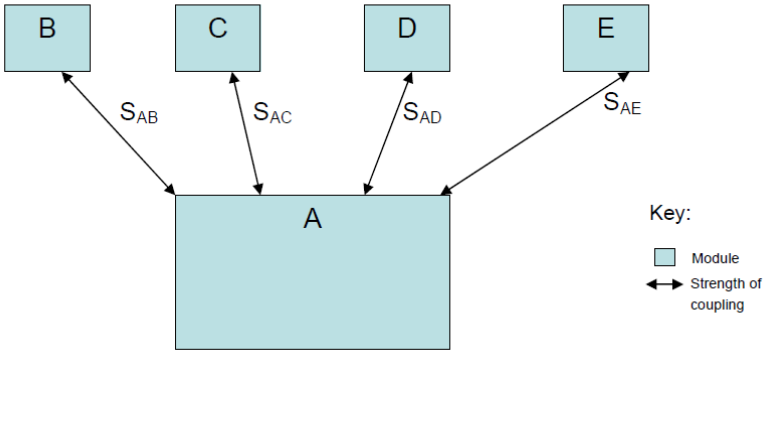
Предотвратяване на ефекта на вълната

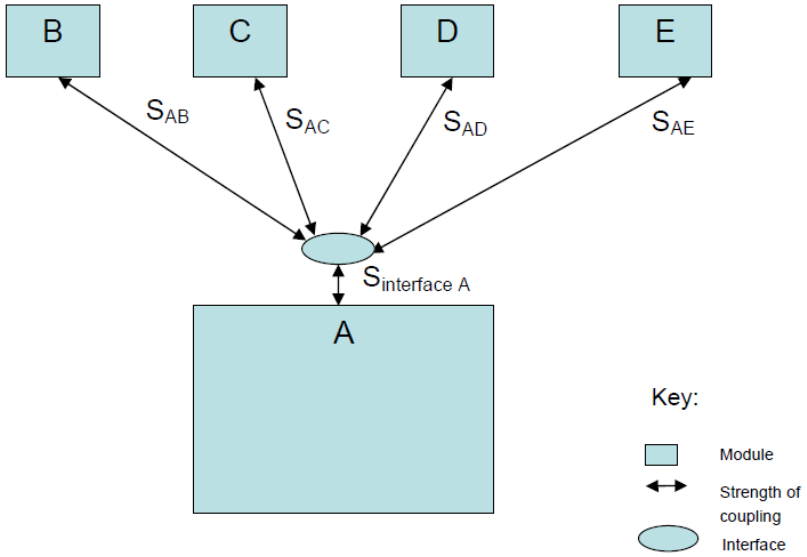
- ***Скриване на информация*** – декомпозиция на отговорността на даден елемент (система или конкретен модул) и възлагането ѝ на по-малки елементи, като при това част от информацията остава публична и част от нея се скрива.
- Публичната функционалност и данни са достъпни посредством специално дефинирани за целта интерфейси.
- Това е най-старата и изпитана техника за ограничаване на промените и е пряко свързана с “очакване на промените”, тъй като именно списъка с очакваните промени е водещ при съставянето на декомпозицията, така че промените да бъдат сведени в рамките на отделни модули.

Предотвратяване на ефекта на вълната

- ***Ограничаване на комуникацията*** чрез ограничаването на модулите, с които даден модул обменя информация (доколкото това е възможно)
- Т.е. – ограничават се модулите, които консумират данни, създадени от модул А и се ограничават модулите, които създават информация, която се използва от модул А.

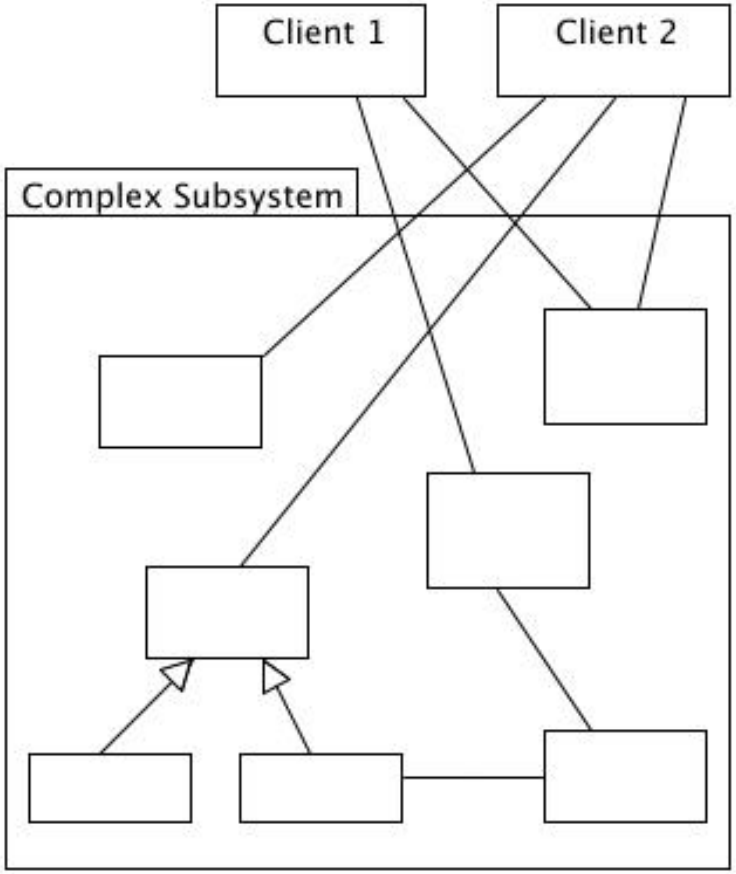
Скриване на информация

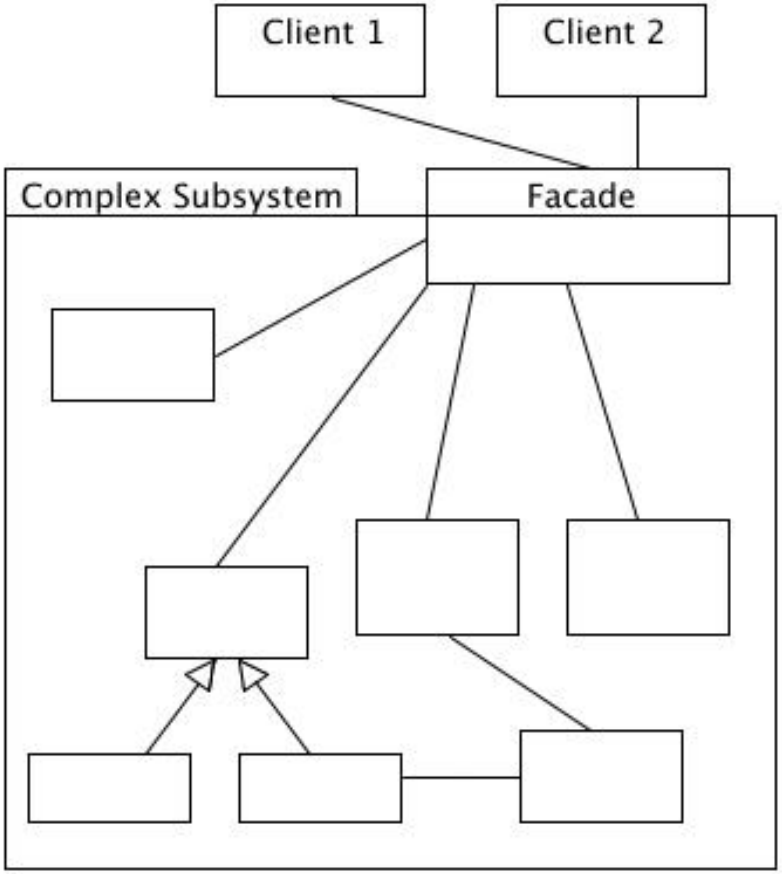




Bachmann, F., L. Bass and R. Nord. Modifiability Tactics. SEI Technical Report. September 2007

Facade





Source: http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/facade.html

Предотвратяване на ефекта на вълната

- ***Поддръжка на съществуващите интерфейси*** – Ако Б зависи от името и сигнатурата на даден интерфейс от А, то съответният синтаксис трябва да се поддържа непроменен. За целта се прилагат следните техники:
  - Добавяне на нов интерфейс – вместо да се сменя интерфейс се добавя нов;
  - Добавя се адаптер – А се променя и същевременно се добавя адаптер, чрез който се експлоатира стария синтаксис;
  - Създава се stub – ако се налага А да се премахне, на негово място се оставя stub – процедура със същия синтаксис, която обаче не прави нищо (NOOP);
  - И т.н.

Предотвратяване на ефекта на вълната

- ***Използване на посредник*** – Ако Б зависи по някакъв начин от А (освен семантично), е възможно между А и Б да бъде поставен „*посредник*“, които премахва тази зависимост.
  - Посредник – wrapper, mediator, façade, adaptor, proxy и т.н....

ТАКТИКИ ЗА ИЗМЕНЯЕМОСТ – ОТЛАГАНЕ НА СВЪРЗВАНЕТО

Отлагане на свързването

- Двете категории тактики, които разгледахме до тук служат за намаляване на броя на модулите, които подлежат на модификации при нуждата от промяна.
- Само че сценариите за изменяемост включват и елементи, свързани с възможността промени да се правят от не-програмисти, което няма нищо общо с брой модули и т.н.
- За да бъдат възможни тези сценарии се налага да се инвестира в допълнителна инфраструктура, която да позволява именно тази възможност – т.н. отлагане на свързването.

Отлагане на свързването

- Различни “решения” могат да бъдат “свързани” в изпълняваната система по различно време.
- Когато свързването става по време на програмирането, промяната следва да бъде изкомуникирана с разработчика, след което тя да бъде реализирана, да се тества и внедри, и всичко това отнема време.
- От друга страна, ако свързването става по време на зареждането и/или изпълнението, това може да се направи от потребителя/администратора, без намесата на разработчика.
- Необходимата за целта инфраструктура (за извършване на промяната и след това нейното тестване и внедряване) трябва да е вградена в самата система.

Отлагане на свързването

- Съществуват много тактики за отлагане на свързването, по-важните от които са:
  - Включване/изкл./замяна на компоненти, както по време на изпълнението (plug-and-play), така и по време на зареждането (component replacement)
  - Конфигурационни файлове, в които се задават стойностите на различни параметри
  - Дефиниране и придържане към протоколи, които позволяват промяна на компоненти по време на изпълнение

Тактики за сигурност (*Security)*

- Тактиките за сигурност могат да бъдат разделени на:
  - Тактики за устояване на атаките (ключалка на вратата)
  - Тактики за откриване на атаките (аларма)
  - Тактики за възстановяване след атака (застраховка)

Тактики за сигурност – устояване на атаки

- Автентикация на потребителите – проверка за това, дали потребителя е този, за който се представя (пароли, сертификати, биометрика и т.н.)
- Оторизация на потребителите – проверка за това дали потребителя има достъп до определени ресурси
- Конфиденциалност на данните – посредством криптиране (на комуникационните канали и на постоянната памет)
- Интегритет – включване на различни механизми на излишък – чек-суми, хеш-алгоритми и т.н.
- Ограничаване на експозицията, т.е. на местата, чрез които можем да бъдем атакувани
- Ограничаване на достъпа – firewall и др., вкл. и средства за ограничаване на физическия достъп

Тактики за сигурност – възстановяване след атака

- Тактиките за възстановяване след атака са свързани с възстановяване на състоянието и с идентификация на извършителя.
- Тактиките за възстановяване на системата донякъде се препокриват с тактиките за Изправност, тъй като извършена атака може да се разгледа като друг срив в работата на системата. Трябва да се внимава обаче в детайли като пароли, списъци за достъп, потребителски профили и т.н.

Тактики за сигурност – възстановяване след атака

- **Тактиката за идентифициране на атакуващия** е поддръжка на audit trail – копие на всяка транзакция, извършена върху данните, заедно с информация, която идентифицира извършителя по недвусмислен начин. Audit Trail-а може да се използва да се проследят действията на извършителя, с цел осигуряване на невъзможност за отричане (non-repudiation), а също и за възстановяване от атаката.
- Важно е да се отбележи, че audit trail-овете също са обект на атака, така че при проектирането на системата трябва да се вземат мерки достъп до тях да се осигурява само при определени условия.

Тактики за изпитаемост (*Testability*)

- Целта на тактиките за изпитаемост е, да подпомогнат тестването, когато част от софтуерната разработка е приключила
- За фазата преди приключването на разработката за това обикновено се прилагат тактики като *code review*
- За тестване на работеща система обикновено се използва софтуер, който чрез изпълнението на специализирани скриптове подава входни данни на системата и анализира резултата от нейната работа
- Целта на тактиките за изпитаемост е да подпомагат този процес

Тактики за изпитаемост

- Запис и възпроизвеждане – прихващане на информацията, която преминава през даден интерфейс.

  Може да се използва както за генериране на входни данни, така и за запис на изходно състояние с цел последващо сравнение.

Тактики за изпитаемост

- Разделяне на интерфейса от реализацията – позволява замяна на реализацията за тестови цели

Тактики за изпитаемост

- Специализиран интерфейс за тестване – предоставяне на интерфейс за специализиран тестващ софтуер, който е различен от нормалния. Това дава възможност различни мета-данни да бъдат предоставени на тестващия софтуер, които да го управляват и т.н.
- Тестващият софтуер може да тества и тестовия интерфейс, чрез сравнение с резултатите, постигнати чрез използване на нормалния интерфейс.

Тактики за изпитаемост

- Вградени модули за мониторинг – Самата система поддържа информация за състоянието, натовареността, производителността, сигурността и т.н. и я предоставя на определен интерфейс. Интерфейсът може да е постоянен или временен, включен посредством техника за инструментиране.

Тактики за използваемост (*Usability*)

- Използваемостта се занимава с това, колко лесно даден потребител успява да свърши дадена задача и доколко системата подпомага това му действие.
- Различаваме два вида тактики за използваемост, всяка от тях насочена към различна категория “потребители”.
  - Тактики за използваемост по време на изпълнението (*runtime*) – насочени към крайния потребител
  - Тактики за използваемост, свързани с UI – насочени към разработчика на интерфейса. Този вид тактики са силно свързани с тактиките за изменяемост
    - Разделяне на интерфейса от реализацията, например при MVC

Тактики за използваемост – по време на изпълнението (1)

- По време на работа на системата, използваемостта обикновено е свързана с предоставянето на достатъчно информация, обратна връзка и възможност за изпълнение на конкретни команди (cancel, undo, aggregate, multiple views и т.н.)
- В областта на Human-Computer Interaction (HCI) обикновено се говори за “потребителска инициатива”, “системна инициатива” или “смесена инициатива”.
- Напр., когато се отказва дадена команда, потребителя извършва “cancel” (потребителска инициатива) и системата отговаря. По време на извършването на това действие обаче, системата предоставя индикатор за прогреса (системна инициатива).

Тактики за използваемост – по време на изпълнението (2)

- Когато става въпрос за потребителска инициатива, архитектът проектира реакцията на системата както при всички останали функционалности, т.е. трябва да се опише поведението на системата в зависимост от получените входни събития. Напр., при cancel:
  - Системата трябва да е подготвена за cancel (т.е. да има процес, който следи за такава команда, който да не се блокира от действието на процеса, който се отказва);
  - Процесът, който се отказва трябва да се преустанови;
  - Ресурсите, които са били заети да се освободят;
  - Евентуално компонентите, които взаимодействат с отказания процес да се уведомят, за да предприемат съответните действия;

Тактики за използваемост – по време на изпълнението (3)

- Когато става въпрос за системна инициатива, системата трябва да разчита на някаква информация (модел) относно потребителя, задачата или моментното си състояние.
- Различните модели изискват различни входни данни за постигането на инициативата. Тактиките за постигането на използваемост по време на системната инициатива са свързани с избора на конкретен модел.

Тактики за използваемост – по време на изпълнението (4)

- Моделиране на задачата – системата знае какво прави потребителя и може да му помогне в зависимост от създадения модел (напр. Auto-Correction);
- Моделиране на потребителя – моделът съдържа информация за това, както потребителя знае за системата, как работи с нея, какво очаква и т.н. Това позволява на системата да бъде пригодена към поведението на потребителя.
- Моделиране на системата – моделът на системата включва очакваното поведение, така че тя да предоставя адекватна обратна връзка. (напр. предвиждане на времето за обработка, за да бъде показан progress bar).