ТЕХНИКИ (ТАКТИКИ) ЗА ПОСТИГАНЕ НА КАЧЕСТВОТО НА СОФТУЕРА

Преговор

- В досегашните лекции разгледахме
  - Специфични архитектурни решения, доказали се в практиката, които решават конкретни проблеми
  - 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.
- В настоящата лекция 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.

Функционалност и декомпозиция

- Функционалността на дадена система, може да бъде постигната посредством структурирането й по много различни начини
- Ако функционалността беше единственото изискване, то системата може да се реализира и като един монолитен блок, без нуждата от описание на вътрешната структура
- Декомпозицията на модули, прави системата разбираема и осигурява постигането на много други цели.
- В този смисъл, декомпозицията на модули е в голяма степен независима от функционалността

СА и нефункционални изисквания

- **СА определя структурите на системата в контекста на нефункционалните изисквания**. Например, дадена система може да бъде разделена на модули така, че няколко екипа да могат да работят паралелно по тях, което намалява времето за пускане на продукта на пазара. Това на практика е нефункционално (бизнес) изискване.
- Интересът на софтуерния архитект към функционалността обикновено е ограничен до въпроса **доколко тя взаимодейства с тези нефункционални изисквания (най-често ограничавайки ги)**.

Функционалност и качества

- Бизнес целите определят качествата, които трябва да бъдат вградени в архитектурата на системата. Тези качества поставят изисквания отвъд функционалните (описание на основните възможности на системата и услугите, които тя предоставя)
- Въпреки, че функционалността и качествата са тясно свързани, *функционалността често е единственото, което се взема под внимание по време на проектирането*.
- Като следствие много системи се преправят не защото им липсва функционалност, а защото е *трудно да се поддържат, трудно е да се смени платформата, не са скалируеми, прекалено са бавни, или пък са несигурни*.
- СА е тази стъпка в процеса на създаването на системата, в която за пръв път се разглеждат качествените изисквания и в зависимост от тях се създават съответните структури, на които се приписва функционалност.

Функционалност и Качества

- Зависят ли функционалността и качествата едно от друго?
- Решенията, който прави архитекта по време на създаването на СА, са определящи за постигането на необходимите нива на съответните качества
  - Например: дадени решения ще доведат до висока производителност, а други до липсата на такава

На днешната лекция

- Тактики за постигане на
  - Изправност/наличност (dependability/availability)
  - Производителност (performance)
  - Сигурност (Security)
  - Удобство при тестването (testability)
  - Удобство при употребата (usability)

ТАКТИКИ ЗА ИЗПРАВНОСТ

Dependability

Тактики за изправност

- Откриване на откази
- Предотвратяване на откази
- Повторно въвеждане в употреба

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.
- Използва се напр. от група компоненти, които солидарно отговарят за една и съща задача; от клиенти, които проверяват дали даден сървърен обект и комуникационния канал до него работят съгласно очакванията за производителност.
- Вместо един детектор да ping-ва всички процеси, може да се организира йерархия от детектори – детекторите от най-ниско ниво ping-ват процесите, с които работят заедно върху един процесор, докато детекторите от по-високо ниво ping-ват детекторите от по-ниско ниво и т.н. – така се спестява мрежови трафик;

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.

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.

Отстраняване на откази

- ***Резерва (Spare)*** – поддръжка на резервни изчислителни мощности, които трябва да се инициализират и пуснат в действие при отказ на някой от компонентите.
- За целта може да е необходима постоянна памет, в която се записва състоянието на системата и която може да се използва от резервната система за възстановяване на състоянието.
- Обикновено се използва за хардуерни компоненти и работни станции.
- Downtime-ът обикновено е от няколко минути до няколко часа.

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

Аспекти на разнородността в софтуерните системи

- Разнородност в проектирането (design diversity)
- Разнородност по данни (data diversity)
- Разнородност по време (temporal diversity)

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

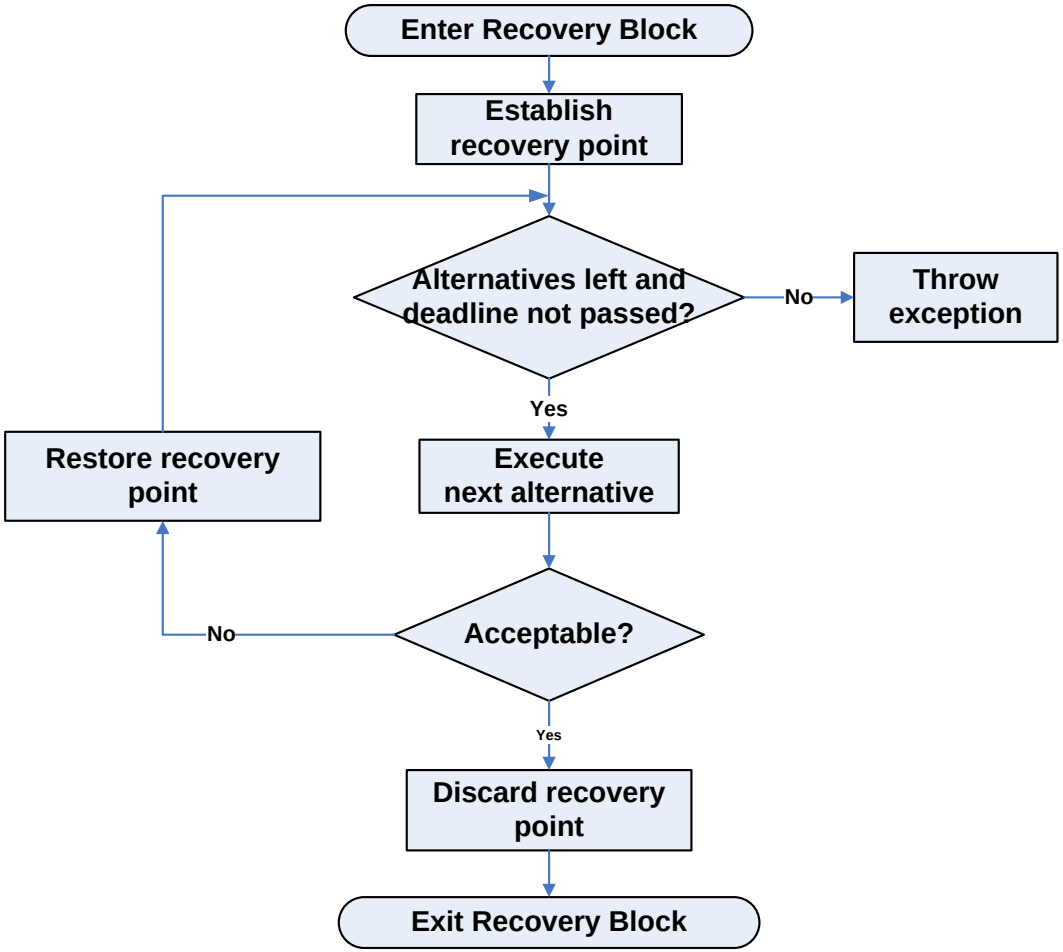
Техники за постигане на разнородност в проектирането

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

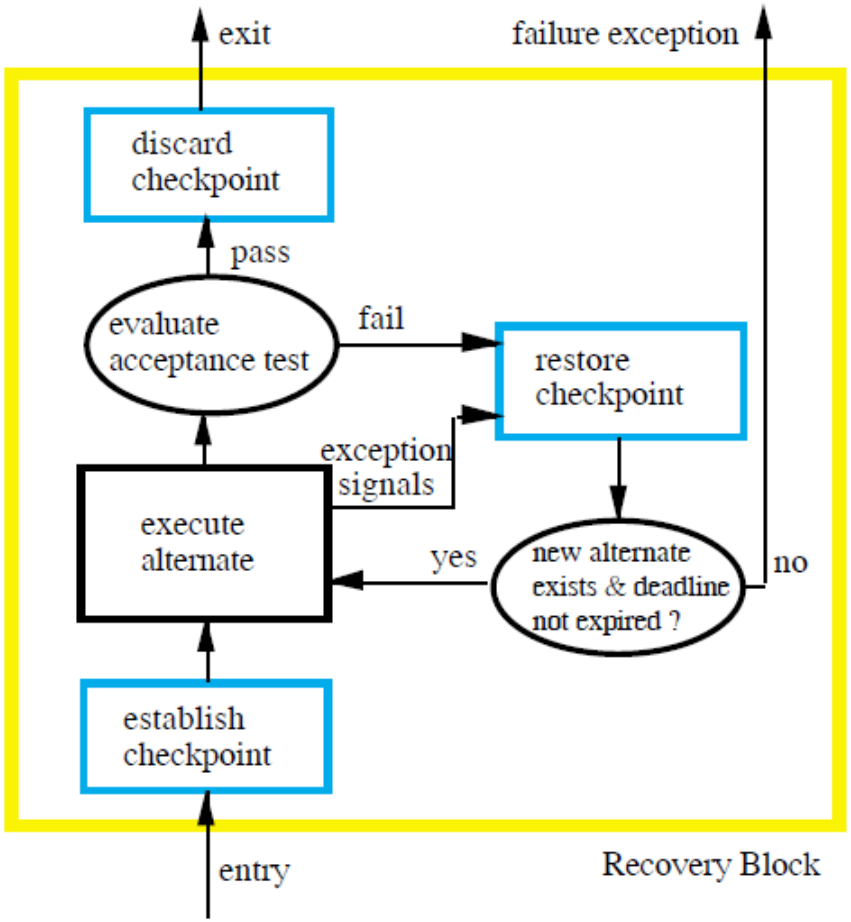
Recovery Blocks

- 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

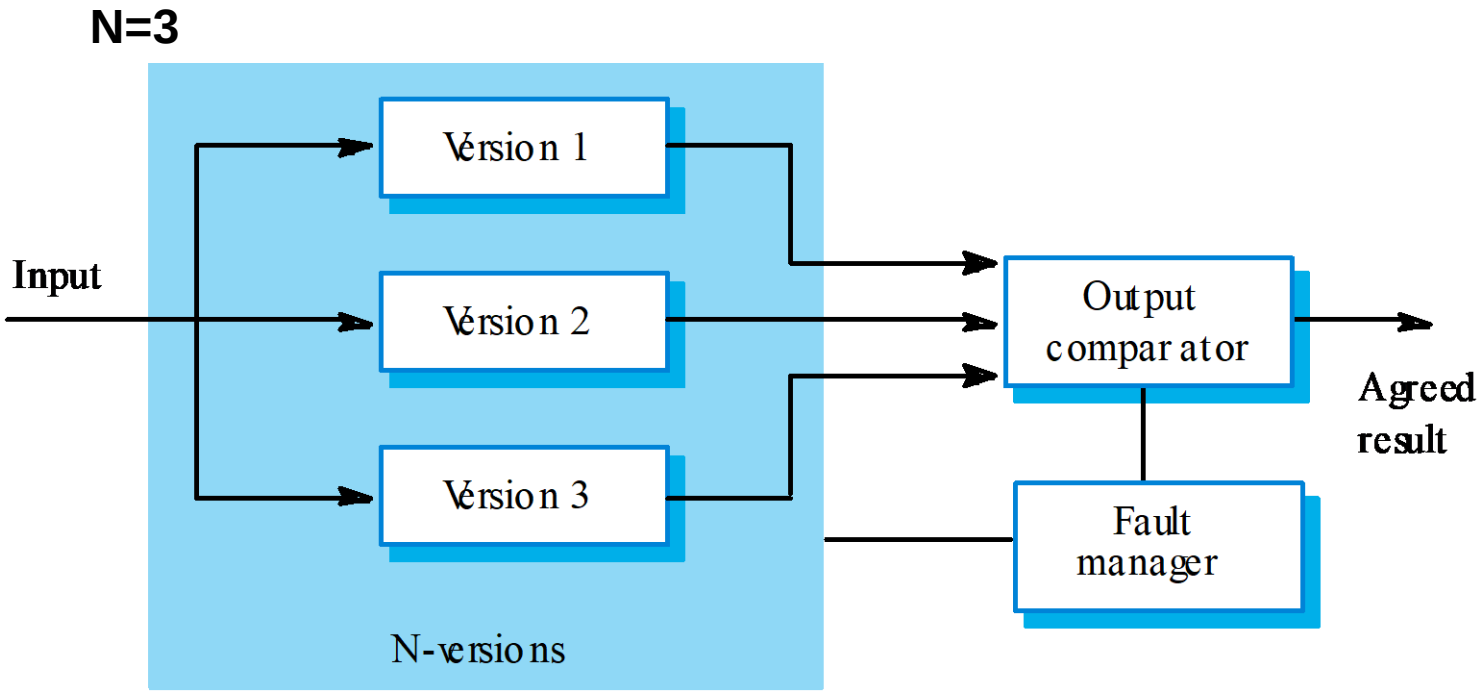


Принцип на Recovery blocks



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 – отстраняване на откази

- **Следене на процесите (Process Monitoring) –** посредством специален процес се следят основните процеси в системата. Ако даден процес откаже, мониторинг процеса може да го премахне, пре-инициализира, да създаде нов екземпляр и т.н.

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

Performance

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 *институция*, 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.

ТАКТИКИ ЗА ИЗМЕНЯЕМОСТ

Modifiability

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

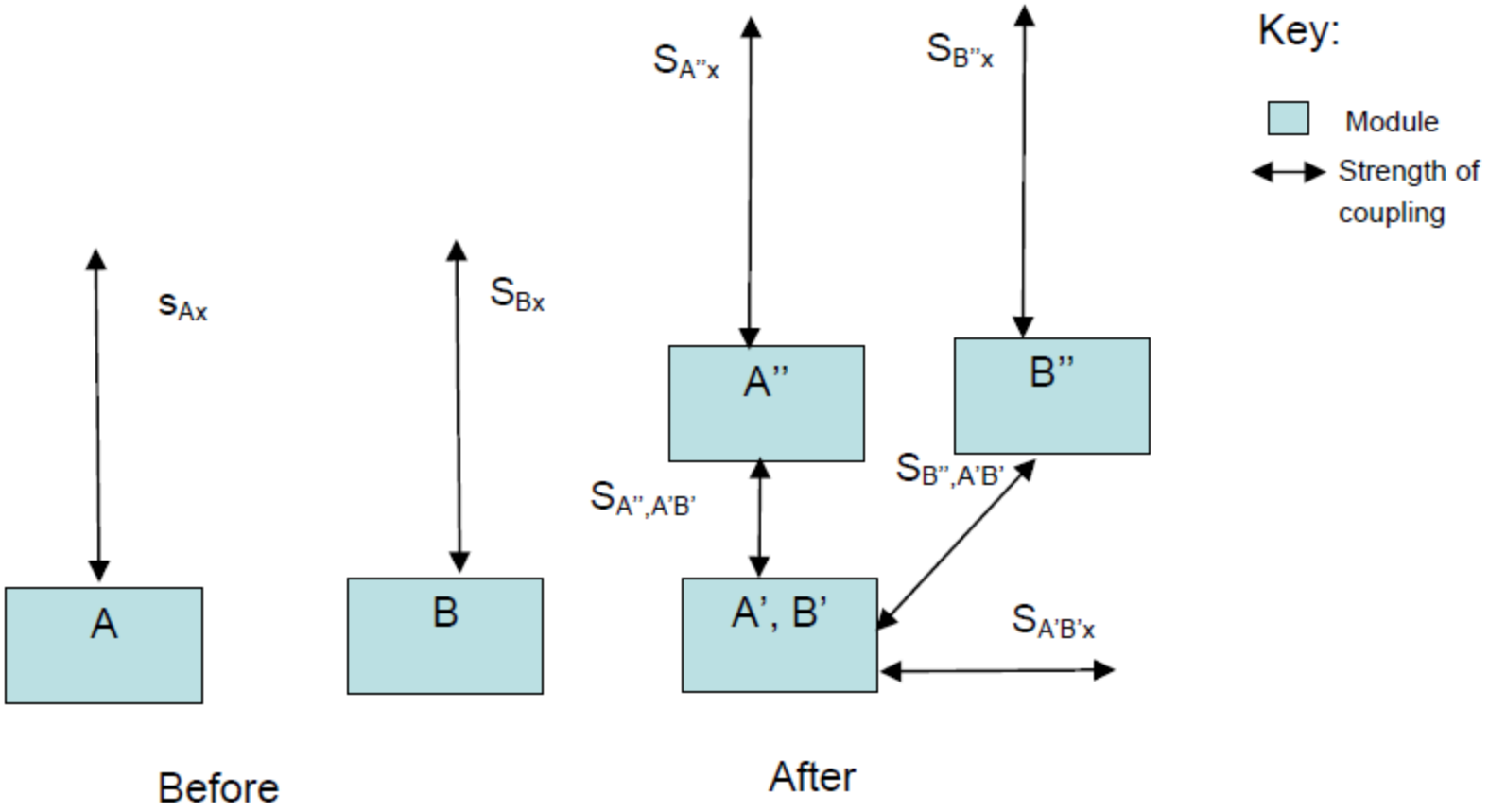
Локализиране на промените

- 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/cohesion) 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.

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*.

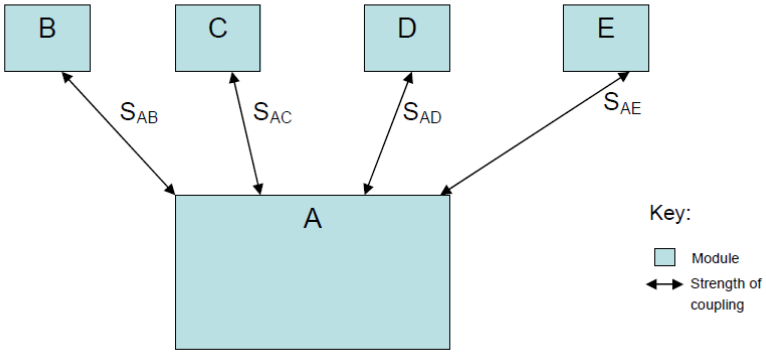
Предотвратяване на ефекта на вълната

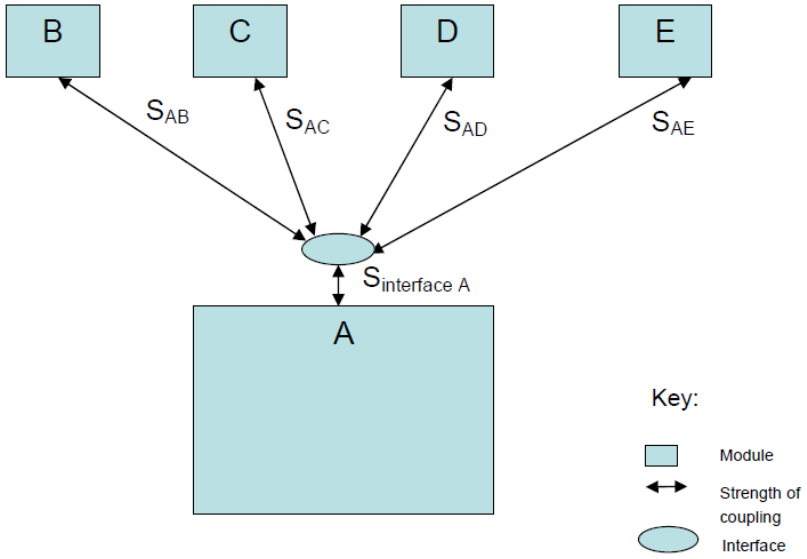
- ***Скриване на информация*** – декомпозиция на отговорността на даден елемент (система или конкретен модул) и възлагането ѝ на по-малки елементи, като при това част от информацията остава публична и част от нея се скрива.
- Публичната функционалност и данни са достъпни посредством специално дефинирани 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.

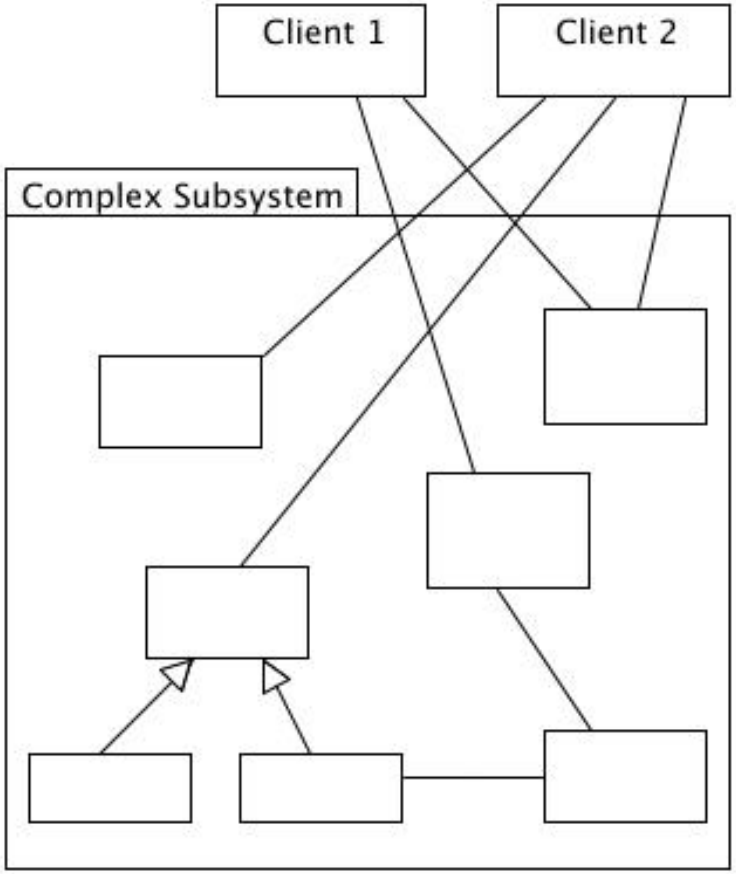
Скриване на информация

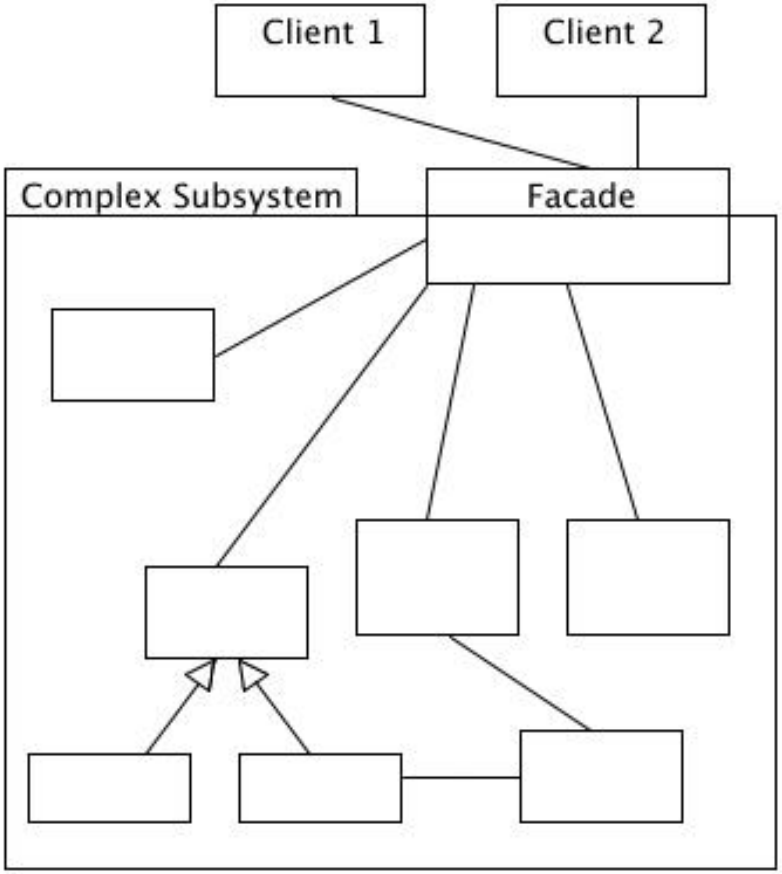




Bachmann, F., L. Bass and R. Nord. Modifiability Tactics. SEI Technical Report. September 2007

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....

- В кои архитектурни стилове се наблюдава приложение на тактиката за предотвратяване на ефекта на вълната?

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